UNITY 3D

Marc Lidon

Programación orientada a objetos // Creación y animación de objetos 3D // Iluminación y partículas

UNITY 3D

UNITY 3D

Marc Lidon

Datos catalográficos
Lidon, Marc
Unity 3D
Primera Edición

Alfaomega Grupo Editor, S.A. de C.V., México

ISBN: 978-607-538-460-3

Formato: 17 x 23 cm Páginas: 560

Diseño de la cubierta y maquetación: ArteMio
Revisor técnico: Pablo Martínez
Correctora: Laura Seoane
Directora de producción: Ma. Rosa Castillo Hidalgo

Unity 3D
Marc Lidon
ISBN: 978-84-267-2682-7 de la edición publicada por MARCOMBO, S.A., Barcelona, España
Derechos reservados © 2019 MARCOMBO, S.A.

Primera edición: Alfaomega Grupo Editor, México, mayo 2019

© 2019 Alfaomega Grupo Editor, S.A. de C.V.
Dr. Isidoro Olvera (Eje 2 sur) No. 74, Col. Doctores, 06720, Ciudad de México.

Miembro de la Cámara Nacional de la Industria Editorial Mexicana
Registro No. 2317

Pág. Web: **http://www.alfaomega.com.mx**
E-mail: **atencionalcliente@alfaomega.com.mx**

ISBN: 978-607-538-460-3

Derechos reservados:
Esta obra es propiedad intelectual de su autor y los derechos de publicación en lengua española han sido legalmente transferidos al editor. Prohibida su reproducción parcial o total por cualquier medio sin permiso por escrito del propietario de los derechos del copyright.

Nota importante:
La información contenida en esta obra tiene un fin exclusivamente didáctico y, por lo tanto, no está previsto su aprovechamiento a nivel profesional o industrial. Las indicaciones técnicas y programas incluidos, han sido elaborados con gran cuidado por el autor y reproducidos bajo estrictas normas de control. ALFAOMEGA GRUPO EDITOR, S.A. de C.V. no será jurídicamente responsable por: errores u omisiones; daños y perjuicios que se pudieran atribuir al uso de la información comprendida en este libro y en el material de apoyo en la web, ni por la utilización indebida que pudiera dársele. d e s c a r g a do en: e y b o oks. c o m

Los nombres comerciales que aparecen en este libro son marcas registradas de sus propietarios y se mencionan únicamente con fines didácticos, por lo que Alfaomega Grupo Editor, S.A. de C.V., México no asume ninguna responsabilidad por el uso que se de a esta información, ya que no infringe ningún derecho de registro de marca. Los datos de los ejemplos y pantallas son ficticios, a no ser que se especifique lo contrario.

Edición autorizada para venta en México y todo el continente americano.

Impreso en México. Printed in Mexico.

Empresas del grupo:

México: Alfaomega Grupo Editor, S.A. de C.V. – Dr. Isidoro Olvera (Eje 2 sur) No. 74, Col. Doctores, Ciudad de México – C.P. 06720. Tel.: (52-55) 5575-5022 – Fax: (52-55) 5575-2420 / 2490. Sin costo: 01-800-020-4396 – E-mail: atencionalcliente@alfaomega.com.mx

Colombia: Alfaomega Colombiana S.A. – Calle 62 No. 20-46, Barrio San Luis, Bogotá, Colombia, Tels.: (57-1) 746 0102 / 210 0415 – E-mail: cliente@alfaomega.com.co

Chile: Alfaomega Grupo Editor, S.A. – Av. Providencia 1443. Oficina 24, Santiago, Chile
Tel.: (56-2) 2235-4248 – Fax: (56-2) 2235-5786 – E-mail: agechile@alfaomega.cl

Argentina: Alfaomega Grupo Editor Argentino, S.A. – Av. Córdoba 1215, piso 10, CP: 1055, Buenos Aires, Argentina, – Tel./Fax: (54-11) 4811-0887 y 4811 7183 – E-mail: ventas@alfaomegaeditor.com.ar

Quiero agradecerle a Jaume Castells la dedicación y el esfuerzo
por enseñarme el camino de la programación:
"Un buen maestro marca la diferencia" y también a Enrique Rendón,
porque en toda buena aventura necesitas un buen compañero.

Plataforma de contenidos interactivos

Para tener acceso al material de la plataforma de contenidos interactivos del libro UNITY 3D, siga los siguientes pasos:

1. Ir a la página: http://libroweb.alfaomega.com.mx

2. Ir a la sección Catálogo y seleccionar la imagen de la portada del libro, al dar doble clic sobre ella, tendrá acceso al material descargable, complemento imprescindible de este libro, el cual podrá descomprimir con la clave UNITY3D.

Nota: Se recomienda respaldar los archivos descargados de la página web en un soporte físico.

Contenido

Capítulo 1. Introducción a Unity 11
1. Introducción 12
2. ¿Qué es Unity? 12
3. Descarga e instalación de Unity 12
4. Proyectos 2D y 3D 14
5. Guardar el proyecto y la escena 16

Capítulo 2. Interfaz de Unity 21
1. Importación de Assets 22
2. Ventana Proyectos (Project) 22
3. Ventana Escena (Scene View) 24
4. Ventana Juego (Game View) 27
5. Ventana Jerarquía (Hierarchy window) 28
6. Ventana Inspector y creación de un GameObject 30

Capítulo 3. Editor de terrenos 39
1. Crear un terreno 40
2. Esculpir la superficie 45
3. Pintar el terreno 48
4. Poner vegetación 51
5. Poner agua en el terreno 54
6. Crear una zona de viento (windzone) 55
7. Editar árboles 56

Capítulo 4. Creación de un escenario modular 65
1. Importar los modelos 66
2. Modelos 67
3. Materiales y texturas 69
4. Parámetros básicos de los materiales 72
5. Colliders y Rigid Bodies 79
6. Model vs Prefabs 83
7. Montar un escenario simple 85
8. Importar Standard Assets y probar el escenario 91

Capítulo 5. Introducción básica de C# con Unity 93
1. Introducción 94
2. Crear y manipular variables 98
3. Trabajar con operadores aritméticos 99
4. Operadores lógicos y de comparación 100
5. Crear declaraciones lógicas con if - else 101
6. Crear declaraciones con switch 103
7. Trabajar con loops 104
8. Crear y llamar funciones 107
9. Entender qué son los Arrays 108
10. Mi primera clase 110

Capítulo 6. Programación orientada a objetos 119
1. Introducción........ 120
2. Clase GameObject 124
3. Acceder a los componentes........ 134
4. Entender las transformaciones 142
5. Vector3........ 146
6. Mover objetos 147
7. Rotar objetos........ 162
8. Escalar objetos 167

Capítulo 7. Creación de un Player en C# 171
1. Introducción........ 172
2. Character Controller 174
3. Movimientos del Character Contoller 177
4. Los Inputs........ 184
5. Mover y rotar nuestro Character Controller 193
6. Saltar obstáculos 197
7. Rigidbody 199
8. Mover un Rigidbody........ 204
9. Controlar el movimiento de un Rigidbody 206
10. Añadir un salto al Rigidbody 208
11. Seguimiento simple de nuestra cámara........ 210
12. Destruir objetos con colisiones 213
13. Tele-transportación con Triggers........ 217
14. Proyecto final 223

Capítulo 8. Raycast y Decals 227
1. Introducción........ 228
2. Abrir puertas con triggers 228
3. Raycast........ 234
4. Cómo obtener información con RaycastHit 241
5. Comunicación con SendMessage 247
6. Decals........ 254
7. Instanciar los Decals con Raycast 261
8. Rotación de nuestros Decals 268
9. Selección de objetos para disparar........ 273
10. Crear un Array de Decals 277

Capítulo 9. UI (Interfaz de Usuario) 289
1. Introducción........ 290
2. Entendiendo el Canvas (lienzo) 290
3. Entendiendo el Rect Transform 297
4. Image........ 304
5. Image Raw 305
6. Text........ 306
7. Button........ 307
8. Slider 309
9. Creación de una mirilla........ 312

10. Creación de un contador de coins 319
11. Creación de una barra de vida 325
12. Hacer daño a nuestro FPSController 331
13. Cómo curar a nuestro FPSController 335
14. Cómo limitar el número de munición 338
15. Pantalla de fallecimiento 348

Capítulo 10. Animación **351**

1. Animación en Unity 352
2. El flujo de trabajo 352
3. Animation Clips 353
4. Ventana Animation 353
5. Animation Controller 360
6. Máquina de estados 362
7. Proyecto de un soldado 365

Capítulo 11. Navigation y Pathfinding **397**

1. Vista general del sistema de navegación en Unity 398
2. Cómo funciona el sistema de navegación 399
3. Construir un NavMesh 401
4. Crear un NavMesh Agent 405
5. Crear un NavMesh Obstacle 410
6. Crear un Off-mesh Link 412
7. Proyectos de Navigation 415

Capítulo 12. Iluminación **419**

1. Introducción 420
2. Iluminación 420
3. Apagar las luces 426
4. Tipos de luces 428
5. Propiedades de las luces 433
6. Iluminación directa e indirecta 434
7. Iluminación Baked 437
8. Iluminación Mixed 439
9. Práctica general 444
10. Ponte a prueba 449

Capítulo 13. Las partículas **451**

1. Introducción 452
2. Sistema de partículas 452
3. Creación de un sistema de partículas 452
4. Editar las propiedades de las partículas 453

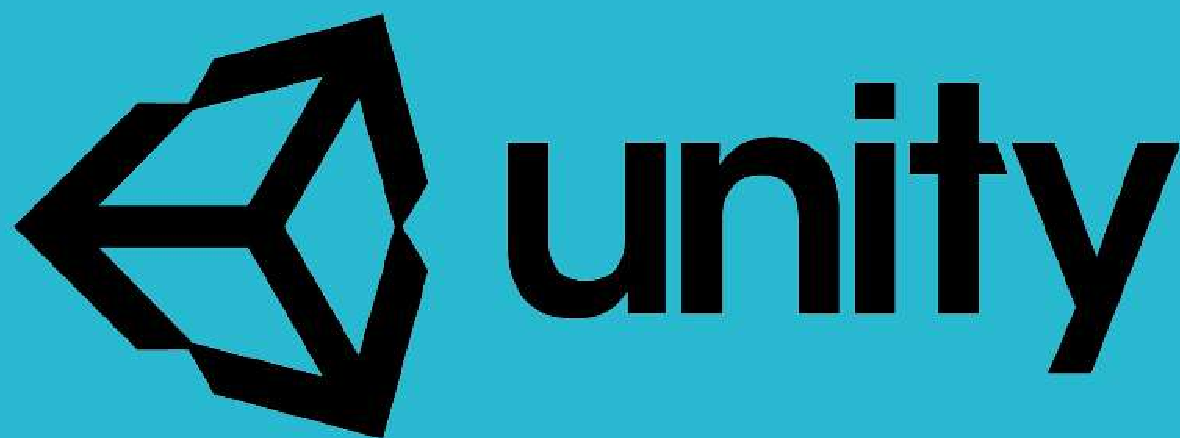
Capítulo 14. Menús y sonido **503**

1. Introducción 504
2. Vista general de sonido 504
3. Empezar con el proyecto 511
4. Creación de nuestro Player 513

5. Crear proyectiles para nuestro player 515
6. Crear un enemigo 523
7. Colisiones 524
8. Explosiones 526
9. Añadir sistema de puntos y vidas 531
10. Escena principal y GameOver 541

Capítulo 1

Introducción a Unity



- Introducción
- ¿Qué es Unity?
- Descarga e instalación de Unity
- Proyectos 2D y 3D
- Guardar el proyecto y la escena

1. Introducción

Primero te doy la bienvenida y las gracias por adquirir esta obra, que pretende documentar de una forma práctica todos los módulos básicos de Unity con todo lo que conlleva aprender un programa de este tipo.

Para empezar con buen pie he creído conveniente hacer una introducción desde cero y crear una base con la que trabajar. Cuando hablo de base me refiero a que el verdadero potencial de Unity reside en la facilidad con la que podemos crear interactividad entre objetos mediante la programación. Entiendo que a muchos les dé miedo la palabra programación por la misma razón que a mí en mis comienzos y es el desconocimiento. Esta es la primera de las razones que nos frenan a la hora de crear grandes proyectos, en todo caso no prometo que después de terminar el libro te conviertas en un súper programador, seamos humildes y tengamos los pies en el suelo, la respuesta es no, pero sí que vas a adquirir conocimientos y habilidades que te permitirán entender cómo se realizan juegos en Unity como empezar a leer código y, lo más importante, a utilizar Unity para crear tus propios proyectos.

2. ¿Qué es Unity?

Unity es un motor de videojuegos que se ha hecho muy popular en los últimos años. En realidad un motor de videojuegos es un conjunto de herramientas que nos facilitan el cálculo de formas geométricas y comportamientos físicos que se utilizan en los videojuegos. Estas herramientas están diseñadas para agilizar el proceso de creación de contenido del juego y no para la resolución de problemas informáticos.

Una de las características que hacen de este motor un referente en la industria de videojuegos es que nos permite la importación de muchos formatos 3D como 3ds, Cinema4D, Blender, FBX y también importar recursos de tipo gráfico, visual y de audio, todo ello posteriormente puede ser optimizado por Unity.

Unity permite construir juegos mediante su editor y un lenguaje de programación, que permite al usuario mediante scripts crear interacción. El usuario puede escoger entre Java Script, C# como lenguajes de programación y podrá consultar en la documentación de las API que proporciona Unity. En esta obra nos centraremos en el lenguaje C#-

Los juegos en Unity se crean mediante escenas que representaran un nuevo nivel o un lugar distinto dentro del juego. Las escenas se crean con el editor de terrenos de Unity o importando modelos propios.

Si no sabes modelar en 3d no te preocupes porque este libro viene acompañado de material adicional para que puedas realizar los proyectos. En el caso de que quieras realizar algún proyecto distinto puede descargar del Asset Store de Unity recursos, en donde encontraras material gratuito y de pago.

3. Descarga e instalación de Unity

Para empezar a trabajar con Unity te recomiendo que visites su página web **https://store.unity.com/es/** y descargues la última versión que tengan en el caso de que no hayas instalado el programa.

Dispones de varas opciones de descarga, para seguir los proyectos del libro puedes descargar la versión gratuita. Esta versión gratuita tiene las herramientas necesarias para aprender a utilizar el programa.

Fig. 1.1

Una vez tengas descargado el paquete instala Unity haciendo clic en el instalador. Un aspecto que te recomiendo es el de no instalar todas las opciones que vienen por defecto porque alguna no las vamos a necesitar y van a ocupar un espacio valioso en su disco duro.

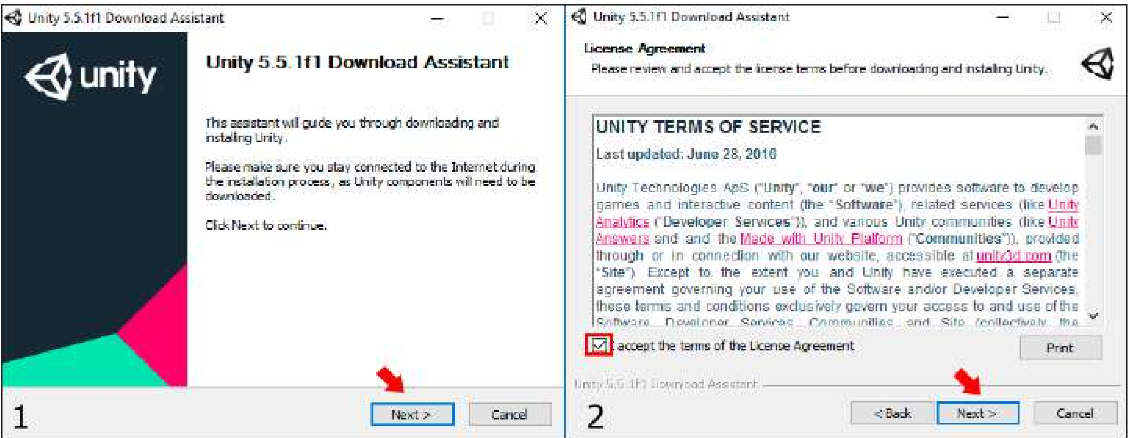


Fig. 1.2

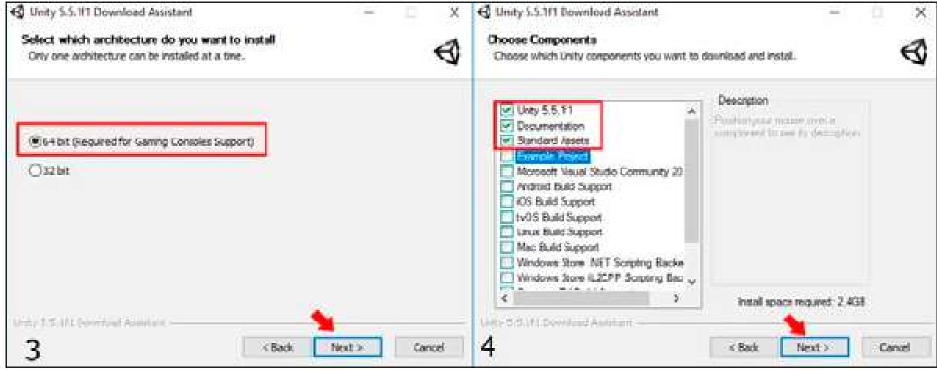


Fig. 1.3

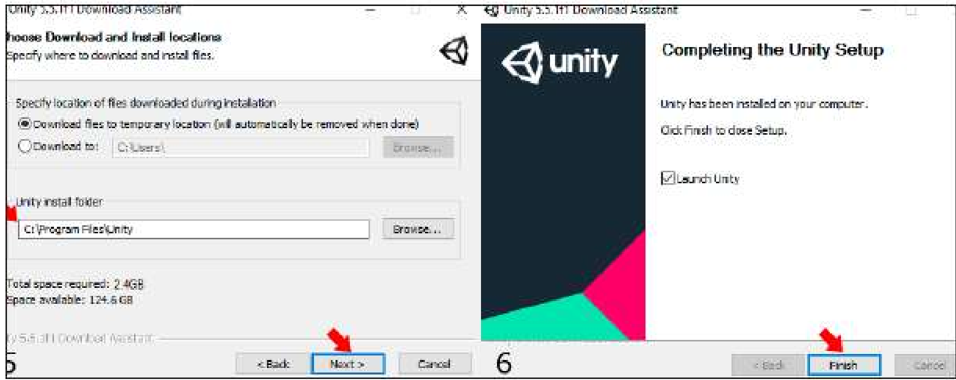


Fig. 1.4

4. Proyectos 2D y 3D

Una vez instalado el programa en tu ordenador, arrancaremos el programa haciendo doble clic en el icono de Unity del escritorio.

Al arrancar el programa se nos abrirá una ventana que nos da la bienvenida y nos pregunta si queremos trabajar offline (sin conexión) o accediendo con una cuenta de Unity.

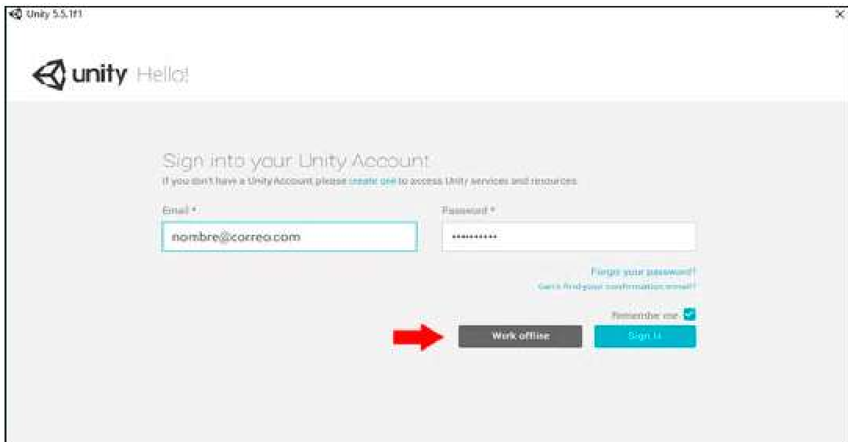


Fig. 1.5

Si pulsamos la opción Work offline se nos muestra la siguiente ventana que nos permite abrir o crear un nuevo proyecto. A continuación voy a resumir que opciones tenemos en esta ventana.

Fig. 1.6

- **La opción 1**: para crear un nuevo proyecto.
- **La opción 2**: para abrir un proyecto que ya tengamos creado. Si te fijas en la parte de abajo veras dos flechas que te indican un menú en donde se muestran los proyectos recientes que tengamos creados. La opción ***On Disk*** muestra los proyectos guardados en el ordenador y la opción ***In the Cloud*** muestra los proyectos que tengamos creados en la nube.
- **La opción 3**: te volverá a mostrar la ventana de acceso a tu cuenta Unity. En el caso de que accedas con una cuenta la ventana te mostrara un icono en la parte superior que viene por defecto o tu avatar personal.

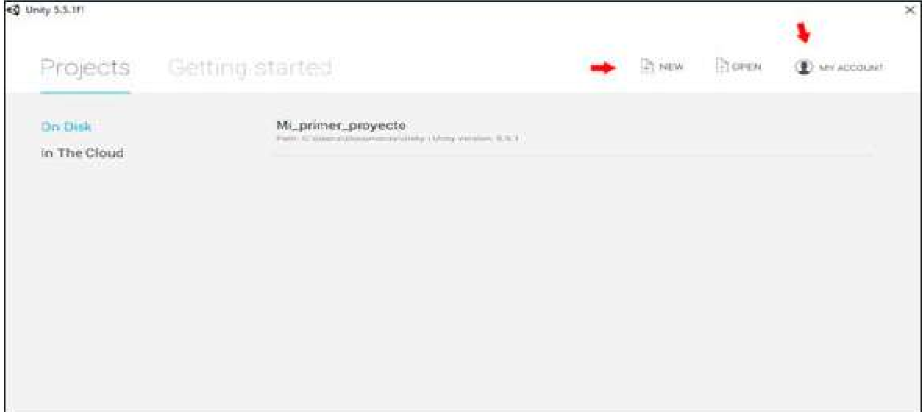


Fig. 1.7

Después de explicar todo lo anterior ya podemos crear un nuevo proyecto, para ello pulsa en el botón **New** que muestra la primera flecha de la imagen anterior. Se nos aparecerá otra ventana que permite, ahora sí, crear y guardar nuestro proyecto.

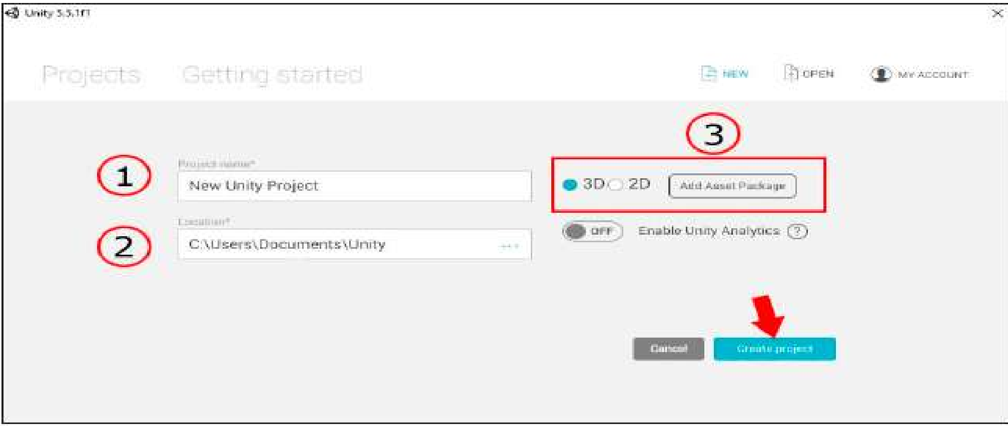


Fig. 1.8

- **Opción 1:** en este cuadro debemos ponerle el nombre a nuestro proyecto, por ejemplo "*Mi_primer_proyecto*".
- **Opción 2:** en este cuadro debemos ponerle la ubicación donde queremos que se guarde.
- **Opción 3:** nos permite crear un proyecto de dos tipos 3D o 2D, en realidad se puede trabajar igual con uno que con el otro. Te recomiendo que selecciones la opción 3D para empezar. También verás que al lado hay un botón que sirve para añadir al proyecto paquetes de ***Assets*** que de momento no vamos a utilizar pero que veremos más adelante.

Una vez has realizado las acciones anteriores puedes pulsar el botón **Create Project** .

5. Guardar el proyecto y la escena

Bien, ya has creado tu primer proyecto y ahora vas a identificar algunos elementos de la interfaz para aprender a trabajar y organizar correctamente tus proyectos. Pero antes quiero que realices dos acciones muy importantes y que a veces caen en el olvido.

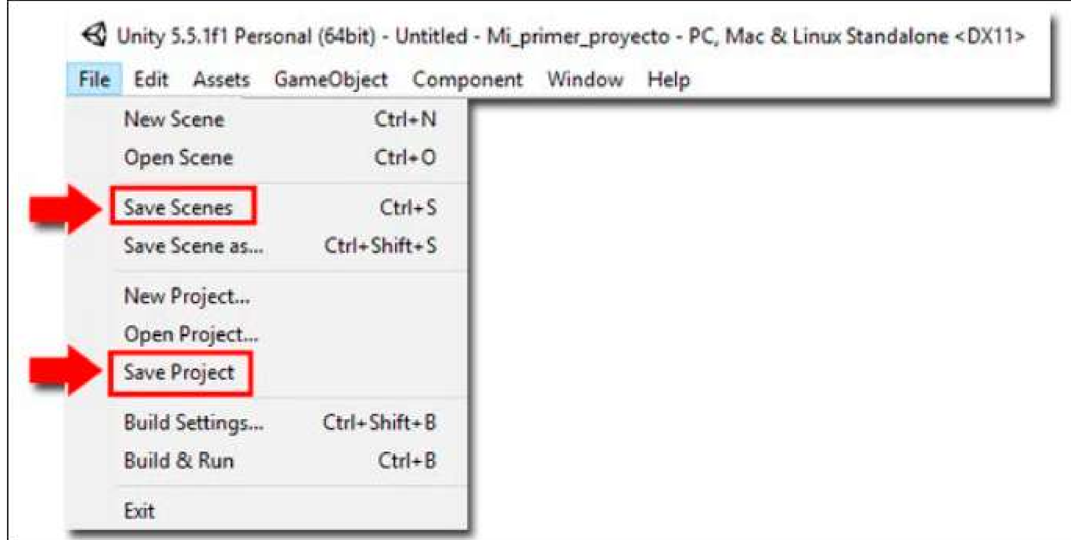


Fig. 1.9

En la parte superior de la interfaz hay un menú, accede a la opción **File> Save Project;** esta acción guardará tu proyecto y, a continuación, desde el mismo menú accede a la opción **File> Save Scenes;** se te abrirá una ventana de navegador que te permite guardar una escena con un nombre, escribe el nombre de **Escena_1** y guárdalo en el lugar que te indica por defecto.

Vista previa de la interfaz

Verás que te aparece un elemento de la interfaz, no te preocupes porque ahora vamos a identificar los elementos principales de esta interfaz, que es la que viene por defecto.

En la siguiente imagen verás enumeradas varias zonas de la interfaz:

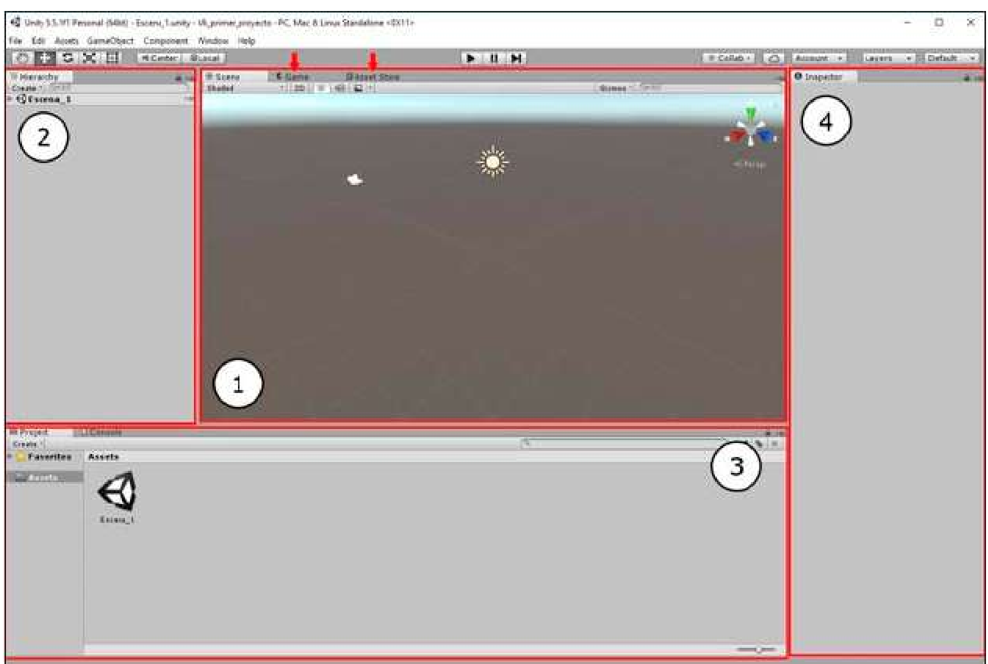


Fig. 1.10

1. Es la ventana escena, en donde construimos mediante objetos las escenas del juego. Verás que dentro de esta escena hay dos pestañas encima señaladas por flechas, son dos ventanas anidadas a este espacio, más adelante reorganizaremos la interfaz para recolocar estos elementos. Si haces clic encima de estas pestañas accederás al contenido de estas.
2. Esta es la ventana Jerarquía ***Hierarchy***, esta ventana contiene todos los elementos de la escena actual. Actualmente en esta versión también organiza los objetos por escenas.
3. La ventana ***Project*** es una ventana que se divide a su vez en dos partes. La parte de la izquierda contiene una estructura jerárquica de carpetas, actualmente solo verás una carpeta que tiene el nombre de ***Assets***, en donde podrás crear nuevas carpetas y poder organizar todos los archivos que necesites para el juego; escenas, modelos, texturas, materiales, scripts y audios, etc. En la parte derecha muestra el contenido de las carpetas.
4. La ventana Inspector es la ventana encargada de albergar todos los parámetros de los objetos que tengas seleccionados. También te va a permitir añadir nuevos parámetros y configurar algunos otros mediante scripts.

Configurar la interfaz

Cuando tenemos una disposición concreta de la interfaz nos referimos al ***layout***. Ahora tiene una visión general de las ventanas que se muestran en el ***layout*** por defecto, pero Unity dispone de otros ***layouts*** pre configurados para facilitar el trabajo al usuario.

Para acceder a los distintos ***layouts*** accede al botón con nombre ***Default*** que encontrarás en la parte superior derecha de la interfaz.

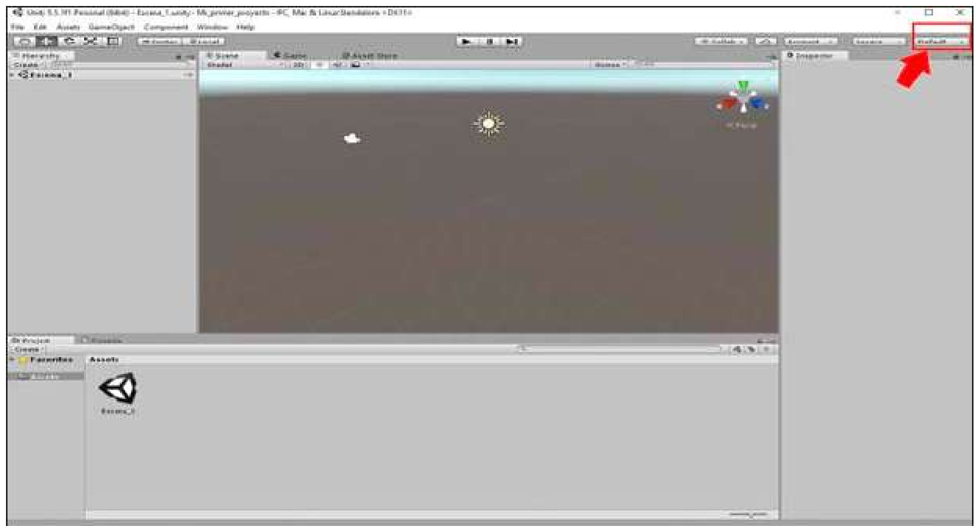


Fig. 1.11

Al hacer clic en el botón se despliega un menú donde podemos seleccionar varias opciones. En este caso vamos a crear uno propio con ayuda de los que ya están creados, eso no quiere decir que más adelante prefieras otra configuración, esto simplemente es un ejercicio para que aprendas cómo puedes crear tu zona de trabajo.

Primero accedes al menú **layout** y seleccionas la opción **Tall.**

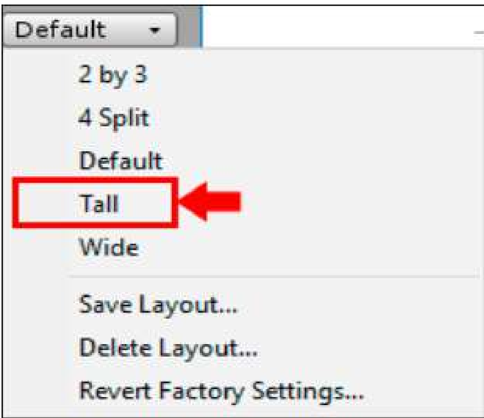


Fig. 1.12

Esta disposición me gusta y va a ser la disposición que se utilizará para todos los ejercicios del libro, con unas cuantas modificaciones que se explican a continuación.

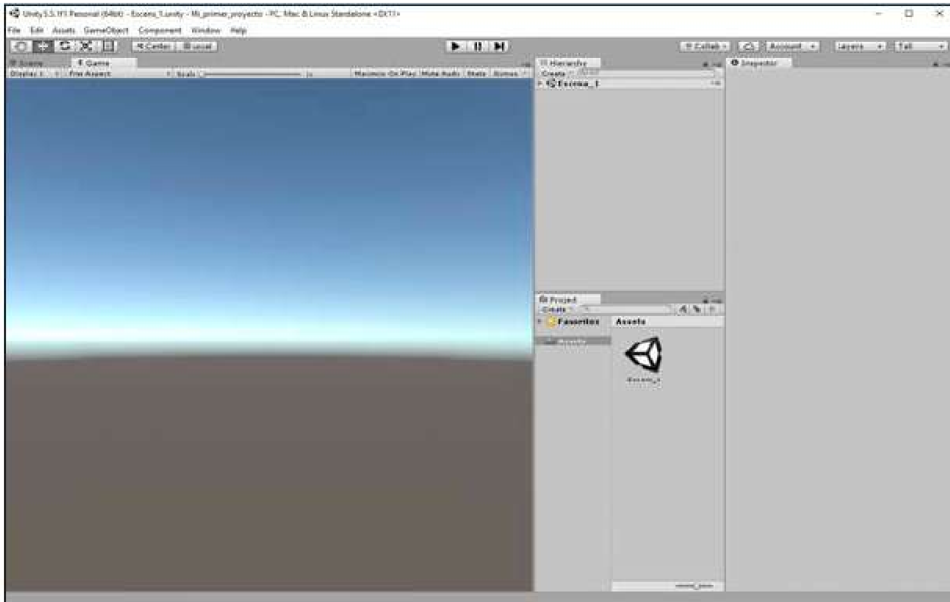


Fig. 1.13

La imagen anterior tiene la interfaz distribuida con el ***layout Tall***, ahora vamos a desanclar una ventana y anclarla en otro lugar de la interfaz, como por ejemplo la pestaña ***Game***. Para realizar esta acción debes hacer clic con el botón izquierdo del ratón y, manteniéndolo pulsado, arrastrar hacia la zona baja de la interfaz. Verás que al llegar a la zona inferior se crea una especie de bloque en gris, es el momento de dejar de pulsar el botón izquierdo del ratón. Si lo has realizado correctamente acabas de recolocar la ventana ***Game*** debajo de la ventana ***Scene***.

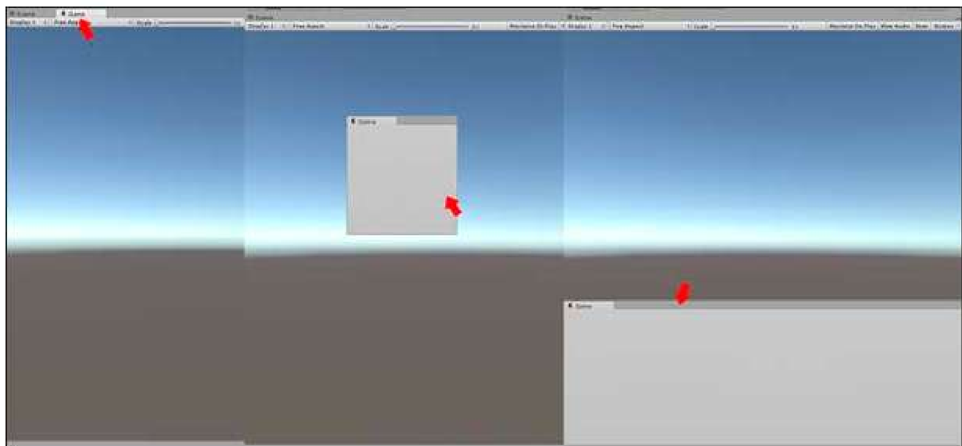


Fig. 1.14

Ahora para finalizar vas a añadir una ventana nueva que no está visible en la interfaz. Unity dispone de muchas herramientas y no todas se muestran en la interfaz, para realizar el ejercicio y añadir una ventana a la interfaz accede al menú principal que se encuentra en la parte superior de la interfaz y dirígete a ***Window>Console.*** A continuación aparecerá la ventana ***Console*** en forma de ventana flotante.

Realiza la misma acción que hiciste con la ventana ***Game*** y sitúa la ventana ***Console*** al lado de la ventana ***Game*** pero esta vez en forma de pestaña. Recuerda que solamente debes arrastrar la ventana manteniendo pulsado el botón izquierdo del ratón a la zona donde quieres colocarla, Unity se encarga de ensamblar la ventana al lugar. Si has realizado con éxito la acción debería quedarte una interfaz parecida a la siguiente imagen:

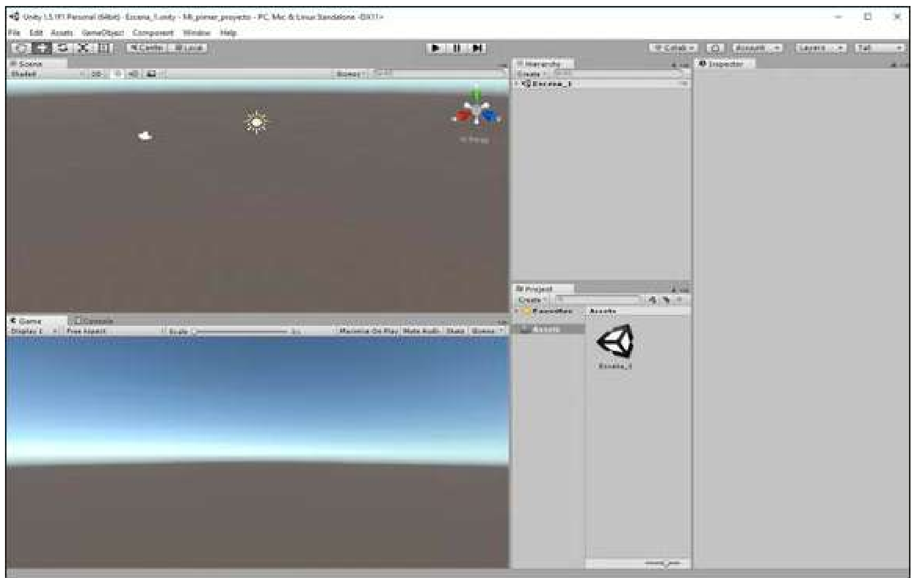


Fig. 1.15

Una vez tengas la interfaz adecuada a tus necesidades deberás guardar la configuración para las siguientes ocasiones. Para guardar el ***layout*** de la interfaz accede al menú de ***layouts*** y selecciona la opción ***Save Layout***. A continuación Unity te pide que le pongas un nombre, por ejemplo: "*Personal*" y pulsa en ***Save*** para guardar la configuración.

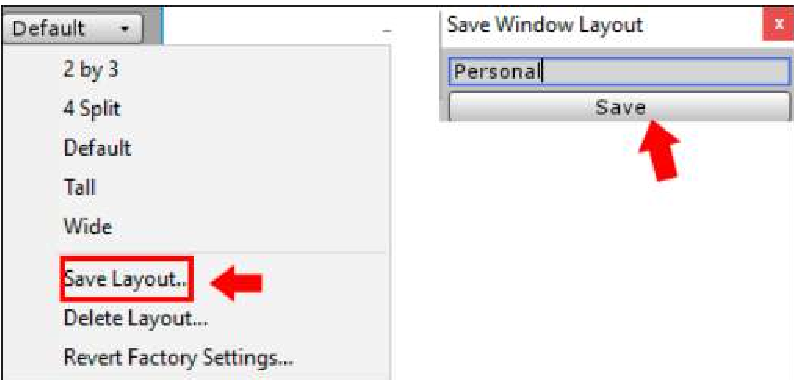


Fig. 1.16

Felicidades, has creado tu primera interfaz personalizada. En capítulos más avanzados es posible que te interese crear otro tipo de ***layouts***, pero ahora para empezar es suficiente.

Capítulo 2

Interfaz de Unity

- Importación de Assets
- Ventana Proyectos (Project)
- Ventana Escena (Scene View)
- Ventana Juego (Game View)
- Ventana Jerarquía (Hierarchy window)
- Ventana Inspector y creación de un GameObject

1. Importación de Assets

La presente obra viene acompañada de material adicional con todo lo necesario para que puedas seguir los proyectos y aprender sin que pierdas detalle. Una de las primeras cosas que vas a tener que aprender es la importación de assets, que es el material necesario para seguir los proyectos de este libro.

En todo proyecto de Unity encontrarás una carpeta llamada Assets. Esta carpeta contiene varias carpetas en su interior en donde se guardan los objetos del juego, scripts, materiales, texturas, etc.

En este capitulo he creado un paquete de assets para que puedas trabajar con las distintas ventanas que se explican.

Para importar el paquete de assets de este capítulo primero ten localizado el material adicional del libro y la ubicación de la carpeta de este capítulo. Luego crea un proyecto nuevo y dale el nombre que quieras. Todos los proyectos serán en 3D siempre que no se diga lo contrario.

Accedemos al menú principal y nos dirigimos a **Assets > Import Package > Custom Package** y buscamos dentro del material del libro en el proyecto 2 el archivo **Capitulo_2.unitypackage**.

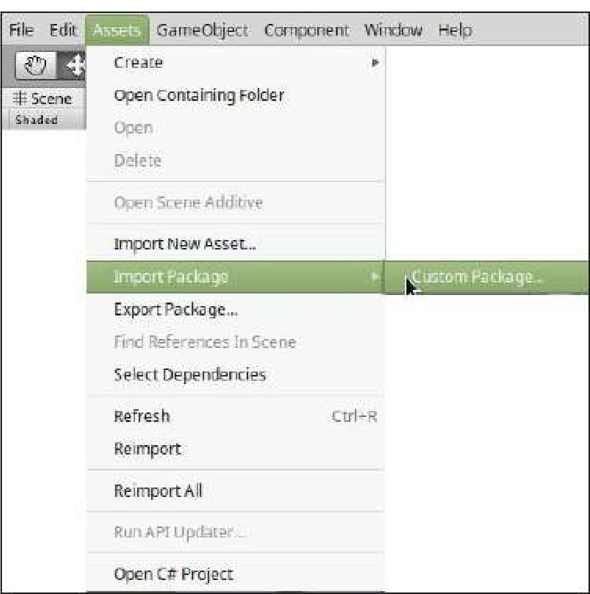


Fig. 2.1

Si todo es correcto, verás que en la ventana **Projects** te aparecen una serie de carpetas que comentaremos a continuación.

2. Ventana Proyectos (Project)

Esta ventana es un reflejo de las carpetas que constituyen tu proyecto y que se encuentran organizadas en tu ordenador, es decir, cada vez que creemos un archivo o una carpeta, esta se creará automáticamente en la carpeta de **Assets** de tu disco y viceversa.

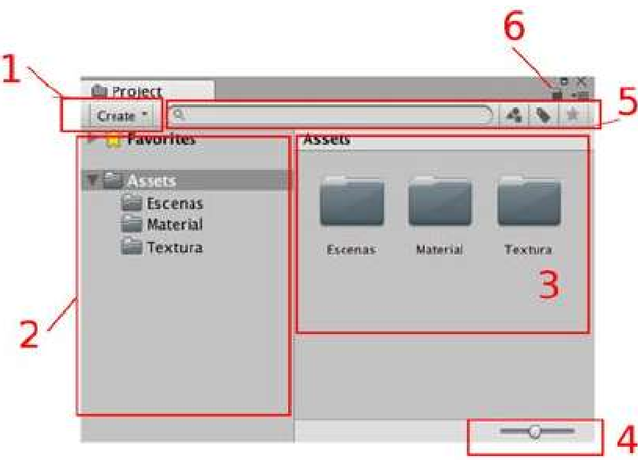


Fig. 2.2

1. Este menú nos permite crear una gran variedad de elementos para el proyecto. Una muy utilizada es la creación de carpetas.
2. Esta sección tiene dos elementos: la carpeta ***Assets*** en donde añadirás nuevas carpetas para ordenar los elementos de tu proyecto y una sección de favoritos que te permite buscar por temática dentro de la carpeta misma.
3. Esta sección te muestra el contenido de las carpetas e irás cambiando según qué carpeta tengas seleccionada.
4. Este tirador permite cambiar el tamaño de los elementos que se muestran en la sección 3.
5. Este menú contiene una caja de textos para poder introducir el nombre del elemento que estás buscando. También puedes hacer búsquedas según el tipo de elemento, el ***label*** que tengan asignado o si se han guardado como favoritos.
6. En esta sección tenemos dos símbolos: un candado que bloquea esta ventana para que no se pueda modificar. El otro símbolo es un menú que podemos utilizar si queremos añadir una pestaña nueva con otro tipo de ventana.

Si has importado los assets anteriores verás que disponemos de tres carpetas: Escenas que contiene una escena llamada mi primera escena: la carpeta materiales que contiene 5 materiales; y una última carpeta que contiene una textura.

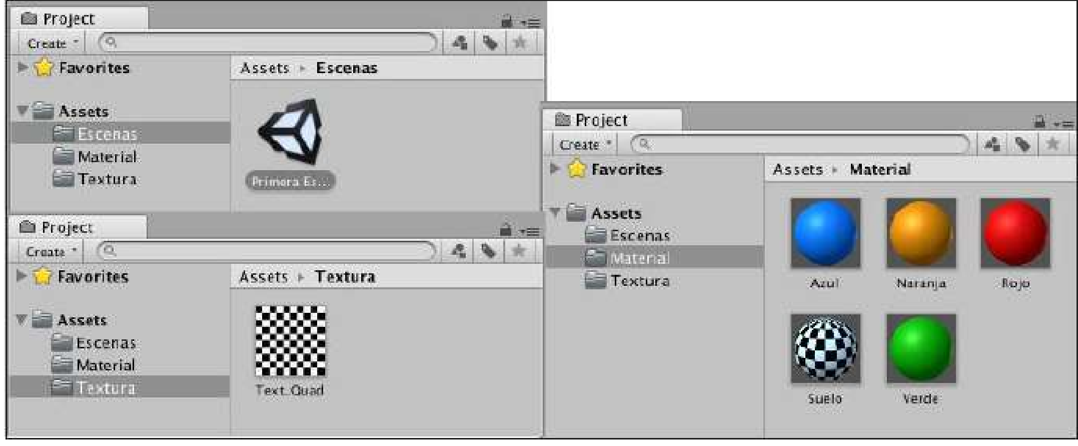


Fig. 2.3

Ahora vamos a crear una carpeta nueva que la vamos a llamar **Prefab**s primero asegúrate de que tienes seleccionada la carpeta Assets y luego, accediendo al menú **Create > Folder.** Seguidamente aparecerá una carpeta nueva a la que debes ponerle un nombre. Para borrar la carpeta simplemente selecciona la carpeta, haz clic con el botón derecho del ratón y seleccionar la opción **Delete.**

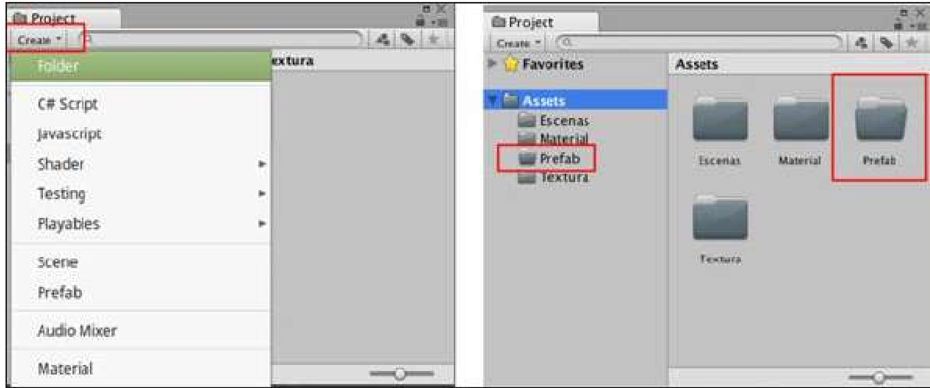


Fig. 2.4

Ahora para hablar de la siguiente ventana primero vamos a acceder a la carpeta Escenas y hacemos doble clic con el botón izquierdo encima de la escena con nombre Primera escena. Se abrirá una escena con varios objetos en la ventana **View.**

3. Ventana Escena (Scene View)

La ventana escena, o también podemos llamarla vista escena, es la parte de la interfaz en la que interactuaremos con el mundo que estamos creando. Esta ventana la utilizaremos para crear nuestras escenas para seleccionar y posicionar personajes, cámaras, luces y todos los demás tipos de objetos de nuestro juego. Poder seleccionar, manipular y modificar objetos en la vista de escena son algunas de las habilidades básicas que vas a aprender para comenzar a trabajar en Unity.

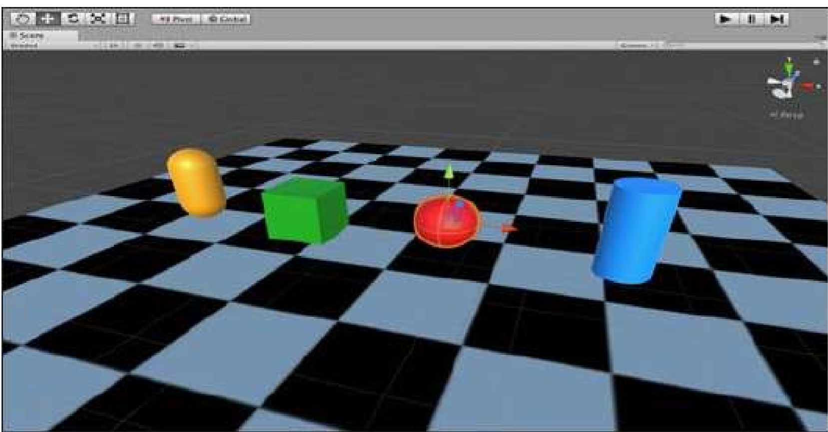


Fig. 2.5

Voy a mostrarte una imagen con todos los objetos básicos que podemos encontrarnos en una escena para que puedas identificarlos cuando los nombre.

Fig. 2.6

Primero vamos a aprender como navegar por el espacio tridimensional. Si haces clic encima del botón de navegación que se encuentra representado por una mano y luego posicionas el cursor encima de la ventana pulsando con el botón izquierdo del ratón podrá desplazar la vista de derecha a izquierda. Esta herramienta tiene tres estados:

1. ***Mover***: haciendo clic con el botón izquierdo y arrastrando encima del visor
2. ***Orbitar:*** pulsando *Alt+ botón izquierdo del ratón* podrá orbitar alrededor del espacio 3D.
3. ***Zoom***: pulsando *Alt + botón derecho del ratón* podrá acercarse o alejarse de la escena. Si dispones de una rueda en el ratón también puede hacerla girar para hacer zoom. En el caso de Mac y Linux Comando (Control) + Shift y el botón derecho del ratón.

En la parte superior derecha de la ventana nos encontramos con un Gizmo. El gizmo nos muestra la dirección en que se encuentra nuestra cámara visor escena. También podemos interactuar con el gizmo. Este objeto te permite visualizar en todos los ejes de coordenadas y también permite ver en dos modos; perspectiva y ortográfica. Si pulsas en el cuadro interior del ***gizmo***, representado en las siguientes imágenes con un color amarillo, podrá cambiar la vista de perspectiva a ortográfica.

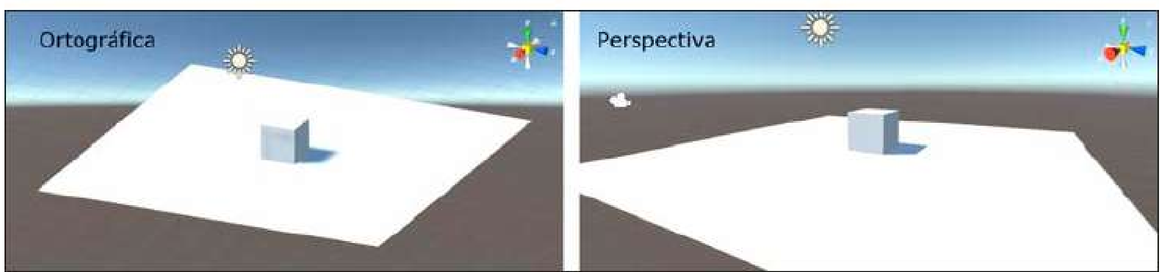


Fig. 2.7

Si pulsas en los conos de colores la vista te mostrará en modo 2d la proyección del eje que haya seleccionado. Por ejemplo, si pulsad en el cono rojo este representa el eje X y te mostrará la vista desde este punto como se muestra en la siguiente imagen. El color verde representa el eje Y y el color azul representa el eje Z.

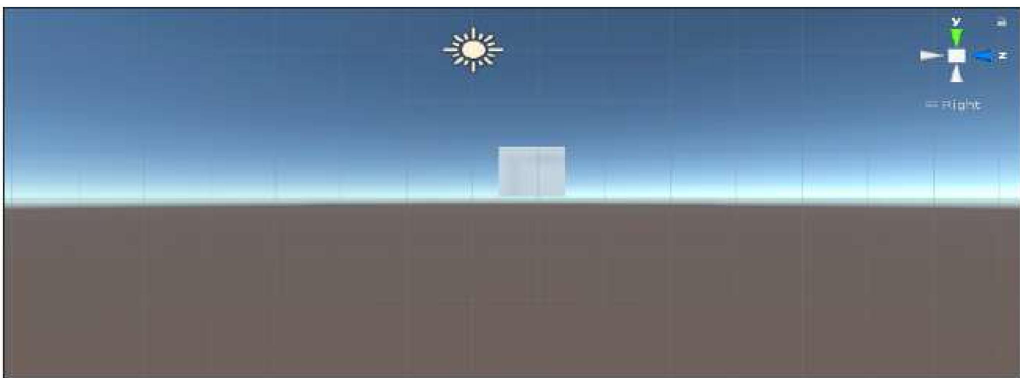


Fig. 2.8

Otro elemento que debes conocer es el menú de escena. El primer menú desplegable te muestra distintos modos de representar la escena dividida en sectores. Te recomiendo que hagas la siguiente prueba: accede al menú ***Shaded*** y escoge la opción ***Shaded Wireframe***; la escena se representará mostrando una vista de alambre dibujada encima del material como se muestra en la siguiente imagen.

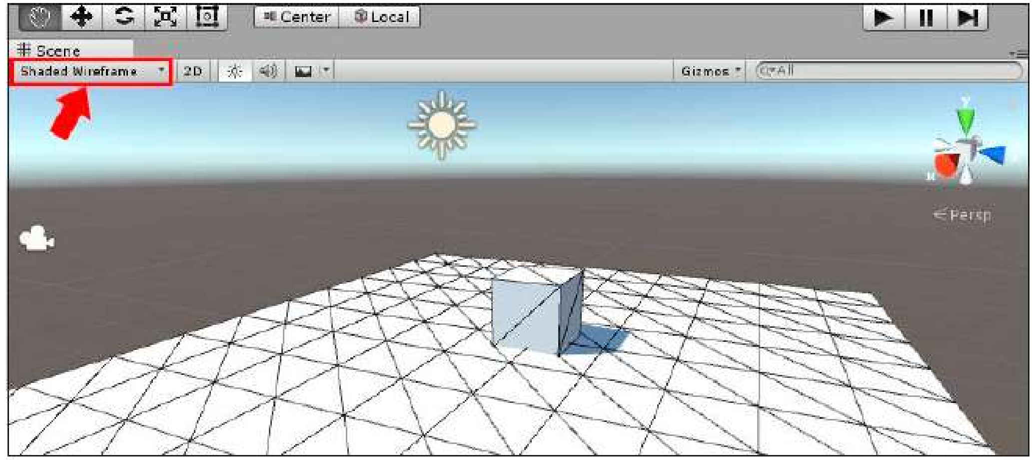


Fig. 2.9

Al lado del menú anterior dispones de una serie de botones que activan y desactivan opciones de la ventana escena. El 2D activa o desactiva esta vista, el botón con el símbolo del sol hace referencia a las luces de la escena, el símbolo del altavoz para el sonido y el botón con el símbolo de la foto alberga un submenú donde puedes marcar o desmarcar una serie de elementos que se desactivan o activan cuando pulsas este icono.

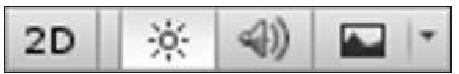


Fig. 2.10

Para finalizar este menú dispones de otra opción con el nombre ***gizmos*** que también te permite activar o desactivar elementos de la escena. Al lado de esta opción dispones

de un buscador, que puede resultarte muy útil cuando trabajes con escenas muy grandes, con múltiples objetos.

Fig. 2.11

4. Ventana Juego (Game View)

Esta ventana es donde podemos ver el resultado de nuestra escena. Es donde se muestra la compilación de todos los objetos, comportamientos, iluminación y todo ello renderizado desde la cámara que tengas dentro de la escena.

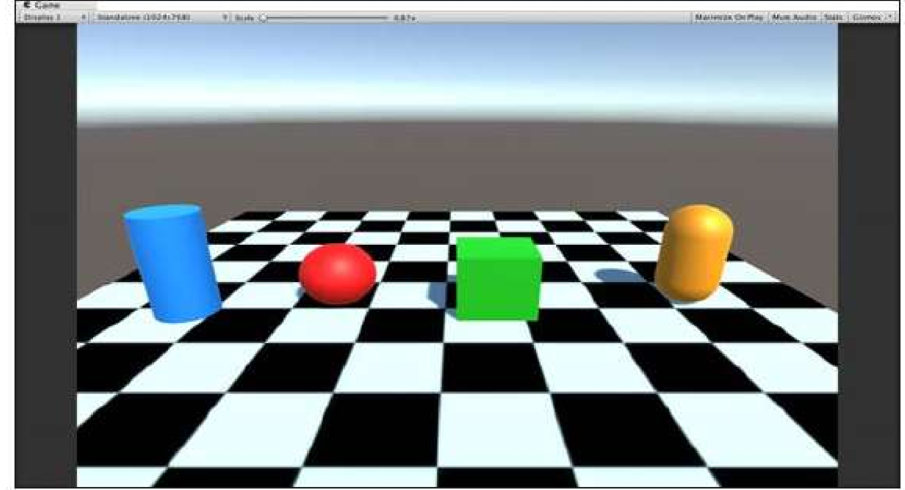


Fig. 2.12

Barra de herramientas del Game View

Tenemos varias opciones para controlar cómo se muestra la imagen final de nuestra escena desde el visor Game. A continuación te detallo algunos aspectos que debes tener en cuenta.

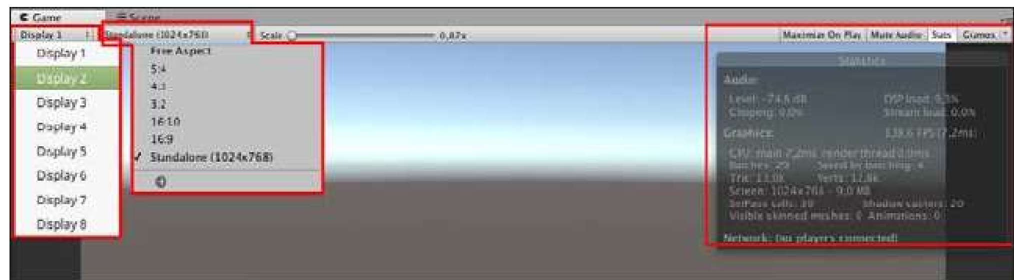


Fig. 2.13

Display: contiene un menú desplegable para escoger la vista de la cámara que contiene la escena, por defecto está configurado para el Display 1.

Free Aspect: disponemos de una serie de valores predefinidos para probar cómo se vería nuestro juego en distintos formatos.

Scale: es un tirado que nos permite hacer **zoom** en la pantalla de juego para ver con más detalle nuestro juego o ver en modo global como se ve desde lejos.

Maximize on Play: al tener activada esta opción, cuando ejecutemos el juego en **Modo Play** la ventana Game se maximizara para que juegue en pantalla completa.

Mute audio: si activamos esta opción silenciaremos cualquier audio que haya dentro del juego cuando entremos en modo Play.

Stats: este botón activa o desactiva la ventana de estadísticas. Esta ventana nos ofrece información del renderizado gráfico y audio mientras esta en modo Play.

Gizmos: este botón activa o desactiva la visibilidad de cierto tipo de gizmos o iconos que veríamos en la ventana escena.

Play Mode

Estos botones de la barra de herramientas nos sirven para controlar el modo de reproducción del editor y ver cómo se reproduce el juego en la ventana Game. Un aspecto muy importante a tener en cuenta cuando trabajamos con este visor es que en modo de reproducción, los cambios que realicemos serán temporales es decir cuando paremos la reproducción de nuestro juego, todos los cambios que hayamos realizado se restablecerán a como estaban antes de entrar en modo de reproducción. La interfaz de usuario del editor se oscurece para advertirte de que estas en modo reproducción.

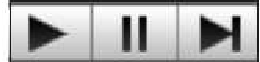


Fig. 2.14

5. Ventana Jerarquía (Hierarchy window)

La ventana Jerarquía contiene una lista de todos los GameObject (objetos,luces,cámaras) que encontramos en nuestra escena. Cuando agregamos un objeto a la escena este aparecerá en la lista y cuando eliminemos un objeto este también desaparecerá de esta lista. La lista se crea en el orden que vamos añadiendo o creando objetos. Puedes ver en la escena que tenemos en el proyecto como están organizados los objetos.

Fig. 2.15

En esta ventana también podemos efectuar varias acciones. Podemos cambiar el orden de nuestros objetos de la lista. Para cambiar el orden debemos seleccionar un objeto de la lista por ejemplo el ultimo elemento que es **Cylinder** y arrastrarlo hacia arriba hasta el tercer lugar que quede entre Directional Light y Plane.

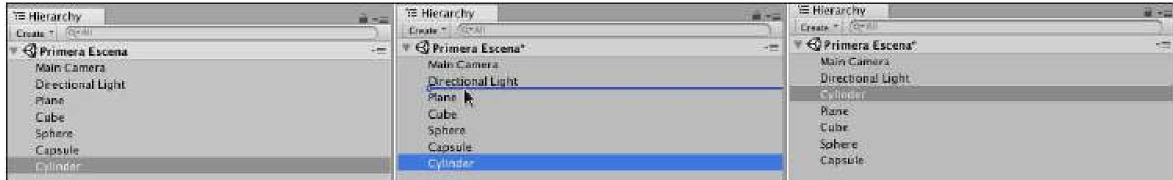


Fig. 2.16

Otra acción que podemos hacer es emparentar objetos. Para que quede más claro vamos a coger dos objetos, la esfera y el cubo. En este ejemplo vamos a hacer que la esfera sea el hijo y el cubo el padre, esto quiere decir que cuando movamos el cubo la esfera también se moverá. Selecciona de la lista la esfera y arrastrarla encima de Cubo. Automáticamente se creará en cubo un subobjeto que es esfera.

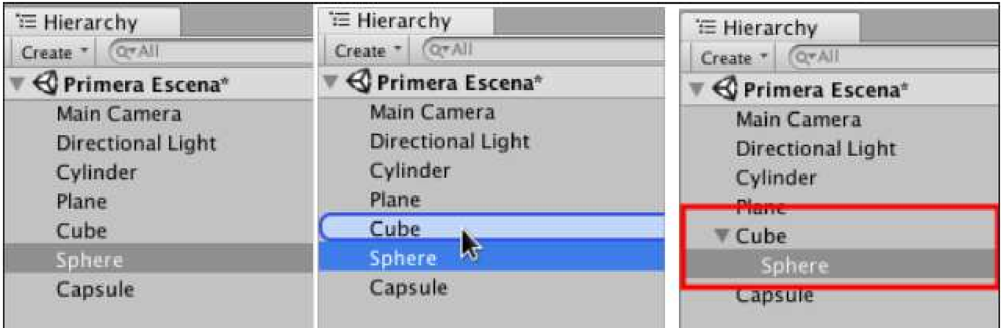


Fig. 2.17

Un objeto padre puede tener varios hijos, y a su vez los hijos pueden ser padres y tener hijos como te muestro a continuación

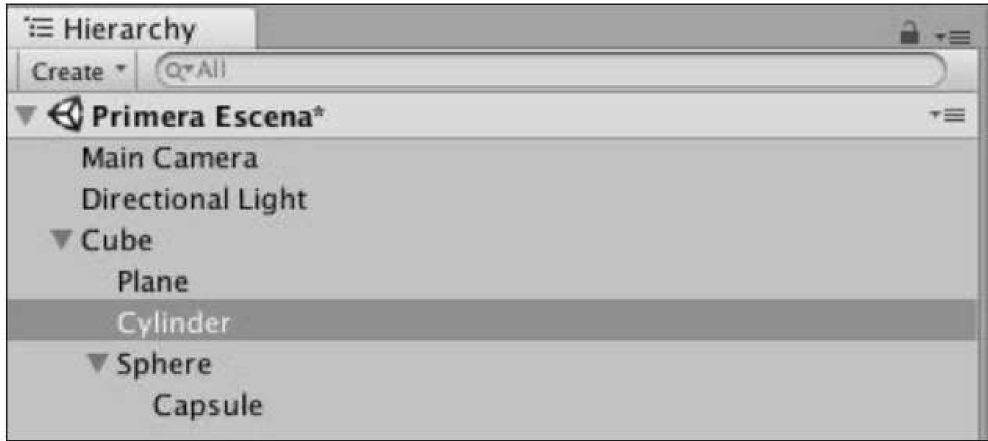


Fig. 2.18

En la imagen anterior puedes ver cómo el objeto **Cube** tiene como hijos a **Plane Cylinder** y **Sphere**, y a su vez **Sphere** es padre de Capsule.

También podemos cambiar el nombre de los objetos primero un clic para seleccionar y luego otro clic encima del nombre del objeto, este se convertirá en una caja de textos donde puedes escribir el nombre.

6. Ventana Inspector y creación de un GameObject

En realidad todo lo que contiene una escena son **GameObjects**. Todos estos **GameObjects** disponen de características distintas, y por ese motivo voy a mostrarte cómo se crean, cómo se manipulan y cómo puedes ver sus características.

Para crear un ***GameObject*** puedes hacerlo de muchas formas distintas una de ellas es en el menú principal acceder a **GameObject>3D Object> Cube**, se creará un objeto llamado **Cube** que podrás ver en la ventana **Hierarchy o Jerarquía.**

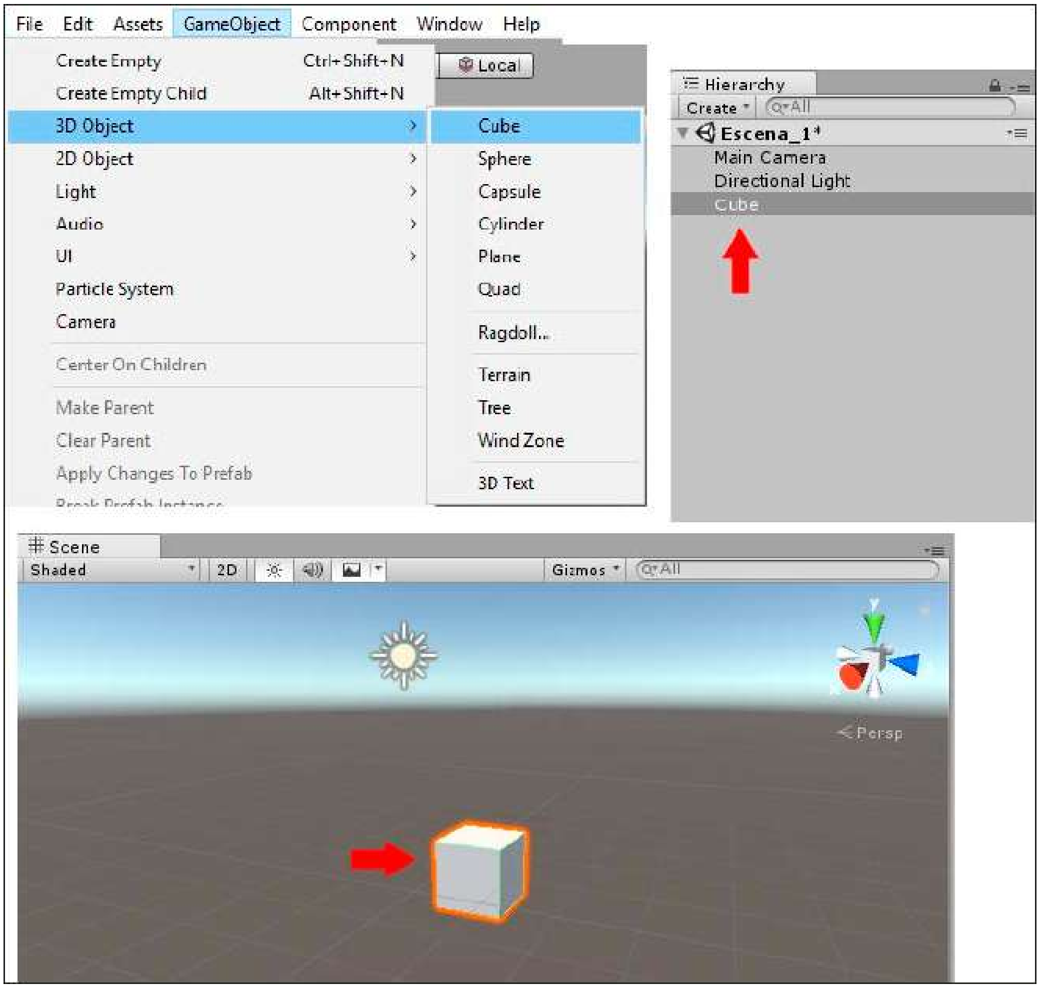


Fig. 2.19

En las imágenes anteriores he decidido utilizar un cubo solo y no la escena completa que tenemos para que se entienda mejor, si lo deseas puedes seleccionar el cubo que tienes de la escena y seguir las explicaciones. Si seleccionas el ***GameObject*** **Cube** en la

ventana ***Hierarchy***, veras que en la ventana **inspector** aparecen las propiedades de este objeto. A continuación voy a explicarle las características de este ***GameObject***.

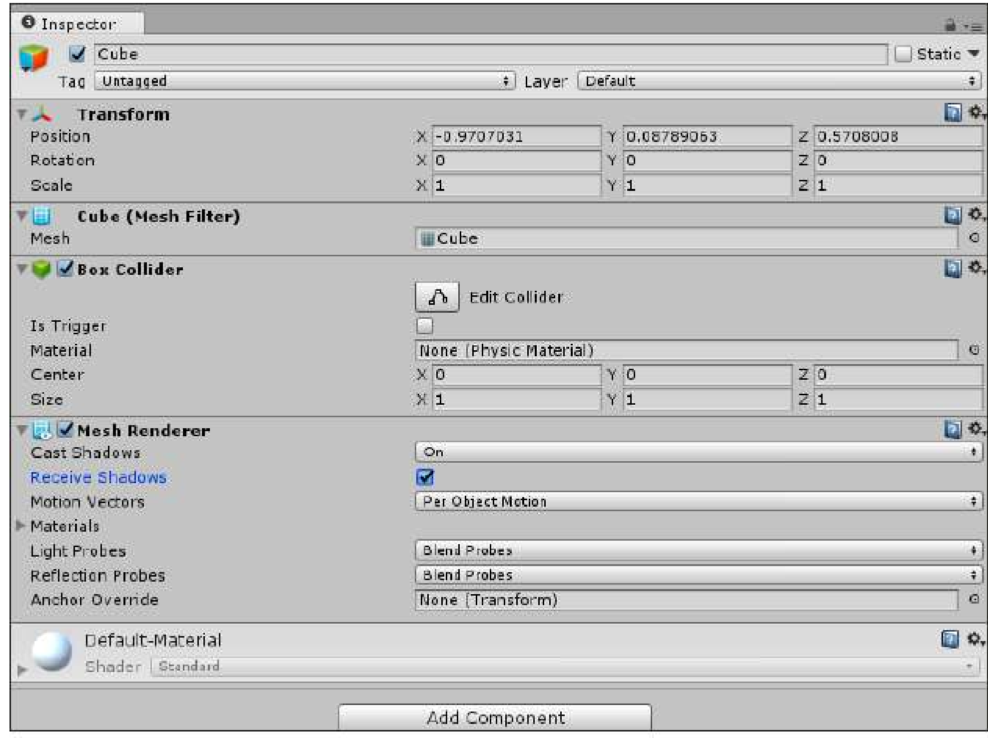


Fig. 2.20

Esta ventana tiene una parte superior y a continuación en su parte inferior se divide en paneles o **componentes** que se pueden colapsar mediante una flecha en la parte superior izquierda.

En la parte superior de la ventana Inspector puedes realizar las siguientes acciones:

El primer elemento es un cubo de colores que en realidad es una herramienta que te permite asignar un icono al objeto para diferenciarlo en una escena del resto de objetos. Por ejemplo en una escena muy grande poder encontrar al jugador.

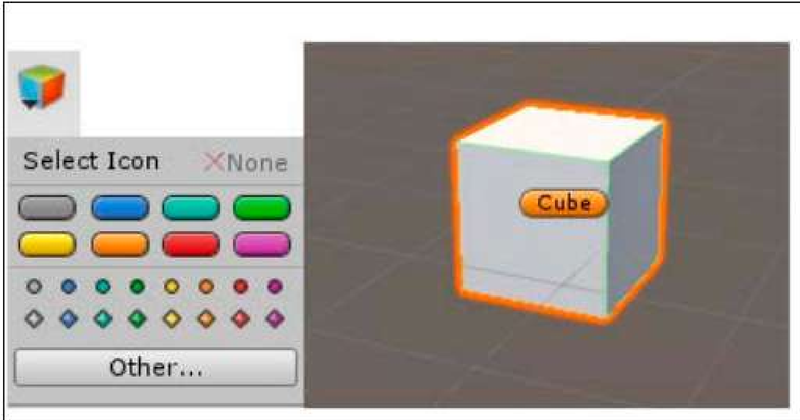


Fig. 2.21

Al lado del selector de iconos dispones de una caja selectora que desactiva el objeto cuando la de seleccionamos y activa el objeto cuando está seleccionada. A su lado puedes cambiar el nombre al objeto, en este ejemplo por defecto llamado **Cube**.

Fig. 2.22

La opción ***Static*** hace referencia a si un objeto es estático o no es estático, esta característica se utiliza según el tipo de iluminación utilicemos y qué función va a realizar este objeto en la escena. En estos momentos déjalo desactivado.

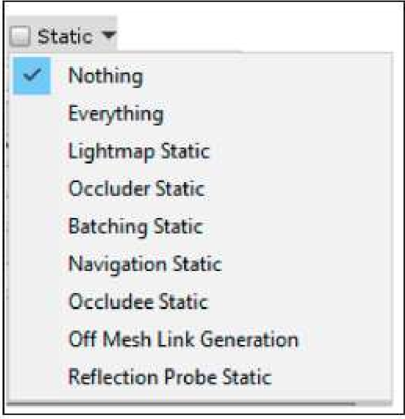


Fig. 2.23

Para terminar esta sección tenemos dos menús el ***Tag*** (etiqueta) y el ***Layer*** (capa). Estos menús te permiten poner etiquetas a los objetos y determinan a qué capas pertenecen. Son herramientas muy útiles en programación cuando queremos llamar a un objeto en concreto.

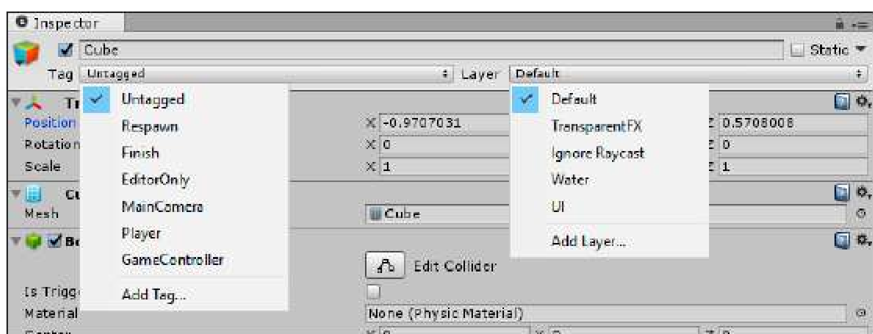


Fig. 2.24

El siguiente componente es el ***Transform*** y te permite manipular el objeto en el espacio 3D. Debes tener en cuenta que los ejes de coordenadas son los que están representados por colores en el **guizmo** de la ventana escena.

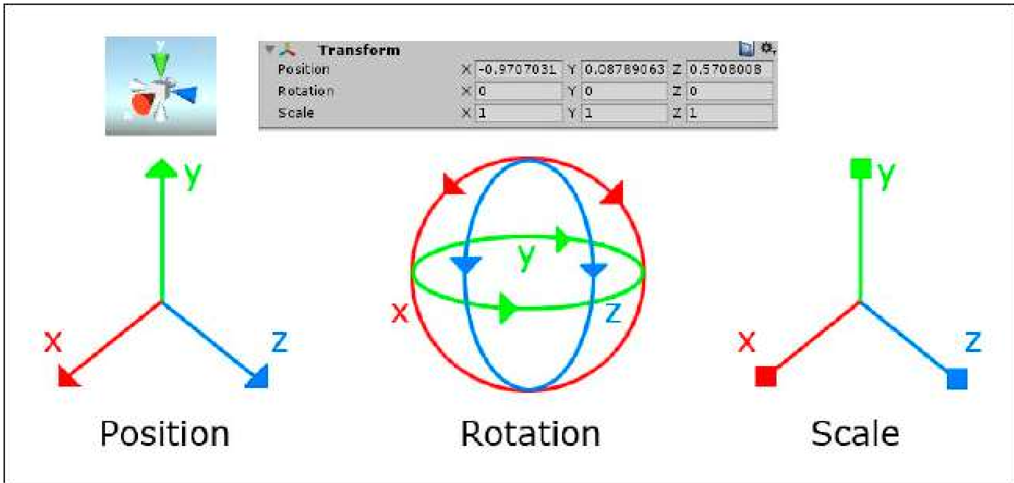


Fig. 2.25

Esta característica te ayudará a identificar hacia dónde quiere realizar la transformación. Si realizas la prueba y cambias de valores los siguientes parámetros verás que el objeto Cube cambia de posición (***position***), de rotación (***rotation***) y de tamaño (***scale***).

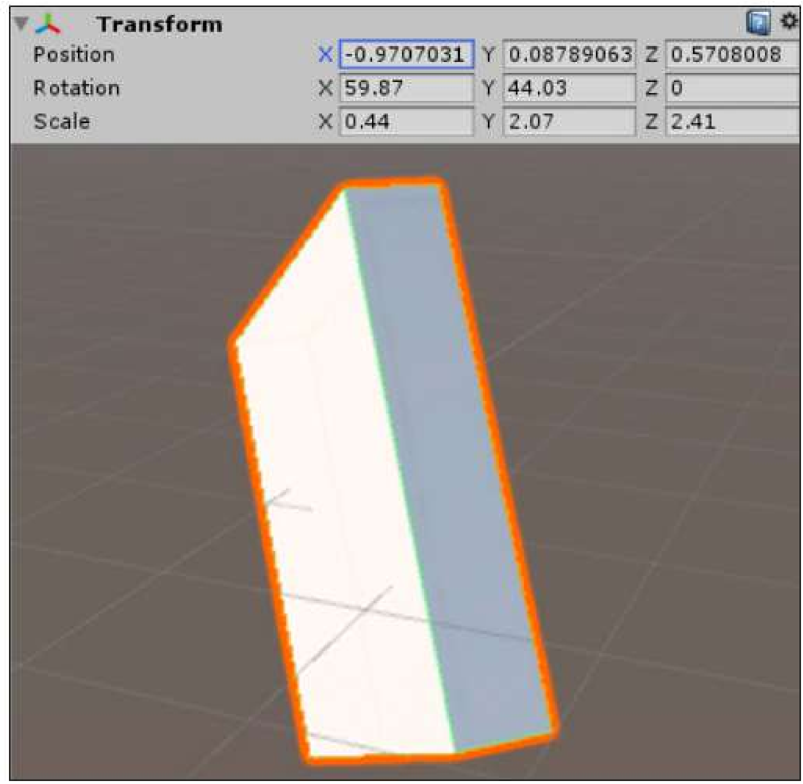


Fig. 2.26

Seguramente que mover objetos en una escena de este modo puede ser una tarea muy complicada. Por ese mismo motivo dispones de una serie de herramientas que no se han explicado anteriormente porque tiene más sentido verlas en este contexto.

En la ventana escena dispone de 3 botones para realizar estas mismas transformaciones en un objeto de una forma más intuitiva.

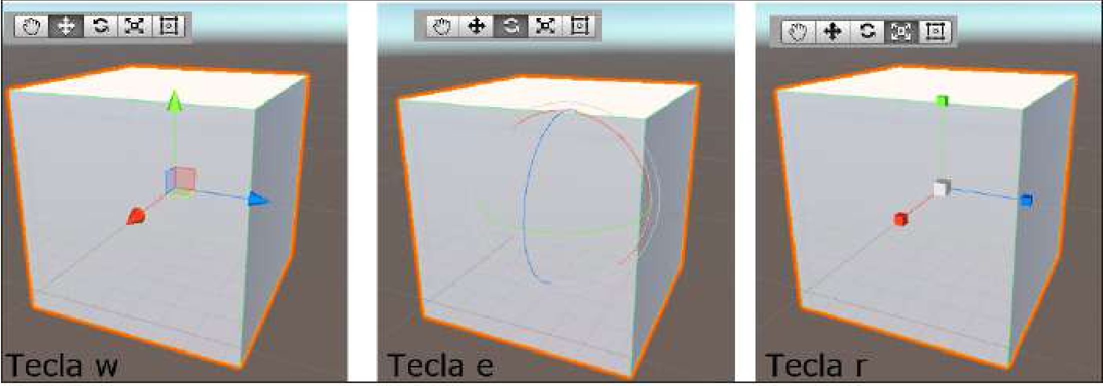


Fig. 2.27

Estas acciones puedes activarlas seleccionando el objeto y pulsando la tecla **W** para mover el objeto tecla **E** para rotarlo y tecla **R** para escalarlo. Para realizar una acción de transformación seleccionas uno de los ejes y pulsando y, manteniendo pulsado el botón izquierdo del ratón, arrástralo para poder desplazar, rotar o escalar el objeto.

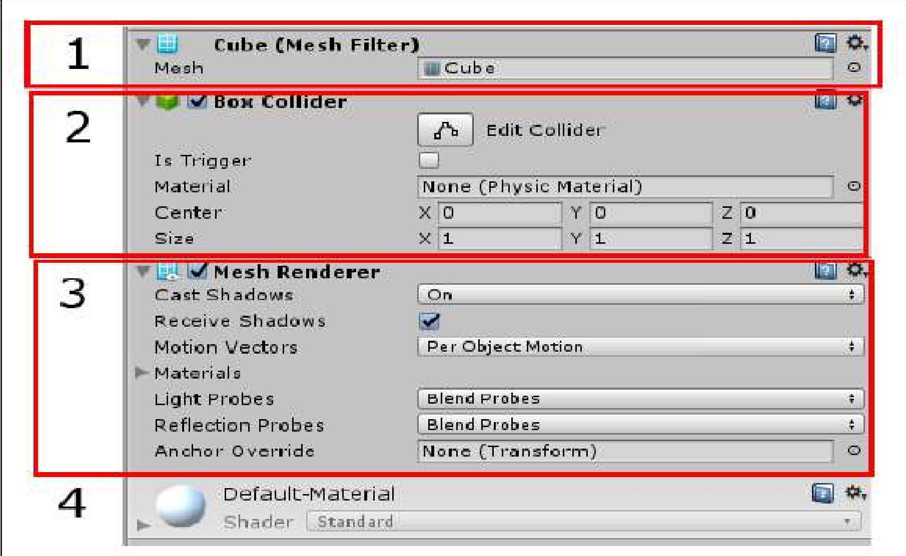


Fig. 2.28

En la imagen anterior verás el resto de componentes del **GameObject:**

1. El componente **Mesh** es el encargado de darle una malla al objeto.
2. El componente **Collider** es una caja que delimita mediante cálculos físicos que superficie del objeto impactará con los otros objetos. Este parámetro se explica con mayor detalle en ejemplos posteriores.
3. El componente **Mesh Renderer** es un conjunto de parámetros que permiten que el objeto pueda ser renderizado por Unity.
4. Los **GameObjects** por defecto como este se les aplica un material por defecto que puede ser configurado posteriormente.

Ahora que has visto los distintos componentes que tiene un GameObject vas a crear uno totalmente de 0.

Primero en la ventana **Hierarchy** (*Jerarquía*) en la opción **Create** selecciona la opción **Create Empty.** Te aparecerá el nombre de **GameObject** en la ventana Jerarquía y en el inspector también tendrás el componente **Transform** pero en la ventana escena no tendrás ningún objeto. Eso se debe a que todavía no le has asignado los componentes que se te han mostrado anteriormente y que son necesarios para tener un **GameObject** en condiciones mínimas.

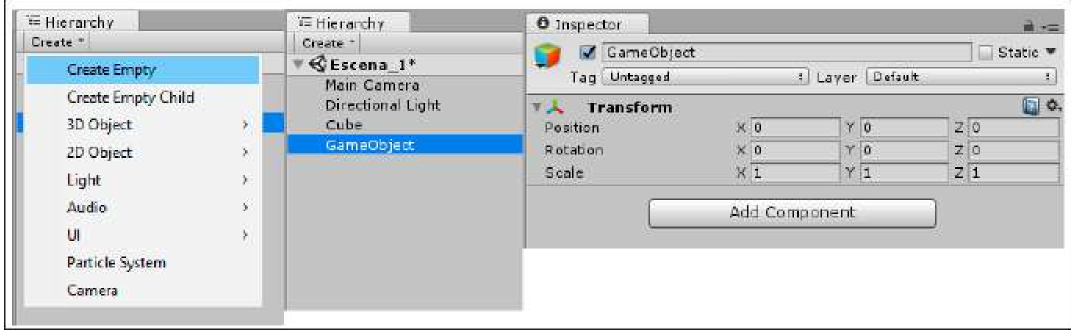


Fig. 2.29

Antes de añadir los componentes le ponemos un nombre en la caja de textos superior.

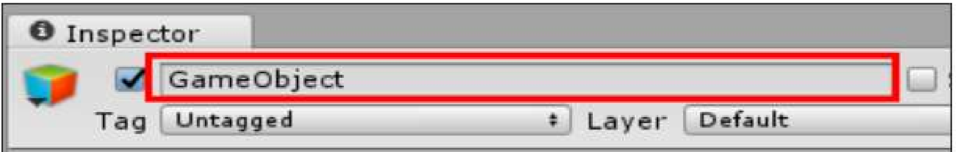


Fig. 2.30

Para añadir los componentes que necesitamos pulsamos en el botón **Add Component**. Se te abrirá un menú con muchas opciones, selecciona la opción **Mesh>Mesh Filter**

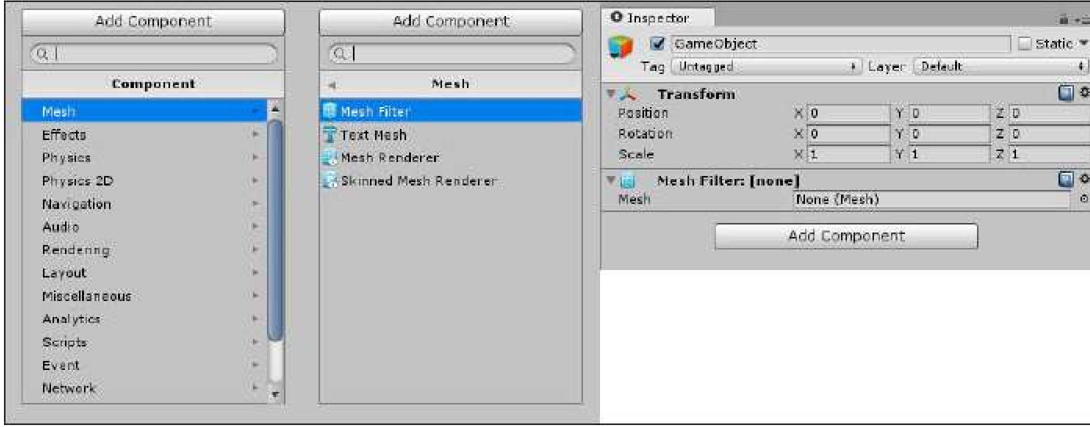


Fig. 2.31

Este componente te permite seleccionar que tipo de superficie tendrá el **GameObject**, para seleccionar el tipo de superficie pulsa en el punto que se encuentra al lado de la caja

de textos y en el menú que aparece selecciona el tipo de superficie como se muestra en la siguiente imagen.

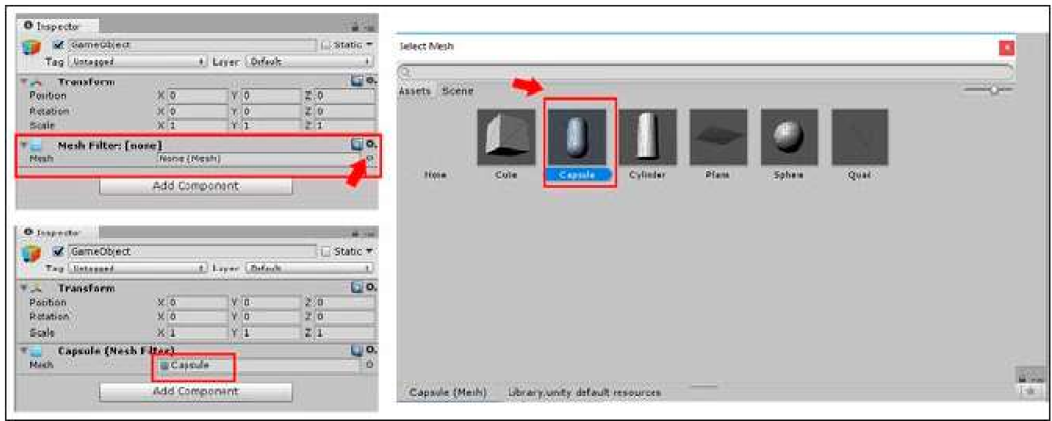


Fig. 2.32

El siguiente paso consiste en que el GameObject pueda ser renderizado por Unity, vuelve a pulsar el botón de **Add Component>Mesh>Mesh Renderer.** Verás que en la ventana aparece el objeto pero tiene un color rosado, esto sucede porque no tiene ningún material aplicado. Para proporcionarle un Material por defecto accede desde las opciones del componente **Mesh Renderer>Materials** que puedes desplegar mediante la flecha que tienes al lado haciendo clic encima de ella y se desplegaran las opciones de esta sección.

Para seleccionar el material pulsa en el punto al lado de la caja de texto de **Element 0**. Se te aparecerá una ventana donde puedes seleccionar varios materiales, escoge uno.

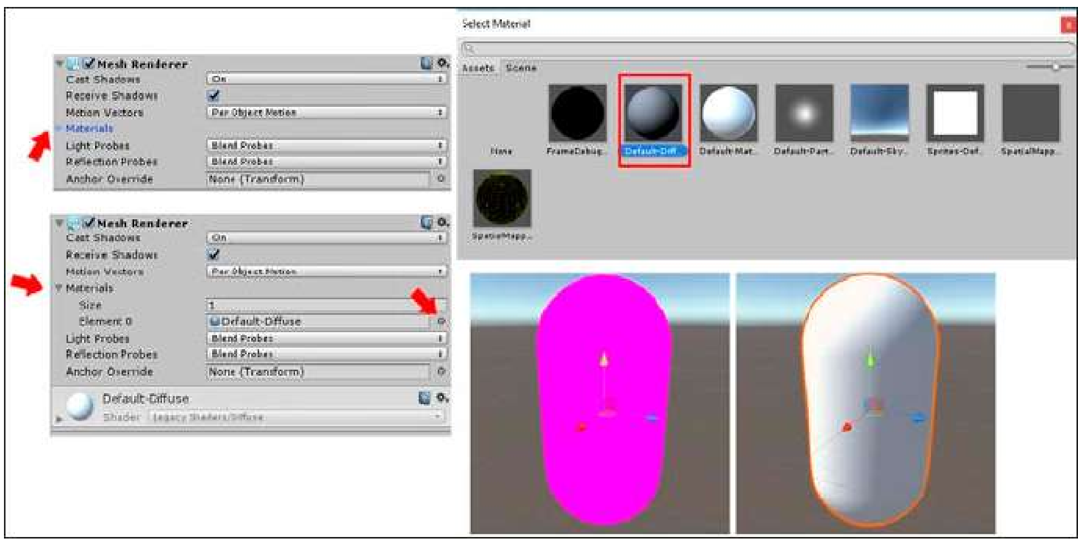


Fig. 2.33

Para finalizar todo el proceso de creación de un **GameObjec**t debes añadir el **collider** accediendo a **Add Component>Physics>Capsule Collider.** En la siguiente imagen se muestra el proceso, en este caso selecciona la opción que tiene la forma de cápsula, por

comodidad, pero eso no quiere decir que no pueda poner ningún otro tipo, puedes experimentar todo lo que desees.

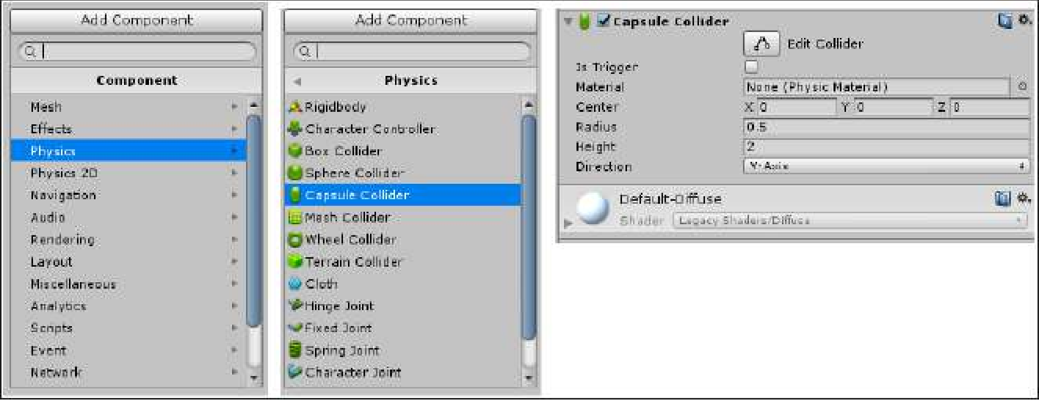


Fig. 2.34

En este primer capítulo no quiero entrar en mucho detalle en las opciones porque es mucho contenido para asimilar sobre todo si empiezas desde cero. Para finalizar este capítulo de introducción voy a proponerte un proyecto final en donde vas a poner en práctica todo lo que has visto en los distintos ejercicios.

- Crear un proyecto nuevo.
- Importar los assets del capitulo
- Guardar el proyecto y la escena.
- Crear ***GameObjects*** desde cero: 1 cubes, 1 esferas, 1 plano
- Poner nombre a los ***GameObjects.***
- Mover, rotar y escalar ***GameObjects.***
- Poner materiales básicos a los ***GameObjects***.

Cuando hayas terminado con esta actividad pasa al siguiente capítulo.

Capítulo 3

Editor de terrenos

- Crear un terreno
- Esculpir la superficie
- Pintar el terreno
- Poner vegetación
- Poner agua en el terreno
- Crear zona de viento *(windzone)*
- Editar árboles

Editor de terrenos

El editor de terrenos nos permite crear terrenos exteriores de una forma muy sencilla y con múltiples herramientas para darle color y textura. Para poder seguir el capítulo deberás crear un proyecto 3D e importar los assets que pertenecen a este capítulo.

Los terrenos son un trabajo muy complejo que conlleva muchas horas de dedicación, importando assets y colocándolos en el lugar que les corresponde. En este capítulo vas a aprender cómo crear un escenario exterior, pintar con texturas, poner vegetación, crear una zona de viento y cómo puedes crear tus propios árboles con el editor de arboles.

Fig. 3.1

1. Crear un terreno

Para crear un terreno nos dirigimos a la barra de herramientas principal y accedemos a **GameObject > 3D Object > Terrain**. Verás que los terrenos consumen bastantes recursos del sistema y si no disponemos de un PC potente, notarás como se ralentiza un poco todo.

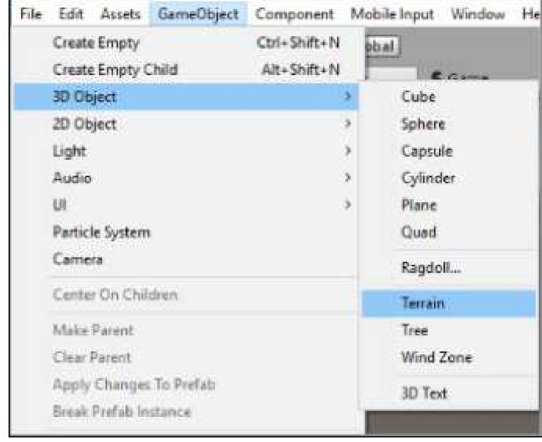


Fig. 3.2

Antes de empezar a trabajar en nuestro terreno vamos a configurar cómo queremos el terreno. Para ello nos dirigimos al panel inspector y accedemos al apartado **Terrain** y en el menú seleccionamos la última opción, como se muestra en la siguiente imagen:

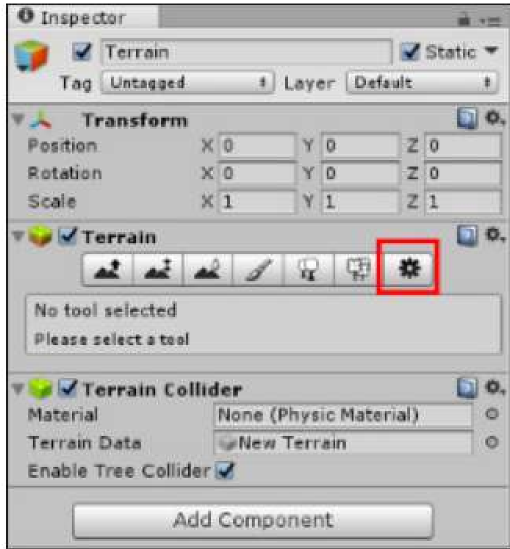


Fig. 3.3

La configuración se divide en apartados que a continuación explico en detalle.

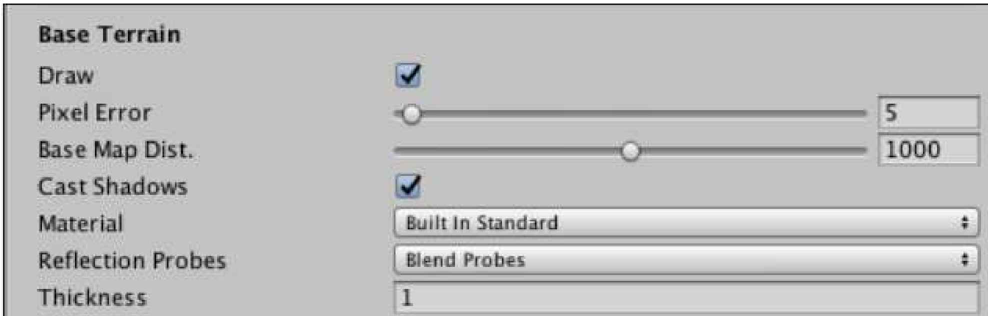


Fig. 3.4

Base Terrain	
Draw	Activa o desactiva la representación del terreno.
Pixel Error	Es la precisión del mapeado entre los mapas del terreno y el terreno generado; los valores más altos indican una exactitud más baja pero una sobrecarga de representación más baja.

<table>
<tr><td>Base Map Distance</td><td>La distancia máxima a la que las texturas del terreno se mostrarán en la resolución completa. Más allá de esta distancia, una imagen compuesta de menor resolución se utilizará para la eficiencia.</td></tr>
<tr><td>Cast Shadow:</td><td>Activa o desactiva las sombras del terreno.</td></tr>
<tr><td rowspan="2">Material</td><td>Es el material utilizado para representar el terreno. Esto afectará a cómo se interpretan los canales de color de una textura de terreno.</td></tr>
<tr><td>• Built in Standard. Este es el material PBR (Phisical-Based Rendering) introducido en Unity 5.0. Para cada capa splat, puede usar una textura para albedo y suavidad, una textura para normal y un valor escalar para ajustar el metal.
• Built in Legacy Diffuse. Este es el material heredado incorporado en el terreno de Unity 4.x y anteriores. Utiliza el modelo de iluminación Lambert (termino difuso solamente) y tiene soporte de mapa normal opcional.
• Built in Legacy Specular. Este material incorporado utiliza BlinnPhong (término difuso y especular) como modelo de iluminación y tiene soporte de mapa normal opcional. Puedes especificar el color especular general y brillo para el terreno.
• Custom. Utiliza un material personalizado de tu elección para representar el terreno. Este material debe usar un sombreado que esté especializado para renderizar el terreno (por ejemplo, debes manejar la textura que se aplique apropiadamente).</td></tr>
<tr><td rowspan="2">Reflection Probes</td><td>Cómo se utilizan las sondas de reflexión en el terreno. Solo es efectivo cuando se utiliza un material estándar incorporado o un material personalizado que admita la representación con reflexión.</td></tr>
<tr><td>• Off. Las sondas de reflexión están desactivadas, skybox se utilizará para la reflexión.
• Blend Probes. Las sondas de reflexión están habilitadas. La mezcla se produce solo entre las sondas. La reflexión predeterminada se utilizará si no hay sondas de reflexión cerca, pero no se producirá ninguna mezcla entre la reflexión por defecto y la sonda.
• Blend Probes And Skybox. Las sondas de reflexión están habilitadas. La mezcla se produce entre las sondas o las sondas y la reflexión por defecto.
• Simple. Las sondas de reflexión están habilitadas, pero no se producirá ninguna mezcla entre sondas cuando hay dos volúmenes superpuestos.</td></tr>
<tr><td>Thickness</td><td>Qué cantidad de volumen de colisión del terreno debe extenderse a lo largo del eje Y negativo. Los objetos se consideran colisionados con el terreno desde la superficie hasta una profundidad igual al espesor. Esto ayuda a evitar que los objetos en movimiento de alta velocidad penetren en el terreno sin utilizar una gran cantidad de colliders continuos.</td></tr>
</table>

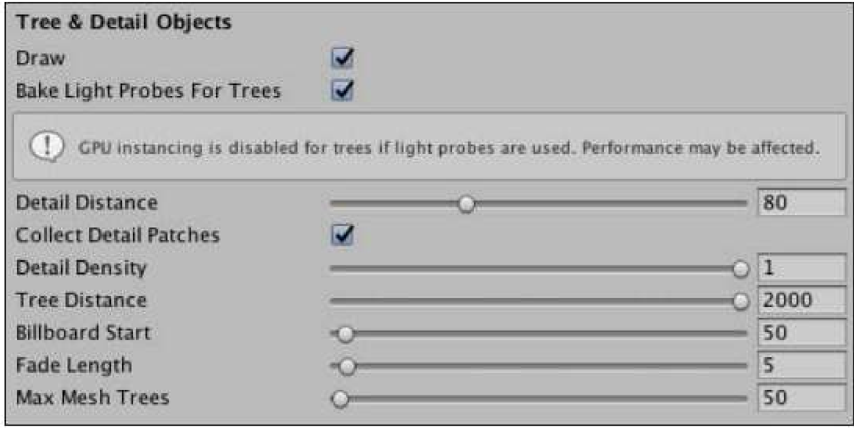


Fig. 3.5

Tree & Detail Objects	
Draw	Activa o desactiva la representación de árboles , hierba y detalles.
Detail Distance	La distancia (de la cámara) más allá de los detalles que serán eliminados.
Detail Density	El número de objetos de detalle / hierba en una unidad de área dada. El valor se puede establecer más bajo para reducir la sobrecarga de renderizado.
Tree Distance	La distancia (desde la cámara) más allá de los árboles que serán sacrificados.
Billboard Start	La distancia (desde la cámara) en la que los objetos del árbol 3D serán reemplazados por las imágenes de la cartelera.
Fade length	Distancia sobre la cual los arboles harán la transición entre objetos 3D y carteleras.
Max Mesh Trees	El número máximo de árboles visibles que se representarán como mallas 3D sólidas. Más allá de este límite, los árboles serán reemplazados por vallas publicitarias.

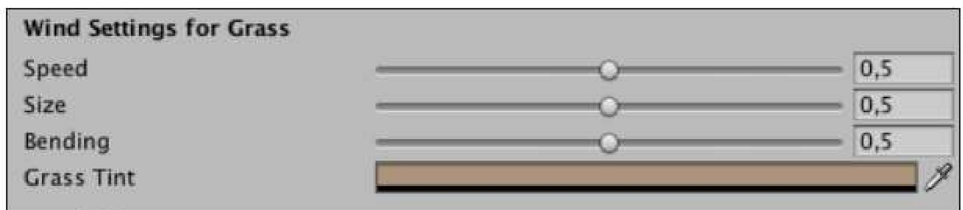


Fig. 3.6

Wind Settings for Grass	
Speed:	La velocidad del viento al soplar el césped.
Size:	El tamaño de las "ondulaciones" en las áreas cubiertas de hierba cuando el viento sopla sobre ellas.
Bending:	El grado en que los objetos de hierba están doblados por el viento.
Grass Tint:	Teñido general del color aplicado a los objetos de la hierba.

Resolution	
Terrain Width	500
Terrain Length	500
Terrain Height	600
Heightmap Resolution	513
Detail Resolution	1024
Detail Resolution Per Patch	8
Control Texture Resolution	512
Base Texture Resolution	1024

Fig. 3.7

Resolution	
Terrain Width	Tamaño del objeto de terreno en su eje X (en unidades unity).
Terrain Length	Tamaño del objeto de terreno en su eje Z (en unidades unity).
Terrain Height	Diferencia en la coordenada Y entre el valor de mapa de altura más bajo posible y el más alto (en unidades del mundo).
Heightmap Resolution	Resolución de píxeles del mapa de altura del terreno (debe ser una potencia de dos más uno, por ejemplo, 513 = 512 1).
Detail Resolution	Resolución del mapa que determina los parches separados de detalles / hierba. Una resolución más alta ofrece parches más pequeños y más detallados.
Detail Resolution per Patch:	Longitud / ancho del cuadrado de parches renderizado con una sola llamada de dibujo.
Control Texture Resolution:	Resolución del "splatmap" que controla la mezcla de las diferentes texturas del terreno.

Base Texture Resolution:	Resolución de la textura compuesta utilizada en el terreno cuando se ve desde una distancia mayor que la distancia de base (véase más arriba).

Heightmap Importar/Exportar

Los botones Importar y Exportar nos permiten establecer o guardar el mapa de altura del terreno en un archivo de imagen en el formato **RAW** que es una imagen en escala de grises. El formato RAW puede ser generado por herramientas de edición de terrenos de terceros (como Bryce) y también puede ser abierto, editado y guardado por **Photoshop o Gimp**. Esto permite la generación sofisticada y la edición de terrenos fuera de Unity.

Fig. 3.8

Explicada las propiedades de configuración, cambia el tamaño de tu terreno de forma que no te consuma muchos recursos. En el ejemplo que te muestro he puesto en el apartado de resolución los siguientes parámetros:

- Terrain Width= 100
- Terrain Length= 100
- Terrain Height = 50

Puedes poner el valor que te vaya mejor para crear tu terreno.

2. Esculpir la superficie

Vamos a ver cómo trabajar con la primera herramienta de las propiedades **Terrain.**

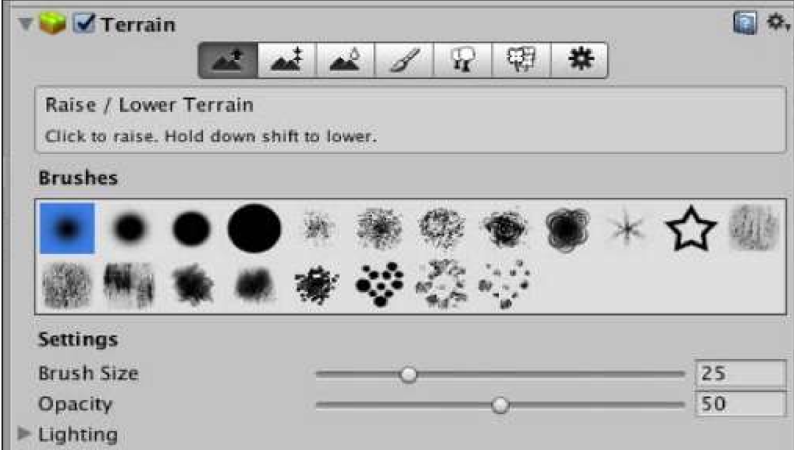


Fig. 3.9

La función es muy simple: tenemos una serie de pinceles (Brushes) con distintas formas que están representadas con una pequeña imagen. Estos pinceles tienen dos opciones para configurar (Settings).

- **Brush Size:** el tamaño, a mayor tamaño nuestros trazos serán más grandes.
- **Opacity:** la opacidad del pincel. A mayor opacidad más fuertes serán nuestros trazos.

Selecciona un pincel y configura el tamaño y la opacidad que quieras. Vamos a la ventana **Scene** y veremos que nuestro cursor se ha convertido en una sombra azulada con la forma de nuestro pincel.

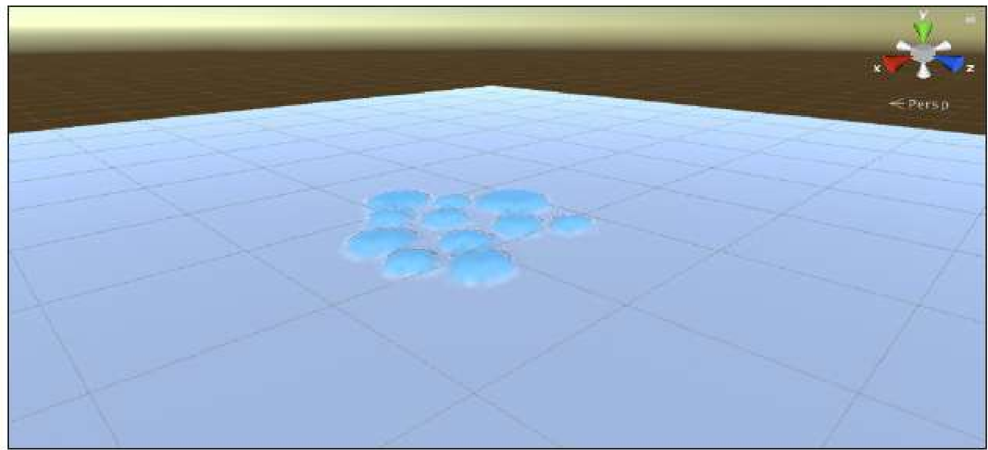


Fig. 3.10

Tenemos dos acciones a realizar con esta opción. Para añadir terreno pulsamos con el Botón Izquierdo del ratón **(BIR).** Para quitar terreno pulsamos Botón Izquierdo del ratón manteniendo pulsada la tecla **SHIFT (BIR+SHIFT).**

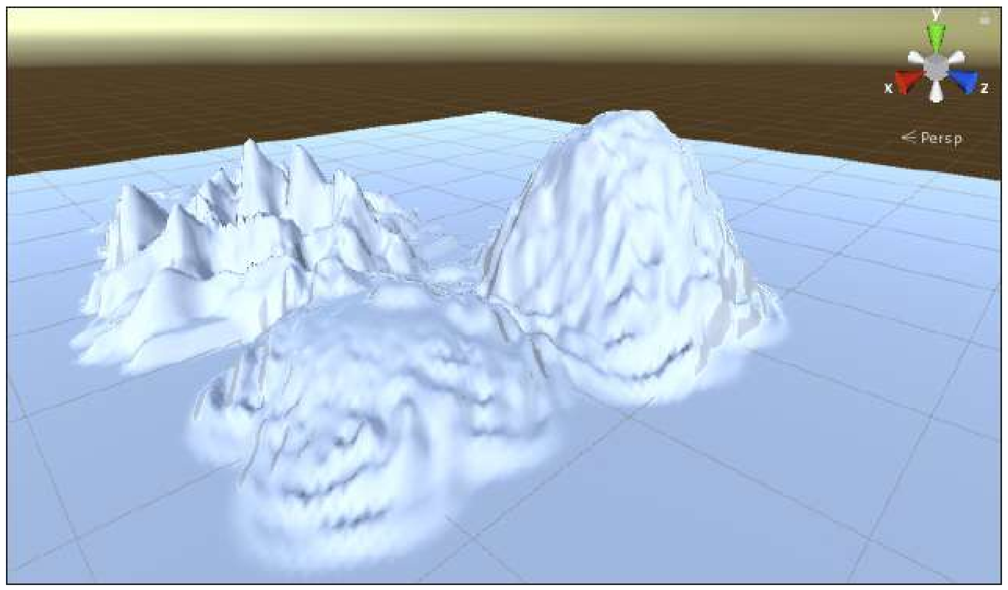


Fig. 3.11

Esta última acción que hemos realizado (la de quitar terreno o hundir), no permite hundir más allá de la base del terreno. Esto sucede porque debemos decirle donde está el nivel 0, es decir tenemos que levantar el terreno a una cierta altura para poder rebajar zonas de nuestro terreno. Lo vemos en el siguiente apartado.

Paint Height

Esta es la segunda opción de la creación de terrenos. Verás que se le añade una opción de configuración el Height. Esta opción determina la posición en la que situaremos nuestro terreno y tiene mucho que ver con la configuración que hemos puesto al principio.

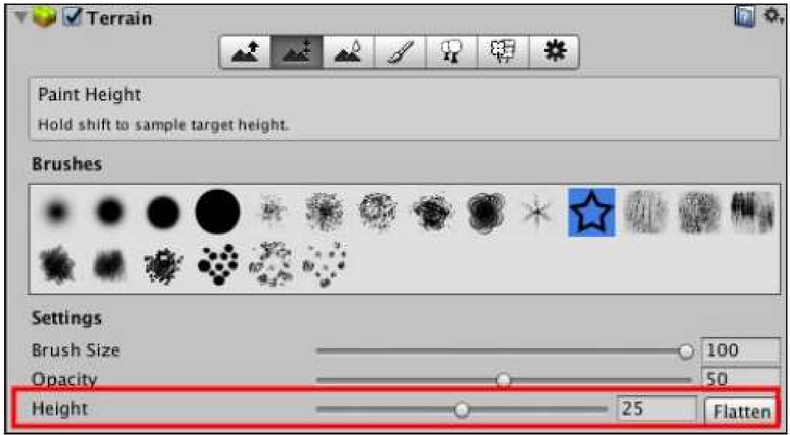


Fig. 3.12

Si recordamos en la configuración del terreno he puesto el valor en **Height** de 50 unidades, esto me permite crear una altura de montañas hasta 50 unidades. En el caso de que quiera crear cráteres o caudales de río tendré que posicionar el terreno por ejemplo a la mitad de la altura máxima. De esta manera ponemos 25 unidades en Height que será la posición base de nuestro terreno y para ejecutar la acción pulsamos **Flatten**.

Fig. 3.13

Ahora si volvemos a la herramienta de elevación de terreno y pulsamos **Shift +BIR** (Botón izquierdo ratón) verás como puedes hundir el terreno hasta 25 unidades. Esta segunda herramienta te permite aplanar el terreno a nivel 0 tanto en las elevaciones como en las hendiduras en el caso de que te hayas equivocado o no te guste el resultado.

Smooth Height

Como su nombre indica, esta opción permite suavizar los relieves del terreno. Contiene las mismas opciones que los pinceles anteriores y la configuración básica; tamaño y opacidad.

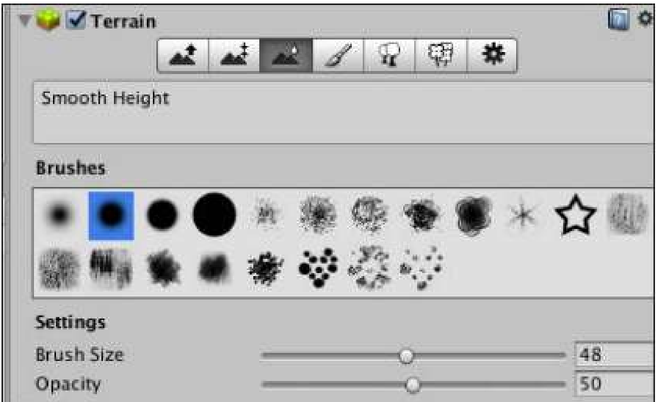


Fig. 3.14

En la siguiente imagen te muestro un mismo relieve: el de la izquierda es sin suavizar y el de la derecha aplicando la herramienta **Smooth Height.**

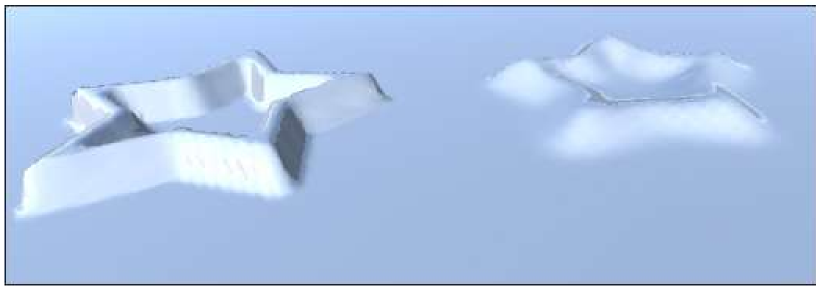


Fig. 3.15

3. Pintar el terreno

Paint Texture

Esta es la opción que nos permite pintar el terreno con texturas, pero en un principio no puede pintar porque no tiene ninguna textura asignada.

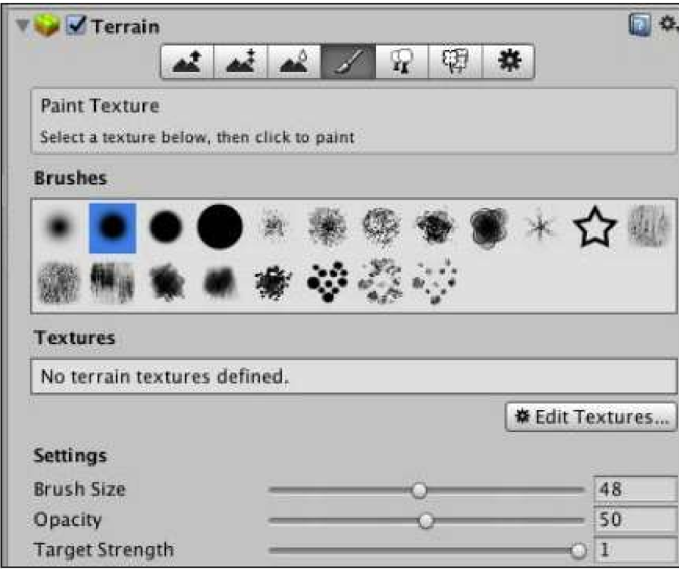


Fig. 3.16

Si miramos en sus opciones verás que tiene un botón llamado **Edit Textures**. Al hacer clic encima veremos un menú emergente que solamente nos permite seleccionar la opción **Add texture**, hacemos clic en esta opción.

Atención: Debes importar los assets del capítulo 3 para que puedas disponer del material necesario.

Se nos aparecerá un menú flotante que nos permite añadir texturas para pintar. El menú tiene dos recuadros, uno para la textura de color y el otro para la textura de relieve. Selecciona el primer recuadro, escoge una textura de hierba (GrassHilAlbedo) y pulsa en el botón Add.

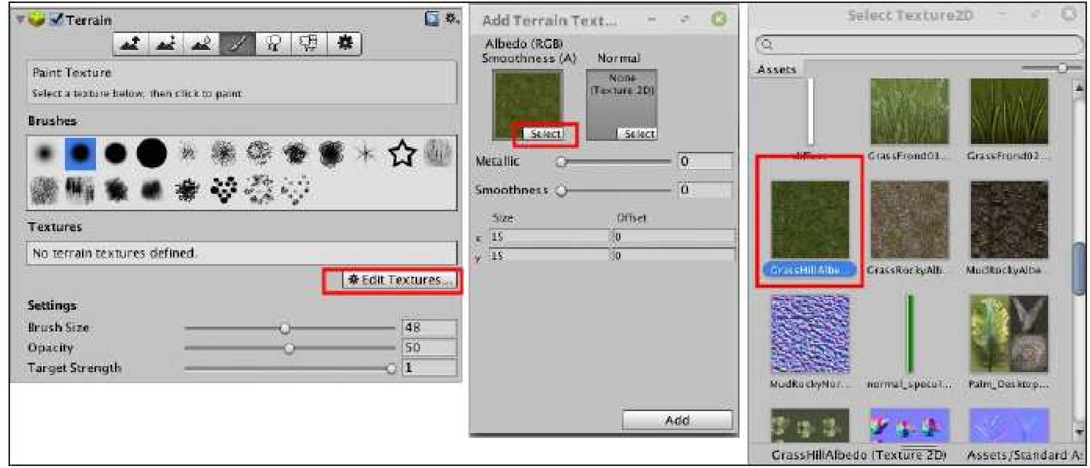


Fig. 3.17

Por defecto Unity pinta todo el terreno con la primera textura. En el caso de que quisiéramos escalar la textura solamente debemos volver a las propiedades de terreno y acceder al menú **Edit Texture > Edit Texture**. Aparece otra vez el menú anterior y podemos variar los parámetros de la textura en X e Y cómo te muestro a continuación.

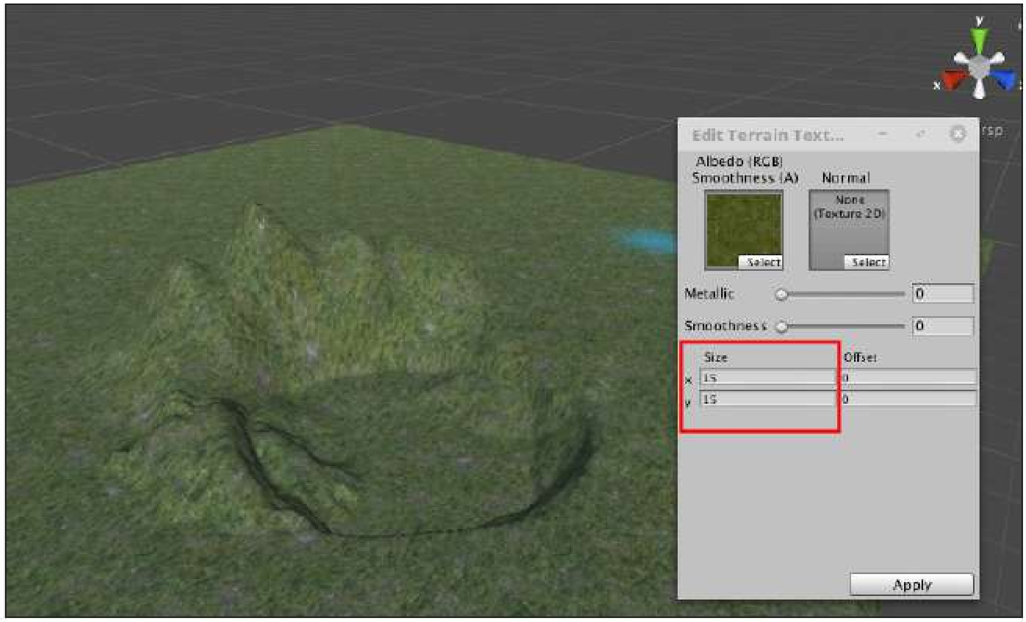


Fig. 3.18

Ahora vamos a añadir una textura de piedra para pintar las montañas. Pulsamos en las propiedades del terreno **Edit Texture> Add** texture y seleccionamos una textura que nos guste **MudRockyAlbedo** esta textura también tiene una textura de relieve (Normal Map) así que hacemos clic en el recuadro de al lado, seleccionamos **MudRocky Normals** y pulsamos en el botón **Add**. Ahora solo tenemos que seleccionar esta textura nueva, seleccionar el tipo de pincel y en la configuración de pinceles debemos tener en cuenta la opción **Target strength**. Esta opción nos permite regular la fuerza con la que pintamos, a mayor fuerza mejor se verá la textura que pintamos.

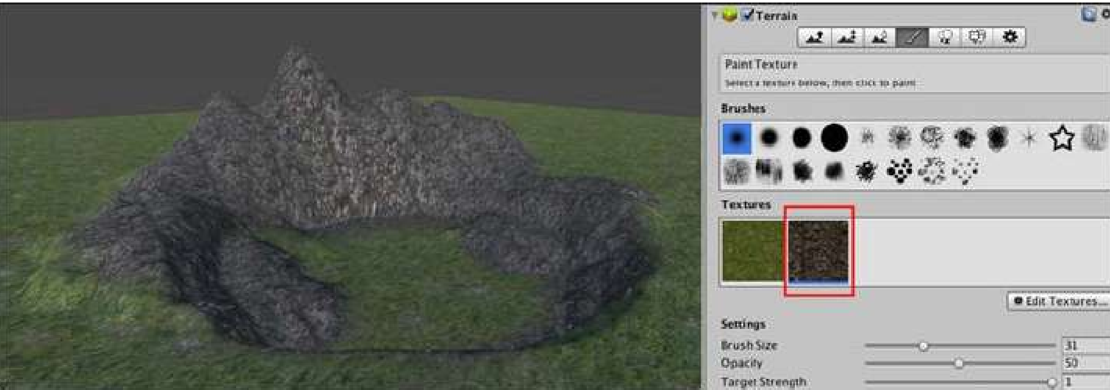


Fig. 3.19

Nota: si quieres tener un entorno más profesional debes intentar que las texturas se difuminen unas con otras, es decir utiliza los parámetros opacity y Target strength con conocimiento buscando el equilibrio en ambas.

4. Poner vegetación

Place Tree

Esta opción nos permite pintar árboles. En realidad los árboles son prefabs que se crearán cuando hagamos un trazo en nuestro terreno.

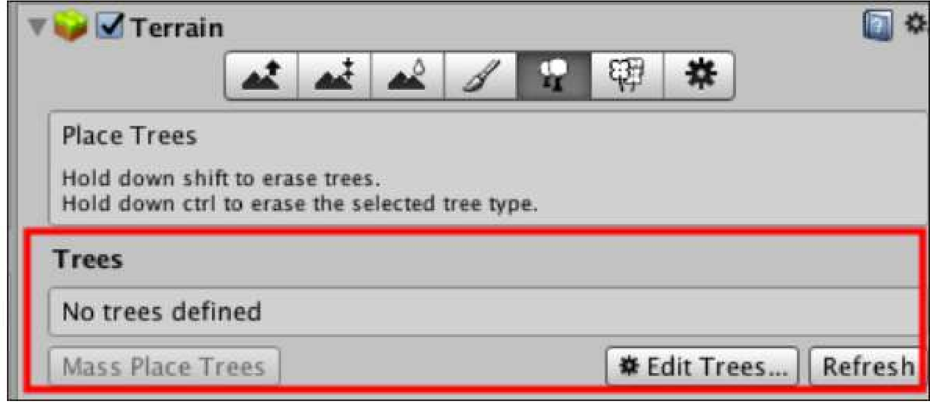


Fig. 3.20

Como en el caso de las texturas debemos hacer clic en la opción **Edit Trees > Add Trees** para añadir estos **prefabs**. Selecciona uno por ejemplo **Palm_Desktop** y confirma pulsando la opción **Add.**

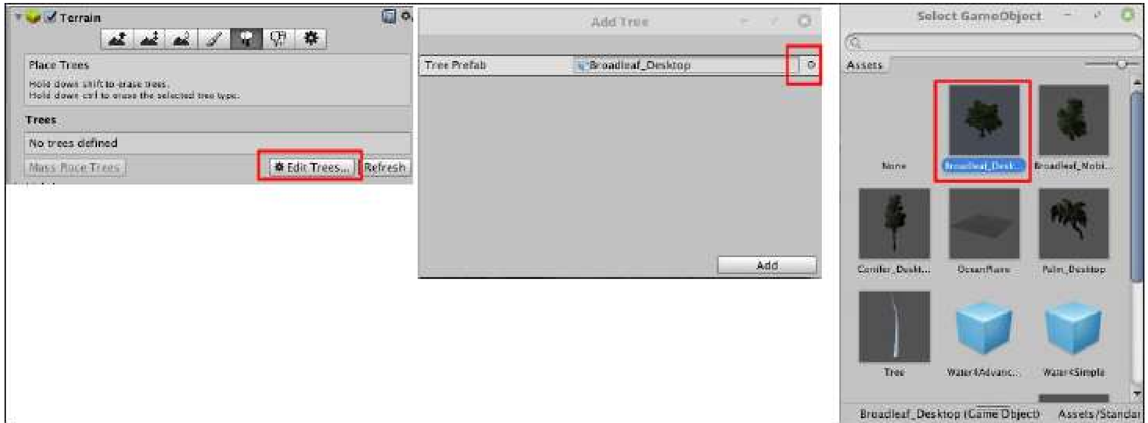


Fig. 3.21

Atención: si has cargado los Assets del capítulo 3 tendrás algunos árboles para utilizar, si no lo has hecho todavía debes cargar este paquete para poder seguir la explicación.

Una vez tenemos un árbol seleccionado se nos aparecerán varias opciones, que nos permiten modificar la configuración de nuestro trazo en el terreno.

Brush size	El tamaño que abarca nuestro pincel a la hora de crear árboles.
Tree Density	La cantidad máxima de árboles que creará en cada trazo. Este parámetro va muy relacionado al tamaño del pincel.
Tree Height	Este parámetro nos permite jugar entre una cantidad de valores que hacen referencia a la altura de los árboles. También tiene la opción Random, que al estar activo hace que la creación de nuestros arboles varíe con cada trazado.
Lock Width to Height:	Esta opción al estar activada, hace que los árboles tengan una proporción entre ancho y alto.
Tree Width	Esta opción por defecto esta desactivada. Se activa al desactivar la opción anterior **Lock Width to Height**, y nos permite crear árboles más delgados aleatoriamente según qué cantidades le configuremos.
Random Tree Rotation	Permite al estar activo que los árboles tomen una rotación aleatoria y de este modo dar un efecto mas realista.

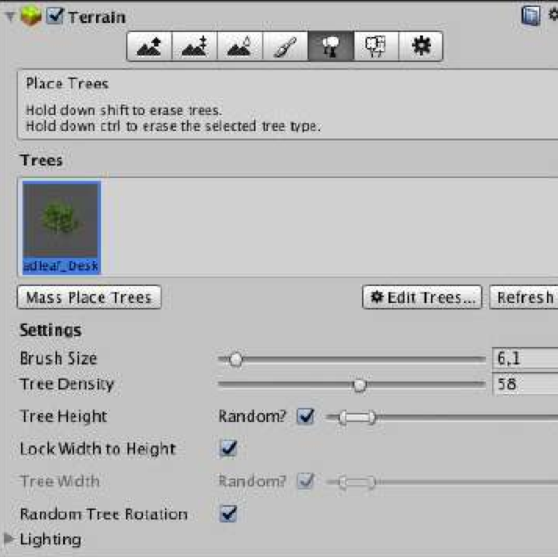


Fig. 3.22

En las opciones de configuración también tenemos un botón llamado **Mass Place Trees;** al pulsarlo se nos abre un menú en donde podemos poner un número de árboles para que Unity nos cree todos los árboles en el terreno aleatoriamente. Esta es una forma muy rápida de crear vegetación pero casi siempre crea árboles en zonas que no deseamos.

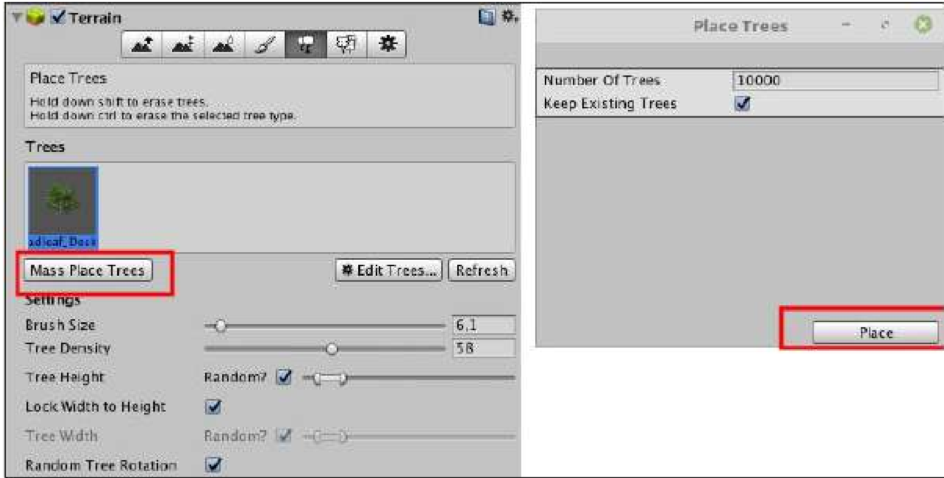


Fig. 3.23

Para borrar todos los árboles que abarca el pincel pulsar SHIFT +BIR. Para borrar los árboles individualmente pulsar CTRL+BIR.

Paint Details

Con esta opción podemos crear vegetación o detalles de otro tipo a partir de texturas con canal alfa o modelos y props como rocas para dar ambiente a nuestro escenario. El funcionamiento es el mismo que hasta ahora tenemos los pinceles y su configuración igual que los árboles.

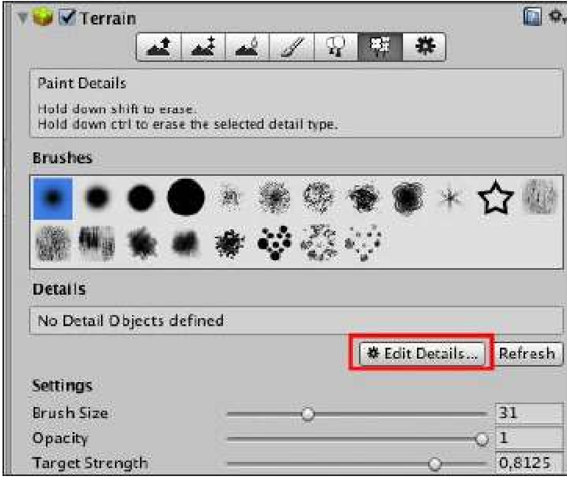


Fig. 3.24

Para añadir una textura con canal alfa debes seleccionar **Add Grass Texture**; si deseas añadir un modelo debes seleccionar la opción **Add Detail Mesh**.

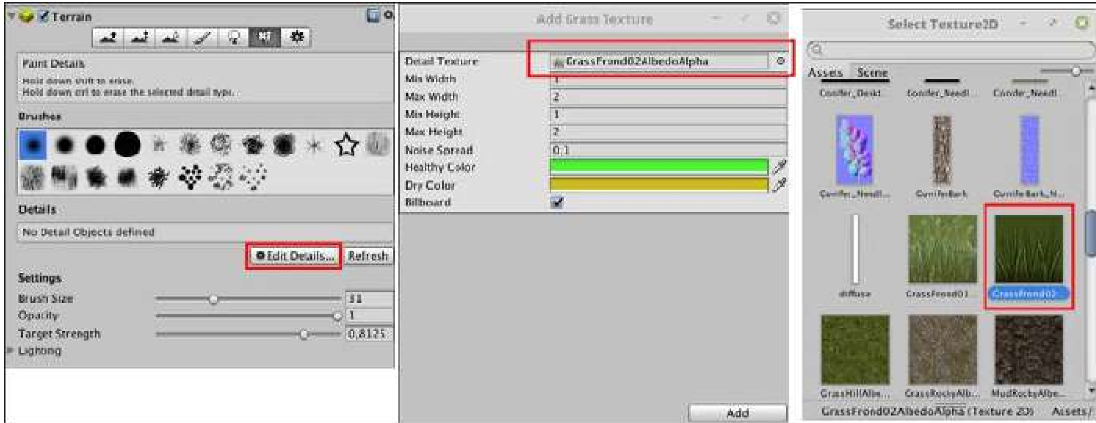


Fig. 3.25

En el menú que se aparece debes escoger en la primera opción para seleccionar la textura que desees; en este caso se ha seleccionado GrassFrond02Albedo. Si deseas cambiar el color o el tamaño de la textura puedes jugar con los parámetros de debajo.

5. Poner agua en el terreno

Unity dispone dentro de la carpeta Environment de una carpeta llamada Water y Water Basic en donde podemos utilizar sus prefabs para poner aguas muy realistas en nuestros terrenos.

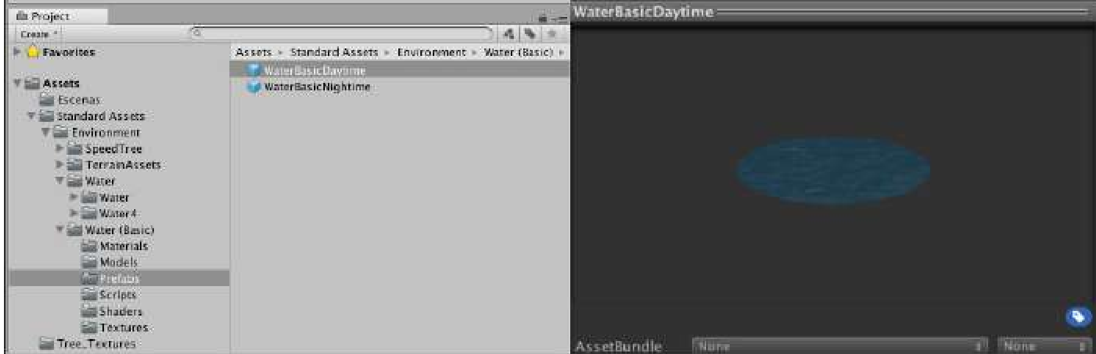


Fig. 3.26

El prefab es un gameObject con características ya predefinidas, para utilizar este prefab desde la carpeta Project selecciona el prefab WaterBasic o el que desees y arrástralo hacia la ventana escena. Ahora debes escalar y posicionar el prefab desde la ventana Inspector, para que tenga el tamaño y la posición adecuados.

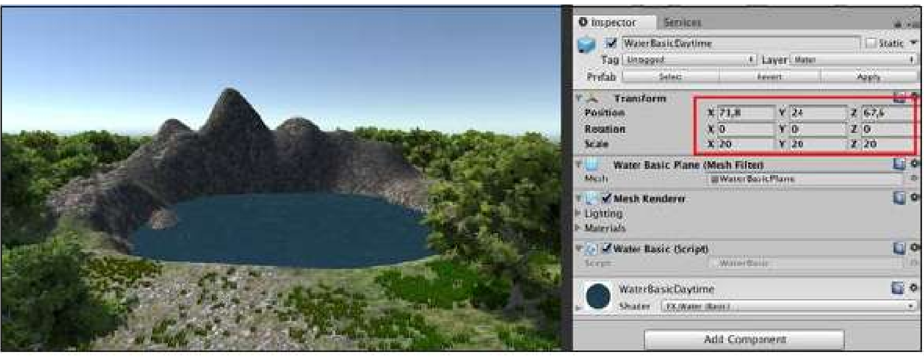


Fig. 3.27

6. Crear una zona de viento (windzone)

Las zonas de viento añaden realismo a los árboles haciendo que estos agiten sus ramas y hojas como si realmente hiciera viento.

Podemos crear una zona de viento accediendo al menú principal en **GameObject > 3D Object > Wind Zone**

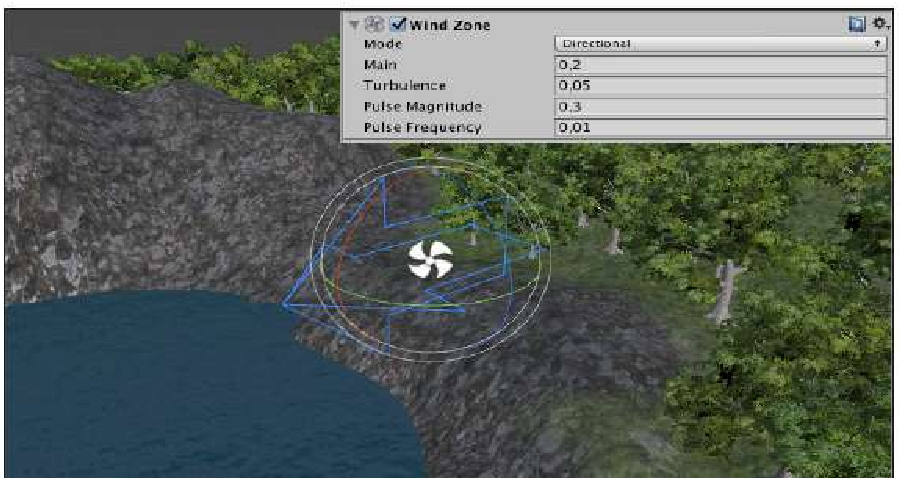


Fig. 3.28

Propiedades de WindZone	
Mode	Esta opción es la que te permite seleccionar entre un wind zone esférico o direccional.
	Spherical. La zona del viento sólo tiene un efecto dentro del radio de la esfera, y va perdiendo fuerza desde el centro hacia el borde. **Directional.** La zona de viento es direccional y afecta a toda la escena en una dirección.

Radius:	Radio de la zona de viento esférica (solo activo si el modo se establece en Esférico).
Main:	La fuerza primaria del viento. Produce una presión de viento que cambia suavemente.
Turbulence:	La fuerza del viento de la turbulencia. Produce una presión de viento que cambia rápidamente.
Pulse Magnitude:	Define cuánto el viento cambia con el tiempo.
Pulse Frequency:	Define la frecuencia de los cambios de viento.

Consejo: para producir un viento general que cambia suavemente, cree una zona de viento direccional. Establece Main a 1.0 o menos, dependiendo de la potencia del viento, ajustar la turbulencia a 0.1, establece la magnitud de pulso en 1.0 o más y para finalizar establecer la frecuencia de pulso a 0.25.

7. Editar árboles

Unity dispone de una herramienta llamada **Tree editor** que permite al usuario crear cualquier tipo de árbol. Esta herramienta es especialmente útil para poder crear distintos tipos de bosques.

Para crear nuestro primer árbol debemos acceder en el menú de herramientas en **GameObject > 3D Object > Tree**.

Una vez creado veras que aparece el tronco de un árbol con un círculo de color amarillo. En el inspector aparece una serie de parámetros. Esta interfaz proporciona todas las herramientas para modelar y esculpir sus árboles. En la ventana de esta interfaz verás dos nodos que son: el nodo raíz del árbol y un único nodo de grupo de ramas, que llamaremos tronco del árbol. En la jerarquía de árbol, selecciona el grupo de rama, que actúa como el tronco de árbol.

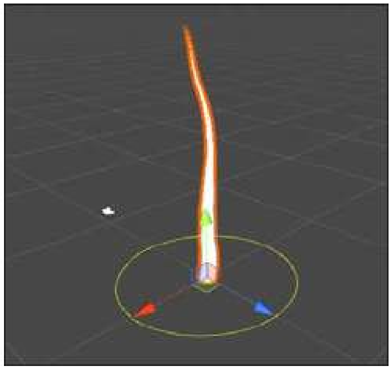


Fig. 3.29

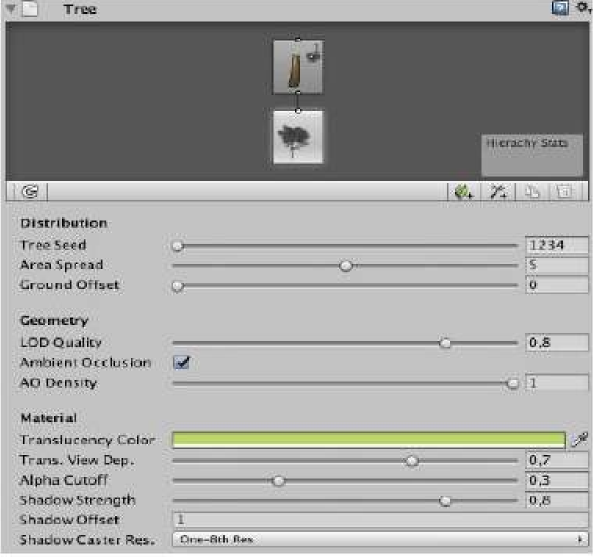


Fig. 3.30

El primer parámetro que vemos en el panel inspector es una ventana que contiene la estructura de nuestro árbol. En la parte inferior derecha veremos una serie de botones que nos permiten añadir elementos a nuestro árbol. Antes de empezar a trabajar con el debemos entender cómo funciona. El nodo que contiene el dibujo de un árbol es el nodo raíz y por tanto este es el nodo base donde se sustenta todos los demás elementos.

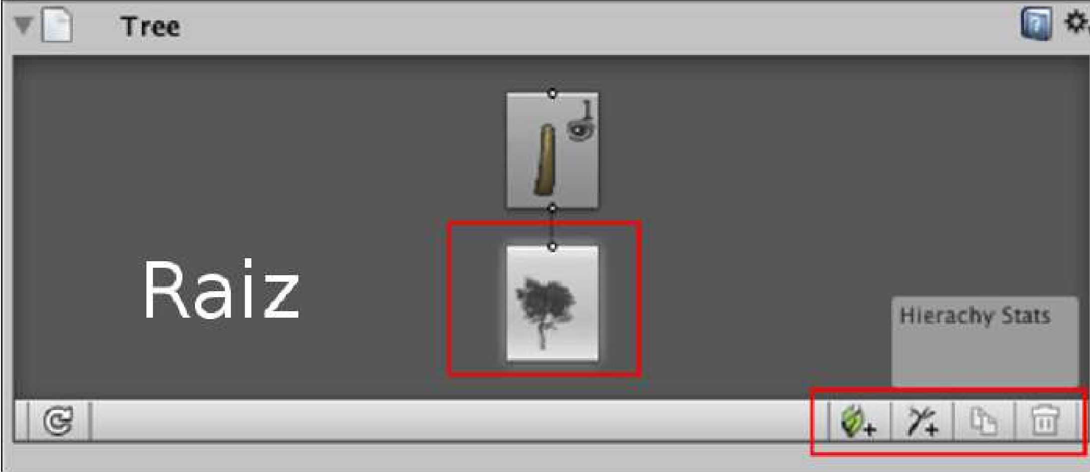


Fig. 3.31

El encabezado

Si hacemos clic en el nodo superior veremos que los parámetros inferiores cambian. Estos nuevos parámetros afectan solamente al nodo seleccionado, excepto al de raíz que afecta a todo el conjunto. El nodo con el icono de una rama es el nodo del tronco y a este nodo le añadiremos nuevas ramas que a su vez tendrán sub-ramas y estos últimos tendrán las hojas. Los nodos disponen de un icono principal que muestra el tipo de elemen-

to que es, el número superior derecho es el número de ramas que existen en este nivel del árbol y la imagen del ojo nos permite activar o desactivar la visibilidad del objeto en la ventana escena.

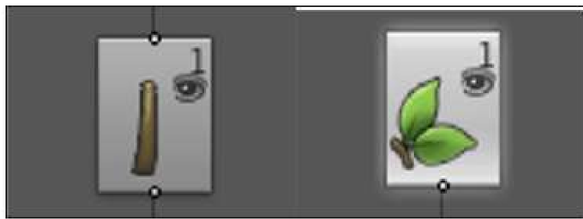


Fig. 3.32

En la parte inferior derecha disponemos de una serie de herramientas para añadir nuevos nodos. De izquierda a derecha la primera herramienta añade grupos de hojas, estas se disponen en niveles igual que las ramas, pero a diferencia de las ramas, las hojas no pueden subdividirse en más niveles. La segunda herramienta añade grupos de ramas al nivel que tengamos seleccionado, es decir si tenemos seleccionado el nivel 1 (el tronco) iremos añadiendo ramas a este. La tercera herramienta duplica el nodo que tengamos seleccionado y la última herramienta elimina el grupo seleccionado.

Fig. 3.33

Editar grupos

Hemos visto cómo podemos crear grupos y ahora vamos a ver cómo podemos editarlos manualmente. Cuando seleccionamos una rama en la ventana de nodos, veras que en la ventana de escena aparece una curva y la recorre una serie de cubos. Estos cubos son puntos de control que podemos seleccionar y desplazar para dar una forma concreta a la rama.

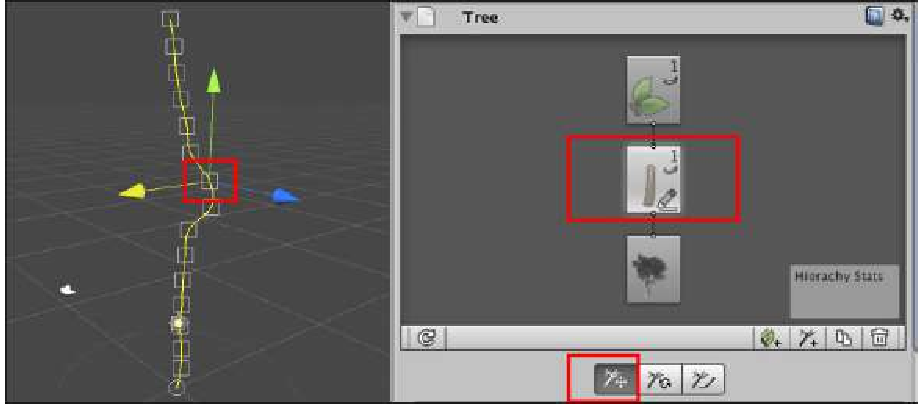


Fig. 3.34

Estos puntos de control pueden ser de desplazamiento de rotación o también Unity nos da la posibilidad de dibujar la rama trazando una trayectoria.

Para seleccionar que el tipo de edición que queremos hacer disponemos de un menú que se encuentra debajo de la ventana de nodos con los siguientes botones:

Para la rotación los puntos de control tomaran la forma de un círculo y para crear un trazo simplemente debemos empezar a trazar una trayectoria encima de del punto de control en donde queremos que empiece.

Propiedades de los grupos de rama

Estas propiedades aparecen siempre que tengamos seleccionado el nodo del grupo de ramas y las propiedades solamente afectarán a ese grupo de ramas.

Distribution

Ajusta y distribuye las ramas en el grupo. Utiliza las curvas para ajustar la posición, la rotación y la escala. Las curvas son relativas a la rama madre o al área extendida en caso de un tronco.

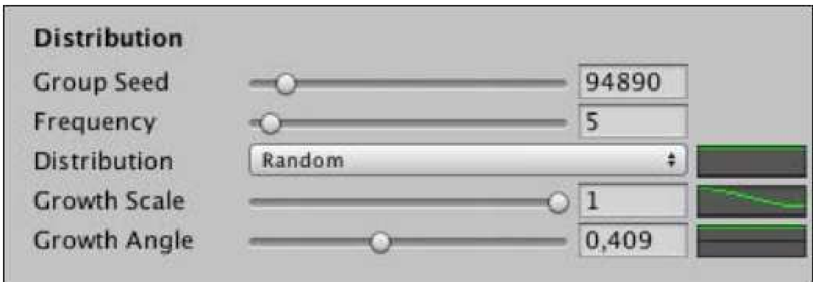


Fig. 3.35

Group Seed	La semilla para este grupo de ramas. Modifica el valor para variar la generación de ramas.
Frequency	Ajusta el número de ramas creadas para cada rama principal.
Distribution	La forma en que las ramas se distribuyen a lo largo de sus padres.
Growth Scale	Define la escala de nodos a lo largo del nodo padre. Utiliza la curva para ajustar y el deslizador para atenuar el efecto.
Growth Angle	Define el ángulo inicial de crecimiento con respecto al padre. Utiliza la curva para ajustar y el deslizador para atenuar el efecto.

Geometry

Selecciona qué tipo de geometría se genera para este grupo y qué materiales se aplicarán. LOD Multiplicador te permite ajustar la calidad de este grupo en relación con la calidad LOD del árbol.

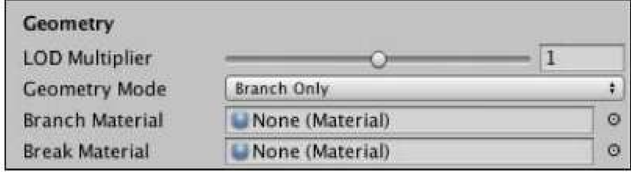


Fig. 3.36

LOD Multiplier	Ajusta la calidad de este grupo en relación con la calidad LOD del árbol, de modo que sea de calidad superior o inferior al resto del árbol.
Geometry Mode	Tipo de geometría para este grupo de rama: Solo rama, Frondas de rama, Frondas solamente.
Branch Material	Primer material para las ramas.
Break Material	Material para cubrir las ramas rotas.

Shape

Ajusta la forma y el crecimiento de las ramas. Utiliza las curvas para afinar la forma, todas las curvas son relativas a la propia rama.

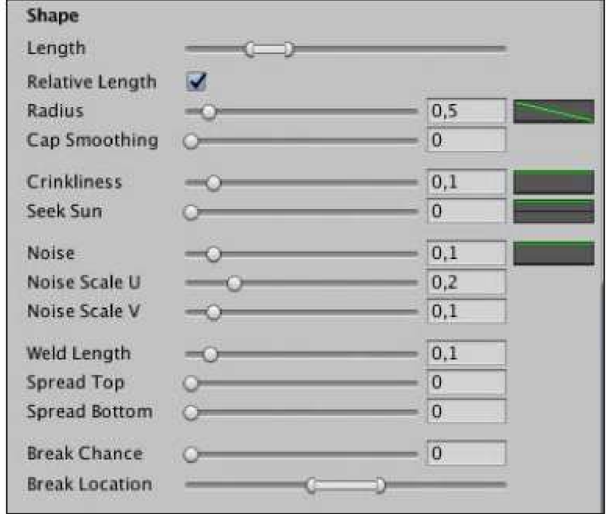


Fig. 3.37

Lenght	Ajusta la longitud de las ramas.
Relative Length	Determina si el radio de una rama está afectado por su longitud.
Radius	Ajusta el radio de las ramas, utiliza la curva para ajustar el radio a lo largo de la longitud de las ramas.
Cap Smoothing	Define la redondez de la tapa / punta de las ramas. Útil para cactus.
Growth	Ajusta el crecimiento de las ramas.
Crinkliness	Ajusta la forma torcida de las ramas, utiliza la curva para afinar.
Seek Sun	Utiliza la curva para ajustar cómo las ramas se doblan hacia arriba / hacia abajo y el deslizador para cambiar la escala.
Noise	Factor de ruido general
Noise Scale U	Escala del ruido alrededor de la rama, los valores más bajos darán un aspecto más vacilante, mientras que los valores más altos dan un aspecto más estocástico.
Noise ScaleV	Escala del ruido a lo largo de la rama, los valores más bajos darán un aspecto más vacilante, mientras que los valores más altos da un aspecto más estocástico.
Flare Radius	El radio de los puntos de luz, esto se añade al radio principal, por lo que un valor cero significa que no hay puntos brillantes.
Flare Height	Define hasta qué punto el tronco comienzan los puntos de luz.
Flare Noise	Define el ruido de los puntos de luz, los valores más bajos darán un aspecto más vacilante, mientras que los valores más altos darán un aspecto más estocástico.
Break Chance	Posibilidad de rotura de una rama, es decir, 0 = ninguna rama está rota, 0,5 = la mitad de las ramas están rotas, 1,0 = todas las ramas están rotas.
Break Location:	Esta gama define dónde se romperán las ramas. Relativo a la longitud de la rama.

Las siguientes propiedades las encontrarás en las ramas hijas del tronco.

Weld Length	Define qué tan arriba en la rama comienza la extensión de la unión.
Spread Top	Factor de propagación de la unión en la parte superior de la rama, en relación con su rama principal. Cero significa que no hay propagación.
Spread Bottom	Factor de ensanchamiento de la unión en la parte inferior de la rama, en relación con su rama principal. Cero significa que no hay propagación.

Wind

Ajusta los parámetros utilizados para animar el conjunto de ramas. Las zonas de viento solo están activas en el modo de reproducción.

Fig. 3.38

Main Wind	Efecto del viento primario. Esto crea un suave movimiento de balanceo y es el único parámetro necesario para las ramas primarias.
Main Turbulence	Turbulencia a lo largo del borde de las frondas. Útil para helechos, palmeras, etc.

Propiedades de los grupos de hojas

Estas propiedades aparecen siempre que tengamos seleccionado el nodo del grupo de hojas. Los grupos de hojas generan la geometría del follaje. Ya sea de primitivas o de mallas creadas por el usuario.

Distribution

Ajusta el recuento y la colocación de las hojas en el grupo. Utiliza las curvas para ajustar la posición, la rotación y la escala. Las curvas son relativas a la rama madre.

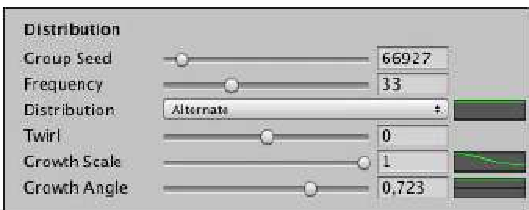


Fig. 3.39

Group Seed	El comienzo para este grupo de hojas. Modifica los valores para variar la generación del comienzo.
Frequency	Ajusta el número de hojas creadas para cada rama principal.
Distribution	Selecciona la forma en que las hojas se distribuyen a lo largo de su padre.

Twirl	Gira alrededor de la rama principal.
Growth Scale	Define la escala de nodos a lo largo del nodo padre. Utiliza la curva para ajustar y el deslizador para atenuar el efecto.
Growth Angle	Define el ángulo inicial de crecimiento en relación con el padre. Utiliza la curva para ajustar y el deslizador para atenuar el efecto.

Geometry

Seleccione qué tipo de geometría se genera para este grupo de hojas y qué materiales se aplican. Si utiliza una malla personalizada, se utilizarán sus materiales.

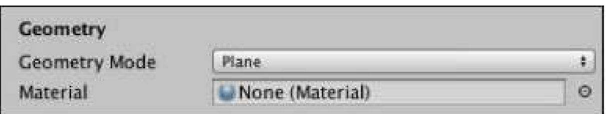


Fig. 3.40

Geometry Mode	El tipo de geometría creada. Puedes usar una malla personalizada, seleccionando la opción Malla, ideal para flores, frutas, etc.
Material	Material utilizado para las hojas.

Shape

Ajusta la forma y el crecimiento de las hojas.

Fig. 3.41

Size	El tipo de geometría creada. Puedes usar una malla personalizada, seleccionando la opción Malla, ideal para flores, frutas, etc.
Perpendicular Align	Ajusta si las hojas están alineadas perpendicularmente a la rama principal.
Horizontal Align	Ajusta si las hojas están alineadas horizontalmente.

Wind

Ajusta los parámetros utilizados para animar el conjunto de hojas. Las zonas de viento solo están activas en el modo de reproducción.

Fig. 3.42

Main Wind	Efecto del viento primario. Por lo general esto debe mantenerse como un valor bajo para evitar que las hojas floten lejos de la rama principal.
Main Turbulence	Efecto secundario de la turbulencia. Para las hojas esto se debe mantener generalmente con un valor bajo.
Edge Turbulence	Define la cantidad de turbulencia del viento que se produce a lo largo de los bordes de las hojas.

Ejemplo de un árbol

Una vez hemos visto los parámetros principales crea un árbol del tipo que desees como el que te muestro a continuación:

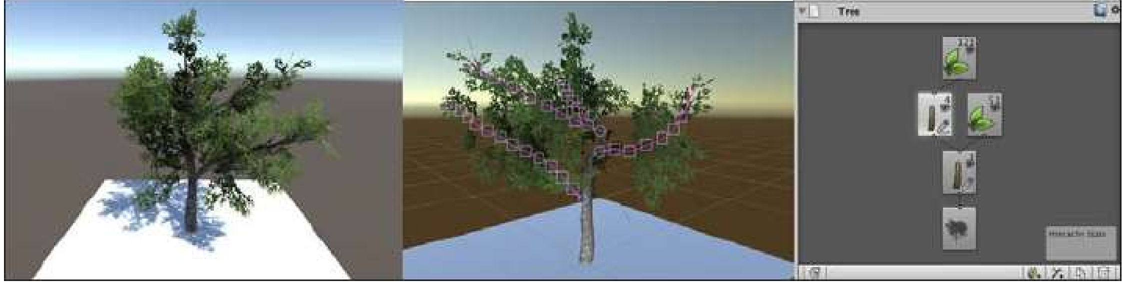


Fig. 3.43

Tienes este ejemplo en el proyecto que acompaña el libro. No debes tener ningún problema si creas nodo por nodo. Cuando tengas que ponerle textura utiliza los que vienen en los **Standard Assets**.

Capítulo 4

Creación de un escenario modular

- Importar los modelos
- Modelos
- Materiales y texturas
- Parámetros básicos de los materiales
- *Colliders* y *Rigid Bodies*
- *Model vs Prefabs*
- Montar un escenario simple
- Importar *Standard Assets* y probar el escenario

1. Importar los modelos

En este capítulo vamos a trabajar con modelos, materiales, texturas y crearemos nuestros prefabs con colisionadores para construir nuestro nivel. Para seguir este capítulo puedes crear un proyecto nuevo e importar el paquete de assets que viene con el material del libro o puedes abrir el proyecto Construcción escenario modular.

Antes de empezar vamos a importar el material de este capítulo 4. Primero debes tener localizada la carpeta con el proyecto de este capítulo y acceder desde la barra principal **Assets > Import Package > Custom Package** y seleccionar el paquete de este capítulo que tiene el nombre de **Modelos_Capitulo 4.unitypackage.**

En el proyecto verás que en la ventana Protector tenemos una carpeta con el nombre Mi_Escenario que contiene las siguientes carpetas:

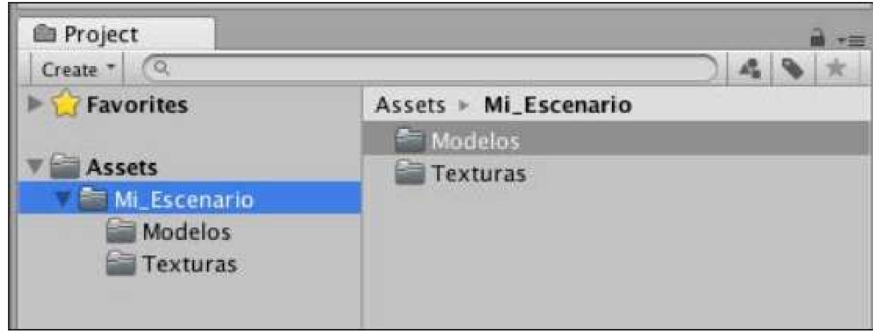


Fig. 4.1

Antes de continuar vamos a crear las carpetas que nos van a hacer falta. Así pues crearemos las siguientes carpetas dentro de Mi_Escenario.

- Materiales
- Prefabs
- Escenas

La ventana inspector debería quedarte como te muestro en la siguiente imagen:

Fig. 4.2

2. Modelos

En esta carpeta tenemos los modelos que han sido creados con un programa de 3d como puede ser Blender, Maya, Lightwave, Modo, Cinema4D etc. En este libro no se enseña cómo se crean estos modelos pero si como podemos trabajar con ellos en Unity.

Si seleccionamos uno de los Modelos, por ejemplo el que tiene como nombre Caja_1, veremos que tiene un añadido en su interior como te muestro a continuación:

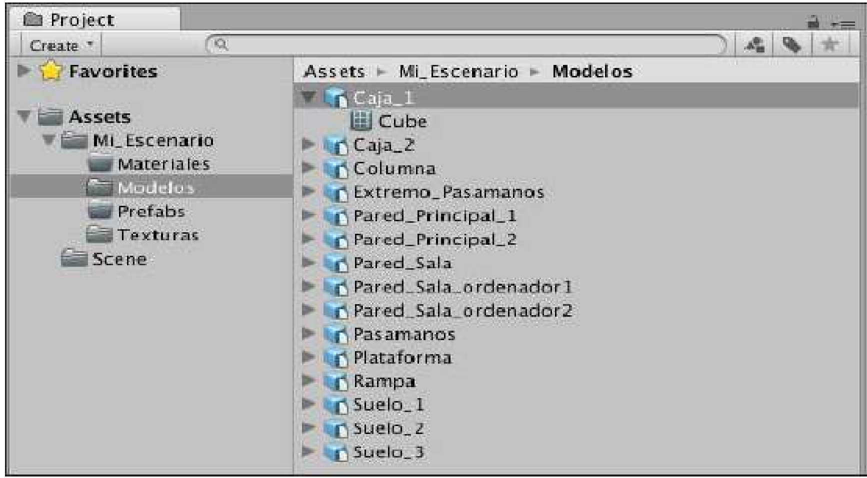


Fig. 4.3

Al seleccionar un Modelo verás que en la ventana inspector aparecen una serie de propiedades del modelo que corresponden a la configuración de su importación. En este apartado disponemos de tres propiedades a configurar dependiendo del modelo.

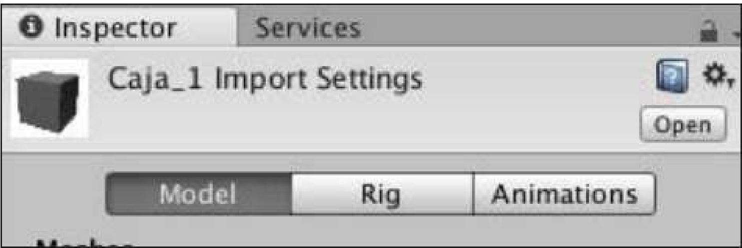


Fig. 4.4

- **Model:** Esta opción contiene todos los parámetros que debemos tener en cuenta sobre la superficie del modelo, que se agrupa en 3 secciones : ***Meshes*** (Superficies), ***Normal & Tangents*** (Referente a la dirección de las normales de las superficies del modelo) y ***Materials*** (Materiales).
- **Rig:** En el caso de que el modelo sea un personaje es posible importar un rig.
- **Animations:** Algunos modelos pueden llevar animaciones.

Para este capítulo solamente vamos a utilizar el modelo como superficie y no vamos a tener ni rig ni animaciones.

Otro aspecto interesante que puedes observar en la ventana inspector es una muestra del modelo visualizado en la parte inferior.

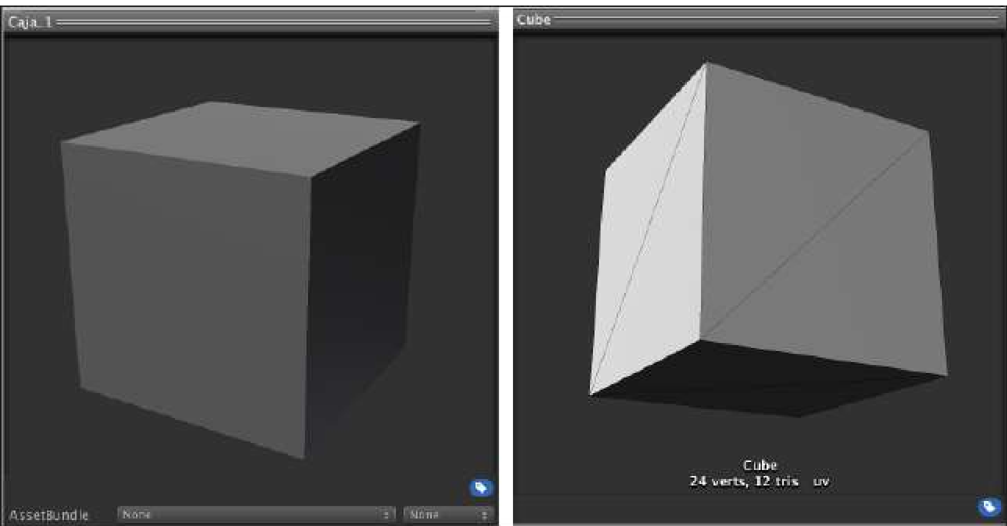


Fig. 4.5

En la imagen anterior el cubo de la izquierda es el modelo con una material por defecto como lo veras en la escena y el cubo de la derecha te muestra la superficie con un número de vértices y triángulos.

Nuestro modelo en escena

A continuación vamos a poner algunos modelos en escena para trabajar con ellos. Para empezar vamos a seleccionar y a arrastrar dentro de la ventana Hierarchy (Jerarquía) 3 modelos; Suelo_1, Pared_Principal_1 y la Caja_1 .

No te preocupes si te quedan un poco desubicados en la escena, porque lo importante es que te queden de una forma parecida a la que te muestro a continuación:

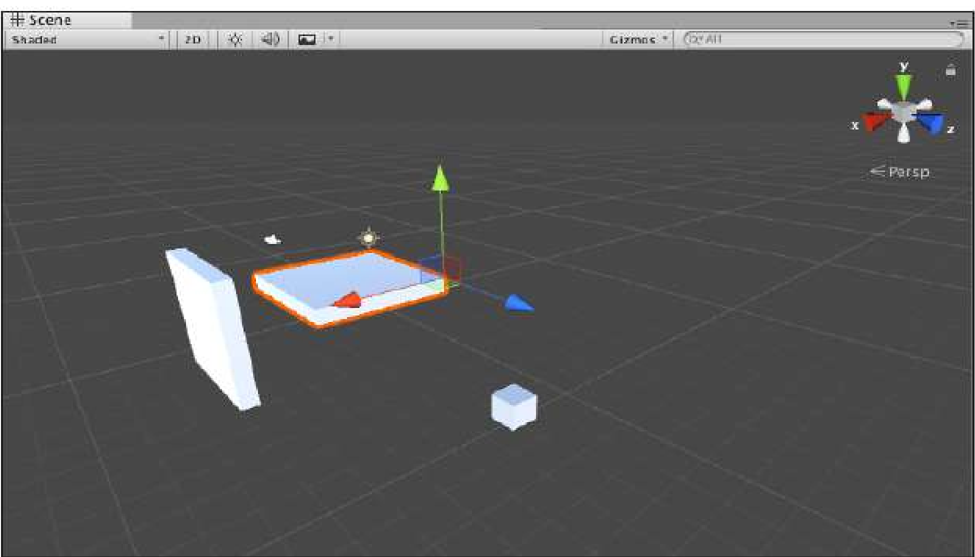


Fig. 4.6

3. Materiales y texturas

De momento nuestros modelos no son gran cosa y es porque tienen un material por defecto que no tiene textura. Bien ahora vamos a crear materiales para nuestros modelos, para ello en la ventana **Project** accedemos a la carpeta Materiales y dentro de ella crearemos un material pulsando botón derecho del ratón y seleccionando la opción **Create>Material** y le ponemos el nombre de Caja_1.

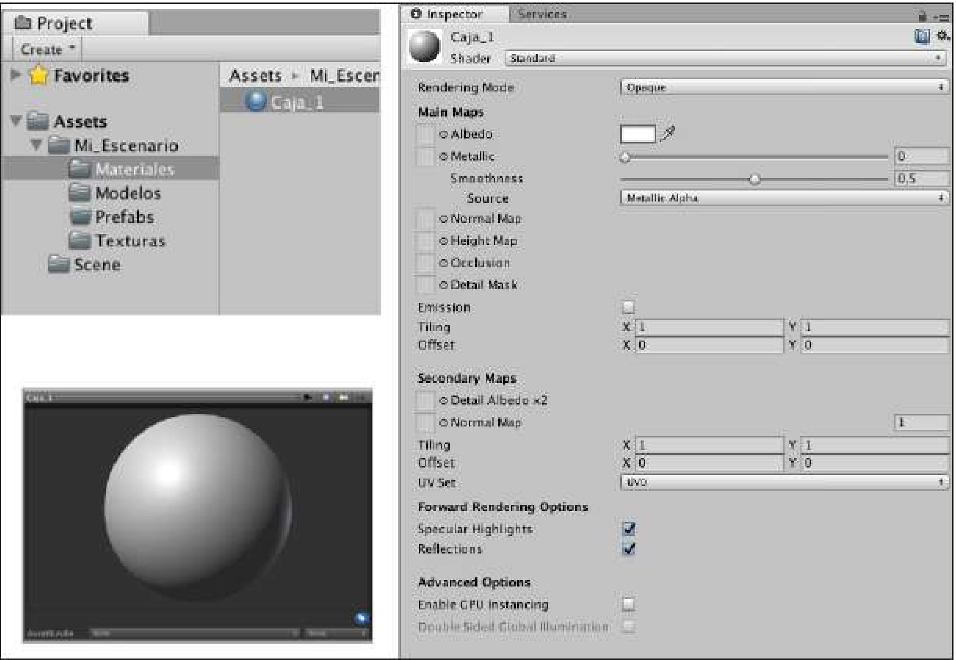


Fig. 4.7

A continuación vamos a ponerle textura a este material pero antes en el menu ***Shader*** donde pone ***Standard*** le damos clic y seleccionamos la opción ***Standard (Specular setup)*** como te muestro en la siguiente imagen:

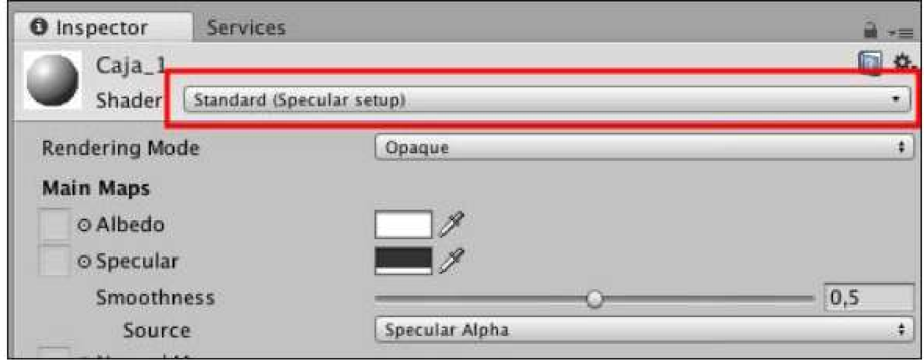


Fig. 4.8

Ahora asegúrate de que **Rendering Mode** se encuentra con la opción Opaque (Opaco) y haz clic encima del punto que encontraras en la opción Albedo. Se abrirá una nueva

ventana con las texturas que se encuentran en la carpeta Texturas, selecciona la textura **Caja_Difuse**.

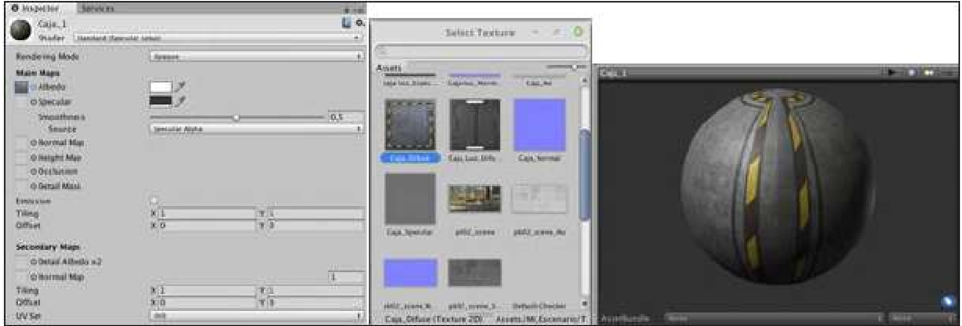


Fig. 4.9

Ahora selecciona el material Caja_1 desde la ventana **Project** y arrástralo encima del modelo Caja_1, como te muestro en la imagen siguiente.

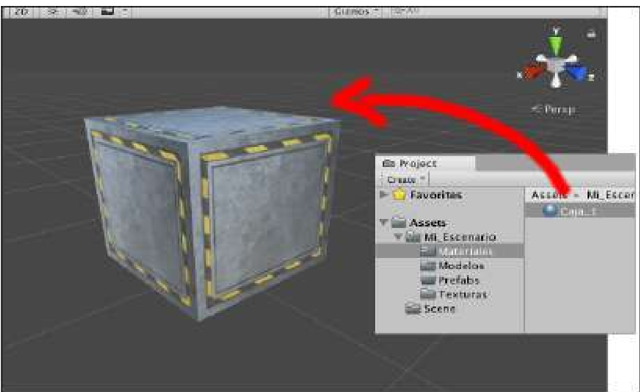


Fig. 4.10

Ahora vemos cómo se le aplica el material al modelo caja_1. El siguiente paso es ponerle al material las texturas de ***Specular***, ***Normal Map*** y ***Occlusion*** como hemos hecho con Albedo. Te dejo una captura de cómo me quedan los parámetros del material Caja_1 y cómo se ve el modelo en escena.

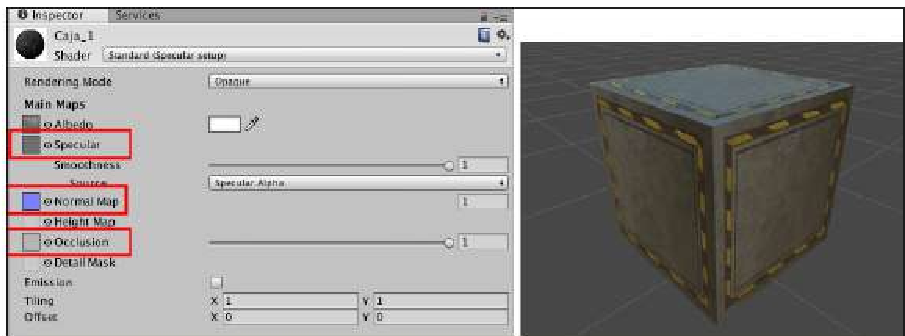


Fig. 4.11

Un apunte que debes tener en cuenta es que cuando le pongas la textura ***Normal Map***, deberás pulsar el botón ***Fix Now*** para que se aplique este tipo de textura.

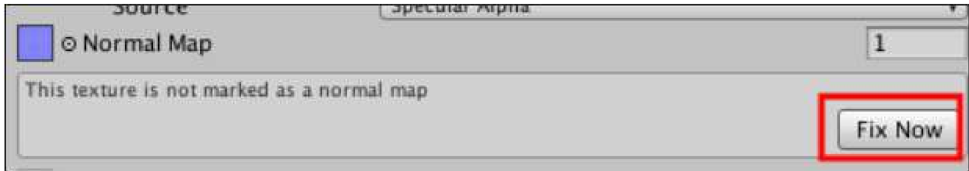


Fig. 4.12

Ahora es el momento de que crees los materiales que faltan para los distintos modelos, para que tengas una pequeña guía de te doy una pista de cuantos materiales debes crear en la siguiente imagen:

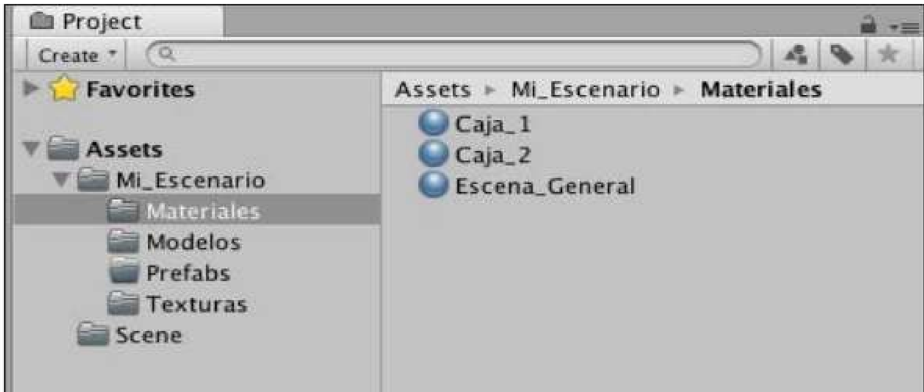


Fig. 4.13

En realidad solamente tienes que crear dos materiales más el de la Caja_2 y otro que he llamado Escena_General que sirve para el resto de modelos. Una vez tengas estos materiales pon todos los modelos en escena y arrastra sus respectivos materiales.

Fig. 4.14

El sombreado de los materiales

Cuando creamos materiales Unity utiliza dos opciones de sombreado el Standard "metallic"y el Standard (Specular setup).

- **Standard "metálico"** dispone de una valor entre (0-1) que identifica si es metálico o no. Si el valor es 1 el parámetro color Albedo controla el color del reflejo especular y la mayoría de la luz se refleja. Cuando el valor es 0 el reflejo especular tendrá el color de la luz proyectada y la superficie de este material apenas tendrá reflejo.
- **Estándar "especular"** en este caso el color especular se utiliza para controlar el color y la fuerza de las reflexiones especulares en el material. Esto nos permite tener un reflejo especular de un color diferente al reflejo difuso.

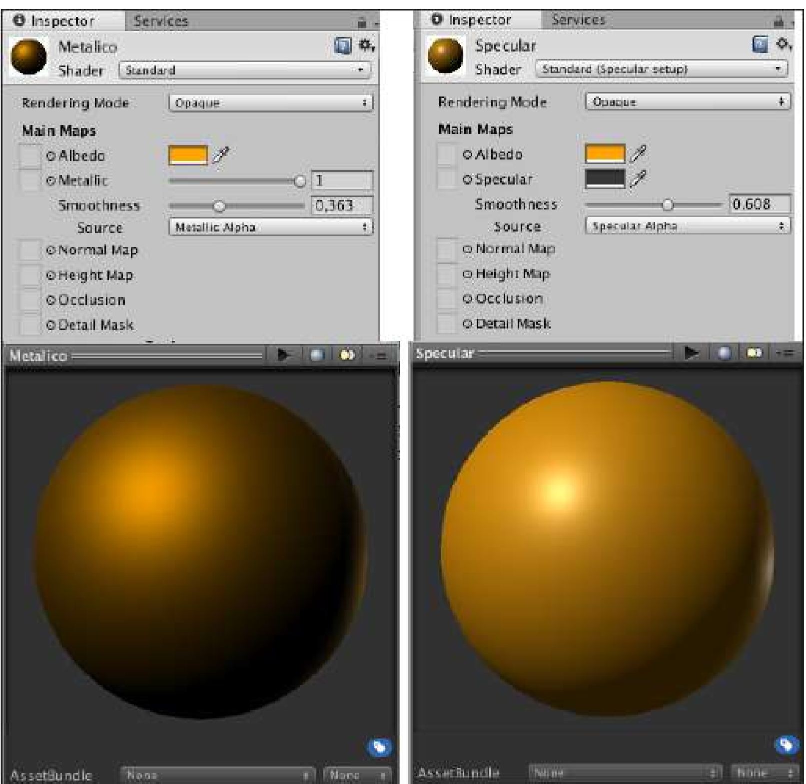


Fig. 4.15

4. Parámetros básicos de los materiales

Ahora que has trabajado un poco con los materiales te resumo sus características más destacadas, para que en un futuro cuando crees tus propios modelos puedas utilizar materiales correctamente.

Rendering Mode

En este apartado nos encontramos con un desplegable en donde podemos seleccionar entre cuatro opciones. El parámetro en sí está enfocado en la transparencia del objeto.

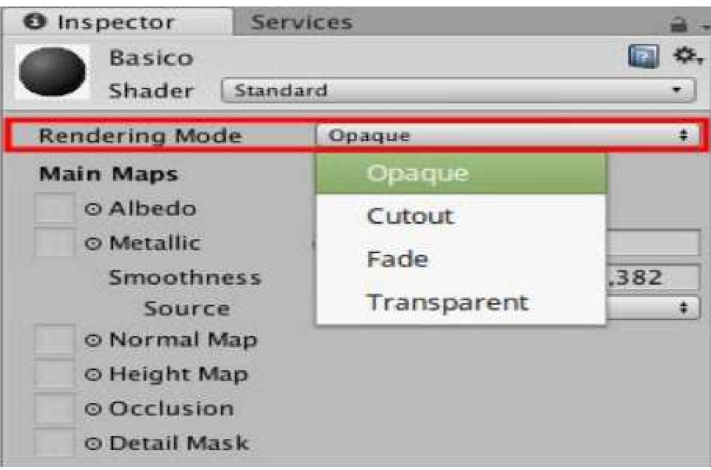


Fig. 4.16

- **Opaque (Opaco):** es el valor predeterminado y adecuado para objetos sólidos normales sin áreas transparentes.
- **Cut out (Recortar):** le permite crear un efecto transparente que tiene bordes duros entre las áreas opacas y transparentes. En este modo, no hay áreas semitransparentes, la textura es 100% opaca o invisible. Esto es útil cuando se usa transparencia para crear la forma de materiales como hojas o tela con agujeros y jirones.
- **Transparent (Transparente):** adecuado para la representación de materiales transparentes realistas, como plástico transparente o vidrio. En este modo, el material tomará valores de transparencia (basados en el canal alfa de la textura y el alfa del ***tint colour***), sin embargo, los reflejos y las iluminaciones se verán con toda claridad, como en el caso de los materiales transparentes reales.
- **Fade (Fundido):** permite que los valores de transparencia desvanezcan por completo un objeto, incluidos los reflejos o reflejos especulares que pueda tener. Este modo es útil si quieres animar un objeto que se desvanece hacia adentro o hacia afuera. No es adecuado para la representación de materiales transparentes realistas.

Main Maps (Mis mapas)

En este apartado disponemos de varios parámetros en donde utilizaremos mapas de textura según necesitemos para nuestros modelos.

Albedo

Controla el color base de la superficie.

Fig. 4.17

Si disponemos de una textura esta se representara en el objeto. En el caso anterior sucede para materiales opacos pero en el caso de que tengamos un material transparente este parámetro puede controlar la transparencia del material con una textura, los valores son blanco completamente opaco y el negro totalmente transparente.

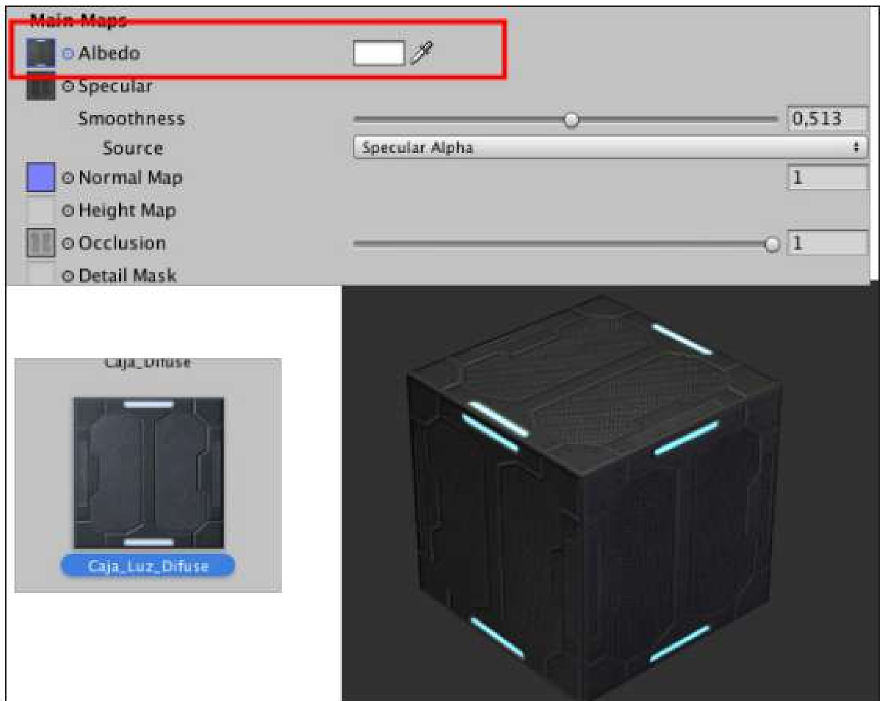


Fig. 4.18

Modo Specular parámetro Specular

Son esencialmente los reflejos directos de las fuentes de luz en su Escena, que generalmente se muestran como reflejos brillantes y brillan en la superficie de los objetos (aunque los reflejos especulares también pueden ser sutiles o difusos).

Cuando se trabaja en modo Especular, el color RGB en el parámetro Especular controla la intensidad y el matiz del color de la reflectividad especular. Esto incluye el brillo de las fuentes de luz y los reflejos del entorno.

Fig. 4.19

Cuando ponemos una textura al parámetro **Specular**, tanto el parámetro **Specular** como el control deslizante **Smoothness** desaparecen. Por otro lado, los niveles especulares para el material están controlados por los valores en los canales Rojo, Verde y Azul de la Textura y los niveles de Suavidad para el material están controlados por el canal

Alfa de la misma Textura. Esto nos unifica el trabajo porque una misma textura define como van a ser las áreas ásperas, lisas, una diferentes valores para la **especularidad** del objeto.

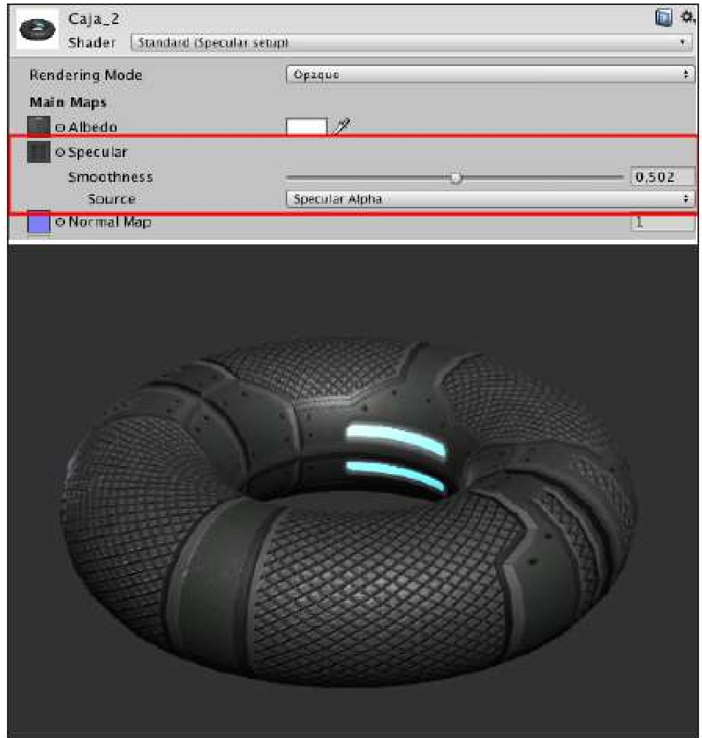


Fig. 4.20

Modo Metallic parámetro Metallic

En este modo a pesar del gran parecido al del **Specular** la reflectividad y el comportamiento de la luz en este tipo de material es variado por los parámetros Metálico (***metallic***) y el nivel de suavizado (***Smoothnes***). Estos dos parámetros bien configurados también nos proporcionan las reflexiones especulares que se siguen generando.

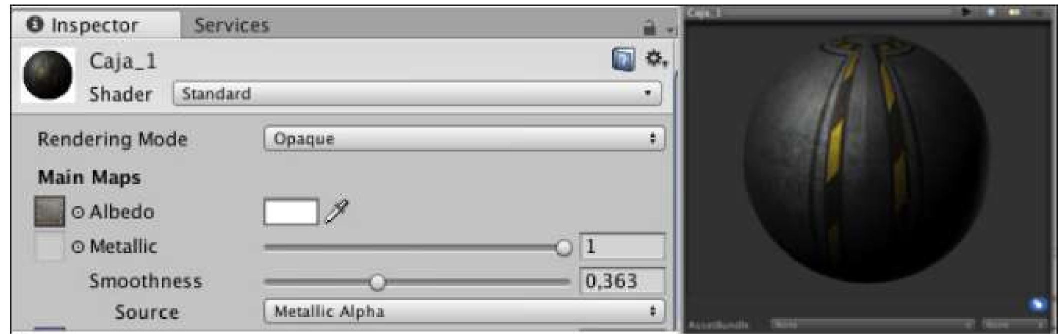


Fig. 4.21

Este Modo sera el que te ponga Unity por defecto siempre que crees un material nuevo, porque a pesar de que se llame ***Metallic*** no es solo para materiales metálicos, la razón es que este parámetro intenta determinar cuánto se parece el material que estás creando a uno metálico. Para que entendamos este concepto debemos entender que en una textura el color blanco representará el valor máximo de ***Metallic*** y el color negro es el valor mínimo de ***Metallic***. De este modo, si tenemos un material para el traje de un guerrero que tiene partes de cuero y metálicas podremos determinar qué zonas deben tener mayor reflexión.

En el caso de que no tengamos ninguna textura este parámetro lo controlaremos mediante un deslizador entre los valores 0 y 1.

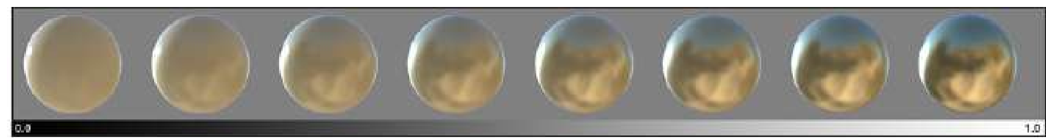


Fig. 4.22

Smoothness

Este parámetro se utiliza tanto en el modo metálico como en el modo especular proporcionando un servicio similar. En mi apreciación personal cuanto más nos acercamos al valor (1) del deslizador mayor es la concentración de la espectacularidad del material tanto en el modo metálico como el modo especular, por el contrario se disipa o suaviza cuanto más nos acercamos al valor (0).

Fig. 4.23

Normal Map

Este tipo de mapas nos permite representar relieve en nuestros modelos a pesar de que estos sean simples planos. Este tipo de texturas son muy utilizadas en los videojuegos, ya que nos permiten dar detalle a nuestros modelos. En los modelos importados de este capítulo puedes ver algunos mapas normales para los modelos que son en realidad cubos.

Los mapas normales son un tipo de mapa de relieve. Son un tipo especial de textura que le permite agregar detalles de la superficie, como protuberancias, ranuras y arañazos, a un modelo que capta la luz como si estuvieran representados por una geometría real. Este tipo de materiales se pueden crear con programas de diseño en 3D.

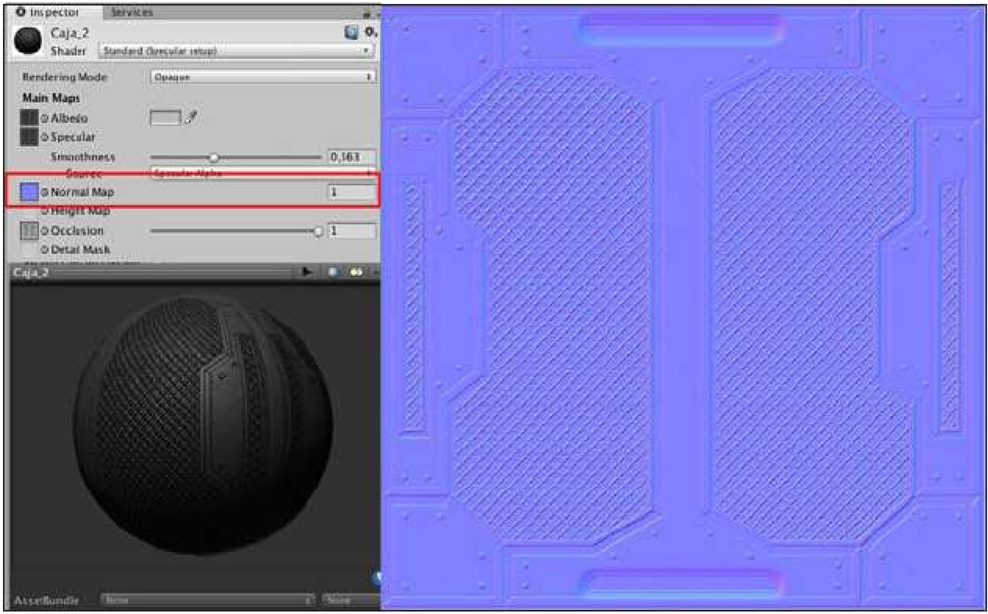


Fig. 4.24

Height Map

Este tipo de mapas o texturas tienen una función parecida a la de las normal maps pero en este caso es mucho más complejo ya que requieren más geometría para que el resultado sea aceptable. El objetivo es también crear relieve en una superficie en el ejemplo del capítulo no utilizaremos ninguna, pero este tipo de texturas se utilizan muchas veces para dar detalles adicionales a superficies con bastante geometría como los terrenos.

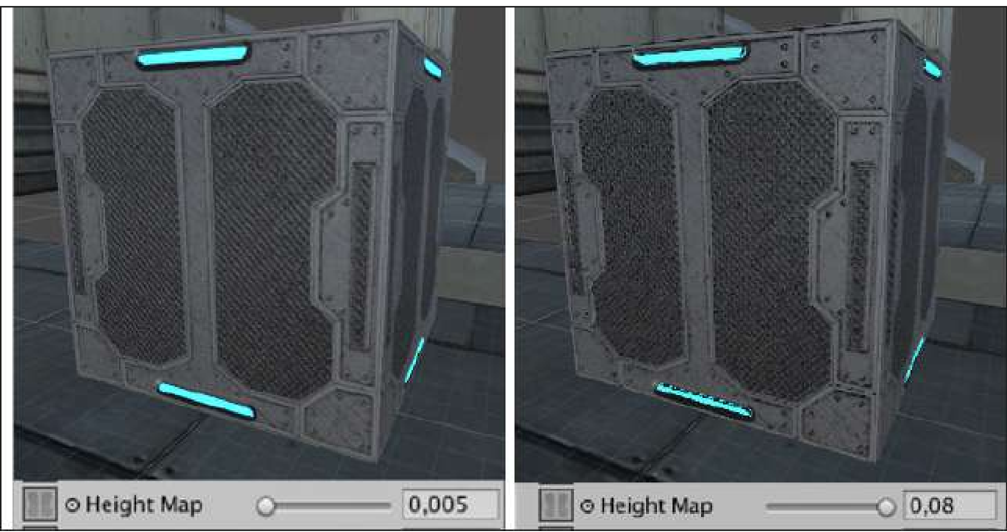


Fig. 4.25

Este tipo de mapas son en tonos de grises en donde el color blanco representa la zona más alta y el color negro la zona más baja.

Oclusion Map

Este tipo de texturas son mapas que intenta crear una iluminación ambiental y es un añadido a la textura principal. Es decir las partes cóncavas de un modelo serán oscurecidas, dando mayor sensación de profundidad. En muchos casos este parámetro combinado con el de normales ayuda a definir mejor el relieve y que se pueda apreciar mejor los detalles de un modelo.

Este tipo de texturas son imágenes en escala de grises, en donde las zonas en blanco son las zonas en donde la iluminación es directa y las zonas negras son donde no recibe iluminación. Este tipo de texturas se suelen crear con programas de creación 3d.

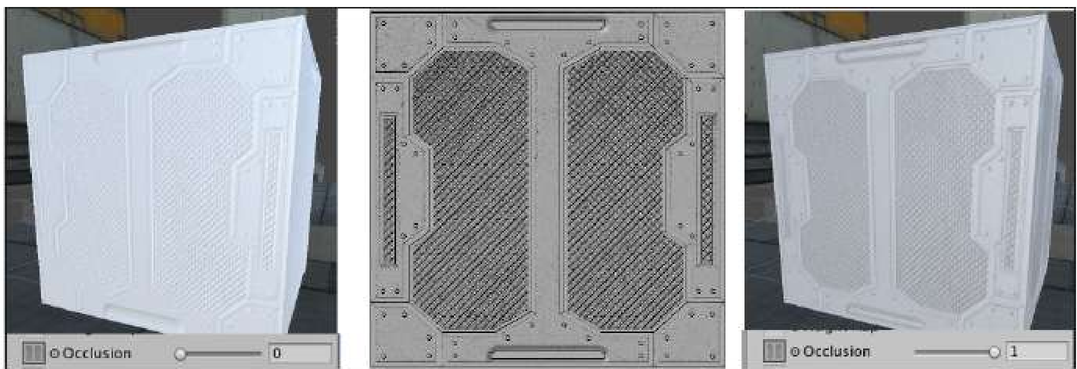


Fig. 4.26

Emission

Este parámetro cuando lo activamos es posible hacer que nuestro material emita luz y poder controlar el color y la intensidad de este. También nos permite incorporarle una textura. Un ejemplo que puedes ver en el ejemplo de este capítulo es la caja dos que dispone de una textura con fondo negro en donde solamente están en blanco las zonas que se quieren iluminar.

Luego puedes utilizar la caja selectora de color para ponerle el color que desees y en la caja de su lado derecho ponerle un valor para la intensidad.

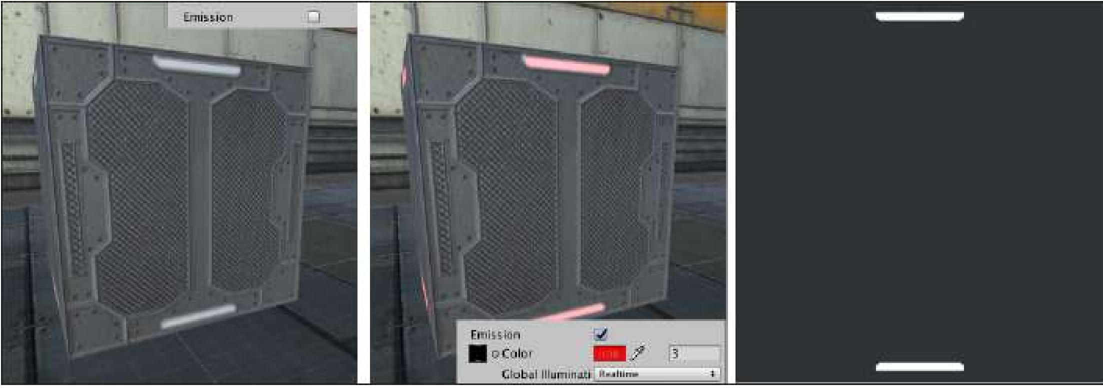


Fig. 4.27

Detail Mask

Este parámetro permite utilizar una textura que actúe en forma de máscara para esconder ciertas zonas de nuestro modelo para aplicar otro tipo de detalle. En otras palabras, sería una forma de mostrar una parte de la textura en ciertas zonas que queremos que se muestren y esconder otras zonas de la textura que no queremos que se muestren. En este capítulo no vamos a entrar en detalle con este parámetro puesto que las texturas ya están preparadas para los modelos.

Parámetros de escalado y repetición

Esta zona se compone del ***Tiling*** en donde podemos especificar cuantas veces repetimos una textura en los ejes (X) e (Y) y el ***Offset*** para desplazarla en los ejes anteriores. Estos parámetros los podemos utilizar siempre que tengamos texturas con un patrón de repetición en estos ejes, si no es el caso es mejor no utilizar estos parámetros.

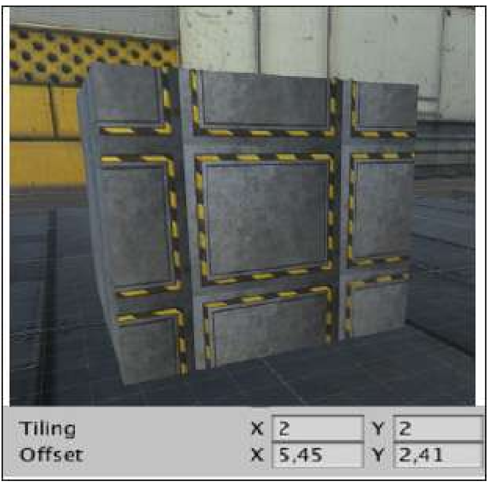


Fig. 4.28

Secondary Maps

Unity nos permite añadir un segundo conjunto de texturas sobre las que tenemos ya aplicadas. Esto es útil cuando queremos agregar algún detalle extra a nuestros modelos. Como hemos dicho en el apartado de escalado y repetición el tipo de texturas que se suele utilizar en estos dos parámetros suelen ser texturas preparadas para repetirse en gran escala.

5. Colliders y Rigid Bodies

Los **colliders** son un componente de los **Gameobject** se adaptan a la forma de nuestros modelos en cierto modo y crean colisiones. Este componente no lo tienen nuestros modelos por que han sido importados. Los **colliders** nos permiten que el objeto cree una colisión con otros objetos cuando estos trabajan con físicas.

Todos nuestros modelos deben tener un **collider**, para que más adelante nuestro player no atraviese las paredes o se caiga en el espacio infinito. Para ponerles un collider debemos seleccionar el objeto desde la ventana Jerarquía o la ventana escena, y desde la ventana inspector añadirle un collider desde **Add Component >Physics > Box Collider**. En este caso como todos los modelos son formas cuadradas utilizaremos el **Box Collider** como te muestro a continuación:

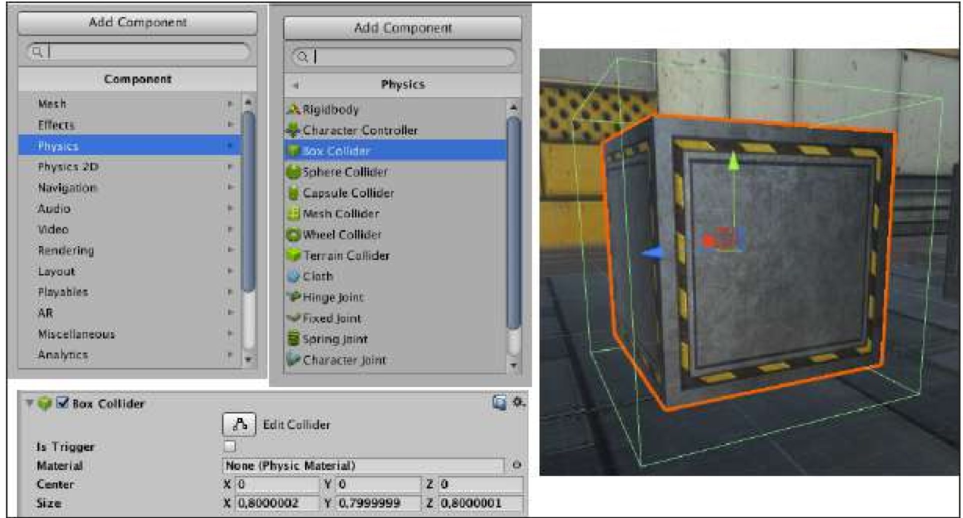


Fig. 4.29

El **Box Collider** anterior es la caja representada en color verde, en la imagen se ha exagerado el tamaño para que se diferencie bien del modelo. Normalmente cuando creas un componente collider este se adapta automáticamente al modelo. Para cambiar el tamaño puedes utilizar el botón Edit Collider y hacerlo desde la ventana escena o puedes utilizar las cajas de valores del apartado Size.

Un **Rigidbody** es otro componente de un **Gameobject** que creará un comportamiento físico en nuestros modelos. Este componente **Rigidbody** se lo vamos a aplicar a las dos cajas que tenemos en nuestra escena. Esto provocará que nuestras cajas reaccionen a la gravedad y reaccionaran a las colisiones.

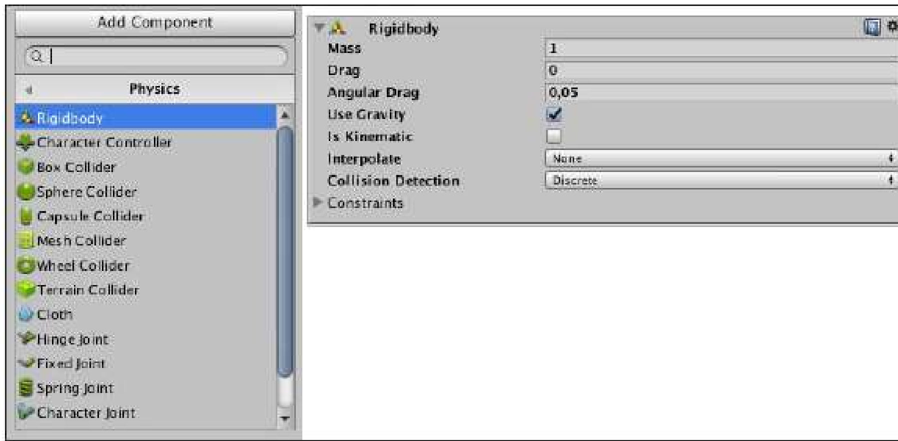


Fig. 4.30

Antes de continuar, si estas siguiendo el capítulo mientras realizas las tareas que te propongo,te habrás encontrado con un inconveniente en el modelo Rampa, puesto que tiene una inclinación y el componente collider toma todo el ancho y todo el alto del modelo. Esto es un problema muy común con los modelos de este tipo vamos a ver como resolverlo.

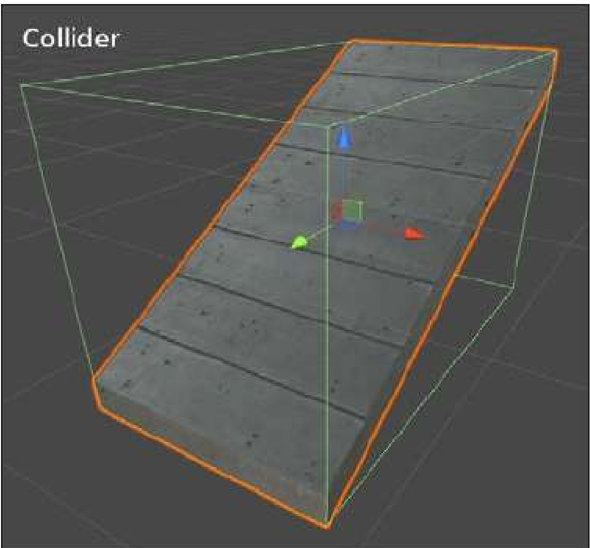


Fig. 4.31

- Primero borra el componente collider del modelo de la rampa.

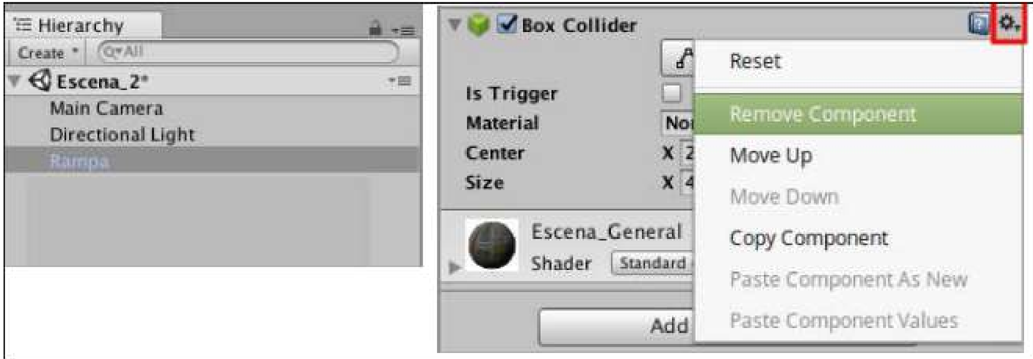


Fig. 4.32

- Dirígete a la ventana Jerarquía y en el botón **Create** seleccionamos la opción **Empty**.
- Se creará un **GameObject** vacío el cual le pondremos el nombre **Collider_Rampa**.
- A este GameObject le añadiremos un BoxCollider y lo escalaremos y lo posicionaremos de manera que se adapte lo máximo posible al modelo.
- Para finalizar arrastra el GameObject Collider_Rampa encima de Rampa para que se emparenten y de este modo ya tenemos el collider.

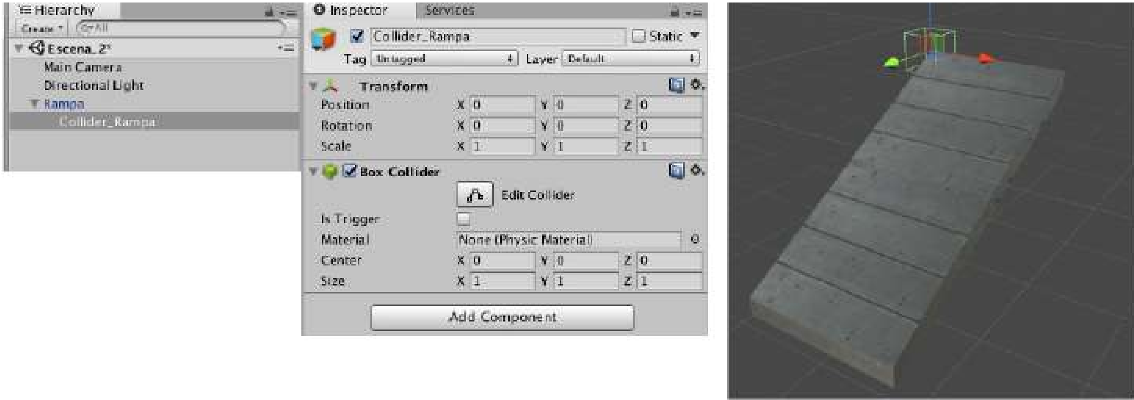


Fig. 4.33

Ahora podemos controlar el objeto Collider_Rampa que es hijo de Rampa utilizando las herramientas de escalado rotación y desplazamiento podemos darle la inclinación que deseemos. Voy a realizar capturas del proceso para que puedas ver como lo hago.

Primero debes utilizar el Gizmo de la ventana Vista y proyectar la vista que se parezca a la que te muestro a continuación:

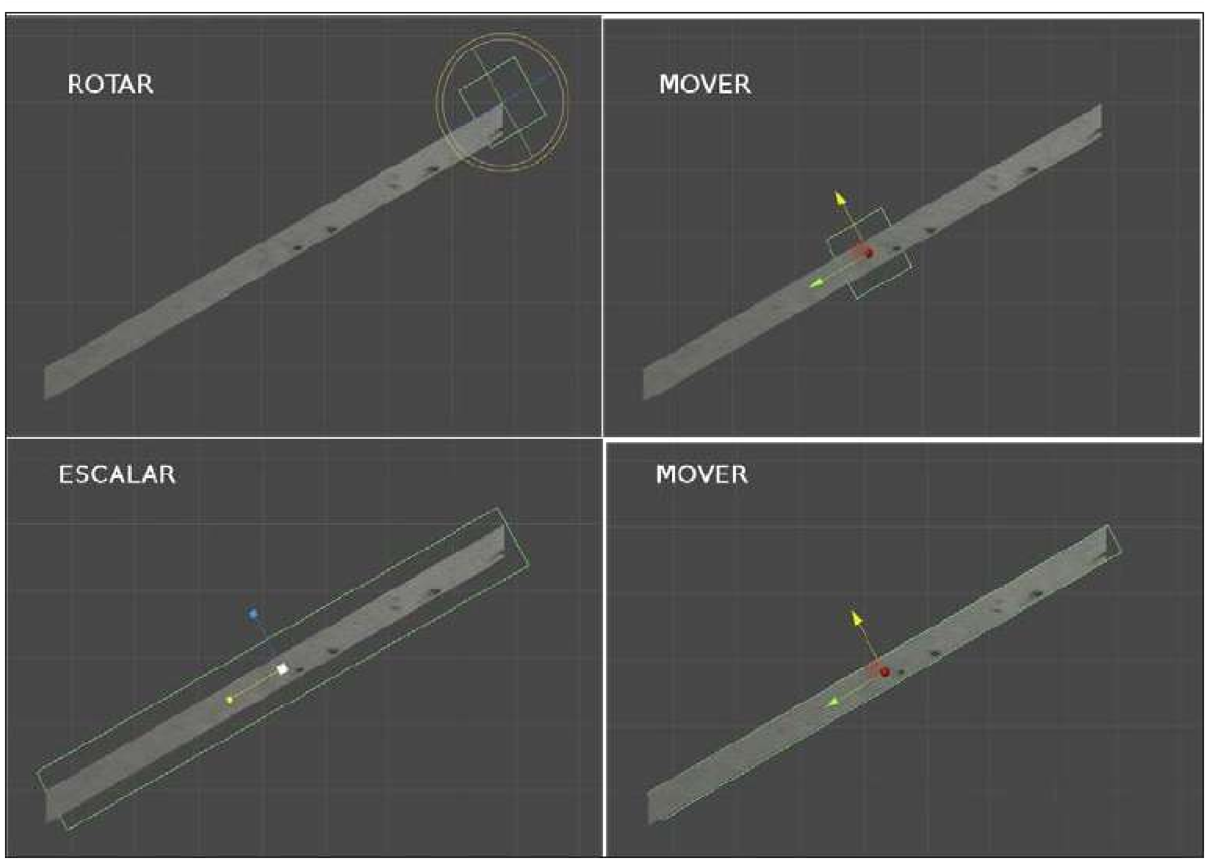


Fig. 4.34

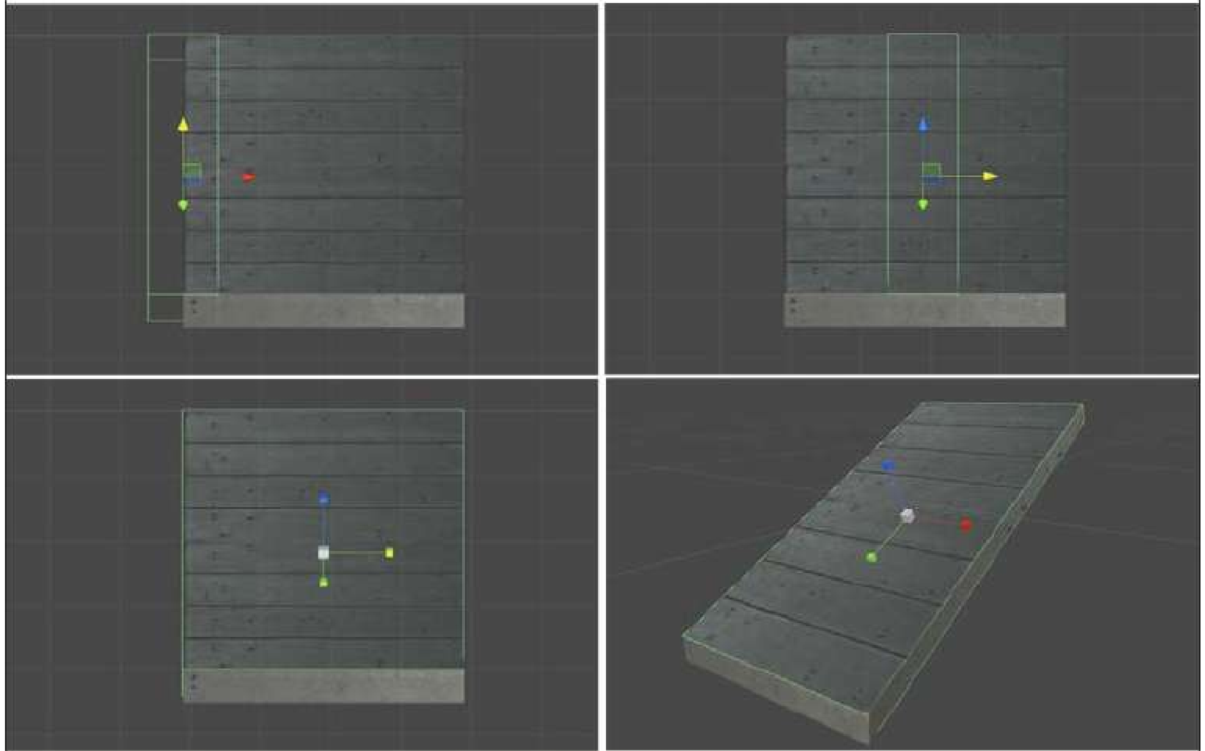


Fig. 4.35

Si todo es correcto debería quedarte la rampa como se muestra en la última imagen. A continuación veremos como preparar nuestros Prefabs para empezar a montar nuestro escenario modular.

6. Model vs Prefabs

Un archivo de modelo puede contener un objeto 3D, como un personaje, un edificio o un mueble. Este modelo es importado como activos múltiples, en donde suele contener la malla del objeto.

Cuando hablamos de **Prefabs** estamos hablando de modelos que hemos configurado en Unity y están listos para ser usados. Es decir los **Prefabs** ya tienen sus texturas puestas, sus animaciones configuradas (si tienen animación), sus colliders, rigidbody etc.

Ahora accederemos a la carpeta Prefabs en la ventana Proyectos y explicaré dos modos de crear Prefabs.

El primero y el más sencillo es configurar nuestro modelo, con sus texturas y sus colliders en la ventana escena y luego desde la ventana Jerarquía seleccionamos el nombre del objeto y lo arrastramos en el interior de la carpeta. Este tomará un color azulado y ya tendrás tu Prefab para utilizarlo siempre que lo necesites.

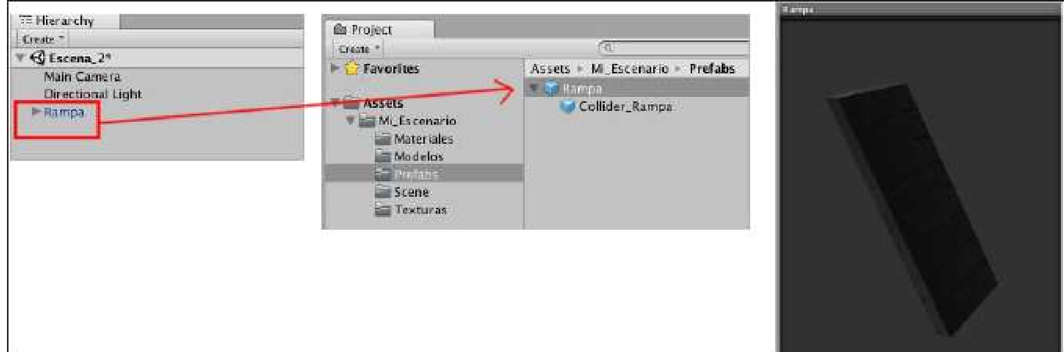


Fig. 4.36

Otra forma es crear un **Prefab** vacío dentro de la carpeta Prefabs haciendo clic con el botón derecho del ratón y acceder a la opción **Create > Prefabs** como te muestro a continuación:

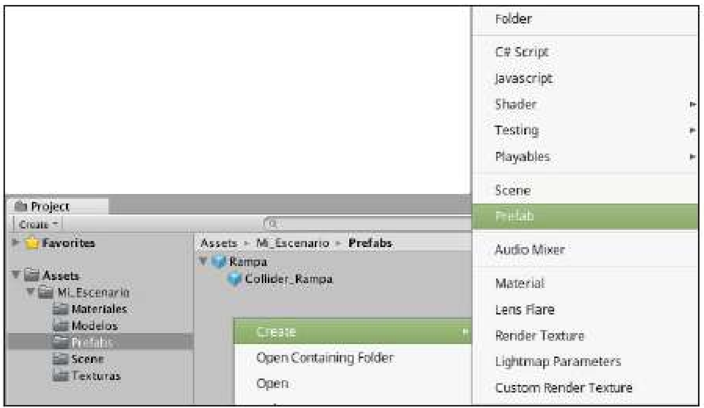


Fig. 4.37

Se creara un Prefab vacío al que le puedes poner el nombre que quieras. Esta forma es muy útil cuando quieres crear configuraciones distintas de tus modelos y guardarlos como distintos Prefabs. Por defecto el nuevo Prefab es de color blanco cuando está vacío, en el momento que nosotros arrastremos un modelo encima de este Prefab tomara el color azul y ya tendremos nuestro Prefab listo para usar.

Fig. 4.38

Ahora debes coger todos los modelos y crear un Prefab de ellos para el siguiente apartado en donde aprenderás a montar partes de un escenario modular.

Fig. 4.39

7. Montar un escenario simple

Este es el momento de colocar todos nuestros Prefabs de forma ordenada para crear una escena simple que podría ser el primer nivel de un videojuego. En la imagen siguiente tenemos un esquema de un escenario muy básico que hemos dividido en dos sectores.

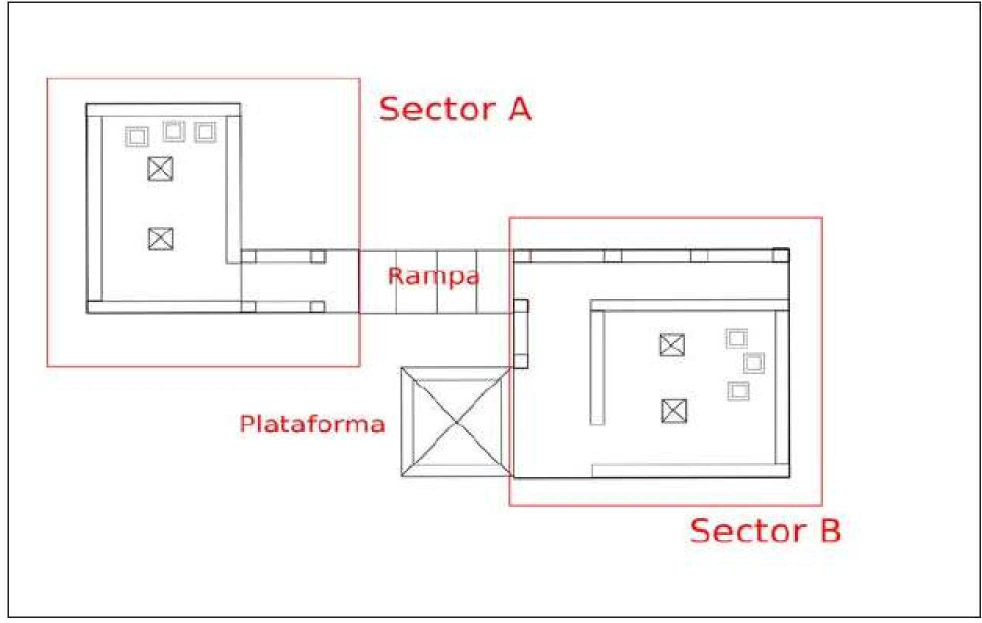


Fig. 4.40

Sector A

Para el sector A primero vamos a crear un GameObject vacío con el nombre de Sector A. Ahora seleccionaremos el Prefab Suelo_1 y lo arrastramos en la escena. En la ventana Jerarquia seleccionamos el Prefab Suelo_1 y lo duplicamos haciendo clic con el botón derecho dentro de la ventana Jerarquía y seleccionando la opción **Duplicate.**

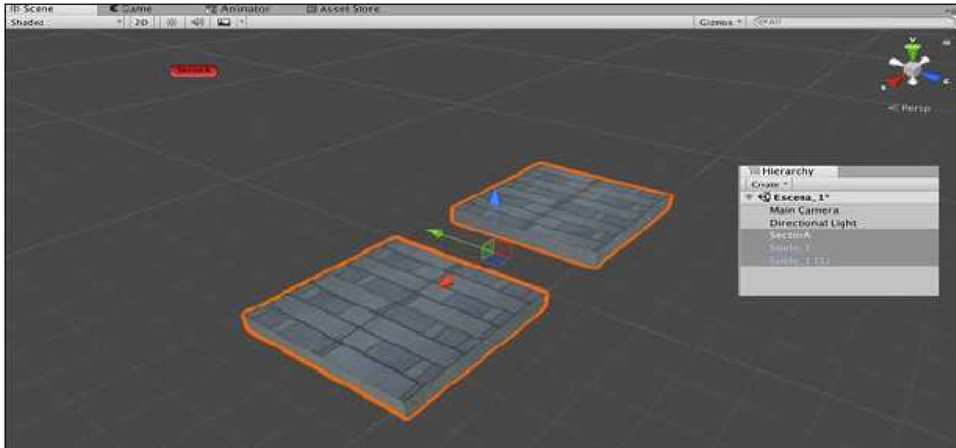


Fig. 4.41

Ahora vamos a colocar estas dos piezas una al lado de la otra bien encajadas para ello selecciona uno de los Prefabs en la escena y luego pulsa la tecla (**V**) del teclado y verás tu cursor se siente atraído por los vértices del Prefab. Con la tecla (**V**) pulsada arrastra tu cursor hasta el vértice opuesto del otro Prefab para que se pongan justo uno al lado del otro.

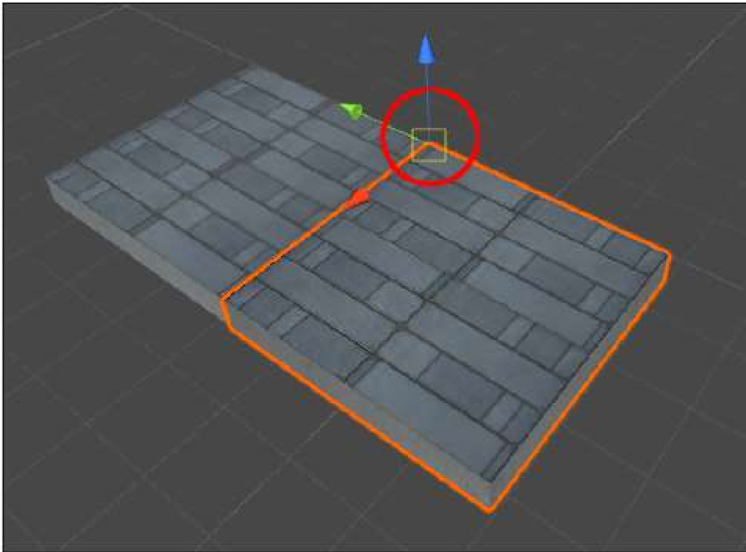


Fig. 4.42

Ahora seleccionamos estos dos Prefabs y los duplicamos a la vez, en este caso si queremos moverlos de forma que siga el espacio de la cuadrícula de la escena movemos los duplicados manteniendo pulsado la tecla_Control.

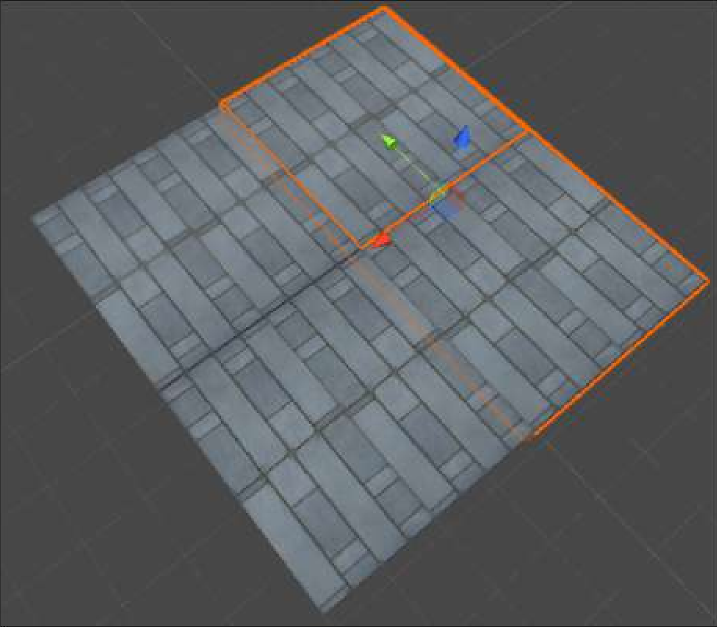


Fig. 4.43

Ahora debemos realizar esta acción varias veces hasta que tengamos el suelo de la superficie del sector A como te he mostrado en el dibujo del mapa del escenario. Una vez tengas una superficie parecida lo dejo a tu elección. Dentro de la ventana Jerarquía vamos a seleccionar todos los Prefabs del suelo y los arrastraremos dentro del GameObject vacío Sector_A. De este modo tendremos ordenados todos los Prefabs. Esto estaría bien para cuando tenemos solo un Prefab en un sector, pero a continuación debemos añadir Paredes y otros elementos, por ese mismo motivo podemos crear varios gameObjects vacíos que sean padres de los distintos elementos como te muestro a continuación:

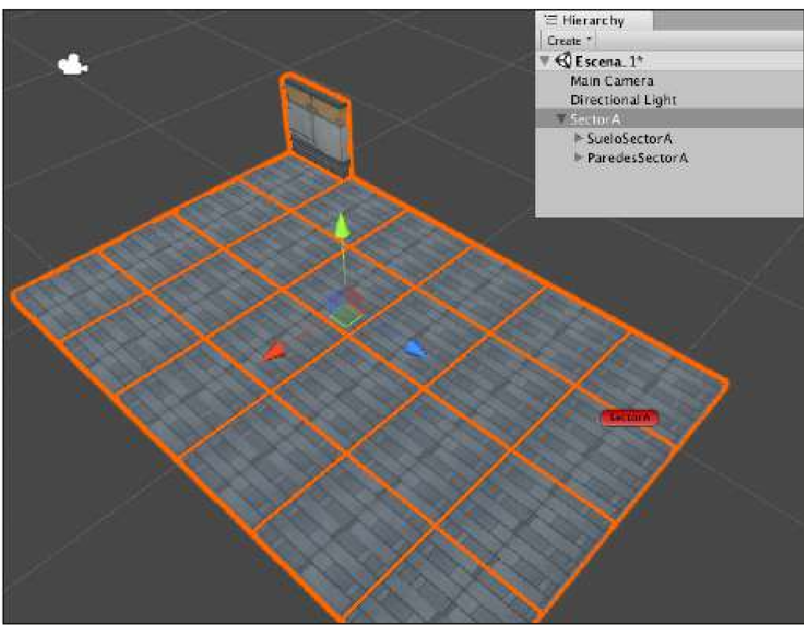


Fig. 4.44

En la imagen anterior verás que en la ventana Jerarquía tenemos El Sector A que es padre de todos los objetos que hay en el escenario excepto la cámara y la luz. Después tenemos un objeto vacío con el nombre Suelo Sector A que contiene todos los Prefabs del suelo, y he añadido otro objeto vacío con el nombre Paredes Sector A que va a contener todas las paredes. Esta es la dinámica que vamos a seguir.

El siguiente paso es que vayas arrastrando en escena los Prefabs por tipo y vayas construyendo el sector A de una forma organizada. Los Prefabs puedes rotarlos y moverlos pero no escalarlos pues estos modelos se crearon de una forma muy simple para que encajaran entre ellos.

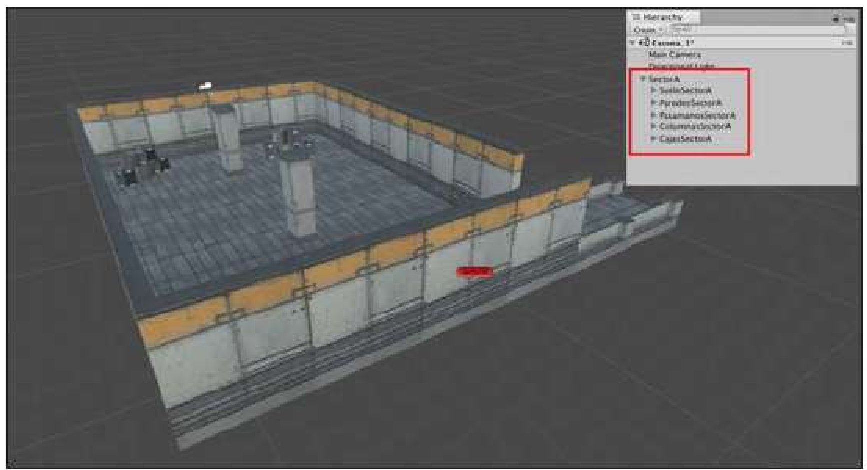


Fig. 4.45

El sector está casi finalizado, como puedes ver tenemos ordenado en la ventana Jerarquía el sector A con todos los elementos que la componen, pero en los Prefabs que he utilizado solo tienen un box collider las columnas las cajas y los pasamanos las paredes y los suelos no llevan porque quería mostrarte otra forma de crear colliders en un escenario y para ser más concretos en un sector específico.

Vamos a crear otro gameObject vacío con nombre CollidersSectorA. Este gameObject que acabamos de crear no tiene de momento ningún parentesco con ningún objeto de la escena. A este objeto CollidersSectorA vamos a agregarle varios componentes de tipo Box Collider que re-situaremos en este sector.

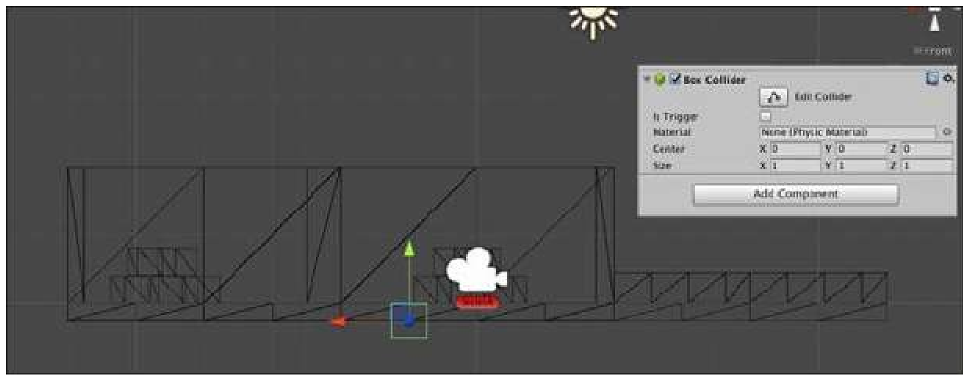


Fig. 4.46

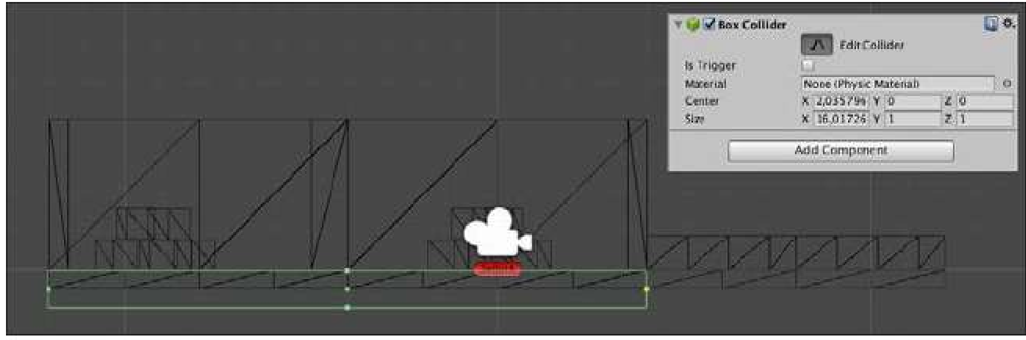


Fig. 4.47

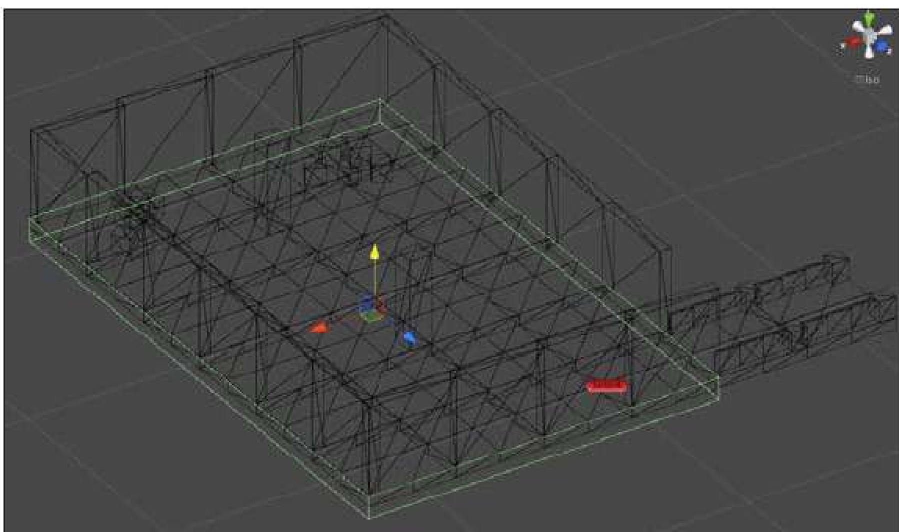


Fig. 4.48

Una vez que tengas el primer boxcollider colocado añadiremos otro componente del mismo tipo a nuestro objeto CollidersSectorA.

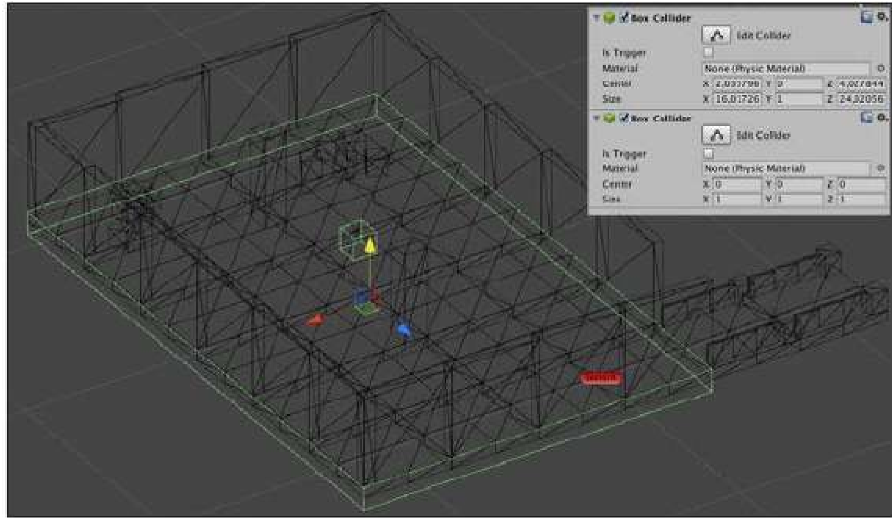


Fig. 4.49

Este nuevo collider lo debemos re-situar utilizando la opción Edit Collider del componente o introduciendo valores en las opciones Center o Size.

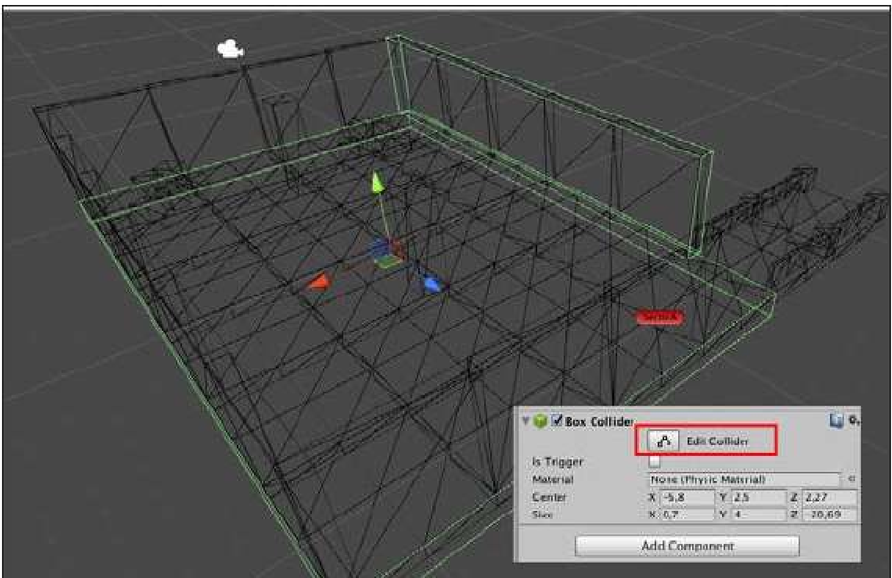


Fig. 4.50

Es una forma de poner colisionadores a nuestra escena, en vez de utilizar un colisionador para cada Prefab, utilizamos 5 colisionadores que abarcan las necesidades de las paredes y suelos. El sector debería verse de la siguiente manera:

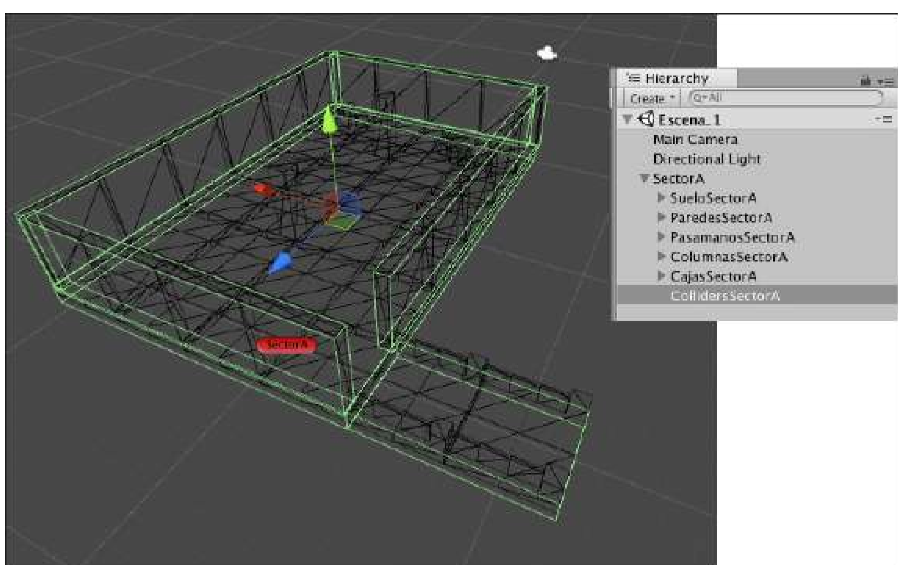


Fig. 4.51

Ahora que ya tenemos el sector A terminado hacemos exactamente el mismo procedimiento para crear el sector B. Los elementos Rampa y Plataforma son elementos que no pertenecerán a ningún sector puesto que son conectores de estos. A continuación te muestro como debería quedar la escena según nuestro esquema.

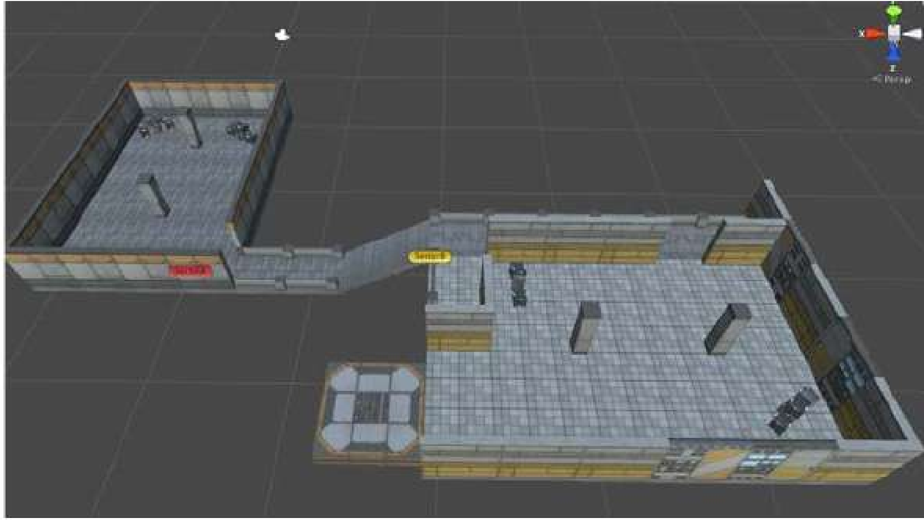


Fig. 4.52

8. Importar Standard Assets y probar el escenario

En el proyecto de este capítulo podrás importar un paquete con Assets en el que encontraremos un Player en primera persona que utilizaremos para poder probar el escenario dentro del juego.

Una vez que cargues el paquete de **Standard Assets** accede a la carpeta **Characters > FirstPersonCharacter > Prefabs > FPSController**, arrastramos este **FPSController** dentro de la escena y lo posicionamos en el **SectorA** como te muestro a continuación:

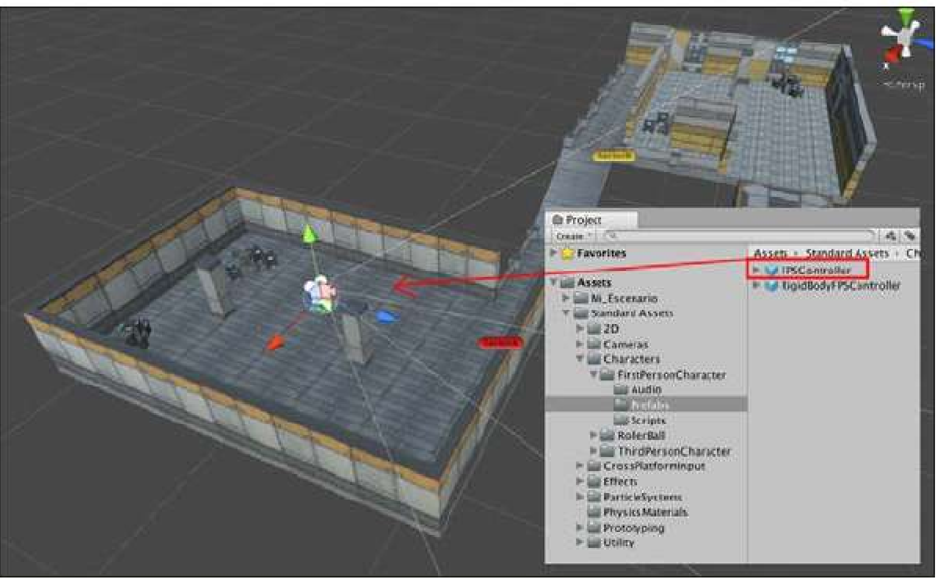


Fig. 4.53

Ahora desactiva o borra la cámara que viene por defecto en la escena como te muestro a continuación.

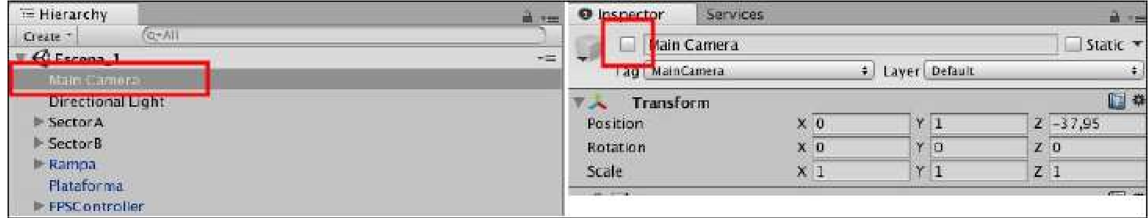


Fig. 4.54

Bien ya tenemos el escenario y un FPS (jugador en primera persona), ahora es el momento de disfrutar, ejecuta el juego y explora el escenario que todo esté correcto.

Fig. 4.55

Fig. 4.56

Capítulo 5

Introducción básica de C# con Unity

- Introducción
- Crear y manipular variables
- Trabajar con operadores aritméticos
- Operadores lógicos y de comparación
- Crear declaraciones lógicas con if - else
- Crear declaraciones con switch
- Trabajar con loops
- Crear y llamar funciones
- Entender qué son los Arrays
- Mi primera Clase

1. Introducción

Este es el primer capítulo en donde vas a empezar a escribir código y tus primeros scripts. Este capítulo se enfoca en darte una base introductoria para que te resulte más sencillo entender los próximos capítulos en los que se empieza a programar el movimiento de nuestro personaje y en donde empezamos a utilizar la API de Unity.

Creando nuestro primer Script

Crea un proyecto nuevo en 3D con el nombre que quieras, por ejemplo capítulo 5. En la ventana **Project** vamos a crear dos carpetas una para guardar nuestras escenas y la otra para guardar nuestros scripts como te muestro en la imagen siguiente.

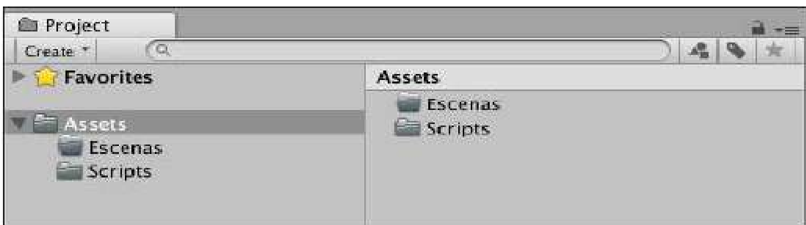


Fig. 5.1

Ahora vamos a crear un script dentro de la carpeta Scripts que hemos creado anteriormente. Seleccionamos la carpeta Scripts y hacemos clic con el botón derecho del ratón y seleccionamos **Create > C# Script**, automáticamente aparecerá un script con el nombre por defecto **NewBehaviourScript.cs**.

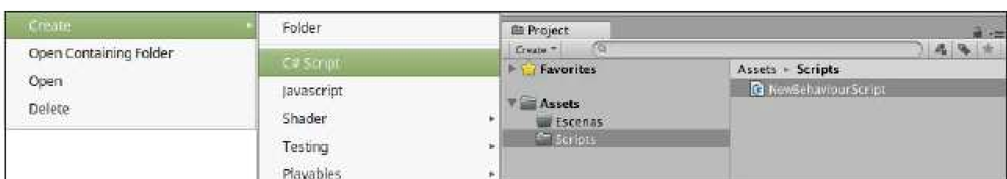


Fig. 5.2

Si quieres puedes cambiarle el nombre al script. A continuación hacemos doble clic encima del script y se nos abrirá el programa **Monodevelop** que nos servirá para editar nuestro script. En esta obra se ha transcrito todos los scripts para mayor entendimiento del lector y se mostrarán siempre en este recuadro que te muestro a continuación.

Script: NewBehaviourScript.cs

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

```
public class NewBehaviourScript : MonoBehaviour
{
    // Use this for initialization
    void Start () {

    }

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

    }
}
```

El primer bloque que encontramos en la parte de arriba:

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

Son necesarias porque es la forma que tenemos en C# de llamar directamente al Engine de Unity y para que se utilicen las colecciones genéricas para empezar a programar. Normalmente cuando creas un script en Unity ya se crean automáticamente, pero en algunos casos deberemos ser nosotros mismos los que agreguemos alguna otra colección.

```
public class NewBehaviourScript : MonoBehaviour
```

Este código que vemos es el nombre de la clase que siempre debe ser el mismo que el nombre del script y después de los dos puntos nos indica a que clase pertenece y en este caso y siempre por defecto pertenecerá a MonoBehaviour. Esta clase debe contener en su interior todo el contenido de nuestro script o dicho de otra manera el comportamiento que queremos que se ejecute.

```
// Use this for initialization
void Start () {
}

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

En este bloque o parte del script es donde declararemos variables y utilizaremos nuestras funciones. En el script tenemos dos comentarios que sirven para informar y siempre van después de // y dos funciones (void Start) y (void Update) que veremos a continuación para que sirven.

Void Start

Para entender cómo funciona vamos a crear un mensaje para cada función que se va a mostrar por la ventana de Consola en Unity. Cuando queremos imprimir un mensaje debemos escribir la palabra Debug.Log(“”); Dentro del paréntesis de momento vamos a escribir entre comillas cuándo queremos poner texto, más adelante veremos qué opciones tenemos. A continuación te dejo el código que me gustaría que probarás.

Script: NewBehaviourScript.cs

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

public class NewBehaviourScript : MonoBehaviour
{
   void Start ()
   {
      Debug.Log(“Nuestro primer mensaje”);
   }
   void Update ()
   {
      Debug.Log(“Este mensaje se va a repetir infinitas veces!!”);
   }
}
```

Una vez hayas escrito el código en tu script debes guardar el archivo seleccionando en la barra de herramientas principal **File > Save**. Ahora minimizamos el editor Monodevelop y nos dirigimos a Unity. Debes localizar la ventana Console en la interfaz si no lo ves puedes abrir esta ventana accediendo al menú principal en la opción **Window > Console**. Se abrirá la ventana Consola como te muestro en la siguiente imagen.

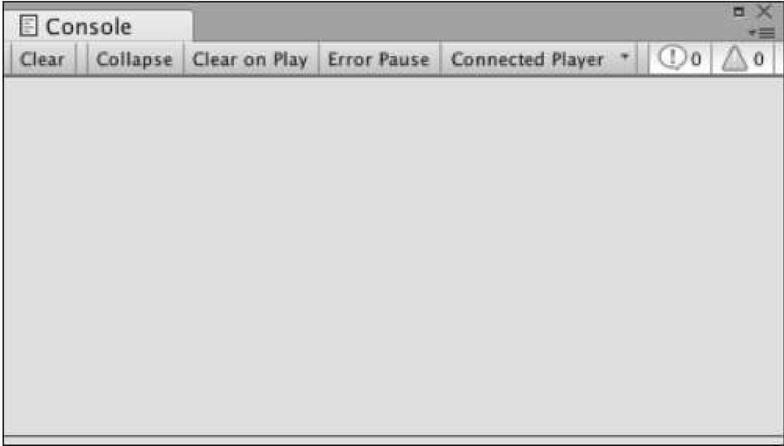


Fig. 5.3

Ahora para que nuestro script funcione debemos agregarlo a un objeto en la escena, como no tenemos ninguno debemos crear un **GameObject** vacío, o por defecto el que tú quieras y arrastrar el script encima del objeto dentro de la ventana jerarquía o con el objeto seleccionado arrastrando el script encima de sus componentes como te muestro a continuación en la siguiente imagen:

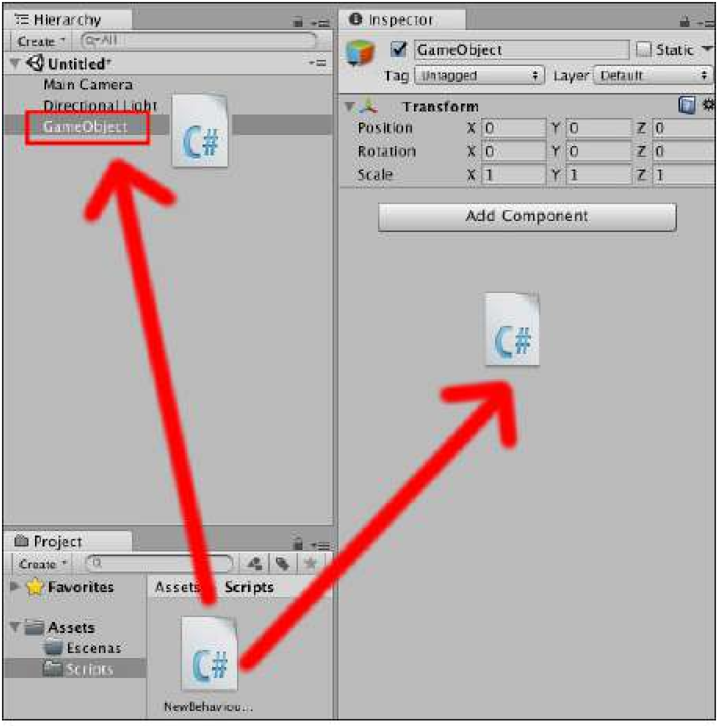


Fig. 5.4

Bien agregado el script ejecutamos el juego haciendo clic en play y pasados unos segundo veremos el siguiente mensaje en consola:

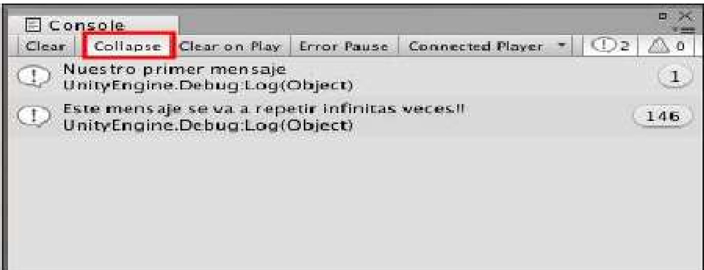


Fig. 5.5

El primer mensaje que se imprime en la consola es el que estaba dentro de la función **Start** si te fijas verás que solo aparece una vez. El segundo mensaje se repite en este caso 146 veces. Verás que he marcado la opción **Collapse** para que el segundo mensaje que estaba dentro de la función **Update** no me salga repetido y solamente se muestre una vez con el numero de repeticiones al lado.

2. Crear y manipular variables

Las variables son contenedores de información, por ejemplo imagínate que tienes tres amigos: Juan, Antonio y María. Ahora imaginate que utilizamos un nombre en clave para referirnos a estos tres amigos. El nombre clave es "jam" que son las letras del principio del nombre. Bien cada vez que digas **jam** estarás refiriéndote a tus 3 amigos, eso mismo es lo que hacemos en programación, damos un nombre fácil de recordar a un valor complejo.

Cuando creamos variables decimos que las estamos declarando y existen varios tipos de variables según el contenido que queremos guardar. Para declarar una variable en C# primero debemos decir si es pública o privada;

Las variables publicas nos permiten comunicar entre los scripts del mismo **GameObject** y otros **GameObjects** y se verán publicadas en la ventana Inspector de Unity.

Las variables privadas también nos permiten comunicar entre los scripts del mismo u otros **GameObjects** pero no se verán publicadas en la ventana Inspector de Unity.

Public/private+ tipo de variable + nombre de la variable;

Para el ejemplo en Unity crea un nuevo GameObject (un cubo) luego crear un nuevo script y llámalo Variables.cs y añadelo al nuevo GameObject:

Script: Variables.cs

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

public class Variables : MonoBehaviour
{
   public int edad = 56;
   public string nombre = “Oscar”;
   public float peso = 1,85f;

   void Start ()
   {
     Debug.Log(“Mi nombre es: “+this.nombre+” tengo “+this.edad+” y mido
“+this.peso);
   }
}
```

Las variables declaradas son:

- int para valores enteros.
- string para valores de cadena de textos siempre entre comillas.
- float para valores decimales y el numero debe finalizar con la letra f.

Para mostrar en consola las variables se utiliza un Debug.Log y dentro del paréntesis se concadena con signos de suma con el nombre de las variables. El término this se utiliza para hacer referencia a la variables del objetos que tiene el script.

Unity nos permite ver las variables que hemos declarado en la ventana inspector porque las hemos declarado públicas, como te muestro en la imagen siguiente:

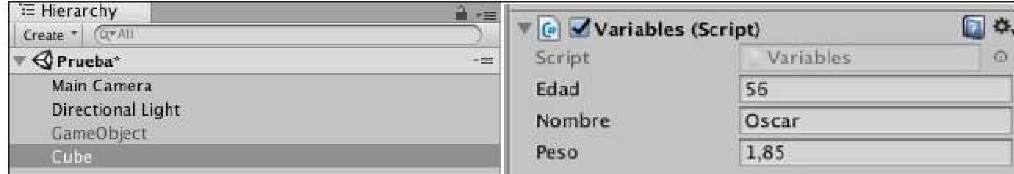


Fig. 5.6

En la imagen anterior, seleccionando el objeto al que le hemos añadido el script nos muestra las variables que hemos declarado públicas en la ventana inspector. Si ejecutamos el juego haciendo clic a play en la consola debería aparecer el mensaje que hemos escrito dentro de la función Start.

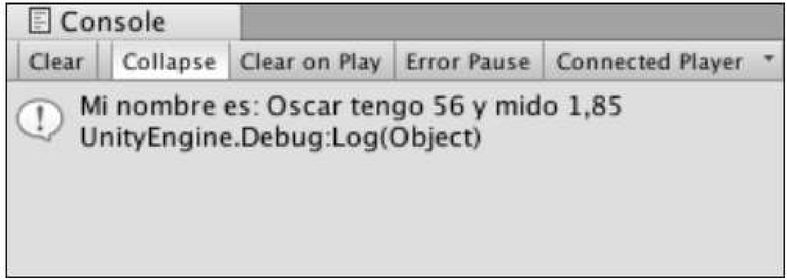


Fig. 5.7

3. Trabajar con operadores aritméticos

Los operadores aritméticos nos permiten realizar operaciones con nuestras variables es decir si tenemos una variable energía = 100 y un enemigo te dispara con un arma de daño= 50 tu energía se le restará 50.

A continuación te muestro todos los operadores que necesitas para hacer tus operaciones. Primero cómo asignamos una operación a nuestras variables:

Suma	x = x + y;	Módulo	x = x % y;
Resta	x = x - y;	Añadir 1	++x
Multiplicación	x = x * y;	Quitar 1	--1
División	x = x / y;		

Vamos a crear un ejemplo para ver como podemos utilizar estos operadores para ello crearemos un nuevo script en C# llamado Aritmeticos.cs (El nombre procura que sea sin acentos y sin la ñ).

Debemos calcular la velocidad que tiene nuestro objeto sabiendo la distancia recorrida y el tiempo que hemos utilizado.

Script: Aritmeticos.cs

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

public class Aritmeticos : MonoBehaviour
{
   private float velocidad;
   public float tiempo = 5,1f;
   public float distancia = 8.1f;

   void Update ()
   {
     this.velocidad= this.distancia/this.tiempo;
     Debug.Log("Mi velocidad es de: "+this.velocidad);
   }
}
```

Las variables declaradas son todas float y públicas excepto la de velocidad porque quiero que solo se muestre en consola. Le hemos dado valores a la variable tiempo y a la variable distancia.
He creado la operación dentro de la función Update para que cuando ejecutemos el juego y cambiemos los valores de tiempo y de distancia, Unity calcule de nuevo los nuevos valores.
Para mostrar en consola el resultado de velocidad utilizamos un Debug.Log.

En Unity podemos borrar el componente script anterior y añadir este nuevo al cubo. El resultado debería verse de la siguiente manera.

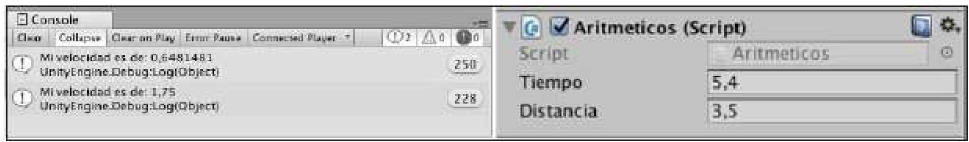


Fig. 5.8

4. Operadores lógicos y de comparación

Los operadores lógicos sirven para obtener dos resultados verdadero o falso estos resultados se les conoce también como valores Booleanos. A continuación te muestro las operaciones:

&&	And (y)	La operación seria verdadero y falso = falso
||	Or (o)	La operación seria verdadero o falso = verdadero
!	Not (no)	La operación seria verdadero no = falso

No tienes que memorizarlo simplemente debes saber que son y para que sirven. A continuación te muestro los operadores de comparación que nos permiten comparar valores en general.

==	igualdad	5==5
!=	desigualdad	4!=0
>	Mayor que	10>2
<	Menor que	2<10
>=	Mayor o igual que	X >= 10 // La x puede ser 10 o mayor que 10
<=	Menor o igual que	X <= 10 // La x puede ser 10 o menor que 10

Este apartado es una especie de esquema en donde te muestro los operadores lógicos y de comparación en el siguiente apartado vamos a ponerlos en practica.

5. Crear declaraciones lógicas con if - else

En la vida diaria tomamos decisiones a cada momento un café (solo o con leche, con azúcar o sacarina etc). Según nuestra toma de decisiones suceden acontecimientos, es decir, imagínate tu personaje en una gruta con dos posibles salidas; la salida de la izquierda o la salida de la derecha.

Resulta que si escoges la de la derecha encuentras el tesoro y si escoges la salida izquierda se te come un dragón. De eso se encarga la declaración if, de comparar situaciones o valores y darnos un resultado según nuestra decisión.

Para declarar un if debemos escribirlo de la siguiente forma:

```
if (x>0)
{
   Lo que sucede si se cumple la comparación del if ;
}
else
{
   Si no se cumple el if pasara esto ;
}
```

Vamos a crear un script llamado **Comparaciones.cs**. En este ejemplo vamos a comparar las mejores puntuaciones de un juego. Tenemos 1 jugadores con 1 puntuación que podremos variar. Si el resultado de estas puntuaciones es >=100 el resultado es Experto si el resultado es >= 50 y <100 el resultado es intermedio y si no se cumple lo anterior es principiante.

Script: Comparaciones.cs

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

public class Comparaciones : MonoBehaviour
{
   private int puntuacion = 35;// Puedes poner el valor que quieras
   void Update ()
   {
      if (puntuacion >= 100)
      {
        Debug.Log (“Experto”);
      } else if (puntuacion < 100 && puntuacion >= 50)
      {
        Debug.Log (“Intermedio”);
      }
      else
      {
        Debug.Log (“Principiante”);
      }
   }
}
```

El siguiente script cuando lo ejecutes en Unity podrás variar la puntuación y en la consola te irán apareciendo los distintos resultados posibles.

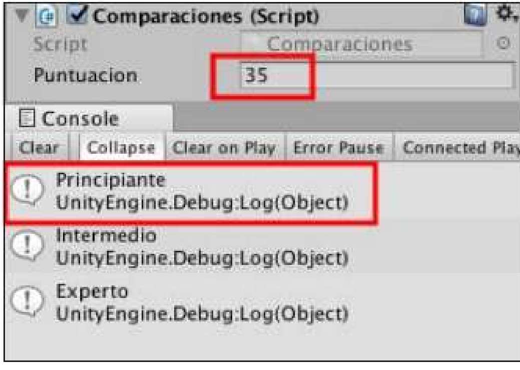


Fig. 5.9

6. Crear declaraciones con switch

El switch nos permite tomar de una sola variable su valor y realizar múltiples operaciones dependiendo del valor de la variable.

Para declarar un switch debemos hacerlo de la siguiente manera:

```
Switch (variable)
{
   case "valor1":
   Debug.Log("Sucede algo");
   break;

   case "valor2":
   Debug.Log("Sucede algo");
   break;

   default:
   break;
}
```

Vamos a realizar un ejercicio muy básico para mostrar las posibilidades de switch. Imagínate que tenemos dos valores para una variable el valor 1 y el valor 2 y queremos que cuando tengamos el valor 1 nuestro personaje tenga una pistola y cuando la variable tenga el valor 2 tenga una escopeta. Para realizar el ejercicio crea un script con el nombre Switch.cs y lo añadimos al cubo, asegúrate de borrar el otro script de la ventana inspector para que solo tenga este script.

Script: Switch.cs

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

public class Switch : MonoBehaviour
{
        public int arma;

     void Start ()
    {
      switch (arma)
     {
        case 1:
        Debug.Log("Pistola");
        break;
```

```
        case 2:
        Debug.Log("Escopeta");
        break;

        default:
        Debug.Log("Nada");
        break;
       }
      }
}
```

En la ventana inspector puedes poner el valor uno y te en la consola te mostrará una pistola y si pones el valor 2 te mostrará escopeta, si pones un valor distinto te mostrará el mensaje por defecto que es nada.

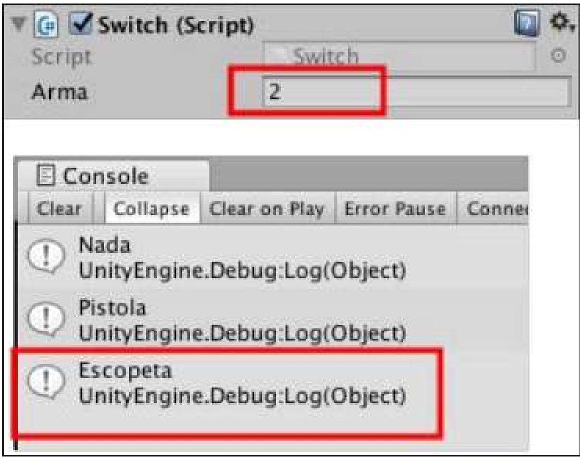


Fig. 5.10

7. Trabajar con loops

Un loop es un bucle una repetición de una condición o serie de condiciones un cierto número de veces o hasta que se cumpla una alguna de estas condiciones. Tenemos que tener cuidado con los loops porque si nos equivocamos podemos crear un loop infinito y bloquear Unity.

While

El primero de los loops que vamos a ver es While que nos permite crear un bucle hasta que se cumple la condición.

```
Variable = 0;

while (variable <= 10)
{
   Debug.Log(variable);
   variable++;
}
```

Primero declaramos una variable con un valor en este caso 0.
Le decimos a while que la condición es que mientras el valor de la variable sea menor o igual a 10 nos muestre por consola el valor de variable pero para que no sea un bucle infinito el valor de la variable tiene que incrementarse 1 cada vez por eso se le añade variable++

#Ejemplo1 para Unity

Crea un script con el nombre **While.cs**. Vamos a crear un script que nos muestre las tablas de multiplicar.

Script: While.cs

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

public class While : MonoBehaviour
{
   private int valor = 0;
   public int tabla;
   private int resultado;
   void Start ()
   {
      while (valor <= 10)
      {
         resultado = valor * tabla;
         Debug.Log (valor+” x “+tabla+” = “+resultado);
         ++valor;
      }

   }
}
```

Ahora si vamos a Unity debemos poner el valor de la tabla que queremos que nos muestre por consola por ejemplo la tabla del 4 y al ejecutar el juego en la consola se nos mostrará la tabla del cuatro.

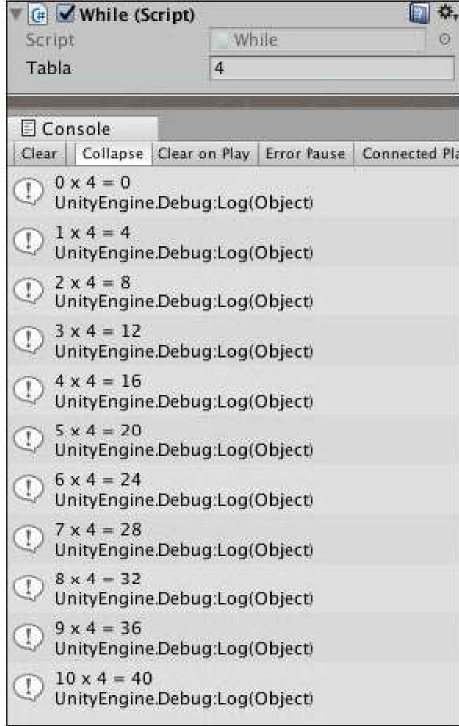


Fig. 5.11

For

El loop for nos permite definir su valor, su limite y su incremento. Es otro loop que podemos utilizar, para escribirlo debemos hacerlo de esta manera:

```
for(x=0; x<= 10; x++)
{
        Debug.Log (x);
}
```

El for contiene una variable x que es igual a 0 en un principio, después tiene la condición de que x tiene que ser menor o igual a 0 y para finalizar este valor se ira incrementando. Cada vez que se cumpla se mostrara en consola el valor de x.
Esto quiere decir que se repetirá 10 veces y que por consola se nos va a mostrar los valores 0,1,2,3,4,5,6,7,8,9,10. El valor 10 es el ultimo ya que el valor 11 sería mayor que 10 y no se cumpliría la condición de x<=10.

Ahora que sabemos como utilizar el loop for vamos a realizar el mismo ejemplo que con el loop **While** de manera que crearemos un script que nos muestre las tablas de multiplicar según el valor de la tabla que queramos.

Script: For.cs

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

public class For : MonoBehaviour
{
   public int tabla;
   private int resultado;
   private int valor;
   void Start ()
   {
      for(valor=0; valor<= 10; valor++)
      {
         resultado = valor * tabla;
         Debug.Log (valor+" x "+tabla+" = "+resultado);
      }
   }
}
```

El resultado debería ser exactamente igual al del ejemplo While descrito anteriormente.

8. Crear y llamar funciones

Ahora que sabemos qué son las variables, cómo podemos operar con ellas tomar decisiones y crear repeticiones según nuestras decisiones, vamos a comprimirlo todo en una función que nos permite contener múltiples acciones dentro, y para ejecutarla solamente necesitamos el nombre de la función.

Para crear una función debemos hacerlo de la siguiente manera:

```
void nombredelaFuncion()
{
   //Aqui ponemos la acción que queremos que ejecute.
}
```

Voy a seguir con el ejemplo de las tablas de multiplicar para que se entienda cómo podemos crear una función que me muestre la tabla de multiplicar que nosotros deseemos. Crea un nuevo script y llámalo **Funciones.cs** y añádelo al objeto que tengas en escena.

Script: Funciones.cs

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

public class Funciones : MonoBehaviour
{
   public int tabla;
   private int resultado;
   private int valor;

   void CreadorTablas ()
   {
      for(valor=0; valor<= 10; valor++)
      {
         resultado = valor * tabla;
         Debug.Log (valor+" x "+tabla+" = "+resultado);
      }
   }

   void Start()
   {
      this.CreadorTablas();
   }
}
```

Después de declarar las variables hemos creado una función con el nombre de CreadorTablas, puedes ponerle el nombre que quieras. Después en su interior hemos creado el loop for que anteriormente habíamos visto para crear una tabla de multiplicar según el valor.
Para llamar a la función debemos poner el nombre de la función como te muestro en el script dentro de la otra función Start.

Si no sabes modelar en 3d no te preocupes porque este libro viene acompañado de material adicional para que puedas realizar los proyectos. En el caso de que quieras realizar algún proyecto distinto puedes descargar del Asset Store de Unity recursos en donde encontraras material gratuito y de pago.

9. Entender qué son los Arrays

Los Arrays son matrices que nos permiten almacenar múltiples objetos en una misma variable. Esta clase solo se encuentra en **Javascript** pero podemos utilizar en **C#** con las **Buitlin arrays** (que son nativas de .Net). Este tipo de **Arrays** son muy rápidas pero no pueden re-dimensionarse.

A continuación te muestro un ejemplo muy básico de cómo podemos utilizar estas **Builtin arrays**:

Script: Arrays.cs

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

public class Arrays : MonoBehaviour
{
   public string[] lista;

   void Start()
   {
      this.lista= new string[4]{"Juan","Maria","Antonio","Lurdes"};
      foreach (string nombres in lista)
      {
         Debug.Log(nombres);
      }
   }
}
```

Primero declaramos una variable de tipo Builtin Array al que le damos el nombre de lista. Si te fijas verás que para decirle que el tipo primero ponemos el tipo de variable que será (int, string, float..) seguido de corchetes y luego le damos el nombre que deseamos.
Dentro de la función Start le damos valores a la variable lista; para ello en C# siempre debemos añadir valores de la siguiente manera new string y entre corchetes puedes poner el numero de elementos o valores que va a tener el Builtin Array, seguido abrimos llaves y entre comillas ya que es de tipo string ponemos los nombres separados por comas.
Para mostrar la lista he utilizado un foreach asignadole una variable de tipo string llamada nombre para mostrar cada elemento del Builtin Array de este modo cuando ejecutes el juego en la consola te mostrara todos los nombres de la lista.

A continuación te muestro como deberías verlo en la ventana Inspector y en la consola.

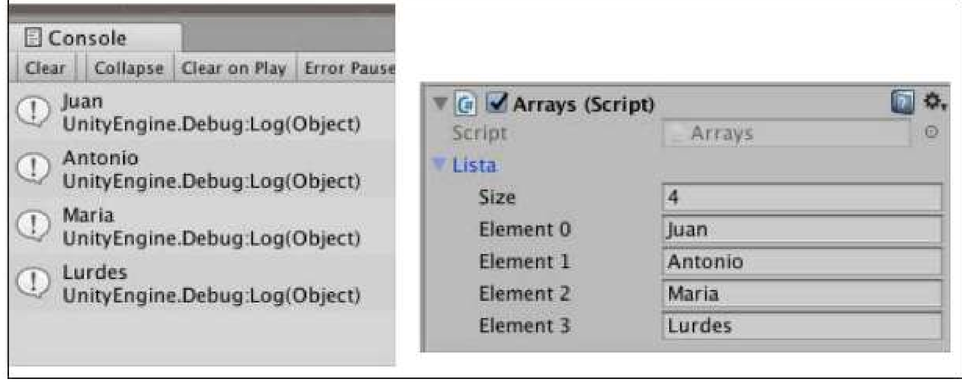


Fig. 5.12

10. Mi primera clase

Hasta ahora el tipo de programación que hemos visto es secuencial es decir empezaba arriba y Unity lo ejecutaba de arriba a bajo. En la **programación orientada** a objetos (POO) es un poco mas compleja en el sentido de que esta basada en la comunicación entre objetos.

Para entender mejor este concepto vamos a realizar una serie de practicas.

1# Atributos

Vamos a crear una tabla de récords en donde cada jugador pueda introducir el nombre, y que nos diga el número de partidas jugadas, el tiempo invertido de juego y si el jugador está activo o no.

Player 1	**Player 2**	**Player 3**
Nombre string partidas int tiempo float Activo bool	Nombre string partidas int tiempo float Activo bool	Nombre string partidas int tiempo float Activo bool

Crea un nuevo proyecto llamado Programacion2 y crea una escena llamada clases. Crear un script C# con nombre infoPlayer. Intenta que el proyecto tenga sus carpetas con los scripts y las escenas ordenadas.

Crea un objeto vacío llamado TablaRecords y le ponemos un script con nombre tablaRecords. Ahora creamos un nuevo script sin añadirlo a ningún sitio y le ponemos el nombre de infoPlayer. Abrimos este script y le ponemos las siguientes variables.

Script:infoPlayer

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

[System.Serializable]
public class infoPlayer
{
   public string nombre;
   public int partida;
   public float tiempo;
   public bool activo;
}
```

En primer lugar, debes borrar todo lo que viene por defecto puesto que vamos a crear una clase que nos proporcione información del player y luego para que podamos ver las variables y los atributos en el inspector de Unity debemos escribir entre corchetes System.Serializable.
El siguiente paso es crear una clase pública para poder acceder desde cualquier script y le ponemos el mismo nombre.
Ahora solamente declaramos las variables que necesitamos; nombre partida tiempo y activo.

Script:tablaRecords

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

public class tablaRecords : MonoBehaviour
{
   public infoPlayer player1;
   public infoPlayer player2;

   void Start ()
   {
      this.player1.nombre = “Paco”;
      this.player1.partida = 8;
      this.player1.tiempo = 6.25f;
      this.player1.activo = true;

      this.player2.nombre = “Maria”;
      this.player2.partida = 2;
      this.player2.tiempo = 4.2f;
      this.player2.activo = false;

      Debug.Log (player1.nombre+”, “+ player1.partida +”, “+ player1.tiempo+”
, “+ player1.activo);
      Debug.Log (player2.nombre+”, “+ player2.partida +”, “+ player2.tiempo+”
, “+ player2.activo);
       }
```

Este script es el que hemos agregado al objeto en Unity. Desde aquí vamos a acceder a la clase infoPlayer para crear nuestros players.
Si te fijas en la primera variable utilizo el nombre de la clase infoPlayer y le doy un nombre en este caso player1. Player1 es un nuevo objeto de la clase infoPlayer.
Ahora en la función Start ya puedo acceder a sus atributos (es decir sus variables) para ello ponemos el nombre de la variable punto y si todo es correcto en monodevelop te aparecen los atributos. Escoge el que necesites y le das el valor que quieras,

En este ejemplo he puesto el player 1 y el player 2 a ti te dejo el 3.
Para finalizar puedes mostrar en la consola los atributos de la clase. Recuerda que te aparecerán los valores cuando ejecutes el juego.

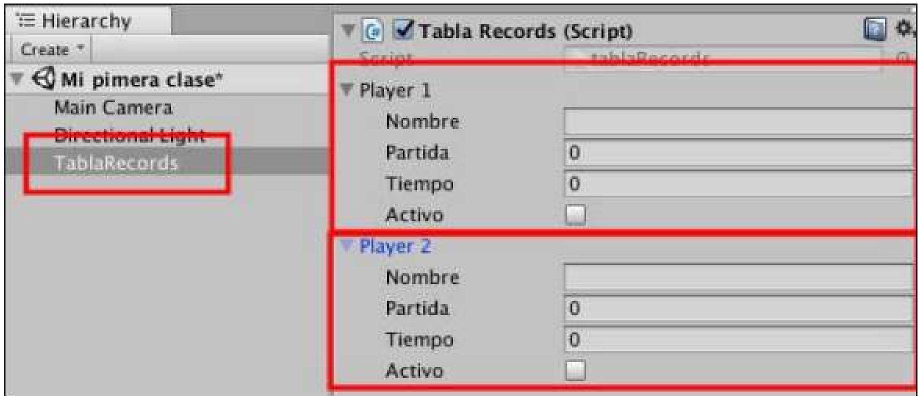


Fig. 5.13

Si ejecutamos el juego veremos que en la consola aparecen los atributos y en la ventana Inspector las cajas de texto se llenan con los valores que les hemos asignado.

Fig. 5.14

2# Métodos

Si seguimos con el ejemplo anterior, también podemos crear funciones dentro de las clases. Para esta actividad vamos a crear una función que nos muestre por consola los atributos de la clase, para no tener que escribir tanto.

Script:infoPlayer

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

```
[System.Serializable]
public class infoPlayer
{
   public string nombre;
   public int partida;
   public float tiempo;
   public bool activo;

       public void MuestraInfo()
   {
      Debug.Log (“Nombre del jugador: “ + nombre);
      Debug.Log (“Numero de Partidas: “ + partida);
      Debug.Log (“Tiempo de juego: “ + tiempo);
      Debug.Log (“Actividad actual: “ + activo);
   }
}
```

Creamos a continuación de las variables una función con el nombre que quieras en este caso es MuestraInfo y dentro escribimos varios Debug.Log con los mensajes y el nombre de la variable que deseamos que se muestren

Ahora que hemos creado este método en la clase solamente debemos substituir en el script tablaRecords los debugs Logs por lo siguiente.

Script:tablaRecords

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

public class tablaRecords : MonoBehaviour
{
   public infoPlayer player1;
   public infoPlayer player2;

   void Start ()
   {
      this.player1.nombre = “Paco”;
      this.player1.partida = 8;
      this.player1.tiempo = 6.25f;
      this.player1.activo = true;

      this.player2.nombre = “Maria”;
      this.player2.partida = 2;
      this.player2.tiempo = 4.2f;
```

```
        this.player2.activo = false;

        this.player1.MuestraInfo();
        this.player2.MuestraInfo();
    }
}
```

3# Constructor básico

Creamos funciones dentro de una clase para que se ejecuten cuando creamos un objeto de esa clase; o, dicho correctamente, se instancia un objeto de esa clase.

En los ejemplos anteriores hemos creado 2 players y ahora en el script infoPlayer vamos a crear el constructor de la clase, que nos va a informar de que se ha creado un nuevo objeto de esta clase.

Script:infoPlayer

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

[System.Serializable]
public class infoPlayer
{
    public string nombre;
    public int partida;
    public float tiempo;
    public bool activo;

    public infoPlayer()
    {
        Debug.Log("Se ha creado un objeto de la clase infoPlayer");
    }
        public void MuestraInfo()
    {
        Debug.Log ("Nombre del jugador: " + nombre);
        Debug.Log ("Numero de Partidas: " + partida);
        Debug.Log ("Tiempo de juego: " + tiempo);
        Debug.Log ("Actividad actual: " + activo);
    }
}
```

Dentro de la clase; debemos crear después de los atributos una función pública con el mismo nombre de la clase dentro ponemos un mensaje.

A continuación vamos a crear los objetos de la clase infoPlayer.

Script:tablaRecords

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

public class tablaRecords : MonoBehaviour
{
   public infoPlayer player1 = new infoPlayer();
   public infoPlayer player2 = new infoPlayer();

   void Start ()
   {
      this.player1.nombre = "Paco";
      this.player1.partida = 8;
      this.player1.tiempo = 6.25f;
      this.player1.activo = true;

      this.player2.nombre = "Maria";
      this.player2.partida = 2;
      this.player2.tiempo = 4.2f;
      this.player2.activo = false;

      this.player1.MuestraInfo();
      this.player2.MuestraInfo();
   }
}
```

Para crear un nuevo objeto diremos que la variable es igual a new seguido del nombre del constructor. Cuando ejecutes el juego en Unity si todo es correcto debe de mostrarte el mensaje que hemos escrito dentro del constructor.

3# Constructor inicializando valores.

También podemos crear los objetos con valores directamente, esto quiere decir que cuando creamos el player1 ya le podemos poner los valores directamente y evitamos tener que escribir tanto código. Para ello vamos al script de la clase y creamos escribimos lo siguiente:

Script:infoPlayer

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

[System.Serializable]
public class infoPlayer
{
   public string nombre;
   public int partida;
   public float tiempo;
   public bool activo;

   public infoPlayer()
   {
      Debug.Log("Se ha creado un objeto de la clase infoPlayer");
   }

   public infoPlayer(string n, int p, float t, bool a)
   {
      this.nombre = n;
      this.partida = p;
      this.tiempo = t;
      this.activo = a;
   }

       public void MuestraInfo()
   {
      Debug.Log ("Nombre del jugador: " + nombre);
      Debug.Log ("Numero de Partidas: " + partida);
      Debug.Log ("Tiempo de juego: " + tiempo);
      Debug.Log ("Actividad actual: " + activo);
   }

}
```

Ahora tenemos el constructor pero con una serie de argumentos.
Dentro del constructor guardamos estos argumentos dentro de las variables que son los atributos que necesitaremos para crear un objeto de esta clase.

Script:tablaRecords

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

public class tablaRecords : MonoBehaviour
{
   public infoPlayer player1;
   public infoPlayer player2;

   void Start ()
   {
      this.player1 = new infoPlayer("Paco",8,6.25f,true);
      this.player2 = new infoPlayer("Maria",2,4.2f,false);

      this.player1.MuestraInfo();
      this.player2.MuestraInfo();
   }
}
```

Nuestro script tablaRecords quedará de la siguiente forma,

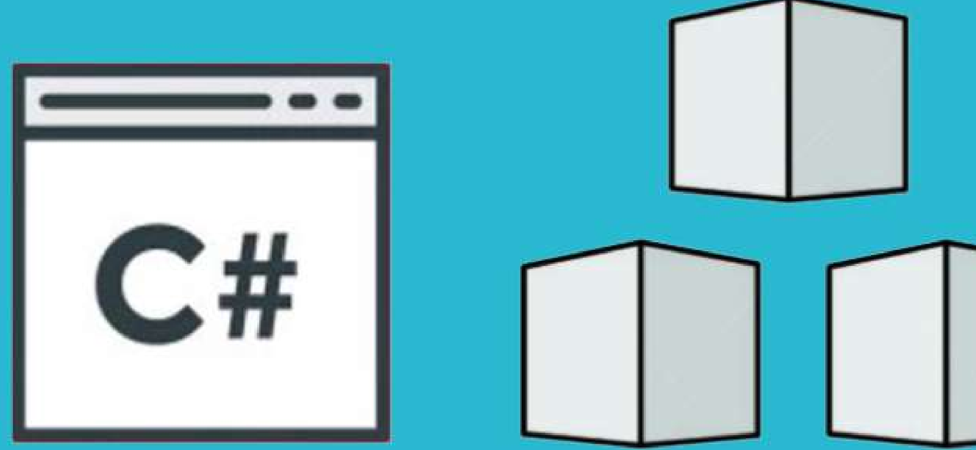
Hasta aquí esta pequeña introducción a la programación en C#, no quiero entrar en muchos detalles de programación básica, sino explicarte un poco las reglas del juego. Estas son algunas de las habilidades que debes entender y utilizar a pesar de que te equivoques, porque en los próximos capítulos empezaremos a ver cómo este lenguaje nos permite controlar objetos y que estos interactúen. La estructura que debes comprender cuando hablamos del concepto, (principalmente abstracto), de programación es:

- Dispones de unas variables para almacenar información compleja.
- Puedes realizar operaciones, comparar valores y tomar decisiones con estas variables.
- Puedes automatizar con bucles operaciones que se repiten continuamente.
- Puedes agruparlo todo en una función para utilizarla más adelante.
- Puedes crear tus propias clases que encapsula todo lo anterior.

En el próximo capítulo empezaremos a poner en practica estos conceptos tan abstractos para que empieces a concretar tus objetivos.

Capítulo 6

Programación orientada a objetos



- Introducción
- Clase GameObject
- Acceder a los componentes
- Entender las transformaciones
- Vector 3
- Mover objetos
- Rotar objetos
- Escalar objetos

1. Introducción

Una vez hemos visto el capítulo de programación básica es el momento de poner en práctica todo lo aprendido mediante actividades prácticas.

No pretendo que este libro se convierta en un simple conjunto de tutoriales, sino en una serie de ejercicios que te permita mejorar en el dominio de Unity.

Para esta tarea el vas a crear un nuevo proyecto llamado Capitulo 6. Una vez tengas abierto el nuevo proyecto, guarda el proyecto. Como ya sabes esta obra viene con material adicional para seguir los capítulos, ten localizado este material adicional para acceder a él.

Bien primero accedemos al menú principal y seleccionamos la opción Assets > Import Package > Custom Package. Se nos abrirá una ventana de navegación del sistema dependiendo de que sistema operativo tengamos Window, o Mac y debemos localizar dónde tenemos guardado el material adicional que acompaña el libro y seleccionar el paquete con el nombre Assets Capitulo6 dentro de la carpeta Proyecto_6.

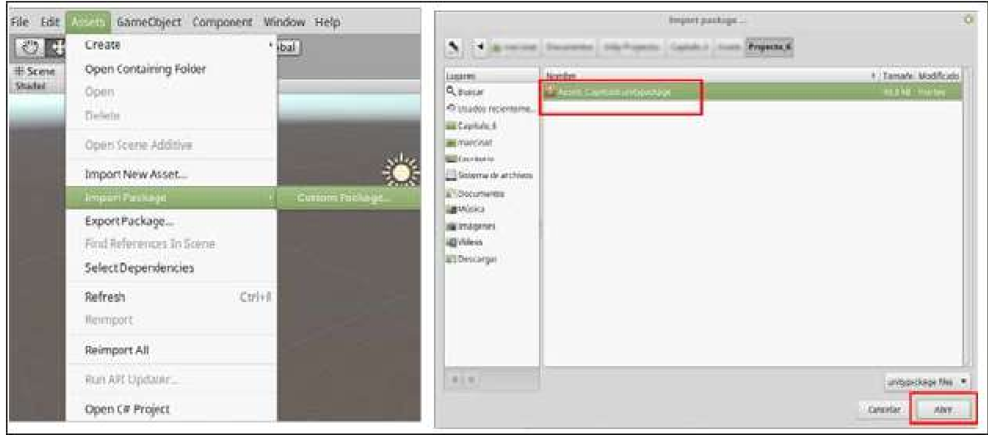


Fig. 6.1

Después de seleccionar el paquete se nos aparecerá una ventana en Unity que nos muestra el contenido de este paquete y nos permite seleccionar que queremos importar del paquete. En nuestro caso vamos a dejarlo todo marcado y le daremos al botón **Import**.

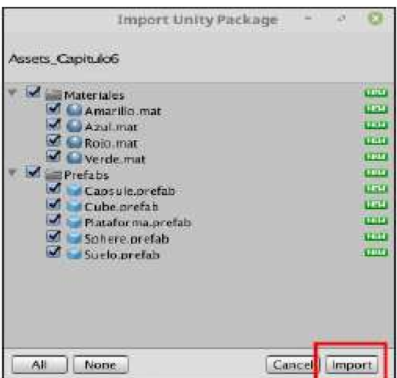


Fig. 6.2

Si lo hemos importado todo correctamente deberías poder ver las siguientes carpetas en la ventana Project.

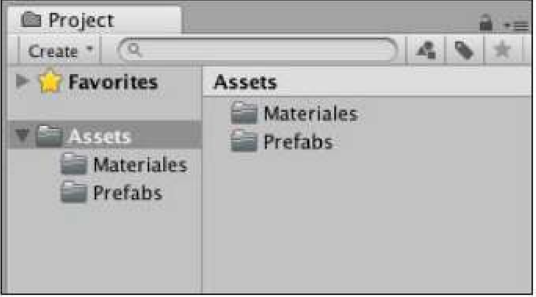


Fig. 6.3

Ya tenemos preparado el proyecto para seguir todas las pautas del capítulo que es uno de los más importantes si nunca has programado porque en este capitulo vamos a ver los siguientes apartados:

Las funciones que voy a mostrarte son referentes a la clase MonoBehaviour. Este primer apartado quiere ayudarte no solo a que entiendas cómo funcionan si no que aprendas también a consultar la documentación de Unity Scripting API. Puedes acceder desde el siguiente enlace:

https://docs.unity3d.com/es/current/ScriptReference/MonoBehaviour.html

Pero, ¿qué es la documentación de Unity? En la siguiente imagen te muestro con qué te vas a encontrar y cómo puedes utilizar esta documentación de una forma inteligente. He numerado las zonas que tienes que tener en cuenta para mejorar la comprensión.

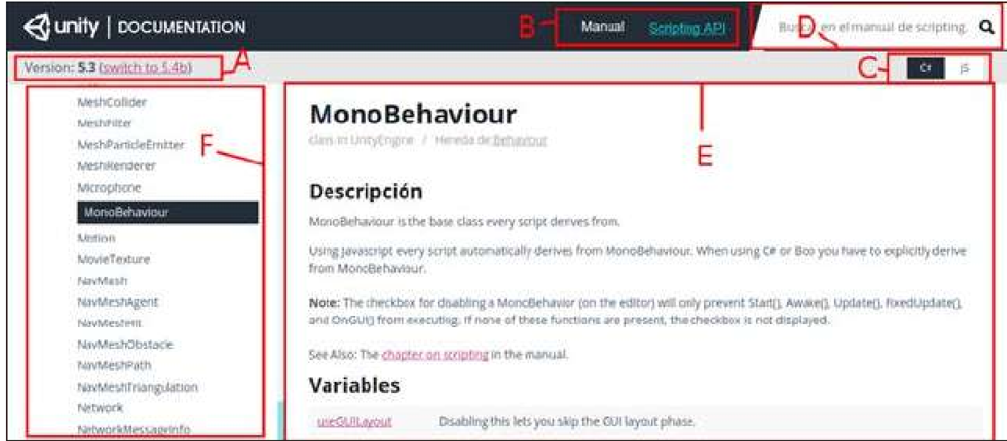


Fig. 6.4

En el apartado A dispones de la versión de Unity a la que estás consultando la información, no quiere decir que de una versión 5,3 a la 5,4 varíe mucho pero es recomendable saber a qué version accedes en la información.

En el apartado B tenemos dos tipos de documentación, el Manual y el Scripting API. El Manual nos proporciona información de cómo utilizar Unity, sus ventanas paneles y todo lo referente al programa en sí y el apartado de API es toda una recopilación de Cla-

ses, funciones, variables, etc. que podemos utilizar en nuestros Scripts para programar en Unity.

En el apartado C verás que puedes seleccionar qué tipo de lenguaje quieres consultar en el apartado Scripting API. Solo hace referencia a las consultas de Scripting. En este libro utilizaremos solamente C#.

El apartado D nos da la posibilidad de buscar una palabra en concreto. Más adelante, cuando tengas dudas sobre algún método en concreto puedes ir directamente al buscador y poner el método que buscas. La documentación te proporcionará una lista con todos los métodos y clases relacionadas con esa palabra.

La zona marcada con una E es la información de la búsqueda en donde te proporcionan descripciones de los métodos y las variables que necesitas.

Por ultimo tienes la sección F que es una lista ordenada alfabéticamente con todas las clases que puedes encontrar en la documentación en el caso de la API, y en el caso del manual una lista organizada por temas referente al programa Unity.

En este enlace verás un apartado Mensajes que en realidad son eventos. Un evento según la programación en C# es cuando ocurre algo. Los eventos habilitan una clase u objeto para notificarlo a otras clases u objetos. La clase que envía el evento recibe el nombre de publicador y las clases que reciben el evento se denominan suscriptores.

Algunas características sobre los eventos:

- El publicador determina el momento en el que se genera un evento; los suscriptores determinan la acción que se lleva a cabo en respuesta al evento.
- Un evento puede tener varios suscriptores. Un suscriptor puede controlar varios eventos de varios publicadores.
- Nunca se generan eventos que no tienen suscriptores.
- Los eventos se suelen usar para indicar acciones del usuario, como los clics de los botones o las selecciones de menú en las interfaces gráficas de usuario.

Dicho lo anterior vamos a poner algunos ejemplos con un ejercicio. Aquí solo te voy a mostrar cómo se escriben dentro de un script y luego explico para qué sirven algunos:

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

public class Evento : MonoBehaviour {

    void Awake()
    {
    Debug.Log("Esto es el Awake");
    }
    void Start()
    {
    Debug.Log("Esto es el Start");
    }
    void Update()
```

```
    {
    Debug.Log("Esto es el Update");
    }
    void OnDisable()
    {
    Debug.Log("El objeto está desactivado");
    }
    void OnEnable()
    {
    Debug.Log("El objeto está activado");
    }
    void OnEnable()
    {
    Debug.Log("El objeto está activado");
    }
    void OnMouseDown()
    {
    Debug.Log("El ratón hace click");
    }
    void OnMouseEnter()
    {
    Debug.Log("El ratón esta encima");
    }
    void OnMouseExit()
    {
    Debug.Log("El ratón sale fuera");
    }
}
```

- **Awake:** es llamado cuando se carga el script y se utiliza para inicializar cualquier variable o estado del juego antes de que comience el juego. Este evento solo se llama una sola vez y después de cargar todos los objetos.
- **Start:** es llamado cuando se habilita un script y justo antes de llamar a cualquiera de los métodos de Update. Al igual que en el Awake solo se llama una sola vez y se ejecuta cuando activamos el juego.
- **Update:** repite todo lo que contenga a cada frame o cuadro. Este evento de MonoBehaviour es muy utilizado para crear comportamientos en un juego.
- **OnDisable & OnEnable:** hemos visto anteriormente cómo podemos activar y desactivar objetos desde la ventana inspector de Unity con estas funciones podemos activar o desactivar el objeto mediante programación.
- **OnMouse(Enter,Down,Exit):** nos permite saber cuando el usuario realiza una serie de acciones con el ratón.

Estas son algunas de las funcionalidades que podemos encontrar en la programación de videojuegos, a continuación iremos profundizando hasta que comprendas como funciona todo.

2. Clase GameObject

Como hemos hecho con los eventos, ahora vamos a acceder a la documentación de GameObject, si te resulta más cómodo te dejo el enlace a continuación o puedes escribir la palabra GameObject en el buscador de la documentación de Unity :

https://docs.unity3d.com/es/current/ScriptReference/GameObject.html

En la documentación encontramos varios aspectos importantes que vamos a tratar en este apartado, pero antes quiero que entiendas la diferencia entre **GameObject** en mayúscula y **gameObject** en minúscula.

Si recordamos cuando creamos un script y lo agregamos o arrastramos encima de un gameObject por ejemplo un cubo, dentro del script para referirnos a el cubo lo escribiremos en minúscula: gameObject. Si queremos hacer referencia a un GameObject externo o crear uno propio utilizaremos GameObject en mayúscula ya que en este caso hacemos referencia a la clase.

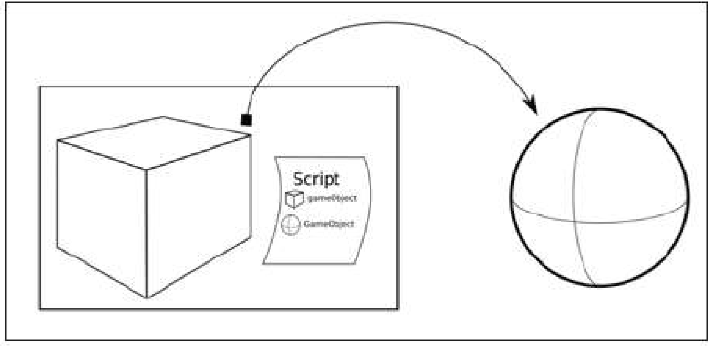


Fig. 6.5

En la imagen anterior representamos que un cubo se le añade un script, dentro del script cuando queremos referirnos al propio objeto utilizamos gameObject. En el caso de que el cubo quiera tomar referencia del objeto redondo dentro del script lo escribiremos en mayúscula GameObject.

Ahora en el proyecto de Unity accede a la carpeta Prefabs y arrastra un cubo al escenario o a la ventana Jerarquía como te muestro en la siguiente imagen:

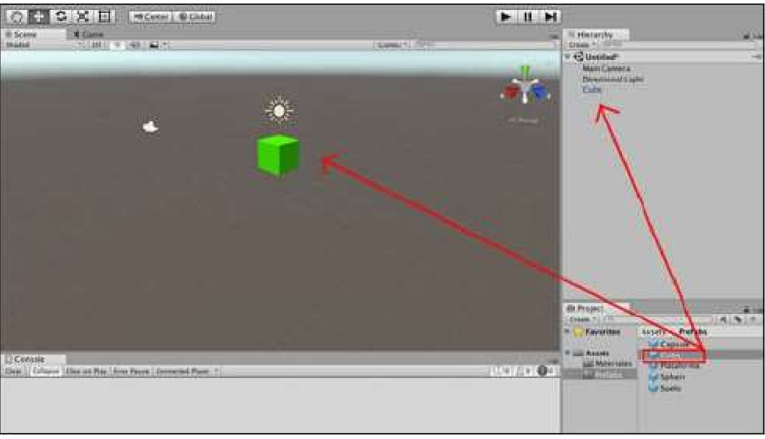


Fig. 6.6

A continuación crea una carpeta nueva dentro de **Project** con el nombre Scripts. Si no recuerdas cómo se hace puedes seleccionar la carpeta Assets y pulsar en el botón **Create** y seleccionar la opción Folder. Te aparecerá una nueva carpeta a la que puedes ponerle un nombre, en este caso Scripts.

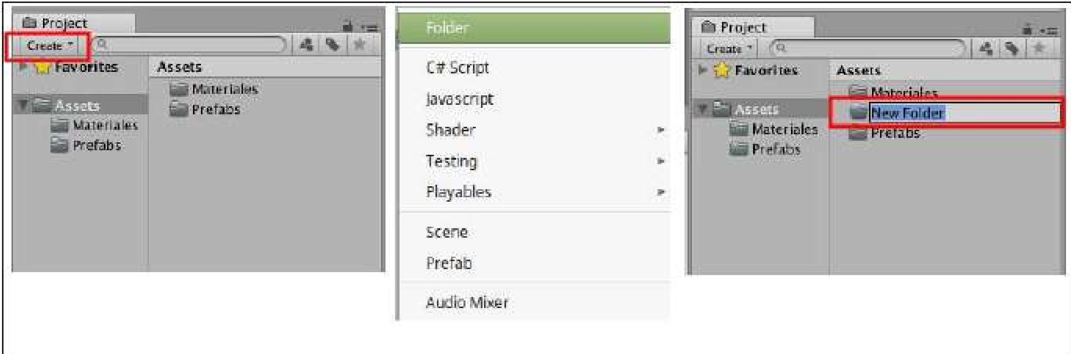


Fig. 6.7

Dentro de la carpeta Scripts vamos a crear un nuevo script con nombre Ejemplo1. Para crear un nuevo script hacemos doble clic encima de la ventana Scripts y dentro de ella hacemos clic en el botón derecho del ratón y en el menu que nos parece selecciona-mos la opción Create > C# Script. Se nos aparece un nuevo Script que podemos renom-brar, en este caso con el nombre Ejemplo1.

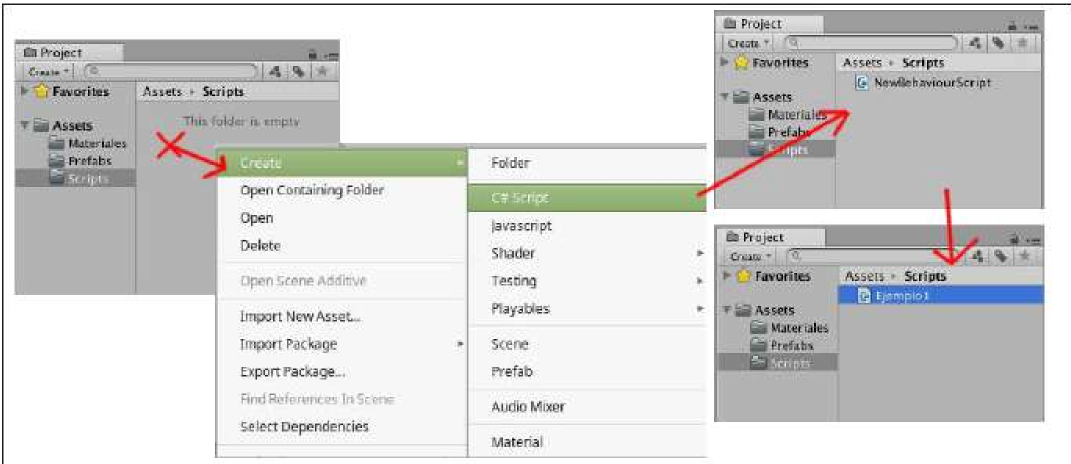


Fig. 6.8

Ahora seleccionamos el Cubo haciendo clic en la escena encima del objeto con el bo-tón izquierdo o desde la ventana Jerarquía encima del nombre del objeto, en este caso con nombre cube. Luego arrastramos el script que hemos creado dentro de la ventana inspector o encima del nombre del objeto cube, como te muestro en la siguiente imagen:

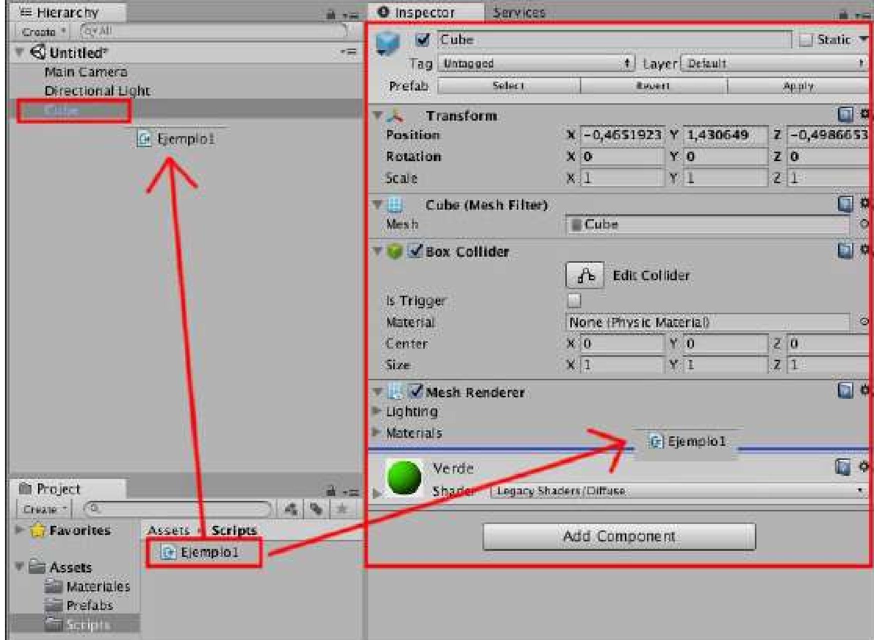


Fig. 6.9

Para finalizar la preparación de la escena vamos a agregar otro prefab a la escena como hemos hecho con el cubo. Añadimos una esfera pero esta no debe llevar ningún script. Te muestro cómo debería quedar la escena:

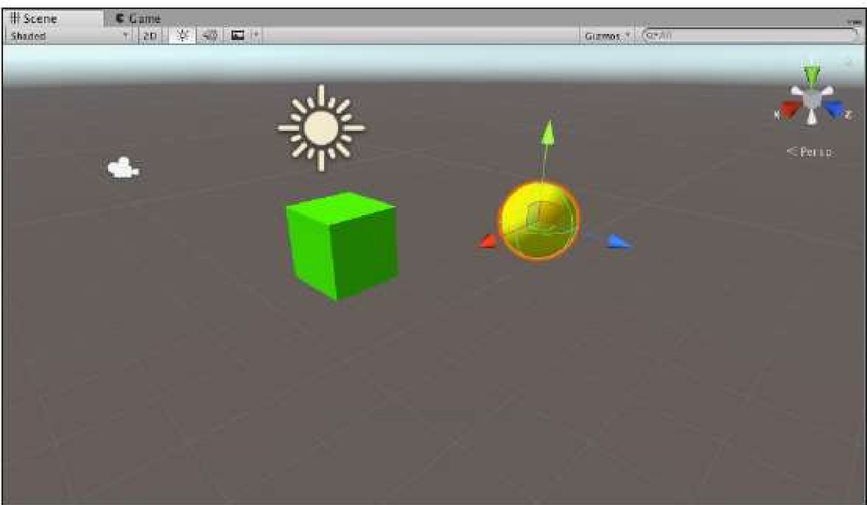


Fig. 6.10

Ahora solo nos queda guardar la escena pero antes vamos a crear una carpeta en la ventana Project con el nombre de Escenas, como hemos hecho con la carpeta script. Para guardar la escena simplemente accedemos al menú y seleccionamos la opción File >SaveScene te aparecerá un ventana de tu sistema operativo dentro de la carpeta Assets, selecciona la carpeta que has creado llamada Escenas y ponle el nombre de Escena 1. Tu escena ha quedado guardada y podemos empezar el ejercicio.

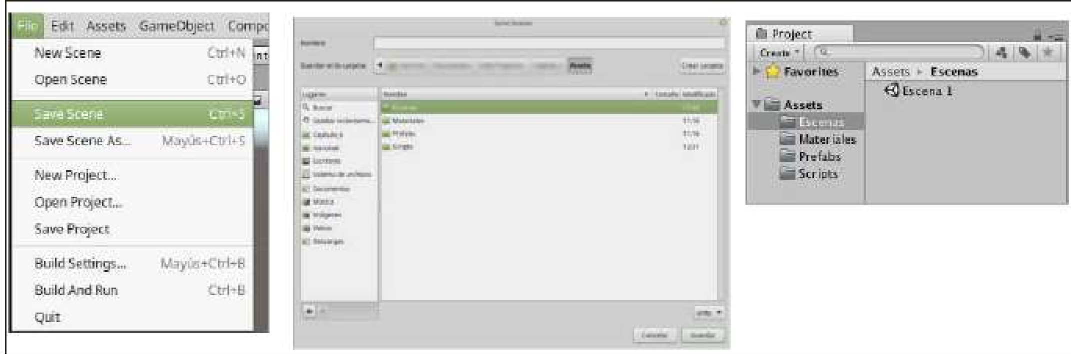


Fig. 6.11

A partir de ahora los enunciados van a ser de la siguiente manera para agilizar el temario.

#1 En una nueva escena crear 2 **GameObjects** un cubo y una esfera (puedes utilizar los Prefabs del material importado). Añadir un script al cubo con nombre Ejemplo1. Para acceder al Script hacemos doble clic encima del archivo Script Ejemplo1 y Unity abrirá el editor Monodevelop y escribiremos el siguiente Script:

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

public class Ejemplo1 : MonoBehaviour
{

    void Start()
    {
        Debug.Log(gameObject.name);
    }

}
```

Si te fijas dentro del Debug.Log hemos utilizado el gameObject en minúscula para referirnos al propio gameObject que en este caso es el cubo. También debes tener en cuenta que las clases tienen métodos y atributos uno de sus atributos es “name” que hace referencia al nombre del objeto.

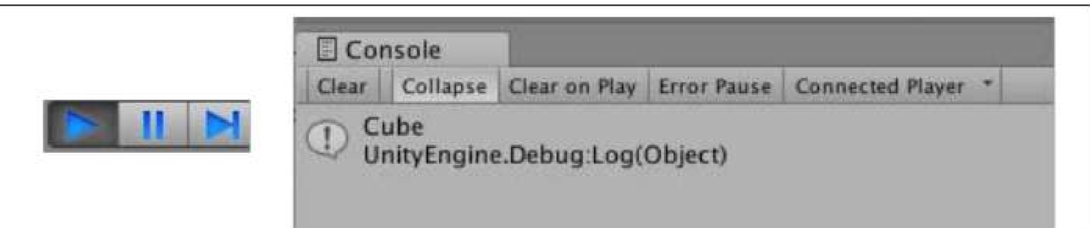


Fig. 6.12

Como puedes ver en la imagen anterior en consola se nos aparece el nombre Cube cuando activamos o ejecutamos la escena.

#2 Ahora vamos a cambiar el nombre que tiene el cubo. Si miras la documentación https://docs.unity3d.com/es/current/ScriptReference/GameObject.html verás que tenemos una variable o atributo llamado name que nos permite acceder al nombre del objeto, y en el ejemplo anterior lo hemos utilizado para que nos muestre el nombre, pero también podemos utilizarlo para cambiar el nombre.

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

public class Ejemplo1 : MonoBehaviour
{
    void Start()
    {
        gameObject.name = “Micubo”
        Debug.Log(gameObject.name);
    }
}
```

Cuando activemos el script, es decir cuando le demos al play verás como cambia el nombre del cubo y como por consola aparece Micubo.

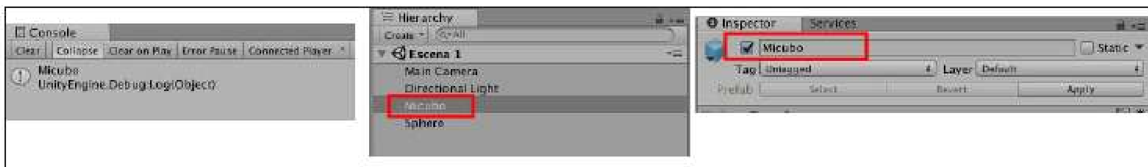


Fig. 6.13

En el ejemplo anterior verás que mientras ejecutemos el juego el nombre del cubo cambia por el nombre de Micubo y puedes verlo en la ventana Jerarquía y en la ventana Inspector.

Hemos visto cómo acceder al gameObject en el que hemos aplicado el script, pero la pregunta es ¿cómo accedemos a un objeto externo?

#3 Ahora vamos a acceder desde el script ejemplo1 del cubo a los componentes del objeto Esfera, arrastrando el objeto a una variable pública de tipo GameObject.

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

public class Ejemplo1 : MonoBehaviour
{
    public GameObject miEsfera;

    void Start()
```

Primero creamos una variable publica de tipo GameObject que nos permite almacenar objetos. La hacemos publica para que aparezca en la ventana inspector.

<pre>   {
        gameObject.name = “Micubo”
        Debug.Log(gameObject.name);
        Debug.Log(miEsfera.name);
    }
}</pre>
	Lo segundo es que muestre por la pantalla de la consola el objeto miEsfera, pero esto no va a funcionar a no ser que al volver a Unity arrastremos el objeto Esfera encima de la ventana inspector, en donde está la variable miEsfera representada.

Para el ejemplo 3 te muestro cómo queda una variable al hacerla pública en la ventana inspector.

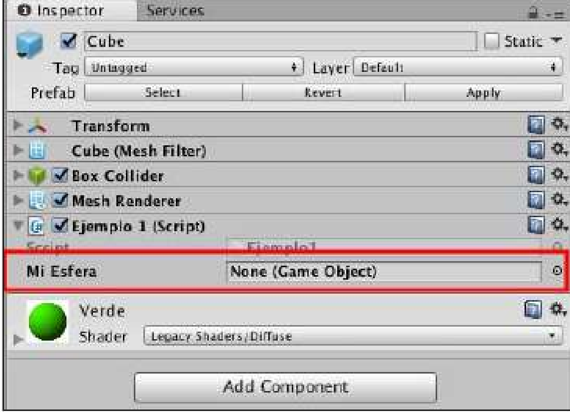


Fig. 6.14

Como puedes ver en la imagen anterior, la variable al hacerse pública se muestra en la ventana Inspector con una caja de textos que pone None (Game Object) esto significa que no hay un objeto asignado para esta variable y debemos asignarlo. Ahora te muestro cómo podemos asignarle manualmente un objeto a esta variable.

Si hacemos clic encima del punto o símbolo en forma de punto que encontramos al final de la caja de textos accederemos a una ventana flotante en donde podemos seleccionar el objeto que queremos.

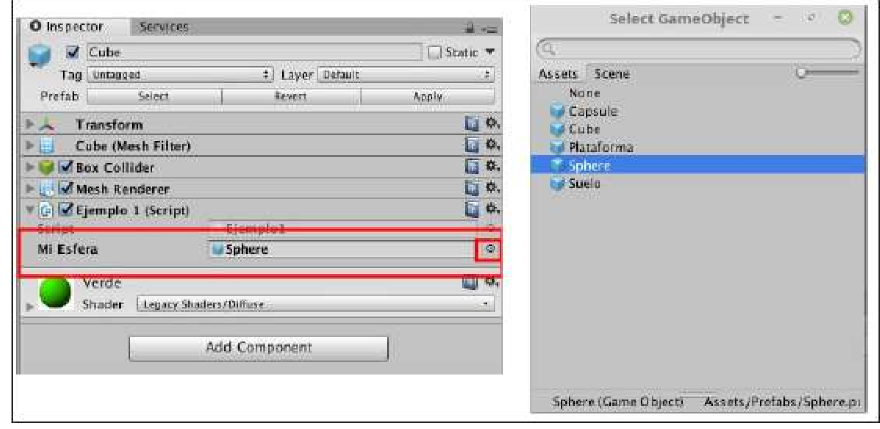


Fig. 6.15

Otra forma de realizar la acción anterior es seleccionar el objeto desde la ventana Jerarquía y arrastrarla encima de la caja de textos de la variable Unity lo asocia automáticamente.

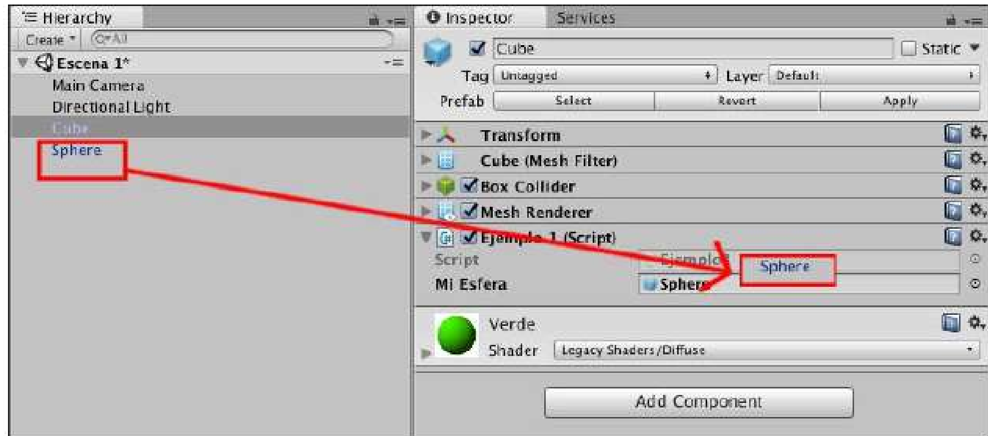


Fig. 6.16

Ahora ejecutamos el juego y podremos comprobar en la consola cómo nos aparece el nombre del objeto Cube con el nombre cambiado y el nombre del objeto externo Sphere que hemos asociado manualmente desde Unity.

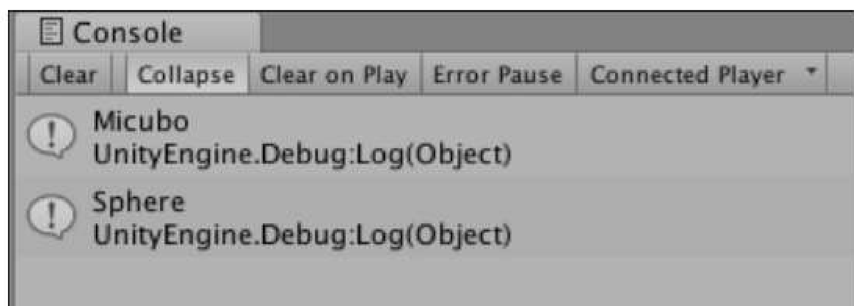


Fig. 6.17

#4 Nuestro siguiente paso es buscar el objeto esfera desde el propio script ejemplo1, pero debemos borrar la variable que hemos hecho anteriormente como te muestro en el siguiente script.

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

public class Ejemplo1 : MonoBehaviour
{
    public GameObject buscarEsfera;
```

```
     void Start()
     {
    gameObject.name = "Micubo"
    Debug.Log(gameObject.name);
    this.buscarEsfera= GameObject.Find("Sphere");
     }

}
```

Creamos una variable de tipo GameObejct con un nombre el que quieras en este caso le he puesto buscar Esfera.el anterior script para eliminar la referencia que hemos puesto en la actividad anterior. También he borrado el Debug.Log que me mostraba el nombre de la esfera y solamente he dejado la del cubo.

Al declarar la variable utilizo un método que puedes ver en la documentación llamado find. Este método buscará un objeto en la escena con el nombre que le pongamos entre paréntesis y comillas, en este caso el valor que le paso es el nombre del objeto Sphere.

Una vez volvamos a Unity veremos que en la ventana inspector que tenemos una nueva variable con el nombre buscarEsfera, pero vuelve a estar vacía. Esta variable tomará la referencia de la esfera o, dicho de otro modo, buscará el objeto con nombre Sphere cuando ejecutemos el juego. En la siguiente imagen te muestro cómo se verá antes y después de ejecutar el juego.

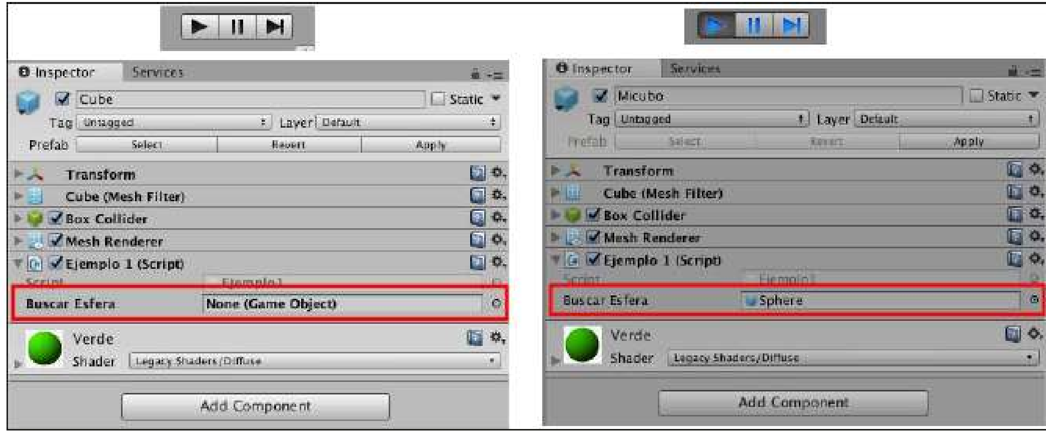


Fig. 6.18

#5 Otra forma de buscar objetos sería igual que la anterior pero utilizando Tags (Etiquetas). Para realizar la actividad vamos a utilizar otro de los Prefabs que disponemos en el proyecto. Selecciona el Prefab Capsule y arrástrala a la escena. Teniendo seleccionada la cápsula accedemos a la ventana Inspector y en el apartado Tag desplegamos su menú y seleccionamos la opción Player.

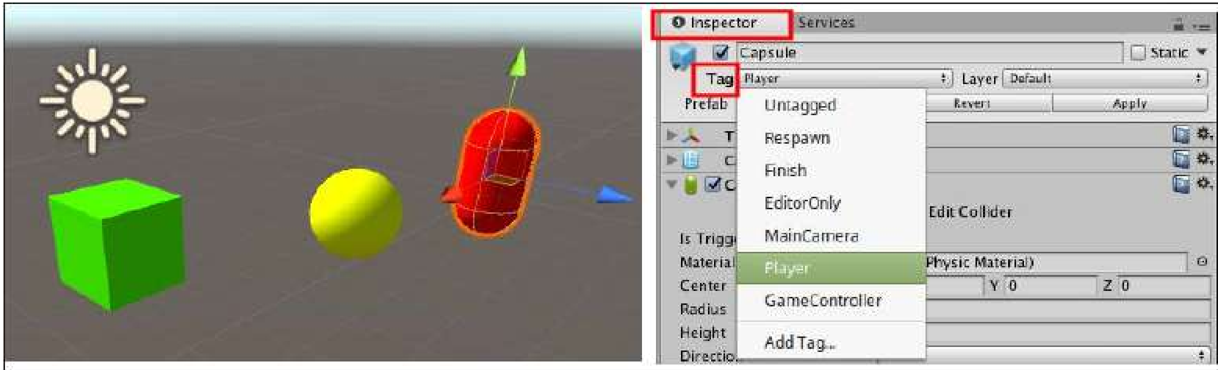


Fig. 6.19

Ahora ya disponemos de un tercer objeto en escena llamado Capsule que ademas tiene una etiqueta con el nombre Player que nos permite desde el script del cubo acceder a él. Seguimos utilizando el mismo script Ejemplo1 como te muestro a continuación:

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

public class Ejemplo1 : MonoBehaviour
{
    public GameObject buscarEsfera;
    public GameObject buscarCapsula;

    void Start()
    {
   gameObject.name = “Micubo”
   Debug.Log(gameObject.name);
   this.buscarEsfera = GameObject.Find(“Sphere”);
   this.buscarCapsula = GameObject.FindGameObejctWithTag (“Player”);
    }
}
```

Creamos una variable pública de tipo GameObject con nombre buscarCapsula, dentro del evento Start declaramos el valor de buscarCapsula que es un GameObject con una etiqueta “Player”.
Cuando ejecutes el juego en esta variable aparecerá contenido el objeto capsule.

Cuando vuelvas a Unity verás que en la ventana Inspector tenemos las variables declaradas públicas vacías, en el momento que ejecutemos el juego veremos cómo toman la referencia de los objetos.

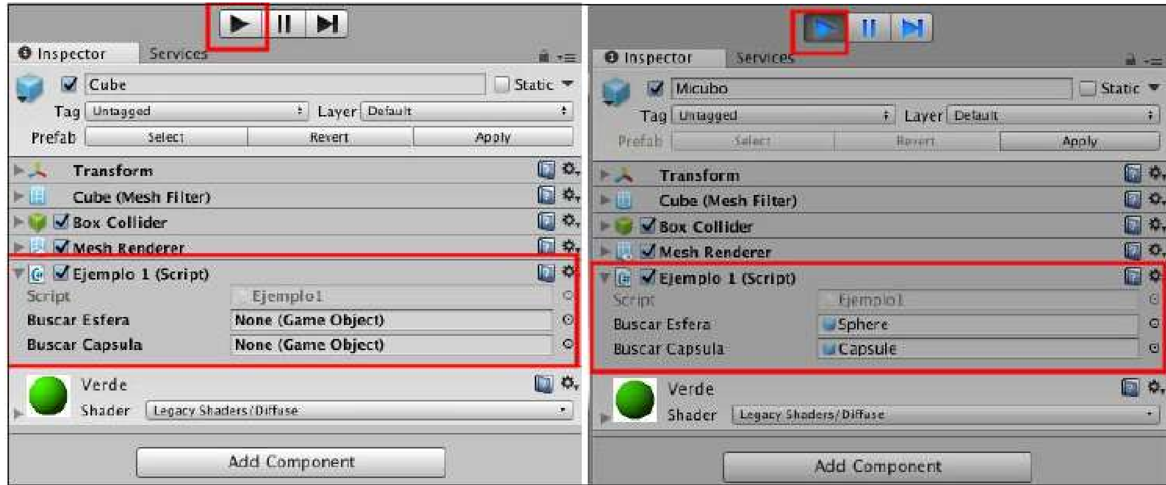


Fig. 6.20

#6 Para finalizar este apartado tienes que saber que también puedes crear GamesObjects utilizando el constructor. El constructor de una clase GameObject tiene el objetivo de que puedas crear nuevos objetos de esa misma clase (como todos los constructores). Para que entiendas mejor a lo que me refiero es que podemos crear nuevos objetos dentro de la escena. Primero te voy a invitar a consultar la documentacion accediendo a este enlace.

https://docs.unity3d.com/es/current/ScriptReference/GameObject-ctor.html

Cuando creamos otro objeto este se crea por defecto con un componente Transform que veremos mas adelante en detalle.

A continuación te muestro como podemos crear un objeto nuevo desde un script.

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

public class Ejemplo1 : MonoBehaviour
{
    public GameObject buscarEsfera;
    public GameObject buscarCapsula;
    public GameObject nuevo;

    void Start()
    {
   gameObject.name = "Micubo"
   Debug.Log(gameObject.name);
   this.buscarEsfera = GameObject.Find("Sphere");
   this.buscarCapsula = GameObject.FindGameObejctsWithTag ("Player");
   this.nuevo= new GameObject ("Nuevo");
    }
}
```

Cuando actives el juego verás que se te aparece un objeto llamado Nuevo. Para mostrarte cómo se ve en Unity he capturado dos imágenes una antes de ejecutar el juego y la segunda imagen ejecutando el juego.

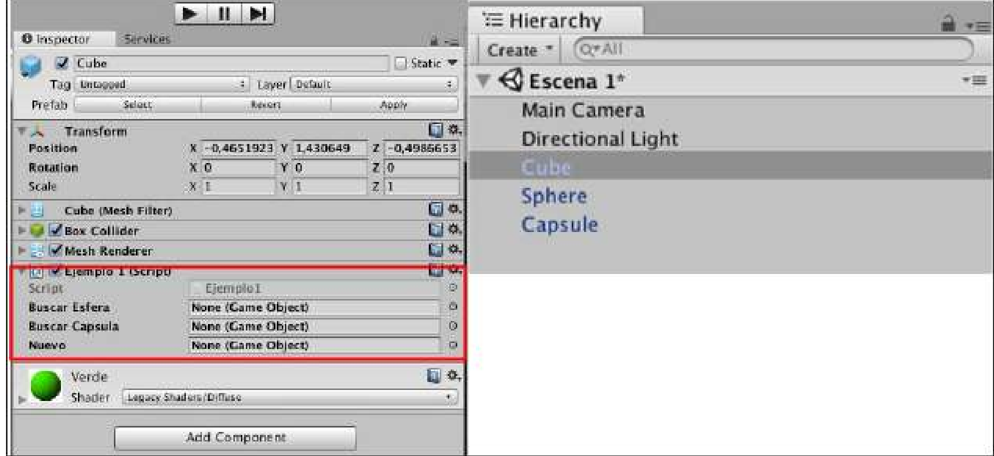


Fig. 6.21

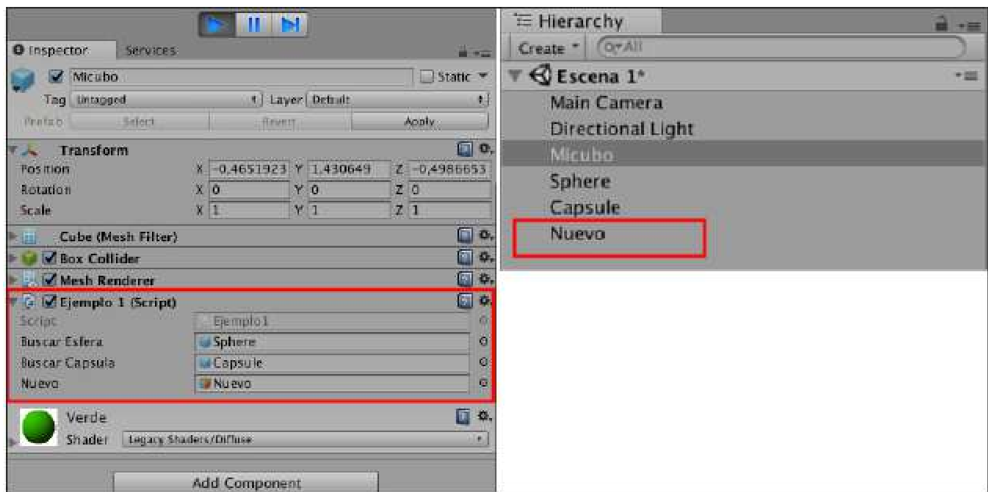


Fig. 6.22

El nuevo objeto que se crea solamente tiene un componente transformación y nombre porque todo objeto en Unity debe tener estos componente como mínimo. También hay que decir que este objeto no tiene nada más y por ese motivo no se puede ver en la ventana escena. A estos objetos les llamamos objetos vacíos y ya los hemos utilizado para ordenar escenarios utilizándolos como padre de los demás objetos. Si tienes dudas puedes revisar el capítulo 4.

3. Acceder a los componentes

Seguimos con los **GameObjects** pero esta vez vamos a ver cómo acceder a los componentes de estos mismos. En los primeros capítulos vimos cómo crear **gameObjects**

y cómo añadir componentes como un collider o materiales; ahora vamos a ver cómo podemos hacer esto mediante scripts. Supongo que te preguntarás por qué tienes que aprender a hacerlo mediante programación cuando puedes hacerlo por el editor; pues la respuesta es muy simple, en ocasiones deberás acceder mediante scripts para poder automatizar componentes en concreto, y porque te da una libertad increíble para trabajar en tus futuros proyectos.

Sé que me repito, pero te aconsejo que tengas la documentación accesible ahora que ya sabes cómo consultarla.

https://docs.unity3d.com/540/Documentation/ScriptReference/GameObject.html

Para explicar el siguiente apartado realizaremos varios ejemplos. Crea una nueva escena accediendo al menú principal en **File > New Scene** como te muestro en la siguiente imagen:

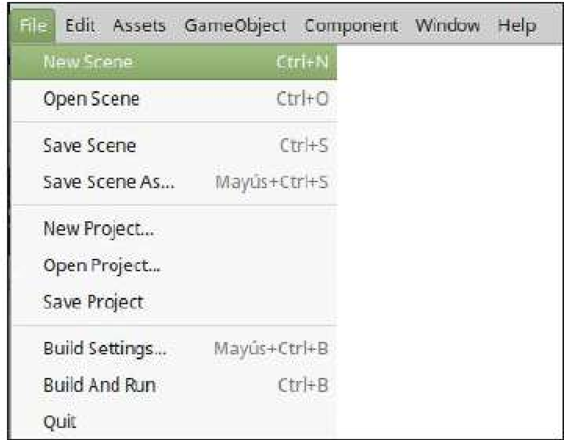


Fig. 6.23

Ahora verás que la escena ha quedado limpia. Seguidamente accede al menú principal otra vez y accede a la opción **Save Scene**. Ponle nombre Escena 2 y guarda la escena en la carpeta de Escenas. Dentro de la Escena 2 arrastra un Prefab cubo como te muestro en la siguiente imagen:

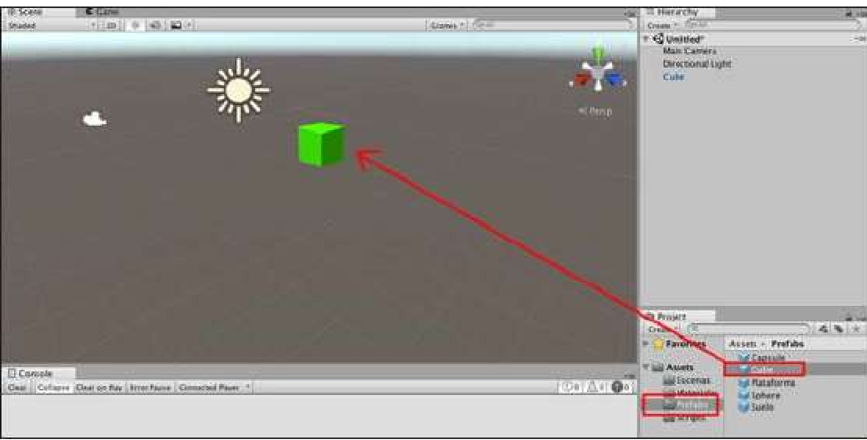


Fig. 6.24

Ahora crearemos un objeto vacío de forma manual, es decir, sin programación. Para ello hacemos clic encima de la ventana Jerarquía en el botón **Create** y seleccionamos la primera opción **Create Empty**. Automáticamente se nos aparecerá en la lista un nuevo objeto con nombre por defecto **GameObject**.

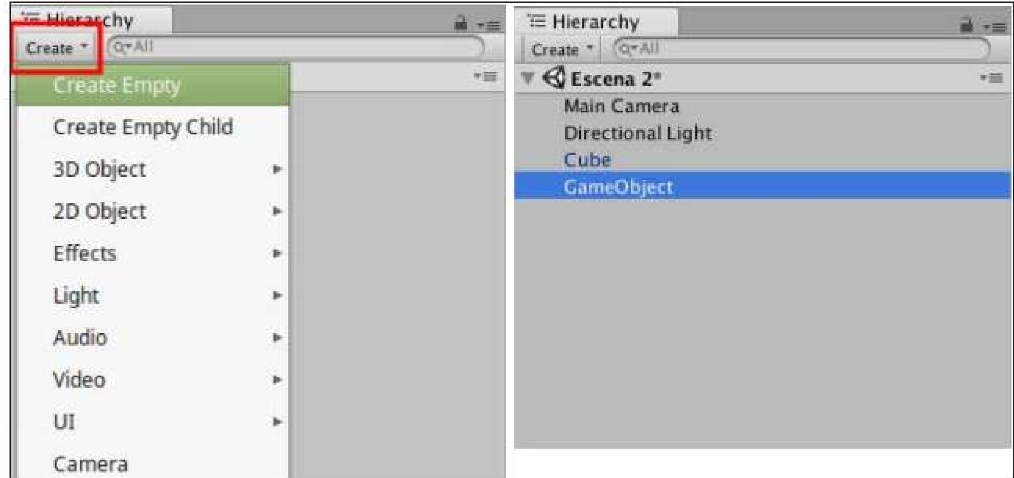


Fig. 6.25

#1 En el **gameObject** vacío le añadimos un script llamado **Components**. Cuando crees el nuevo script y le hayas puesto nombre guárdalo en la carpeta Scripts para mantener ordenado el proyecto. Dentro del Script **Components** vamos a cambiar el nombre y vamos a mostrar la posición en (x,y,z) por consola. Esto significa a que vamos a acceder a sus componentes, el del nombre ya lo hemos visto a continuación te muestro el script.

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

public class Components : MonoBehaviour {

     void Start()
     {
       gameObject.name = “Esfera”
     }

    void Update()
    {
       Debug.Log(gameObject.transform.position);
    }
}
```

Primero, como el script está dentro del objeto utilizamos directamente gameObject dentro de la función Start. Como vimos en el apartado anterior para cambiar el nombre utilizamos .name y le damos el valor del nombre entre comillas.
Después, dentro de la función Update utilizamos el Debug.Log y dentro de los paréntesis hacemos referencia al objeto en sí como gameObject, seguido de un punto para acceder al componente transform. Esto nos devuelve 3 valores, uno para cada eje (x,y,z)
Puedes cambiar el valor de x, y, z manualmente desde la ventana inspector para ver como cada vez te muestra por consola un valor distinto. Esto sucede porque dentro de la función update ser repite continuamente.

A continuación te muestro el resultado que me muestra la ventana Inspector y la consola.

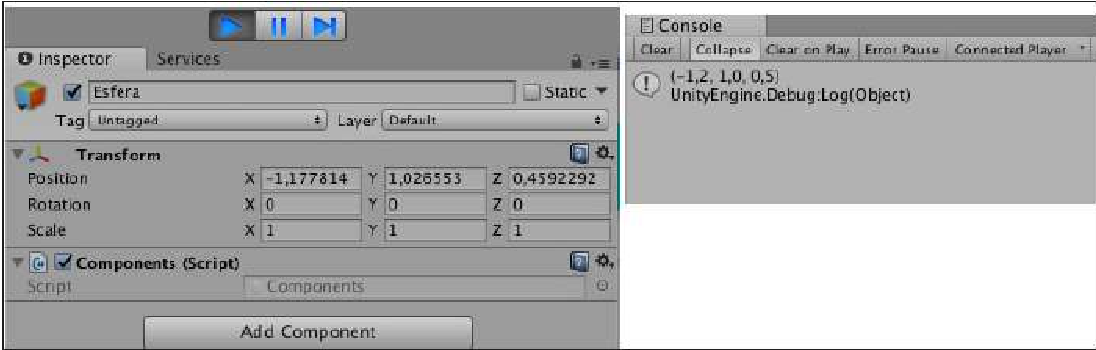


Fig. 6.26

Este ejemplo anterior simplemente es para mostrarte de que mediante programación podemos controlar con mucha precisión ciertos aspectos de los objetos. También verás que por consola solamente nos muestra un decimal en la posición. Antes de continuar quiero que entiendas que las transformaciones son de tres tipos:

- Posición: que nos permiten desplazar los objetos por toda la escena.
- Rotación: que nos permite rotar los objetos
- Escalado: que nos permite cambiar el tamaño de los objetos.

Todas las transformaciones se basan principalmente en tres ejes X, Y, Z que nos dan una posición, una rotación y un escalado más precisos. Estas tres coordenadas pueden ser de utilizadas de un modo Global o un modo Local. De momento vamos a ir viendo poco a poco los 3 tipos y cómo podemos acceder a ellos.

#2 Ahora creamos un nuevo script que llamaremos **ComponenteExterior.cs;** vamos a ver cómo accedemos al componente transform de un objeto externo, en este caso del cubo que tenemos en escena. Primero selecciona el GameObject vacío y le eliminar el script que tenía anteriormente de la siguiente manera:

Selecciona el **GameObject** vacío y, accediendo a la ventana Inspector en el componente Script, haz clic encima del símbolo que te marco en la imagen siguiente y en el menú que aparece seleccionamos la opción Remove Component; el componente script será eliminado.

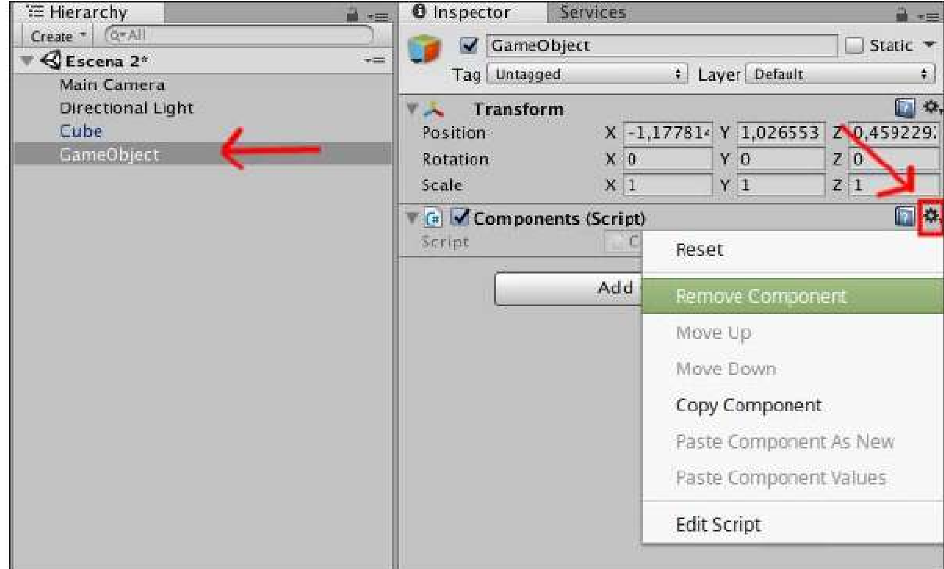


Fig. 6.27

Ahora, si ya has creado el nuevo script con el nombre de **ComponenteExterior.cs**, se lo agregamos al **Game Object** vacío. El componente nuevo de tipo script debe quedar como en la siguiente imagen:

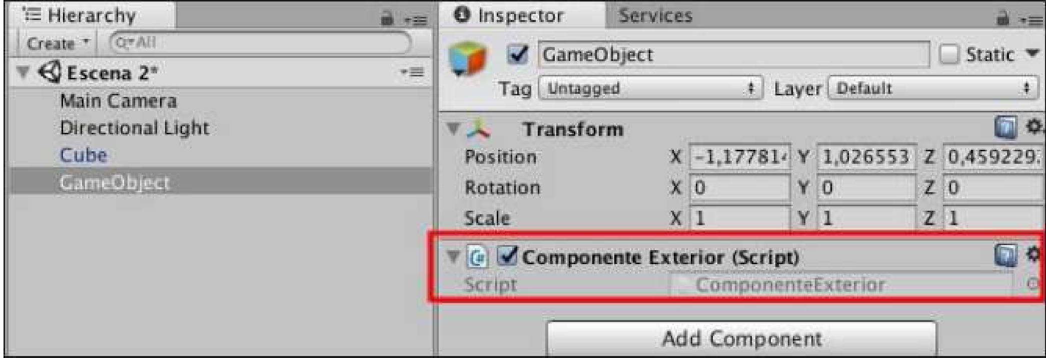


Fig. 6.28

A continuación te muestro el script que debe mostrarte la posición en cada eje por separado del Cubo. Recuerdo que estos ejercicios son para familiarizarnos con la forma de acceder a los componentes.

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

public class ComponenteExterior : MonoBehaviour
```

```
{
    public GameObject miCube;
    public float cubeX;
    public float cubeY;
    public float cubeZ;

    void Start()
    {
   this.miCube = GameObject.Find("Cube");
   this.cubeX = miCube.transform.position.x;
   this.cubeY = miCube.transform.position.y;
   this.cubeZ = miCube.transform.position.z;
   Debug.Log(cubeX);
   Debug.Log(cubeY);
   Debug.Log(cubeZ);
    }
}
```

Primero debemos crear una variable de tipo GameObject para contener al cubo que he llamado miCubo y luego como necesitamos mostrar 3 valores con decimales he creado 3 variables de tipo float para los ejes del cubo x,y ,z.
Dentro de Start primero vamos a buscar el objeto Cube que en este caso como solo hay un cubo podemos utilizar el método Find con el nombre "Cube".
Ahora ya tenemos el cubo dentro de una variable y podemos acceder a su primer eje de transformación. Por eso cojo las variables que he creado para almacenar el valor en x,y,z. Estas variables serán igual a la variable miCube, que en realidad es el cubo y que accedemos a su componente transform en la posición x,y ,z.
Por último, solo tenemos que mostrar las variables dentro del Debug.Log.

En la siguiente imagen te muestro cómo queda la ventana inspector y el resultado en consola:

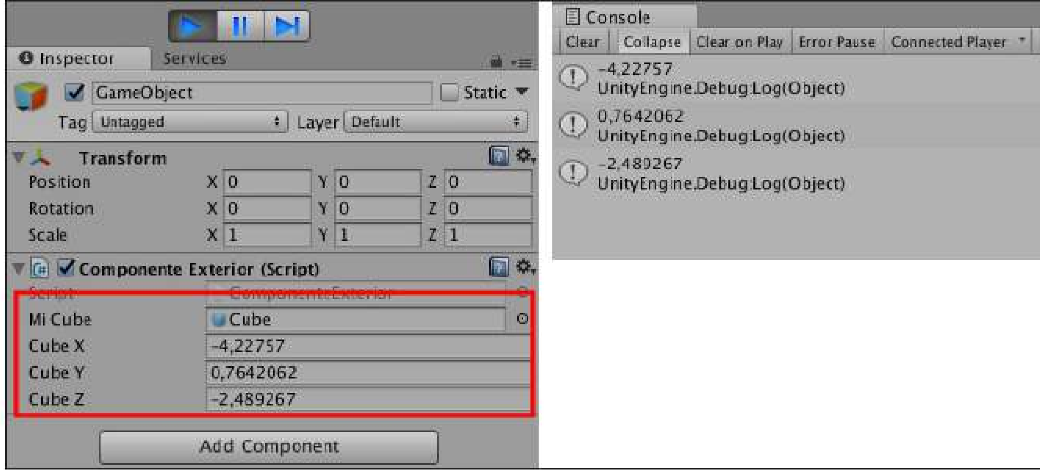


Fig. 6.29

El componente de transformación de momento lo dejaremos aquí y lo retomaremos más adelante con ejemplos mucho más prácticos. Ahora vamos a ver cómo podemos acceder a otro tipo de componentes. Esta vez vamos a guardar la información en variables para mayor comodidad.

#3 Ahora, ¿cómo lo hacemos para acceder a los otros componentes? Por ejemplo, ¿cómo podemos acceder al collider del cubo y cambiarle el tamaño de su collider?

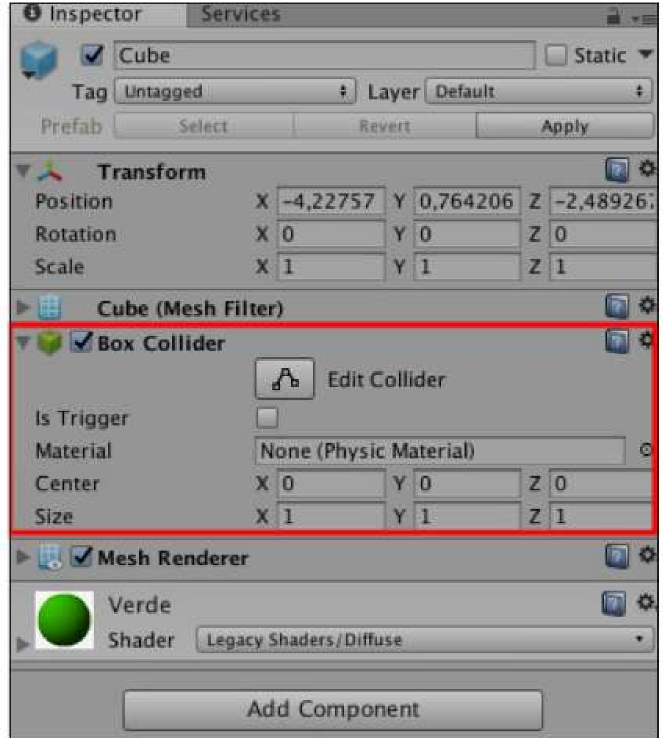


Fig. 6.30

Si te fijas en la imagen anterior verás que el cubo tiene un componente llamado Box Collider. Dentro de este componente tiene un parámetro llamado **Size** que es el encargado de darle un tamaño en los ejes x, y, z. Para poder guardar la información en una variable de este componente utilizamos un método llamado **GetComponent** que vas a ver en el script. Si lo deseas puedes acceder a este enlace para más información de este método:

https://docs.unity3d.com/es/current/ScriptReference/GameObject.GetComponent.html

Para la siguiente actividad elimina el script del GameObject vacío y crea uno nuevo con el nombre de **ComponenteCollider.cs** y agrégalo a este.

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

public class ComponenteCollider : MonoBehaviour {
```

```
    public GameObject miCube;
    public BoxCollider miCollider;

    void Start()
    {
   this.miCube = GameObject.Find("Cube");
   this.miCollider = this.miCube.GetComponent<BoxCollider>();
   this.miCollider.size = new Vector3 (5f,5f,5f);
    }
}
```

Primero debemos crear una variable pública de tipo GameObject para contener al cubo que he llamado miCubo y luego necesitamos un contenedor de tipo BoxCollider para almacenar la información del Collider del cubo.
Dentro de Start primero vamos a buscar el objeto cube; que en este caso como solo hay un cubo podemos utilizar el método Find con el nombre "Cube".
Ahora ya tenemos el cubo dentro de una variable pero todavía no podemos acceder a su componente BoxCollider, primero debemos obtener acceso y es por ese motivo que hemos creado la variable miCollider. Para tener permiso utilizamos el método GetComponent
seguido del tipo entre <> y acabamos con();
Ahora la variable miCollider puede acceder a las características del componente BoxCollider y podemos definir un tamaño.
El tamaño necesita tres parámetros o valores decimales y por eso hemos creado un Vector3. Si no comprender muy bien que es no pasa nada porque lo veremos en otro tema.
Si todo ha salido bien deberías ver como el collider del cubo se hace 5 unidades más grande.

En la imagen siguiente te muestro cómo ya hemos hecho en los ejemplos anteriores, cómo accedemos a un objeto externo y a su componente desde el script del GameObject vacío. La variable pública MiCube de tipo GameObject toma de referencia el Objeto Cube de la escena y la variable pública MiCollider de tipo BoxCollider toma de referencia del BoxCollider del objeto Cube.

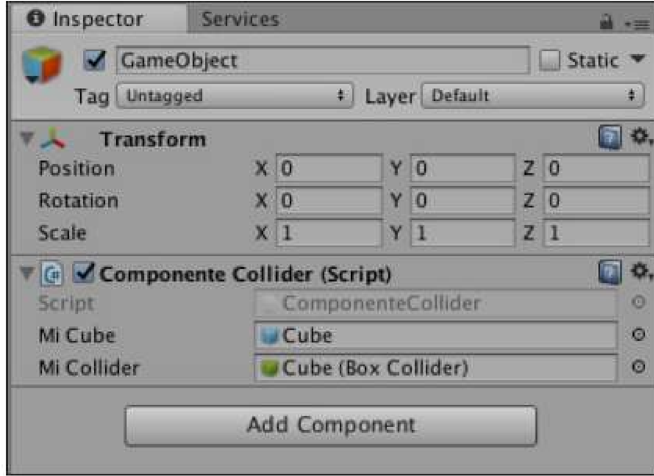


Fig. 6.31

Con el juego en ejecución podrás ver, si seleccionas el cubo, cómo su componente Collider tiene un tamaño de 5 unidades para todos sus ejes como te muestro en la siguiente imagen:

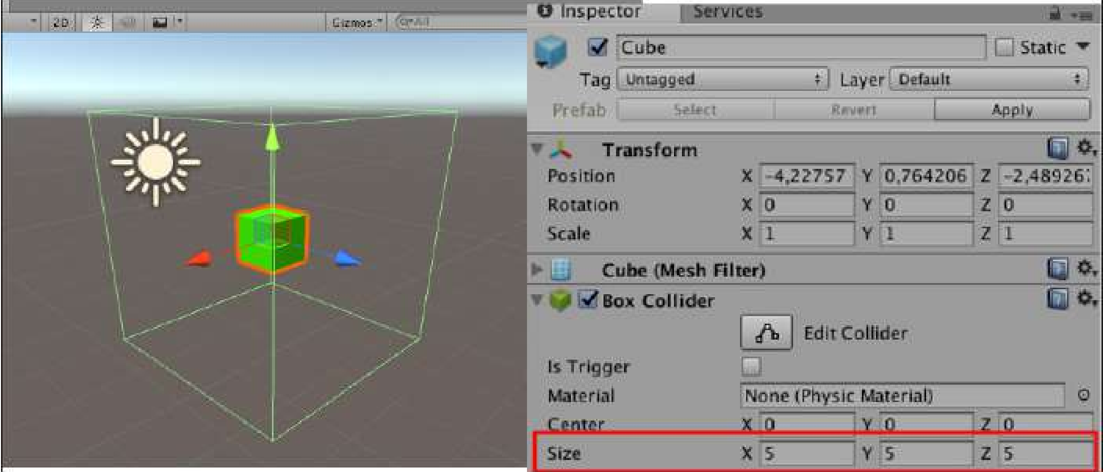


Fig. 6.32

Hasta ahora hemos demostrado que con un script podemos tener acceso a los objetos y a sus componentes, tanto al propio objeto que tiene el script como a un objeto externo. En el próximo apartado vamos a ver en mayor detalle el componente de transformación.

4. Entender las transformaciones

Ya sabemos cómo podemos acceder a gameObjects y a sus componentes, es lo que necesitamos para empezar a trabajar con las transformaciones.

En el inspector de Unity tenemos tres opciones de transformación que son posición, rotación y escalado. Estas tres opciones se caracterizan por un vector3 en cada una de sus transformaciones. Un vector 3 se caracteriza por un valor en x,y,z. Estos valores son coordenadas que nos permiten posicionar, rotar o escalar un objeto.

Si queremos empezar mover nuestros objetos vamos a tener que entender que es un Vector3.

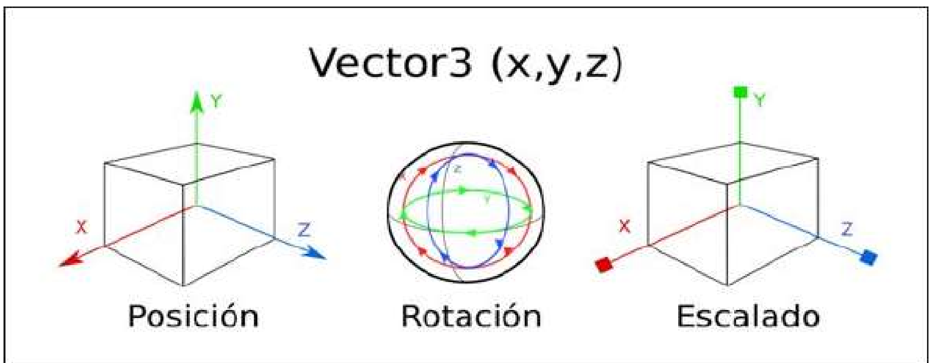


Fig. 6.33

En el inspector de Unity tenemos tres opciones de transformación que son posición, rotación y escalado. Estas tres opciones se caracterizan por un vector3 en cada una de sus transformaciones. Un vector 3 se caracteriza por un valor en **x, y, z**. Estos valores son coordenadas que nos permiten posicionar, rotar o escalar un objeto.

Puedes mirar la documentación referente a la clase **Transform** accediendo al siguiente enlace:

https://docs.unity3d.com/540/Documentation/ScriptReference/Transform.html

Para empezar a ver cómo funciona vamos a crear una nueva escena con el nombre Escena3. Guárdala en la carpeta Escenas como hemos hecho desde el comienzo del capítulo. Los objetos que vamos a utilizar son un cubo y una esfera, arrastra estos objetos de los Prefabs o si lo prefieres puedes crear los objetos tú mismo. En el caso de que crees los objetos desde el menú principal accediendo a GameObject > 3D Object > Cube u otro tipo de objeto, puedes utilizar los materiales que dispones en la carpeta Materiales en la ventana Project. Para diferenciar estos objetos vamos a ponerles nombres según el color que tienen; en el caso del ejemplo que voy a mostrar el cubo tendrá el nombre Verde y la esfera tendrá el nombre Amarillo, como te muestro en la siguiente imagen:

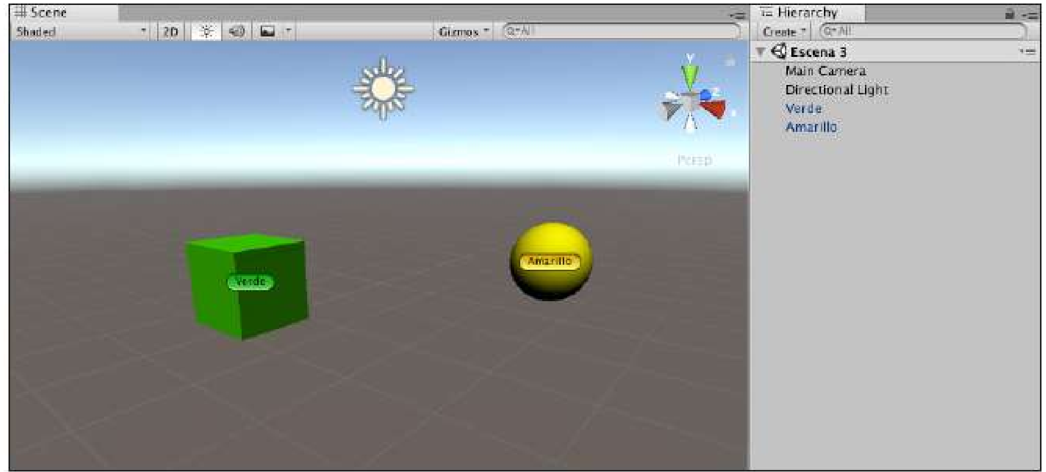


Fig. 6.34

#1 Crearemos un Script llamado Posiciones.cs y se lo agregamos al objeto verde. Este script tiene el objetivo de cambiar de posición el objeto Verde. A continuación te muestro como es el script.

Script Posiciones.cs

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

public class Posiciones : MonoBehaviour {
    public Transform direccion;
```

```
    void Start()
    {
    this.direccion = gameObject.GetComponent<Transform>();
    this.direccion.transform.position = new Vector3 (5f,5f,5f);
    }
}
```

Declaramos una variable de tipo Transform para almacenar la información de este tipo que se compone de la posición, la rotación y el escalado.
En la función Start accedemos al componente Transform para eso decimos que la variable dirección sea igual al propio objeto y guardamos el componente de tipo transform. Siempre utilizaremos esta fórmula GetComponent<El Tipo>();
Ahora podemos acceder desde la variable al componente de transformación, en concreto al de posición y para cambiar su posición creamos un vector 3 con 3 valores, uno para cada eje de coordenadas (x, y, z).

Si hemos escrito bien el script al acceder a Unity y ejecutar el juego veremos que la posición del cubo en la ventana Inspector tiene el valor de 5 en todos sus ejes.

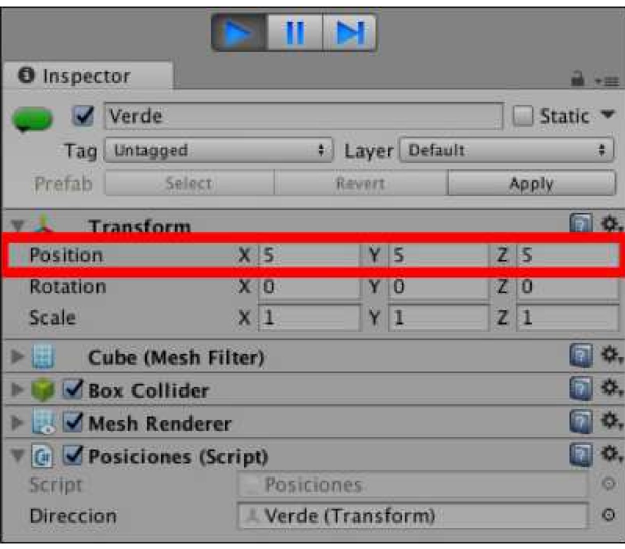


Fig. 6.35

#2 Desde el mismo script vamos a desplazar el objeto Amarillo, que se debe tomar una nueva posición cuando ejecutes el juego. Las partes azules son las que añadimos.

Script Posiciones.cs

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

```
public class Movimiento : MonoBehaviour
{
        public Transform direccion;
        public GameObject miAmarillo;
        public Transform direcAmarillo;

    void Start()
     {
        this.direccion = gameObject.GetComponent<Transform>();
        this.direccion.transform.position = new Vector3 (5,5,5);
        this.miAmarillo = GameObject.Find("Amarillo");
        this.direcAmarillo = this.miAmarillo.GetComponent<Transform>();
        this.direcAmarillo.transform.position = new Vector3 (-2, 5, 1);
     }
}
```

Primero debemos crear una variable de tipo GameObject para guardar la información del objeto que queremos obtener y otra variable de tipo Transform para guardar la información del componente transform del objeto que queremos obtener.
En la función Start utilizamos la variable miAmarillo y guardamos el objeto utilizando el método Find para que busque en la escena el objeto con nombre Amarillo.
Una vez tenemos localizado el objeto accederemos a su componente Transform y guardaremos la información en la variable direcAmarillo.
Ahora podemos acceder desde esta variables al componente de transformación en concreto al de posición y para cambiar su posición creamos un vector 3 con 3 valores, uno para cada eje de coordenadas (x,y,z) En este ejemplo le he puesto los valores que me han parecido.

Si todo es correcto, la esfera de color amarillo tomará la nueva posición como te muestro a continuación:

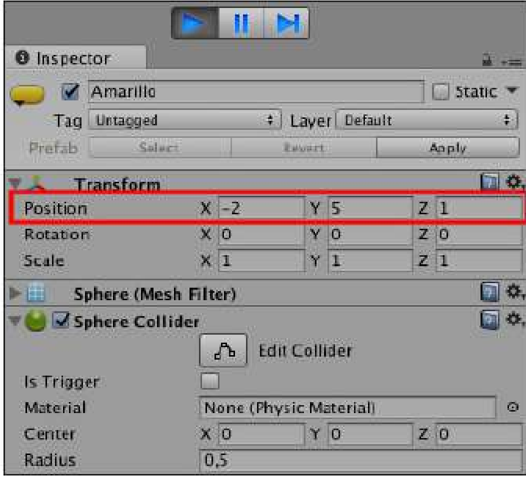


Fig. 6.36

5. Vector3

Antes de continuar veras que hemos encontrado una estructura que es el Vector3 que nos permite dar valores a las posiciones x, y, z. Te recomiendo que te mires la documentación:

https://docs.unity3d.com/540/Documentation/ScriptReference/Vector3.html

Si te fijas en la documentación veras que disponemos de una serie de variables estáticas que nos permiten resumir algunas posiciones que son muy comunes.

Vector3	Variable estática
new Vector3 (0,0,0);	Vector3.zero
new Vector3 (1,1,1);	Vector3.one
new Vector3 (0,1,0);	Vector3.up
new Vector3 (0,0,1);	Vector3.forward
new Vector3 (1,0,0);	Vector3.right

También se puede acceder a los valores de la tabla anterior desde **transform** de la siguiente forma.

transform	Variable estática
transform.zero	Vector3.zero
transform.one	Vector3.one
transform.up	Vector3.up
transform.forward	Vector3.forward
transform.right	Vector3.right

Esto nos será de gran utilidad cuando aprendamos a mover nuestros objetos mediante teclado. En la siguiente imagen te represento un dibujo de como un cubo avanzaría una unidad utilizando el Vector3.forward.

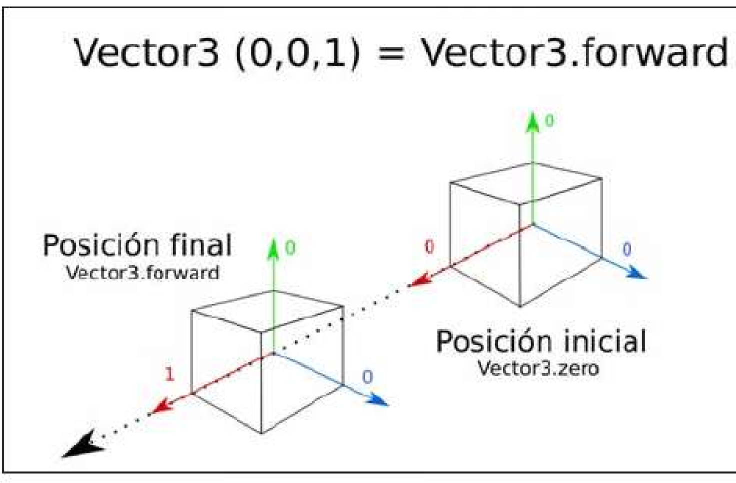


Fig. 6.37

6. Mover objetos

Ahora que ya hemos jugado con las posiciones de los objetos vamos a ver como podemos moverlos o crear una animación continua de estos objetos. Para mover objetos de forma continua cada frame utilizaremos la función Update que tenemos en los scripts por defecto cuando creamos uno. La función Update se encarga de repetir continuamente todo lo que le pongamos dentro.

Para este ejemplo vamos a continuar con nuestra Escena 3 y los objetos Verde y Amarillo pero vamos a añadir otro objeto de la carpeta Prefabs, la cápsula de color rojo a la que le cambiaremos el nombre de Capsule por el nombre de Rojo. En la siguiente imagen te muestro los objetos que debemos tener en la escena:

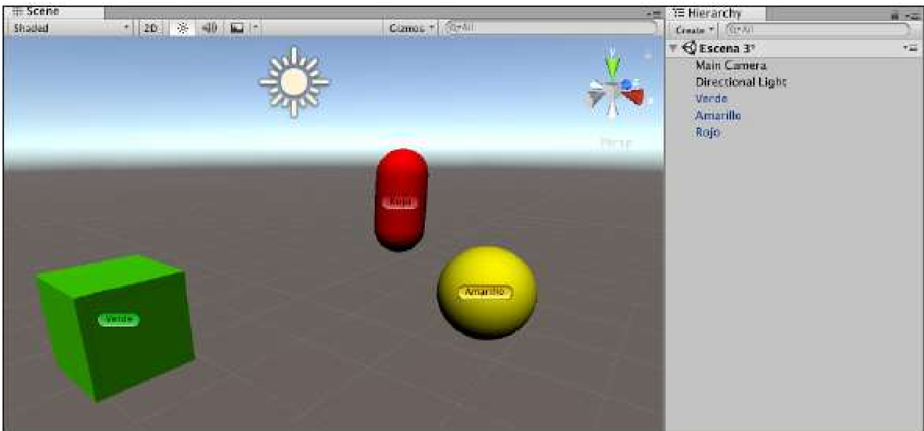


Fig. 6.38

En este caso vamos a utilizar los Vector3 que hemos aprendido en el apartado anterior. Vamos a utilizar el **forward** (eje azul), **right** (eje rojo), **up** (eje verde). Los ejes a los que me refiero son a los ejes de coordenadas globales de Unity.

#1 En este ejemplo elimina el componente script que tenía el objeto Verde y crea un nuevo script con el nombre MovConstante.cs. Este script lo vamos a agregar al objeto Verde para no variar.

Script: MovConstante.cs

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

public class MovConstante : MonoBehaviour
{
   public GameObject miAmarillo;
   public GameObject miRojo;

   public Transform direcVerde;
   public Transform direcAmarillo;
   public Transform direcRojo;

   void Start ()
   {
      this.miAmarillo = GameObject.Find ("Verde");
      this.miRojo = GameObject.Find ("Rojo");
      this.direcVerde = gameobject.GetComponent<Transform> ();
      this.direcAmarillo = this.miAmarillo.GetComponent<Transform> ();
      this.direcRojo = this.miRojo.GetComponent<Transform> ();
   }
   void Update ()
   {
      this.direcVerde.transform.position += transform.forward;
      this.direcAmarillo.transform.position += transform.up;
      this.direcRojo.transform.position += transform.right;
   }
}
```

Las variables las he agrupado en dos: las dos primeras son variables de tipo GamObject para poder localizar los objetos Amarillo y Rojo, es por ese motivo que les he puesto el nombre de miAmarillo y miRojo. Las siguientes variables son de tipo transform y las vamos a utilizar para guardar la información del componente transform de los objetos de escena, para el objeto Verde le he dado el nombre de direcVerde (direc hace referencia a la dirección).

En la función Start localizamos los objetos que necesitamos para las variables de tipo GameObject. Simplemente decimos que this.miAmarillo es igual al GameObject ahora le decimos que objeto y para encontrarlo utilizamos el método Find seguido del nombre del Objeto entre comillas y paréntesis.

Una vez hemos localizado los objetos para this.miAmarillo y this.miRojo, podemos almacenar el componente Transform de las otras variables. La que se diferencia es la de this.direcVerde porque es el objeto que lleva el script, por ese motivo utilizaremos gameObject en minúscula y accederemos a su componente Transform con GetComponent como hemos visto hasta ahora.
En las otras dos variables por ejemplo en direcAmarillo debe ser igual, primero debemos decirle el objeto en este caso tenemos localizado el objeto en la variable miAmarillo luego podemos utilizar el GetComponent para guardar la información. Para la variable direcRojo realizamos la misma operación.
En la función Update tenemos una operación aritmética expresada con un símbolo += esto es lo mismo que:
this.direcAzul.transform.position=this.direcAzul.transform.position+transform.forward;
Porque dentro de Update accedemos a la variable que tiene acceso al componente Transform y le sumamos la posición que tiene + la posición de forward que es (0,0,1).
En realidad Update repite la secuencia infinitas veces entonces la posición inicial es (0,0,0)+(0,0,1)=(0,0,1). En el primer frame o cuadro en el segundo sera lo siguiente (0,0,1)+(0,0,1)=(0,0,2), eso quiere decir que en el eje z en el segundo frame va avanzar 2 unidades.

A continuación te muestro como se verá la ventana inspector del objeto Verde al ejecutar el juego.

Fig. 6.39

Time.deltaTime

Vamos a ver un método de la clase Time que nos permite utilizar tiempo y no frames en la función Update. Es una variable estática dentro de la clase time y almacena valores float, puedes consultar más información en la documentación de Unity:

https://docs.unity3d.com/540/Documentation/ScriptReference/Time.html

¿Qué és esto exactamente? La respuesta es que Update utiliza un framerate o un rango de fotogramas para ejecutar el juego. Con esto quiero decir que si tu ordenador es

más potente la ejecución del juego será mucho más rápida, por el contrario si tu PC es antiguo el juego será bastante más lento. Por eso exite este método Time que es la clase y deltaTime que es el método. Cuando decimos delta hacemos referencia a un cambio en una cantidad; eso significa que deltaTime es un cambio en la cantidad de tiempo en segundos desde el último fotograma.

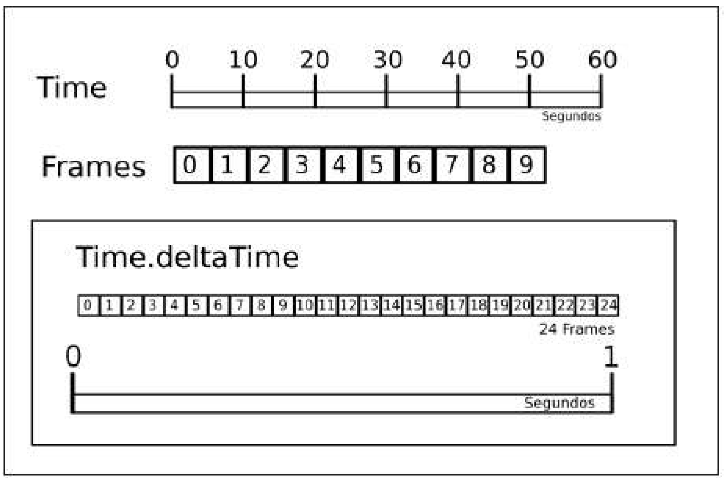


Fig. 6.40

Vamos a utilizar el método **Time.deltaTime** para no depender del rango de fotogramas de la función Update, y crearemos una variable velocidad que será de tipo float y a la que multiplicaremos al tiempo para acelerar el movimiento de las esferas.

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

public class MovConstante : MonoBehaviour
{
   public GameObject miAmarillo;
   public GameObject miRojo;

   public Transform direcVerde;
   public Transform direcAmarillo;
   public Transform direcRojo;

   public float velocidad;

   void Start ()
   {
      this.miAmarillo = GameObject.Find ("Verde");
      this.miRojo = GameObject.Find ("Rojo");
      this.direcVerde = gameobject.GetComponent<Transform> ();
```

```
        this.direcAmarillo = this.miAmarillo.GetComponent<Transform> ();
        this.direcRojo = this.miRojo.GetComponent<Transform> ();
        this.velocidad = 5f;
    }
    void Update ()
    {
        this.direcVerde.transform.position+=transform.forward*Time.deltaTime*-
this.velocidad;
        this.direcAmarillo.transform.position+=transform.up*Time.deltaTime*-
this.velocidad;
        this.direcRojo.transform.position+=transform.right*Time.deltaTime*this.
velocidad;
    }
}
    {
       this.direcAzul.transform.position += transform.forward*Time.deltaTime*-
this.velocidad;
       this.direcVerde.transform.position += transform.up*Time.deltaTime*this.
velocidad;
       this.direcRojo.transform.position += transform.right*Time.deltaTime*-
this.velocidad;
    }
}
```

Simplemente hemos multiplicado Time.deltaTime para que utilice el tiempo en vez del los fotogramas por segundo del Update y al crear una variable velocidad con valor 5 los objetos se moverán a una velocidad más lenta que anteriormente.

Mover objetos con Transform.Translate

La clase Transform tiene un método que se llama Translate que nos permite desplazar un objeto en cualquiera de los ejes x, y, z con la opción de escoger en qué tipo de espacio en relación al objeto (Local, Global). Si quieres consultar la documentación te dejo el enlace:

https://docs.unity3d.com/540/Documentation/ScriptReference/Transform.Translate.html

Cuando trabajamos en 3D tenemos que tener otro factor en cuenta: qué tipo de referencia utilizamos para los ejes de coordenadas, el local o el global. Para distinguir de una forma muy sencilla cuál es el que nos conviene debemos preguntarnos si queremos que el movimiento esté basado en el objeto o en el espacio 3D. Si queremos que el movimiento se base en el objeto debemos utilizar la relación Local, por el contrario si queremos que siga una única dirección o la del espacio 3d usaremos la relación Global.

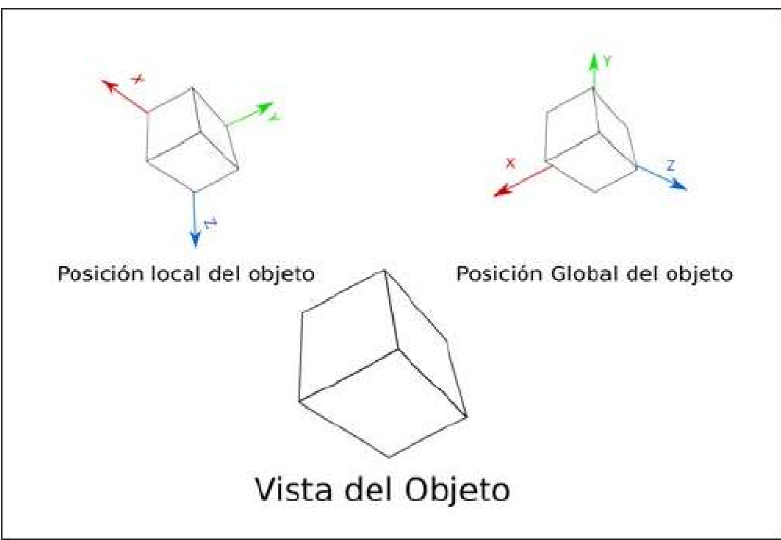


Fig. 6.41

Para esta actividad elimina todos los objetos de la escena excepto el objeto Rojo (La capsula). Crea un nuevo script con el nombre **Translate.cs** y se lo agregas al objeto Rojo. Antes de acceder al script selecciona el objeto Rojo y le ponemos los siguientes valores al componente **Tansform** de la ventana Inspector. **Rotation (60,90,30).**

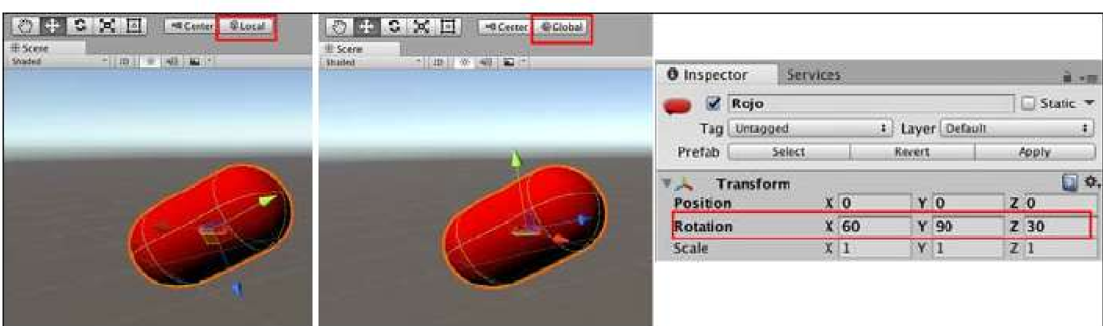


Fig. 6.42

En la imagen anterior verás que te muestro cómo son los ejes del objeto en modo Local y modo Global. A continuación te muestro cómo podemos animar el objeto en modo local:

Script: Translate.cs

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

```
public class Translate : MonoBehaviour
{
   public Transform direcRojo;
   public float velocidad = 5f;
   private float acceleracion;

   void Start ()
   {
   this.direcRojo = gameObject.GetComponent<Transform> ();
   this.acceleracion = this.velocidad * Time.deltaTime;
   }

   void Update ()
   {
   this.direcRojo.transform.Translate (Vector3.forward * this.acceleracion,
Space.World);
   }
}
```

En este caso no estoy aplicando la fórmula de aceleración igual a velocidad divido por el tiempo. En realidad he creado una variable privada para mi comodidad en donde guardo el valor del tiempo multiplicado por la velocidad. De este modo no tengo que escribir tanto dentro de la función Update.
En la función Start guardo el componente Transform dentro de la variable direcRojo y utilizo la variables aceleración de tipo float para guardar la velocidad multiplicada por el Time.deltaTime.
Dentro de la función Update en la variable que contiene la información de transformación ya no utilizamos el método position, ahora utilizamos el método Translate que nos pide un Vector3 en el que utilizamos los métodos forward, multiplicado por la variable aceleración que es la velocidad por el Time.delta.Time y el ultimo parámetro es decirle que referencia en el espacio utiliza la local (según el objeto) o la global (según el espacio). A continuación te muestro como quedaría según si utilizamos el espacio Local o Global.
Ejemplo:

```
transform.Translate(Vector3.forward, Space.Self);      //Local
transform.Translate(Vector3.up, Space.World);          //Global
```

Si lo deseas puedes variar este mismo script la dirección del Vector3 y el método de Space para hacerlo Local o Global. Es recomendable que intentes variar algunos parámetros para ver qué sucede.

Mover objetos de un punto A a un punto B

Este ejemplo es para mover un Gameobject desde un punto A que tiene un vector3 hasta un punto B que tiene otro vector3.

Para realizar esta actividad crea una nueva escena con el nombre de Escena 4 y guarda la escena en la carpeta escenas, si has seguido el capítulo desde el principio deberías tener 4 Escenas guardadas como te muestro en la siguiente imagen:

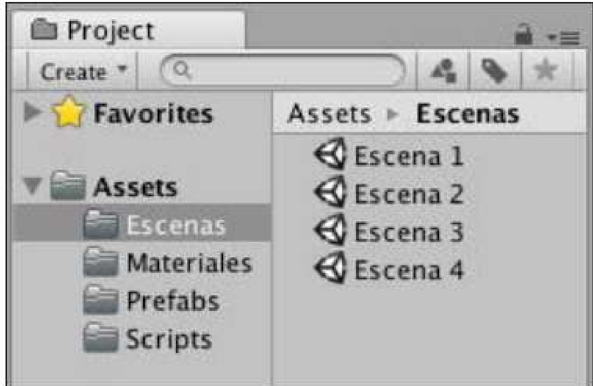


Fig. 6.43

Ahora vamos a montar la escena arrastrando los siguientes prefabs: dos cápsulas, dos plataformas y un suelo. El suelo tiene un material azul, las plataformas de color amarillo y las cápsulas un color rojo. Vamos a renombrar la cápsula con el nombre de **Player**. A las plataformas vamos a ponerles el nombre de **PosicionA** y **PosicionB.** Para que tengas una idea de como lo he montado te muestro la siguiente imagen.

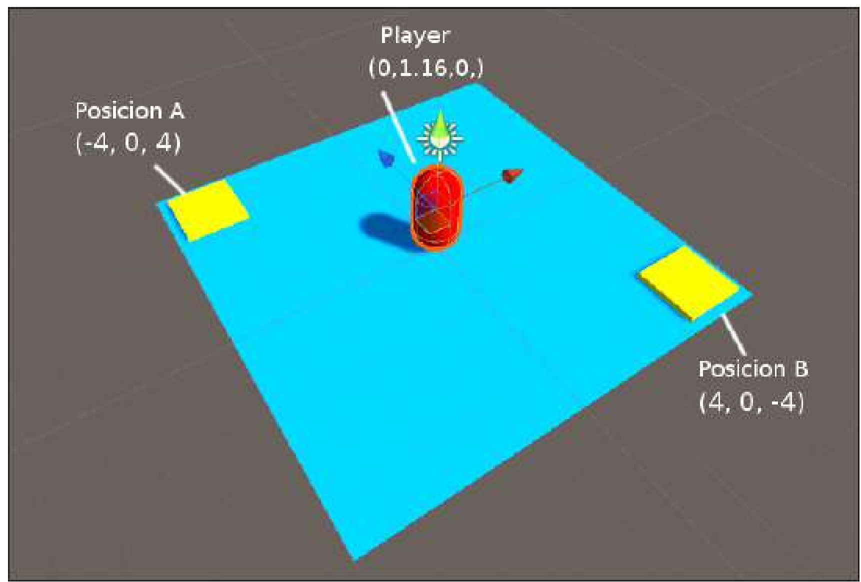


Fig. 6.44

Existen varias formas de hacer esto, y por ello te recomiendo que mires el siguiente enlace de la documentación de Unity:

https://docs.unity3d.com/ScriptReference/Vector3.html

Para este ejemplo he utilizado dos métodos de Vector3; uno es el **Lerp** y el otro es el método **MoveTowards**. Para este ejemplo vamos a crear dos scripts, uno se va a llamar **MoveLerp.cs** y el otro se va a llamar **MoveTowards.cs** en donde veremos cómo utilizar uno y el otro. Empezamos por el **MoveLerp**.

Script:MoveLerp.cs

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

public class MoveLerp : MonoBehaviour
{
   public Transform vectorPlayer;
   public GameObject posicionA;
   public GameObject posicionB;
   public Vector3 posA;
   public Vector3 posB;

   void Start ()
   {
      this.vectorPlayer = gameObject.GetComponent<Transform> ();
      this.posicionA = GameObject.Find ("PosicionA");
      this.posicionB = GameObject.Find ("PosicionB");
      this.posA = this.posicionA.transform.position;
      this.posB = this.posicionB.transform.position;

   }

   void Update ()
   {
      this.vectorPlayer.transform.position = Vector3.Lerp(this.posA,this.
posB, Time.time*0.5f);
   }
}
```

Las variables que necesitamos son una tipo Transform para guardar la posición de nuestro objeto Player y luego dos variables de tipo GameObject para guardar los objetos PosicionA y PosicionB. Para mayor comodidad he creado dos variables de tipo Vector3 en donde guardaremos la posición de los objetos PosicionA y PosicionB.
En la función Start utilizo el método GetComponent para acceder al componente Transform del objeto Player. En las variables posicionA y posicionB utilizo el método Find y el nombre de los objetos para tener referencia de ellos en la escena. Las variables this.posA y this.posB son de tipo Vector3 y por lo tanto solo pueden guardar información de este tipo así pues this.posA será igual a la posición que tiene el objeto PosicionA y this.posB será igual a la posición del objeto PosicionB.
En la función Update hemos dicho que utilizaríamos el metodo Lerp. Así que decimos que this.vectorPlayer (que es la variable que almacena el componente Transform del objeto Player) accediendo a su posición sea igual a un vector3 con el método Lerp que nos pide 3 argumentos el primero una posición inicial que la hemos guardado en la variable posA, el segundo argumento una posición final la cual la hemos guardado en la variable posB y el tercer y último argumento un valor de float para el tiempo al que le he añadido el tiempo con Time.time multiplicado por un valor de 0,5f.

En realidad el método Lerp nos permite mover un objeto de una posición a otra con una interpolación de movimiento suave. Si quieres explorar más sobre este método puedes acceder al siguiente enlace de la documentación de Unity.

https://docs.unity3d.com/ScriptReference/Vector3.Lerp.html

El resultado del script al ejecutar la escena debería ser que el objeto **Player** se posicione encima del objeto **PosicionA** y luego se desplace hacia el objeto **PosicionB**. En la siguiente imagen te muestro como se ve el componente script una vez activamos la escena.

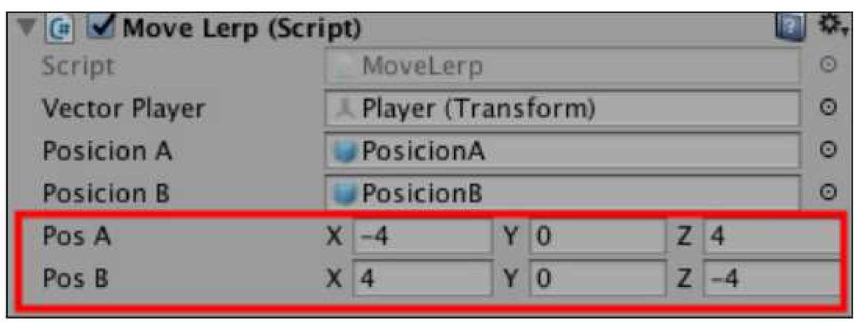


Fig. 6.45

Para el siguiente método creamos un nuevo script con el nombre de MoveTorwards si no lo hemos hecho todavía y antes de agregarlo al objeto Player asegurémonos de eliminar el componente script MoveLerp.cs porque podemos tener conflictos entre scripts.

Script: MoveTowards.cs

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
public class MoveLerp : MonoBehaviour
{
        public Transform vectorPlayer;
    public GameObject posicionA;
    public GameObject posicionB;
    public Vector3 posA;
    public Vector3 posB;

    void Start ()

    {
       this.vectorPlayer = gameObject.GetComponent<Transform> ();
       this.posicionA = GameObject.Find (“PosicionA”);
       this.posicionB = GameObject.Find (“PosicionB”);
       this.posA = this.posicionA.transform.position;
       this.posB = this.posicionB.transform.position;
```

```
        }

        void Update ()
        {
                this.vectorPlayer.transform.position = Vector3.MoveTorwards(-
this.posA,
        this.posB, Time.time*0.5f);
        }
}
```

Si lo comparamos con el anterior el script MoveLerp todo es igual exceptuando el método del Vecto3 dentro de la función Update.

El método MoveTowards mueve un punto a lo largo de una línea recta hacia un punto marcado.

La fórmula de este método es **Vector3.MoveTorwards**(Vector3 inicial, Vector3 como punto final, valor float con una cantidad máxima de distancia). Esta función la tenemos que considerar como una forma de acercarnos a un punto destino.

Si quieres ver algún otro ejemplo puedes acceder al enlace que te facilito a continuación:

https://docs.unity3d.com/2018.1/Documentation/ScriptReference/Vector3.MoveTowards.html

De todos modos estas actividades son ejercicios básicos para que vayas entendiendo y practicando con los scripts no te preocupes si no entiendes el objetivo a largo plazo, simplemente realiza las actividades y disfruta del resultado las piezas irán encajando capítulo a capítulo.

Mover objetos de ida y retorno

Para este ejemplo vamos a crear unos objetos móviles como si de un videojuego de plataformas se tratara. Para seguir bien todos los pasos vamos a crear un nueva escena como ya hemos explicado en el inicio de este capítulo y la guardamos en la carpeta Escenas con el nombre de Escena 5. En esta nueva escena vamos a utilizar de la carpeta Prefabs un suelo para tener una superficie y un Prefab plataforma.

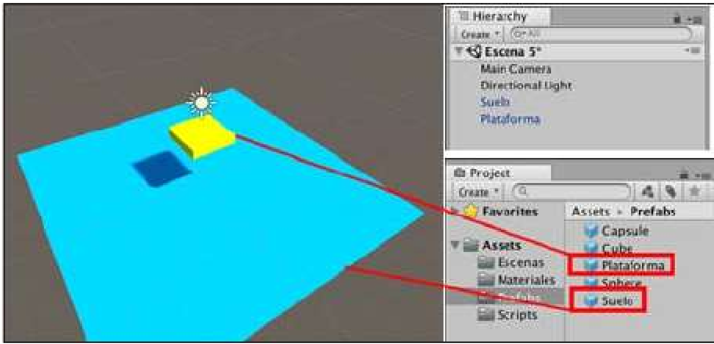


Fig. 6.46

Ahora vamos a seleccionar el objeto plataforma y lo vamos a duplicar dos veces. Para duplicarlo accedemos a la ventana Jerarquia (Hierarchy), y pulsamos el botón derecho del ratón encima del nombre plataforma y en el menú que nos aparece seleccionamos la opción duplicate.

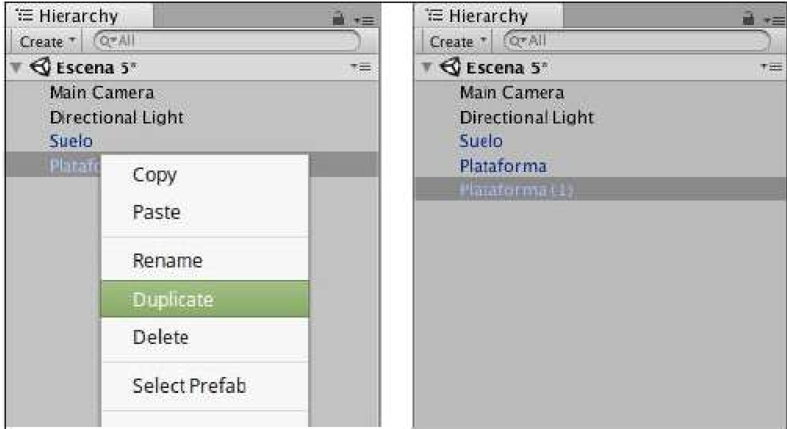


Fig. 6.47

En la imagen anterior hemos creado un duplicado de plataforma y nos hace falta otro más. Una vez que tengamos creado dos duplicados, que los puedes diferenciar porque tienen al lado del nombre una numeración entre paréntesis, vamos a ponerles nombres, a la Plataforma (1) le ponemos el nombre de PosicionA y a la Plataforma (2) el nombre de PosicionB.

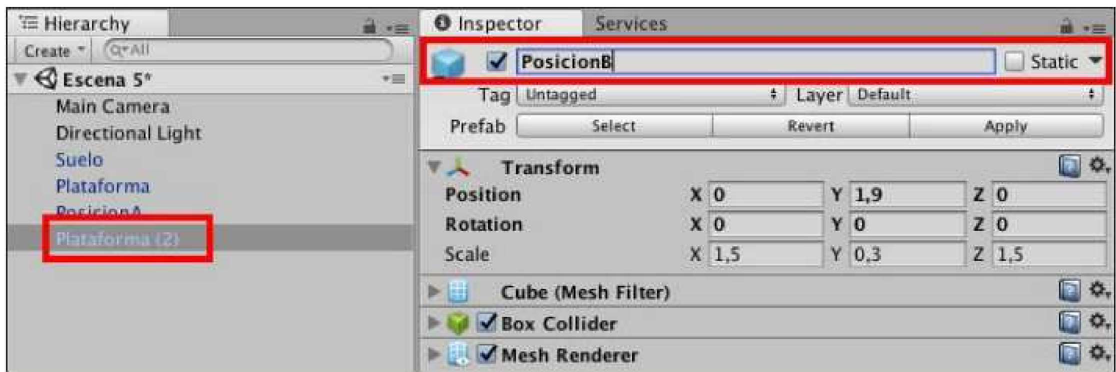


Fig. 6.48

De momento nuestra escena tiene un suelo, una Plataforma y dos objetos llamados Posición A y Posición B. Para diferenciar la Plataforma de las Posiciones a los objetos Posiciones les vamos añadir un material de color rojo. Si no lo recuerdas puedes acceder a la ventana Project en la carpeta Materiales, seleccionar el material llamado Rojo y arrastrarlo encima o bien de los nombres en la ventana Jerarquía o bien en los propios objetos dentro de la ventana escena.

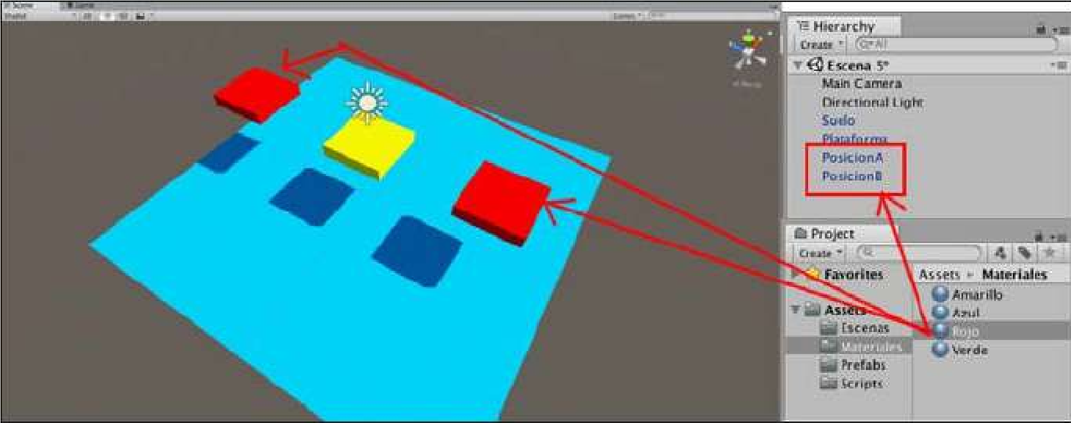


Fig. 6.49

Ahora que ya tenemos el escenario medio montado voy a proporcionarte las posiciones de cada uno de los objetos de manera que la escena quede igual que en el ejemplo que te voy a mostrar.

La plataforma que es de color amarillo tiene la posición en los ejes x,y,z (0,2,0), el objeto PosicionA tiene la posición (0,2,4), para el objeto PosicionB (0,2,-4) y el objeto suelo (0,0,0).

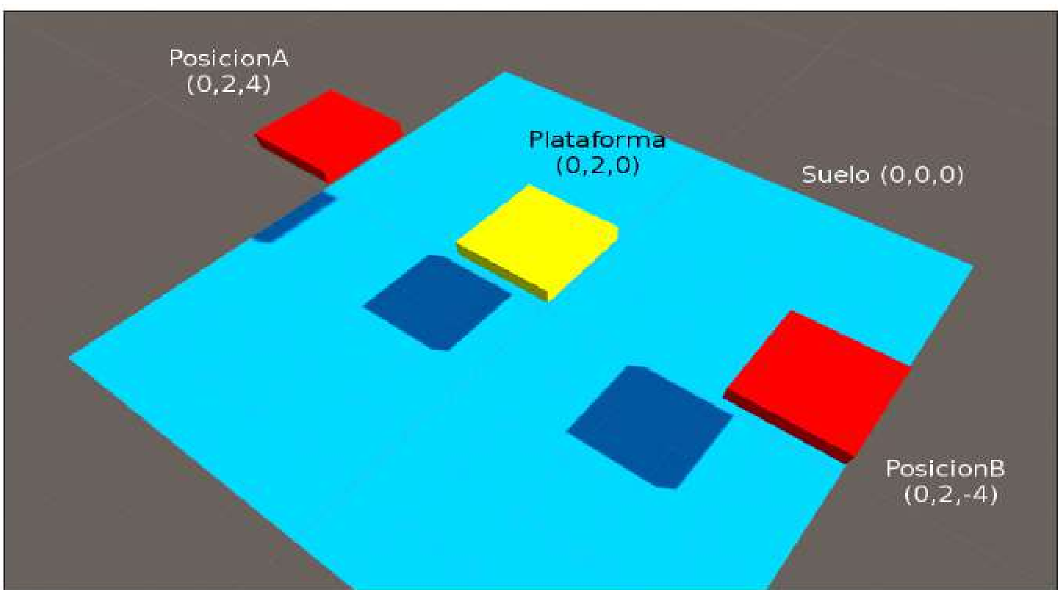


Fig. 6.50

Recuerda que las coordenadas las encontrarás seleccionando el objeto y accediendo a la ventana Inspector. En donde puedes introducir los valores haciendo clic encima de los distintos ejes, en este caso hacemos referencia a la posición.

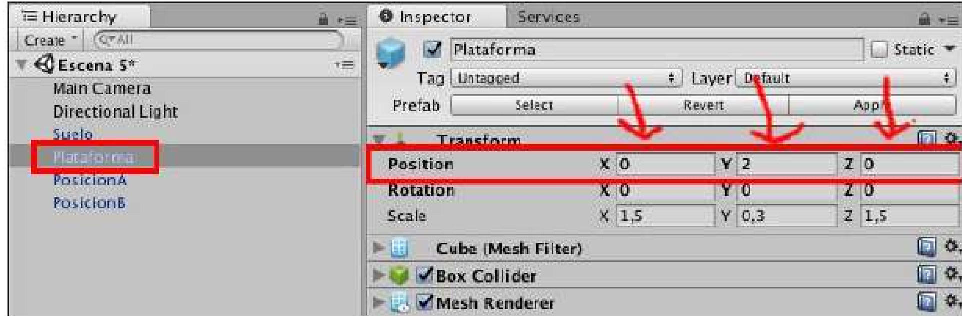


Fig. 6.51

Bien, ya tenemos nuestro escenario preparado con los nombres correctos; el objetivo es intentar que la Plataforma realice un movimiento de ida y vuelta desde el objeto PosicionA hasta el objeto PosicionB. Para ello crearemos un Script con el nombre de MovPingPong.cs y se lo añadiremos al objeto Plataforma.

Para realizar este movimiento voy a utilizar un método que viene de una estructura del UnityEngine que es Mathf con una colección de funciones matemáticas. El método se llama PingPong, esta función necesita los argumentos siguientes:

PingPong (tiempo , distancia)

Si quieres investigar más por curiosidad te facilito el enlace a la documentación:

https://docs.unity3d.com/2018.1/Documentation/ScriptReference/Mathf.PingPong.html

A continuación te muestro qué debe contener el script para que nuestra plataforma se mueva de un lugar a otro utilizando este método.

Script: MovPingPong.cs

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

public class MovPingPong : MonoBehaviour
{
   public Transform miPosicion;
   public float velocidad;
   public float rango;
   public float distancia;
   private float posX;
   private float posY;
   private float posZ;
   void Start ()
   {
      this.miPosicion = gameObject.GetComponent<Transform> ();
      this.posX = this.miPosicion.transform.position.x;
      this.posY = this.miPosicion.transform.position.y;
```

```
        this.posZ = this.miPosicion.transform.position.z;
        this.velocidad = 2f;
        this.distancia = 4.7f;
        this.rango = -2.4f;
    }

    void Update ()
    {
        this.posZ = Mathf.PingPong (Time.time*this.velocidad, this.distancia)
        +this.rango;
        this.miPosicion.transform.position = new Vector3 (posX,posY,posZ);
    }
}
```

En la parte referente a las variables he creado una variable de tipo Transform con el nombre miPosicion para guardar la información del componente Transfrorm (Este paso se realiza para mostrar de donde saco la posición del objeto). Después necesitamos tras valores de tipo float que serán públicos y los guardaremos con los nombres velocidad, rango y distancia. Las siguientes variables las he creado para mayor comodidad al utilizar la función PingPong , son privadas porque no hace falta que se muestren en el inspector y son de tipo float para almacenar un valor decimal de las posiciones x, y, z del objeto.
En la función Start primero decimos que miPosicion guarde la información del componente Transform del propio objeto. Luego igualamos las variables privadas Posx para que guarde la posición en x que tiene el objeto, la variable Posy guarda la posición y del objeto y por último la Posz guarda la posición del objeto en z.
Declaramos los siguientes valores para las variables this.velocidad que como el nombre indica es la velocidad a la que queremos que la plataforma se mueva, this.distancia que se encarga de darle una distancia de recorrido a la plataforma y this.rango que nos permite corregir el recorrido para precisar mejor las distancias.
En la función Update vamos a utilizar la posición del objeto en el eje z que ya hemos guardado en this.posz y utilizaremos la función Mathf.PingPong en donde al tiempo le multiplicaremos la velocidad y le daremos una distancia con la variables this.distancia, estos son los valores que necesita por defecto, pero para mejorar el recorrido le sumamos la variable this.rango.
Para finalizar igualamos la posición que tiene el objeto accediendo desde la variable this.miPosicion y le damos una nueva posición creando un nuevo Vector3 con tres valores float que son las variables que hemos creado. Si te fijas en this.posZ hemos aplicado la función Mathf.PingPong: esto debería darnos como resultado un movimiento de la plataforma en el eje Z.

Antes de ver el resultado te preguntarás que de dónde he sacado la distancia y el rango de las variables. En muchas ocasiones como son variables públicas estos valores puedo manipularlos desde la ventana Inspector mientras el juego se ejecuta, pero en este caso he buscado los valores de la plataforma en el eje z, cuando se aproxima al objeto PosicionA y cuando se aproxima al objeto PosicionB sin que la Plataforma llegue a tocar estos objetos.

El resultado de la posición de la Plataforma en el eje z cuando nos acercamos a PosicionA es 2,3 y cuando nos acercamos a PosicionB es -2,4. Con estos dos valores ignorando el signo menos al sumarlos me dan la distancia total que quiero que la plataforma recorra, el problema es que Mathf.PingPong empieza a calcular con valores positivos y

es por eso que le sumo un rango para corregir la distancia y que me tome el recorrido en su parte negativa. A continuación te muestro un gráfico en donde se explica cómo saco el rango de este ejemplo:

Fig. 6.52

7. Rotar objetos

Hasta ahora hemos estado jugando con el desplazamiento de los objetos y probando algunos métodos para desplazarlos, ahora vamos a entender como podemos rotar objetos. Principalmente primero tenemos que acceder al componente **Transform** como hemos hecho hasta ahora, guardando la información en una variable y si recordamos cuando queríamos acceder a su posición escribíamos **transform** un punto que enlazaba con el método y position, bien pues dentro de **Transform** podemos acceder a otros métodos y esta vez quiero mostrarte dos métodos: **rotation** que nos permite rotar el objeto en modo global y **localRotation** que nos permite rotar en modo local.

https://docs.unity3d.com/2018.1/Documentation/ScriptReference/Transform.html

En el enlace anterior tienes todos los métodos y atributos que puedes utilizar con **Transform** pero en este caso me voy a centrar en uno en concreto que es el **Transform. rotation.** Puedes acceder a la documentación accediendo al siguiente enlace.

https://docs.unity3d.com/2018.1/Documentation/ScriptReference/Transform-rotation.html

Si accedemos a la documentación sobre este método nos encontramos con los Quaternions que son usados para representar las rotaciones. Los Quaternions se basan en números complejos y esto nos permite poder modificar el componente de rotación de un objeto utilizando los ejes (x, y, z, w). No voy a entrar en mucho detalle sobre el tema pero sí que vamos a ver como se utilizan. Me gustaría que conocieras un método de Quaternion que es el Euler que nos pide 3 valores decimales para darle una rotación a un objeto de la siguiente manera Quaternion.Euler (float x, float y, float z). A continuación de dejo un enlace de con la documentación sobre el método Euler:

https://docs.unity3d.com/2018.1/Documentation/ScriptReference/Quaternion.Euler.html

Rotar objeto con transform.rotation

Para el siguiente ejemplo crea una nueva escena y guárdala en la carpeta escenas con el nombre Escena 6. Para nuestra escena solo vamos a utilizar un cubo, puedes arrastrar de la carpeta Prefabs el objeto Cube que tiene un material de color verde.

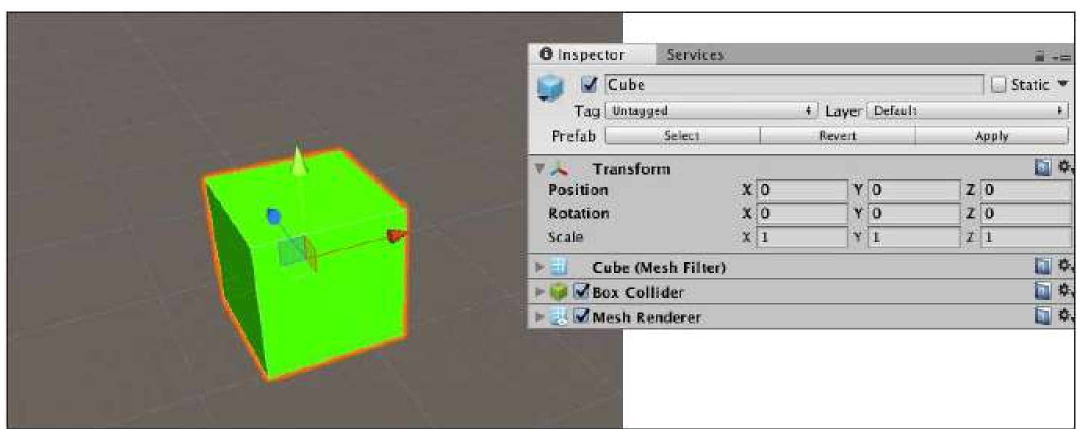


Fig. 6.53

El ejercicio consiste en hacer rotar el objeto mediante programación con el método **transform.rotation** al que deberemos utilizar un Quaternion. Para ello crea un nuevo script con el nombre **Rotacion.cs** y lo agregas al objeto **Cube** de la escena.

Script: Rotacion.cs

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

public class Rotaciones : MonoBehaviour
{
   public Transform miRot;
   public float velocidad;

   void Start ()
   {
      this.miRot = gameObject.GetComponent<Transform> ();
      this.velocidad=45f;
   }
   void Update ()
```

```
    {
        this.miRot.transform.rotation = Quaternion.Euler (0f,this.velocidad*Ti-
me.time,0f);
    }
}
```

Como siempre creamos una variable de tipo Transform para guardar la información del transform del gameObject, en este caso la he llamado miRot. También he creado una variable velocidad de tipo float para controlar la velocidad de rotación.
Dentro de la función Start guardo la información del Transform del gameObject Cube dentro de la variable miRot y le doy un valor a la velocidad.
Dentro de la función Update accedo al componente transform pero esta vez a la que se encarga de rotar que es rotation. Para rotation debo darle un Quaternion y en este caso he utilizado un Quaternion.Euler, que me permite ponerle valores en x y z . En este caso le he puesto el valor que contiene la variable velocidad multiplicado por Time.time en el eje y .

Ahora si volvemos a Unity verás cómo en el inspector te aparecen las variables y cuando ejecutes el juego y el cubo empezara a rotar en su eje y.

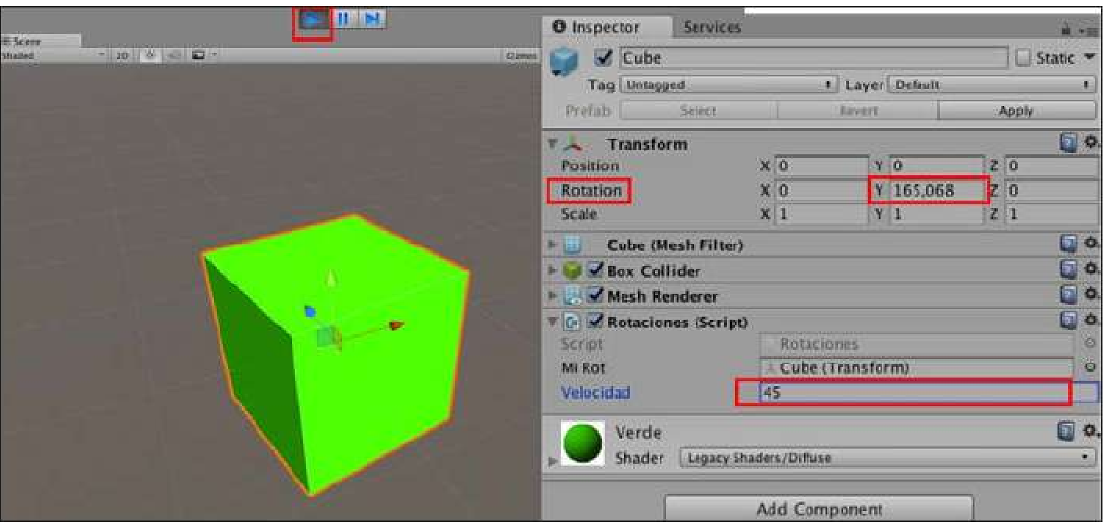


Fig. 6.54

Rotar objeto con transform.Rotate

A diferencia del anterior este método nos pide dos argumentos para rotar un objeto:

- Un vector 3 con un **eulerAngles** en donde especificamos los grados de rotación para cada eje.
- Un valor de **Space** que puede ser Global Space.World o puede ser Local Space.Self.

Este aspecto nos permite especificar a cómo queremos la rotación en relación al mundo o al propio objeto.

Puedes acceder a la documentación de este método desde el enlace que te facilito a continuación:

https://docs.unity3d.com/2018.1/Documentation/ScriptReference/Transform.Rotate.html

Para este ejemplo elimina el cubo que teníamos en escena, seleccionándolo desde la ventana Jerarquía haciendo clic en el nombre con el botón derecho y seleccionando la opción **Delete.**

Añade a la escena dos cubos de la carpeta Prefabs y les cambiamos los materiales arrastrando el material Azul encima de uno de los cubos y arrastrando el material Amarillo encima del otro cubo. Ponles como nombre a los cubos el color que tienen Azul y Amarillo. Para que tengas más claro como debe quedar el escenario mira la siguiente imagen:

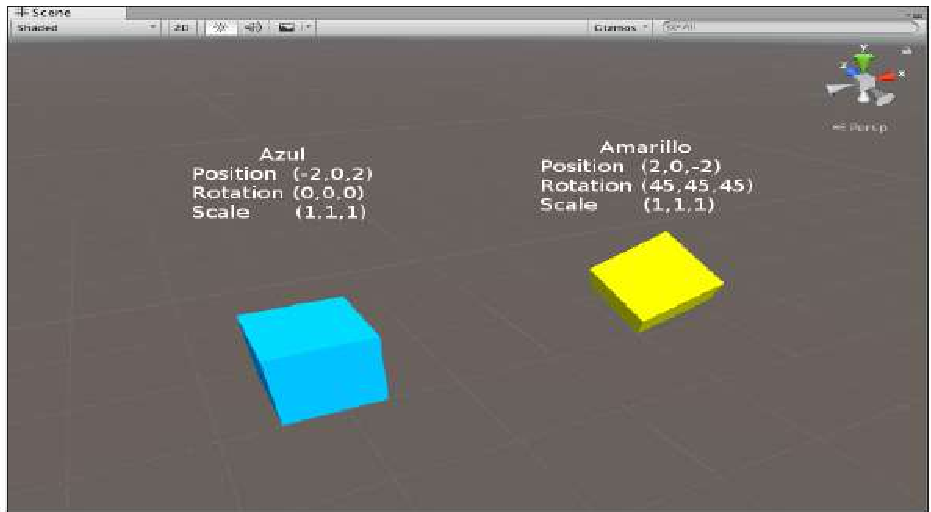


Fig. 6.55

Tenemos un cubo con el nombre Azul y otro cubo con el nombre Amarillo. Las posiciones son las que te muestra la imagen: para el cubo Azul la posición es (-2,0,2) y para su rotación es cero en todos sus ejes. Para el cubo amarillo su posición es (2,0,-2) y para su rotación le damos el valor 45 para todos sus ejes.

Ahora crearemos un script con el nombre Rotate.cs, recuerda de que todos los scripts debes guardarlos en la carpeta scripts. El script Rotate.cs lo arrastramos encima del cubo Azul o encima del nombre Azul en la ventana de Jerarquía. A continuación vamos a crear una rotación utilizando el **tansform.Rotate** en modo global para el cubo Azul y en modo local para el cubo Amarillo

Script: Rotate.cs

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

```
public class Rotaciones : MonoBehaviour
{
   public Transform rotAzul;
   public GameObject miAmarillo;
   public float grados;

   void Start ()
   {
      this.miAmarillo = GameObject.Find("Amarillo");
      this.rotAzul = gameObject.GetComponent<Transform> ();
      this.grados = 90f;
   }

   void Update()
   {
      this.miAmarillo.transform.Rotate(new Vector3(0f,this.grados*Time.delta-
Time, 0f), Space.Self);
      this.rotAzul.transform.Rotate(new Vector (0f, this.grados*Time.deltaTi-
me, 0f), Space.World);
   }
}
```

Creamos una variable pública de tipo GameObject para referenciar el objeto Amarillo por eso mismo le he puesto el nombre de miAmarillo. Otra variable para guardar el componente Transform del objeto en si que es el cubo azul por eso le he puestro de nombre rotAzul y para finalizar una variable pública de tipo float con el nombre de grados, para poder controlar cuantos grados giran en la animación.
En la función Start guardamos la referencia del el objeto Amarillo con el método Find en la variable miAmarillo. En la variable rotAzul guardamos la información del componente Transform y para finalizar le doy el valor de 90 grados a la variable this.grados.
Dentro de la función Update utilizamos la variable this.miAmarillo y en este caso como ya estamos dentro del objeto podemos acceder a su componente transform.Rotate en donde le daremos una nuevo Vector3 donde quiero que rote en el eje y, por eso mismo el primer valor es 0 y el segundo le doy el valor de la variable this.grados multiplicado por Time.delta.Time para que me tome la rotación en segundos. El eje z también le damos un valor de 0 cerrando el paréntesis y después de coma le damos el argumento Space en donde para mi Amarillo quiero que tome la referencia de rotación local por eso utilizo Space.Self. Para la variable this.rotAzul que hace referencia al cubo Azul utilizamos la misma estructura pero en el argumento Space le ponemos World para que utilice la rotación en modo Global.

Una vez guardado el script vamos a Unity y cuando ejecutes el juego veras como los cubos empiezan a rotar. El cubo Azul rota en modo Global y el cubo Amarillo en modo Local.

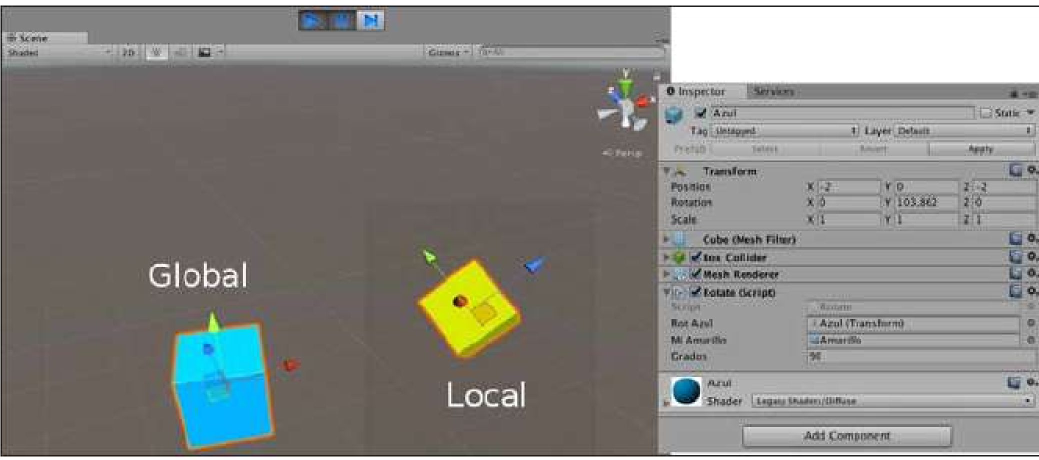


Fig. 6.56

8. Escalar objetos

Este es el último de los componentes que veremos en este capítulo, utilizaremos la escena anterior, no hace falta que borres ninguno de los objetos anteriores. Añadimos un cubo de la carpeta de los Prefabs a nuestra escena y desde la ventana Inspector nos aseguramos de que este nuevo cubo está posicionado en el centro de la escena Position(0, 0, 0).

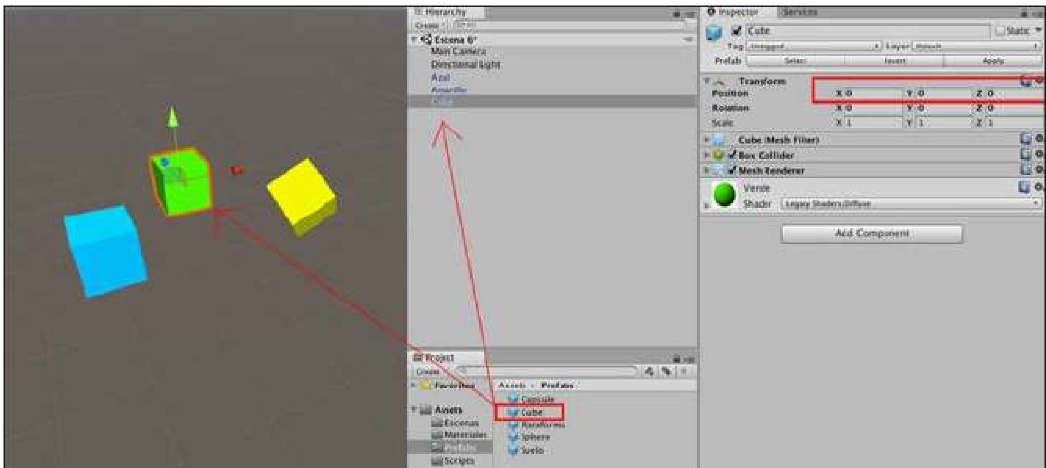


Fig. 6.57

Vamos a crear un nuevo script con nombre Escalar y vamos a agregarlo a nuestro cubo verde. Para escalar un objeto debemos mirar en la documentación de Unity el apartado de Transform. Si lo prefieres puedes acceder desde el enlace que te proporciono a continuación.

https://docs.unity3d.com/2018.1/Documentation/ScriptReference/Transform.html

Si miras la documentación encontrarás en el apartado de Propiedades la opción localScale. Esta propiedad nos permite escalar el objeto mediante un vector3. Para realizar

la escala de nuestro cubo verde de una forma progresiva y animada escribiremos el siguiente script:

Script: Escalar.cs

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

public class Escalar : MonoBehaviour
{
   public Transform escalaCubo;
   public float velocidad;

   void Start ()
   {
      this.escalaCubo = gameObject.GetComponent<Transform> ();
      this.velocidad = 2f;
   }

   void Update ()
   {
      this.escalaCubo.transform.localScale += new Vector3 (0.5f, 0.5f,
   0.5f ) *this.velocidad*Time.deltaTime;
   }
}
```

En este caso sigo creando una variable pública para almacenar el componente Transform para seguir practicando como se accede a los componentes. También he creado una variable de tipo float para poder controlar la velocidad a la que se escala.
Dentro de la función Start utilizo la variables this.escalarCubo para acceder a el componente Transform del propio objeto. También le doy un valor de 2f para la velocidad.

Dentro de la función Update utilizaremos la variable this.escalaCubo para acceder a la propiedad localScale y utilizaremos la simbología de += para sumarle el propio vector localScale que es (1,1,1) con el nuevo vector3 que le agregamos que es (0.5f,0.5f,0.5f), a este nuevo vector lo multiplico por la velocidad que a su vez está multiplicado por Time.deltaTime para que tome el tiempo en segundos.

Al ejecutar el juego con el script agregado al cubo verde, verás cómo el cubo empieza a crecer infinitamente. Es un ejemplo muy sencillo que puedes probar hacer con cualquier objeto que crees. Las siguientes imágenes muestran cómo queda la escena antes de ejecutar el juego y con el juego en ejecución.

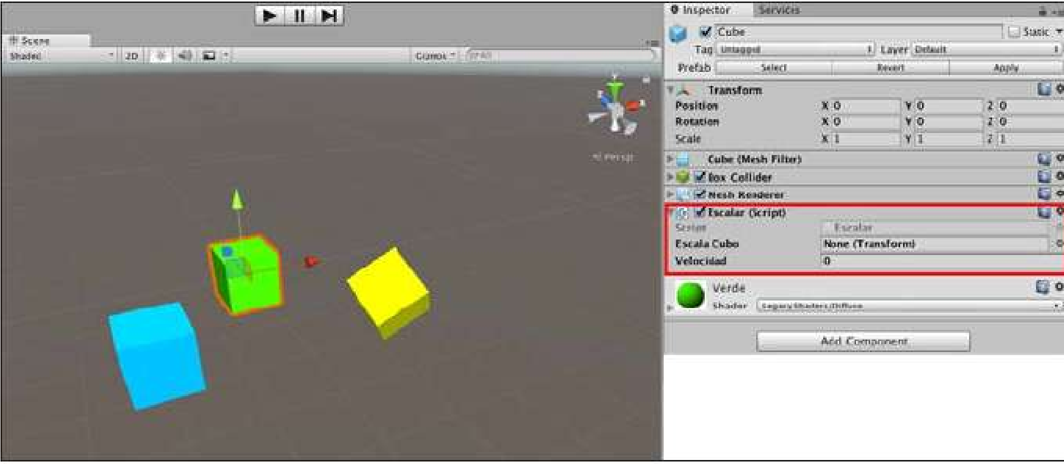


Fig. 6.58

En la siguiente imagen el cubo con el juego en ejecución:

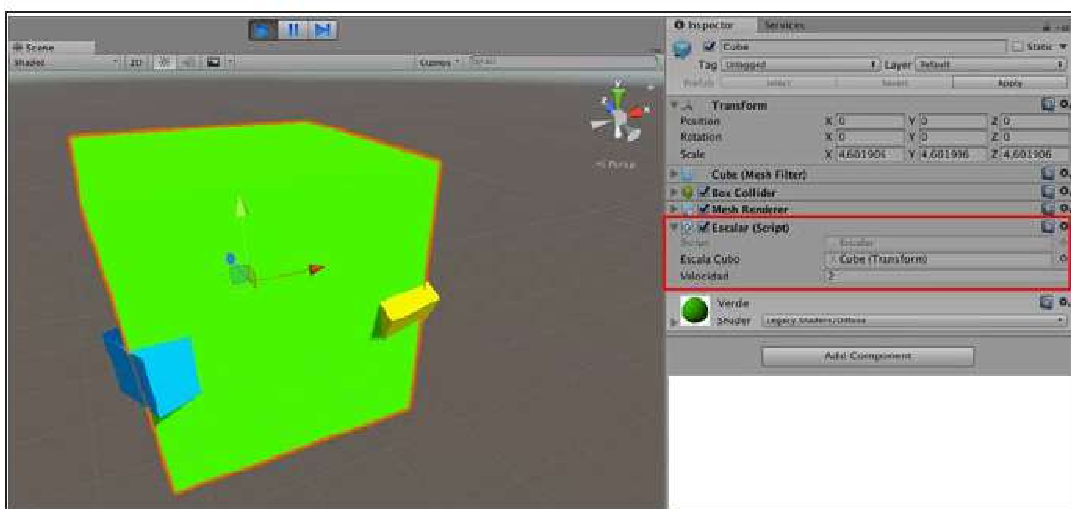


Fig. 6.59

En este capítulo hemos empezado a utilizar la programación para acceder a los componentes, en especial al componente Transform, que nos permite mover, rotar y escalar los objetos. Los métodos que hemos utilizado son para mostrarte que disponemos de una documentación con muchas posibilidades para descubrir. Si has realizado todas las actividades o casi todas de este capítulo te felicito y te invito a pasar al próximo capítulo en donde empezaremos a controlar los objetos mediante teclas.

Capítulo 7

Creación de un Player en C#

- Introducción
- Character Controller
- Movimientos del Character Controller
- Los Inputs
- Mover y rotar nuestro Character Controller
- Saltar obstáculos
- Rigidbody
- Mover un Rigidbody
- Controlar el movimiento de un Rigidbody
- Añadir un salto al Rigidbody
- Seguimiento simple de nuestra cámara
- Destruir objetos con colisiones
- Tele-transportación con Triggers
- Proyecto final

1. Introducción

Este capítulo es uno de los más interesantes porque ya no solo vamos a mover, rotar y escalar objetos, ahora los vamos a controlar mediante **inputs** o entradas de teclado. Si has comprendido bien el capítulo anterior en donde se explica cómo se accedena los **gameObjects** y como manejar las transformaciones a modo básico mediante scripts, en este capítulo vas a divertirte mucho.

Uno de los objetivos de este capítulo es que comprendas bien el componente **Character Controller** y cómo podemos utilizarlo para mover a tu personaje mediante entradas de teclado. En un principio trabajaremos con un objeto simple como la cápsula.

El otro objetivo es que puedas crear los controles básicos (también con entradas de teclado) de un objeto con Rigidbody y puedas crear tu propio Player básico para que puedas moverlo por un escenario que en este caso en particular será una pelota.

Para finalizar el capítulo aplicaremos todo lo aprendido creando una escena donde una pelota recolectará monedas (sin contarlas) y crearemos una zona de reinicio por si esta pelota cae en el vacío.

Para seguir el capitulo en el material que acompaña el libro tenemos dentro de la carpeta Proyecto_7 dos paquetes para importar: uno es el **Assets_Capitulo_7.unitypackage** que es el que vamos a necesitar para el principio, y el otro es el **Práctica_Capitulo_7.unitypackage** que es el que importaremos al final del capítulo.

Para empezar crea un nuevo proyecto 3d en Unity y ponle el nombre que quieras, yo te propongo como nombre Capitulo 7. Una vez lo hayas creado importamos el primer paquete **Assets_Capitulo_7.unitypackage** del material que acompaña la obra. Para importar accedemos al menú principal **Assets > Import Package > CustomPackage...** Se abrirá una ventana de tu sistema en donde debes buscar donde está guardado el material adicional de la obra. Una vez lo encuentres en la carpeta Proyecto_7 selecciona el paquete con el nombre **Assets_Capitulo_7.unitypackage;** se nos aparece una nueva ventana en donde podemos ver el contenido del paquete y Unity nos da la posibilidad de seleccionar qué queremos importar. En este paquete disponemos de dos carpetas, una con los materiales y la otra con Prefabs; déjalo todo seleccionado. Para que ejecute la importación hacemos clic encima del botón Import que se encuentra en la parte inferior derecha.

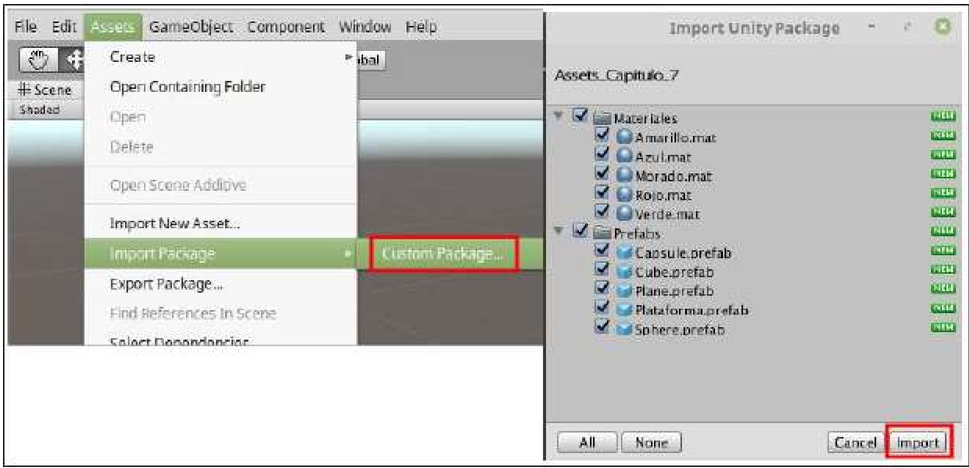


Fig. 7.1

En nuestra ventana Project deberían aparecer estas dos carpetas Materiales y Prefabs. En la imagen anterior puedes ver qué contiene cada carpeta. Para continuar con el capítulo solo nos queda crear dos carpetas más, una con el nombre Scripts y otra con el nombre Escenas. Para crear estas carpetas asegúrate de tener seleccionada la carpeta Assets dentro de la ventana Project y después accede a la opción **Create > Folder** como te muestro en la siguiente imagen:

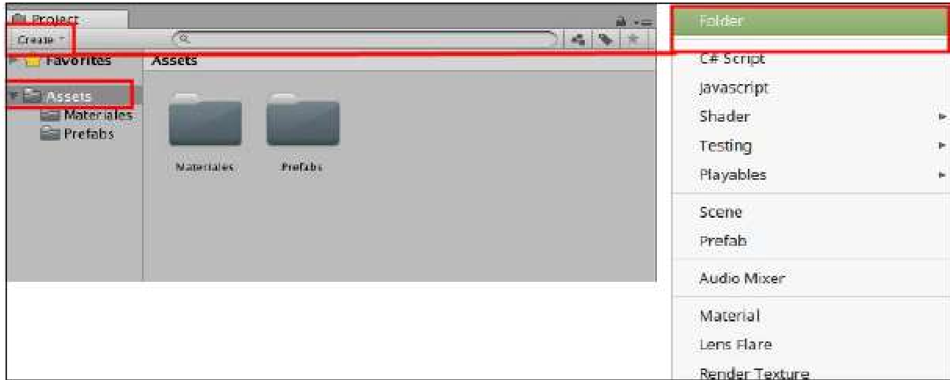


Fig. 7.2

Con la acción anterior conseguimos crear una nueva carpeta a la que debemos darle un nombre, por ejemplo Scripts. Realiza una vez más este proceso para crear una carpeta más con el nombre Escenas como te muestro a continuación.

Fig. 7.3

Para finalizar, guarda el proyecto accediendo al menú principal **File> Save Project** y guarda la escena accediendo al menú principal **File> Save Scene**, en este caso se abrirá una ventana del sistema; selecciona la carpeta Escenas y guarda la escena con el nombre **Escena_1**. Ahora en Unity ya puedes ver cómo en la ventana **Project** en la carpeta Escenas se ha guardado la escena.

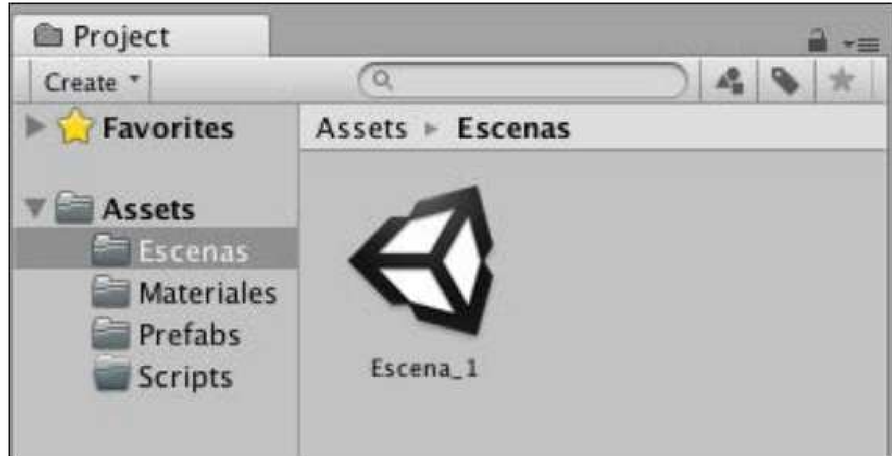


Fig. 7.4

2. Character Controller

En el capítulo anterior hemos visto cómo podemos mover nuestros objetos de una forma automática. Para ver cómo podemos controlar nuestros objetos de forma manual debemos conocer el componente **Character Controller**.

El **Character Controller** es un componente que se utiliza como su nombre indica para controlar un Player o personaje en primera y tercera persona sin utilizar físicas, que también veremos más adelante.

Para empezar necesitamos un plano y una cápsula. Accede a la ventana **Project** en la carpeta Prefabs y arrastra el Prefab con nombre Plane o bien al escenario o bien a la ventana Jerarquía. Luego hacemos lo mismo con el Prefab Capsule. Puedes ver como queda en la siguiente imagen.

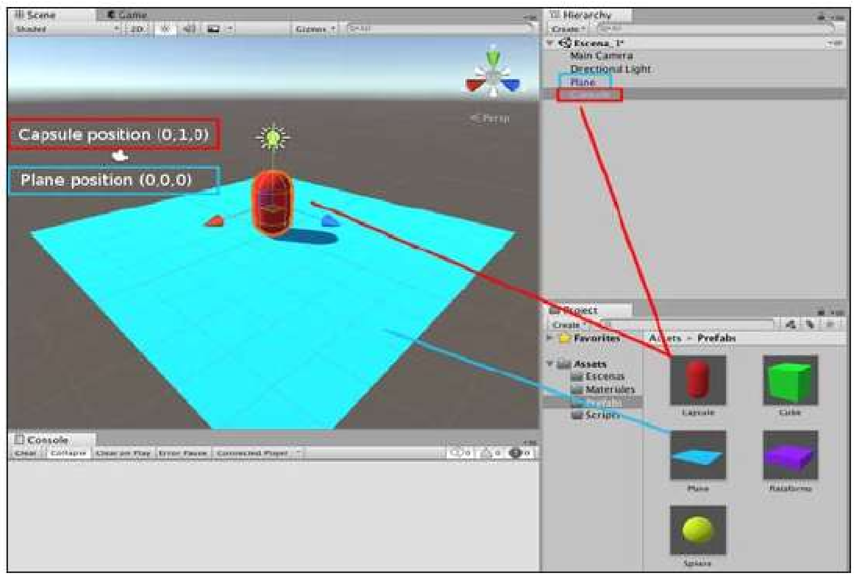


Fig. 7.5

Si seleccionamos la cápsula haciendo clic encima de su nombre en la ventana Jerarquía, y en la ventana Inspector veremos sus componentes. Primero cambiaremos el nombre de Capsule por el de Player como te muestro en la siguiente imagen:

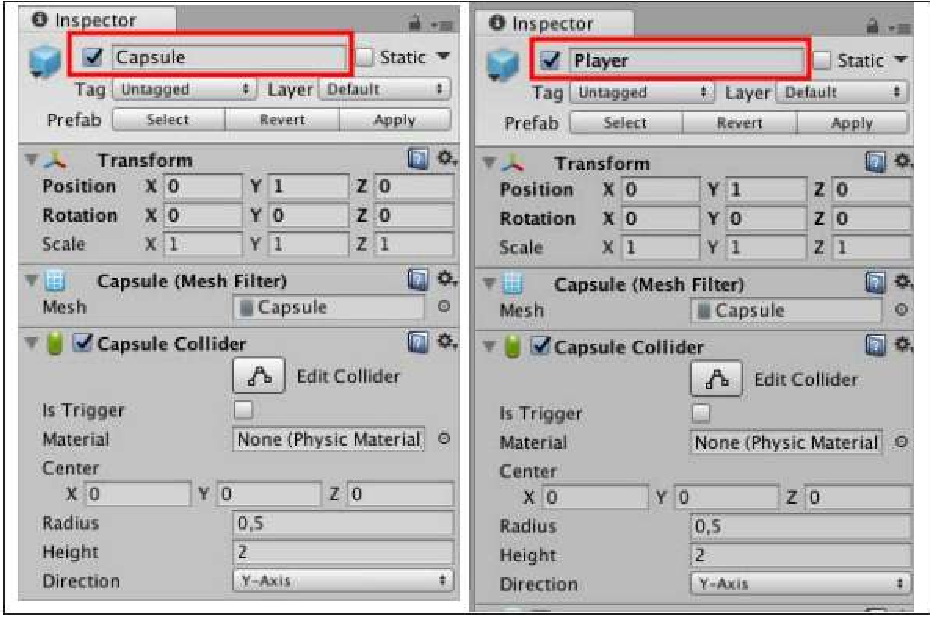


Fig. 7.6

A la cápsula que ahora tiene el mombre de Player vamos a añadirle un nuevo componente desde la ventana I**nspector>Add Component> Physics > Character Controller**. Ahora el objeto player ya dispone del componente **Character Controller.**

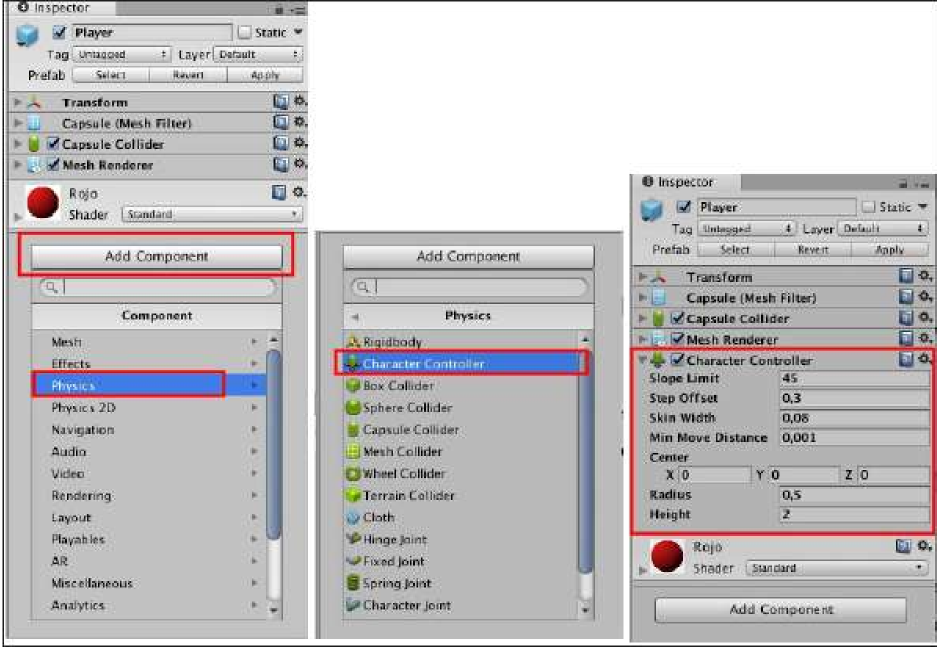


Fig. 7.7

A continuación te detallo las distintas funciones que caracterizan este componente. No tienes por qué sabértelas de memoria, te las detallo a modo de consulta para cuando sepas cómo utilizarlo.

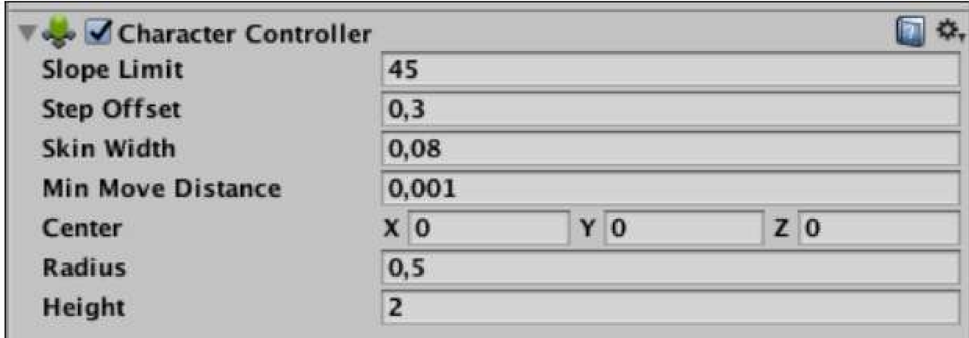


Fig. 7.8

Propiedades	Función
Slope Limit	Limita el colisionador al subir sólo pendientes menos empinadas (en grados) que el valor indicado.
Step Offset	El personaje subirá una escalera sólo si está más cerca del suelo que el valor indicado. Este valor no debe ser mayor que la altura del **Character Controller** o generará un error.
Skin widht	Dos colisionadores pueden penetrar entre sí tan profundamente como su ancho de piel. Una anchura baja de la piel puede hacer que el personaje se quede pegado. Un buen ajuste seria el 10% del radio.
Min Move Distance	Si el personaje trata de moverse por debajo del valor indicado, no se moverá en absoluto. En la mayoría de situaciones este valor debe dejarse en 0.
Center	Esta propiedad ayuda a compensar el colisionador en forma de capsula en el espacio global, sin que afecte en el giro del personaje.
Radius	Longitud del radio del colisionador. Es la anchura del colisionador.
Height	Es la altura del colisionador. Es decir cuanto mayor es el valor, más se alarga el colisionador a lo largo del eje Y.

El Character controller se asemeja a un Collider en forma de cápsula, y que vamos a utilizarlo para crear movimiento. A continuación vamos a ver cómo podemos darle movimiento mediante scripts y entender cómo funciona.

3. Movimientos del Character Contoller

Como siempre y antes de entrar en detalle dale un rápido vistazo a la documentación de Unity para ver que métodos y atributos tiene este componente. Puedes acceder desde el enlace que te facilito a continuación.

https://docs.unity3d.com/2018.1/Documentation/ScriptReference/CharacterController.html

Ya tenemos la escena, nuestro Player con su componente **Character Controller** , ahora vamos a ver como podemos acceder a este componente para poder darle un movimiento desde un script.

Primero debes crear un script dentro de la carpeta scripts con el nombre ControladorPlayer, selecciona la carpeta Scripts dentro de la ventana Project y pulsa en el botón Create en la parte superior de la ventana y selecciona Create > C# Script se creara un nuevo script al que seguidamente debes ponerle el nombre de ControladorPlayer.

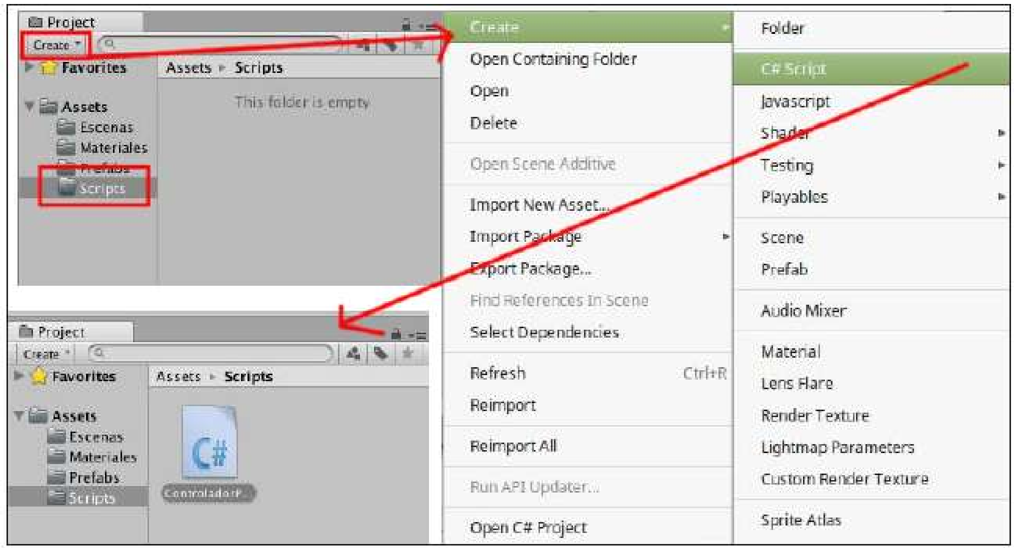


Fig. 7.9

Este Script lo arrastramos encima del objeto Player en la ventana Escena o encima del nombre Player en la ventana Jerarquía para añadirle el script. Una vez añadido hacemos doble clic encima del Script ControladorPlayer.cs y automáticamente se nos abrirá el editor Monodevelop con el que podremos editar el script.

Script: ControladorPlayer.cs

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

public class ControladorPlayer : MonoBehaviour
```

```
{
   public CharacterController controlador;

   void Start ()
   {
      this.controlador = gameObject.GetComponent<CharacterController> ();
   }
}
```

Primero creamos una variable pública de tipo Character Controller con el nombre controlador. Esta variable es donde guardaremos la información del componente CharacterController.
En la función Start() accedemos al componente y guardamos la información en la variable.
Recuerda que hay que guardar el script desde Monodevelop para que se ejecuten los cambios.

Bien de momento hemos accedido al componente CharacterController ahora volviendo a la documentación de Unity vemos que CharacterController tiene dos métodos públicos para mover los objetos; **Move y SimpleMove.** A continuación vamos a ver como podemos utilizarlos.

Mover objetos con SimpleMove dirección Global

Seguimos con la misma escena y el mismo script, vamos a mover nuestro player con la función **SimpleMove.** Esta función nos permite mover un objeto con la siguiente fórmula: **SimpleMove(Vector 3 speed)**; los argumentos son un vector 3 en donde le daremos un valor a uno de los 3 ejes (x, y, z) , para que se mueva en ese eje en concreto. El otro argumento es multiplicar por un valor de velocidad .

Script: ControladorPlayer.cs

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

public class ControladorPlayer : MonoBehaviour
{
   public CharacterController controlador;
   public float speed;

   void Start ()
   {
      this.controlador = gameObject.GetComponent<CharacterController> ();
      this.speed=15f;
```

```
    }
    void Update ()
    {
        this.controlador.SimpleMove (Vector3.right * this.speed*Time.
deltaTime);
    }
}
```

Seguimos con el mismo script pero esta vez hemos añadido una nueva variable de tipo float con nombre speed y en la función Start() le he dado el valor de 15f. Puedes ponerle el valor que quieras, este valor es orientativo para el ejemplo.

En la función Update vamos a utilizar el método SimpleMove. Para acceder al método primero ponemos la variable this.controlador que almacena la información del componente CharacterController y accedemos a su método poniendo un punto y SimpleMove. SimpleMove nos pide un Vector3, en este caso le he dado un Vector3.right para que avance por el eje (X), recuerda que Vector3.right es lo mismo que Vector3(1,0,0). El siguiente argumento lo he multiplicado por la variable de tipo float speed que es 15, que a su vez también he multiplicado por Time.deltaTime para que la animación se ejecute en segundos. Recuerda que hay que guardar el script desde Monodevelop para que se ejecuten los cambios.

Si todo es correcto cuando volvamos a Unity veremos en la ventana Inspector las variables públicas y cuando ejecutemos el juego nuestro Player se desplazará en dirección al eje x.

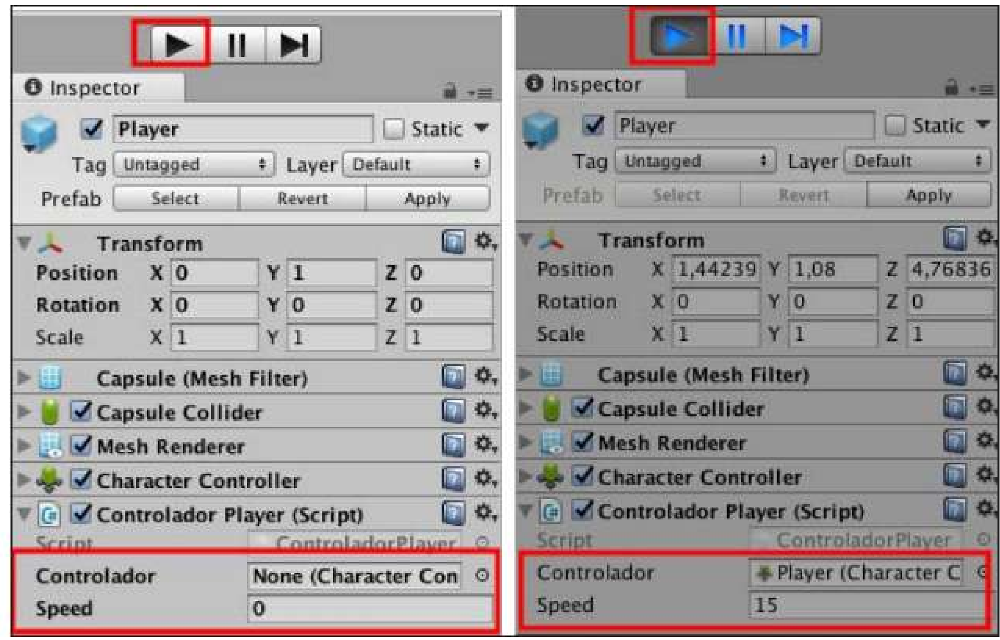


Fig. 7.10

En este ejemplo es un movimiento simple en modo global, porque si rotáramos el objeto en el eje (Y) por ejemplo 45 grados a la izquierda y ejecutamos el juego veremos

que el objeto sigue desplazándose en la misma dirección que en el ejemplo anterior, ignorando completamente el eje de coordenadas local del objeto.

Fig. 7.11

Mover objetos con SimpleMove dirección Local

Para hacer que nuestro objeto tome la dirección en modo local utilizaremos el método **transform.TransformDirection** antes del Vector 3, como te muestro a continuación con el mismo script:

Script: ControladorPlayer

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

public class ControladorPlayer : MonoBehaviour
{
    public CharacterController controlador;
    public float speed;

    void Start ()
    {
        this.controlador = gameObject.GetComponent<CharacterController> ();
        this.speed=15f;
    }
    void Update ()
    {
```

```
      this.controlador.SimpleMove (transform.TransformDirection (Vector3.ri-
ght *             this.speed*Time.deltaTime));
   }
}
```

Dentro del paréntesis de SimpleMove accedemos a la transformación de Direccion y abrimos un nuevo paréntesis donde ponemos la fórmula de SimpleMove.
Recuerda que hay que guardar el script desde Monodevelop para que se ejecuten los cambios.

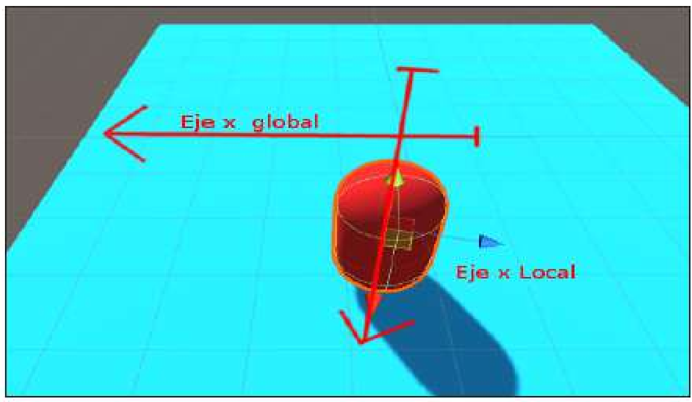


Fig. 7.12

Si has escrito el script correctamente y lo has guardado, nuestro Player debería moverse siempre en dirección al eje x local. Puedes comprobarlo seleccionando el objeto Player y cambiarle el valor de Rotación en el eje Y . Después ejecuta el juego y verás como la cápsula toma una dirección distinta al del ejemplo anterior.

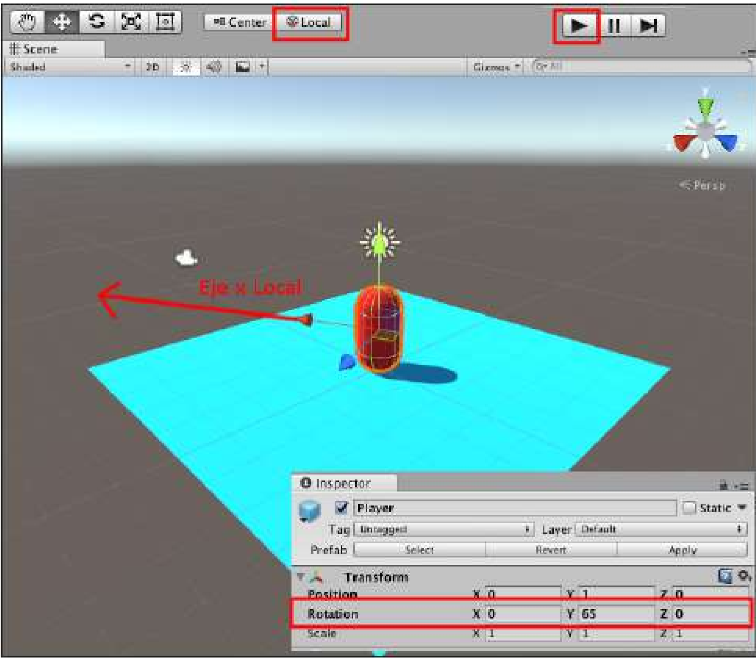


Fig. 7.13

Mover objetos con Move dirección local

Este método es parecido al anterior, con la diferencia de que en este no se le aplica gravedad. Es decir, en los ejemplos anteriores nuestro objeto tocaba el suelo y se movía, con el método Move si no le aplicamos algún tipo de fuerza para que toque el suelo la animación se ejecutará en la posición en la que se encuentre el objeto en ese momento. Dicho de otra forma, tenemos que crear una fuerza que haga de gravedad.

Para esta función necesitaremos una variable para la dirección de tipo Vector3, otra para la velocidad de tipo float y una variable para la gravedad de tipo float.

Para el ejemplo primero vamos a eliminar del objeto Player el script **ControladorPlayer** de la ventana Inspector; para ello selecciona el objeto Player y, en su ventana Inspector en el componente Script hacemos clic encima del icono con forma de engranaje y seleccionamos **Remove component.** Para que lo tengas más claro te lo muestro en la siguiente imagen:

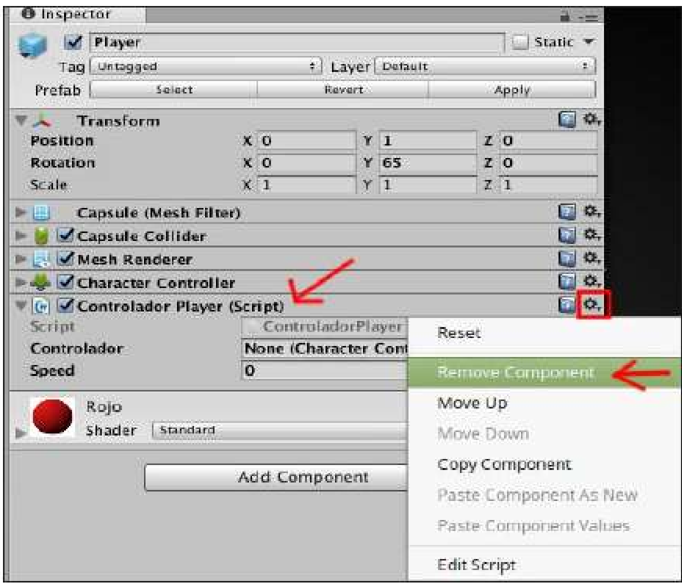


Fig. 7.14

Ahora podemos crear un nuevo script llamado **ControladorPlayerMove.cs**. Recuerda guardar todos los scripts que vayamos creando en la carpeta Scripts. Arrastra este nuevo script encima del objeto **Player** como hemos hecho en el ejemplo anterior y por último haz doble clic encima del script para editarlo en **Monodevelop**.

Script: ControladorPlayerMove.cs

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

public class ControladorPlayer : MonoBehaviour
```

```
{
    public CharacterController controlador;
    public float speed;
    public float gravedad;
    public Vector3 direccion;

    void Start ()
    {
        this.controlador = gameObject.GetComponent<CharacterController> ();
        this.speed = 2f;
        this.gravedad = -9.8f;
    }

    void Update ()
    {
        this.direccion = transform.TransformDirection(Vector3.right);
        this.direccion.y = this.gravedad*Time.deltaTime;
        this.controlador.Move(this.direccion * this.speed * Time.deltaTime);
    }
}
```

Primero creamos una variable pública de tipo CharacterController que llamaremos controlador y otra de tipo float para la velocidad llamada speed como en el ejemplo anterior. Ahora crearemos dos variables también públicas, una de tipo float que será para almacenar el valor de la gravedad y la otra variable de tipo vector3 con nombre dirección para guardar la dirección en la que queremos que se desplace nuestro objeto.
En la función Start() referenciamos la variable this.controlador para que acceda al componente CharacterController de nuestro objeto. A la velocidad le he dado para este ejemplo un valor 2f, pero puedes ponerle el valor que creas conveniente. Por último le damos el valor negativo a la variable gravedad para que sea una fuerza que empuje hacia abajo y no hacia arriba.

En la función Update() he utilizado primero la variable dirección para almacenar la transformación local del vector3.right que hace referencia al eje(x). Después he utilizado la misma variable dirección en el eje y, para que tome el valor de gravedad y como estamos dentro del Update lo he multiplicado por Time-deltaTime para que tome de referencia el tiempo en segundos.

Para finalizar simplemente accedemos al método Move y substituimos la transformacion del Vector3 por la variable direccción , los demás argumentos queda igual multiplicar por speed y por Time.deltaTime . Recuerda que hay que guardar el script desde Monodevelop para que se ejecuten los cambios.

Algo muy gracioso que te va a suceder cuando ejecutes el script es que cuando el Player se salgas del plano va a caer lentamente y avanzando al mismo tiempo. Recuerda que en el script hemos tomado de referencia para desplazar el objeto el eje x local. A continuación te muestro cómo se vería la ventana inspector, sin ejecutar y cuando el juego se ejecuta.

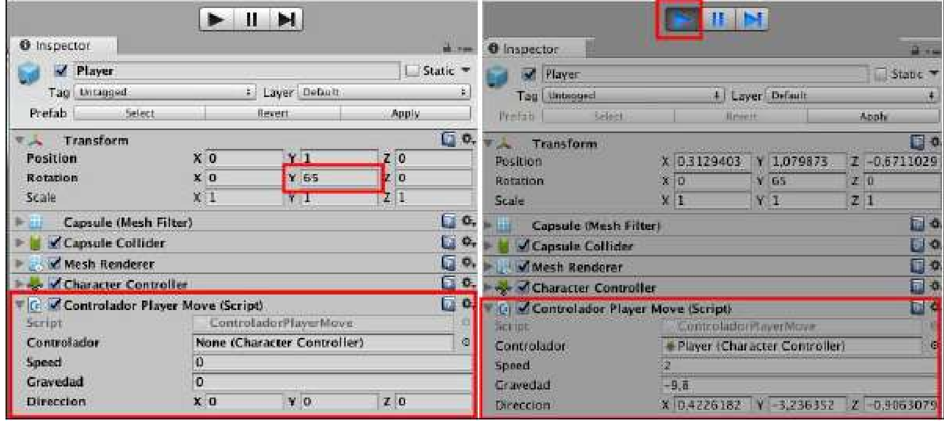


Fig. 7.15

Estos ejemplos son importantes porque a partir de ahora utilizaremos estos métodos mediante **Inputs**, de este modo ya no tendremos una pequeña animación del objeto sino que empezaremos a controlar el movimiento de nuestro objeto.

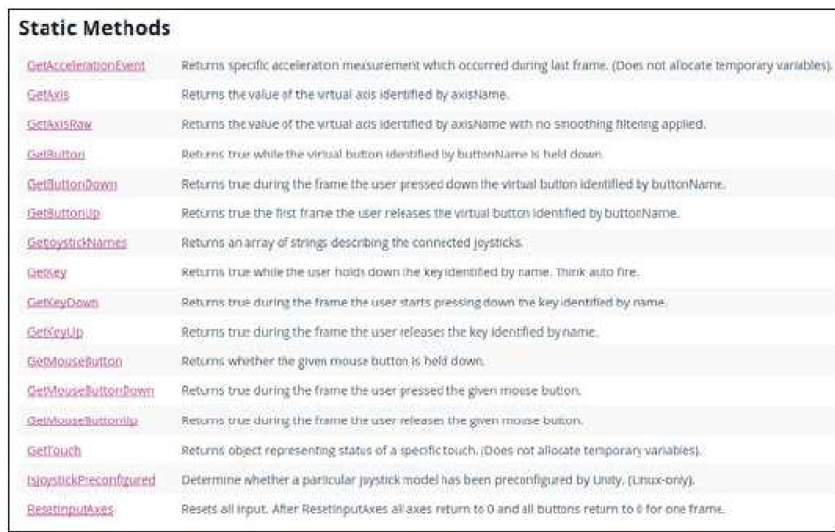
4. Los Inputs

Es la forma que utilizaremos para detectar teclas pulsadas del usuario. Voy a pedirte que mires la documentación de Unity solo para que veas de qué disponemos para trabajar con inputs. Puedes acceder desde este enlace:

https://docs.unity3d.com/2018.1/Documentation/ScriptReference/Input.html

En primer lugar verás que se divide en propiedades estáticas y funciones estáticas. Ahora me quiero centrar en las funciones estáticas que nos permiten recoger información de los botones virtuales.

En esta documentación se recogen los cuatro tipos de inputs que se utilizan: los inputs de teclado, o de joystick/joypad, los inputs de ratón y los inputs táctiles. En esta obra vamos a ver el funcionamiento de los de teclado y ratón.



Static Methods

GetAccelerationEvent	Returns specific acceleration measurement which occurred during last frame. (Does not allocate temporary variables).
GetAxis	Returns the value of the virtual axis identified by axisName.
GetAxisRaw	Returns the value of the virtual axis identified by axisName with no smoothing filtering applied.
GetButton	Returns true while the virtual button identified by buttonName is held down.
GetButtonDown	Returns true during the frame the user pressed down the virtual button identified by buttonName.
GetButtonUp	Returns true the first frame the user releases the virtual button identified by buttonName.
GetJoystickNames	Returns an array of strings describing the connected joysticks.
GetKey	Returns true while the user holds down the key identified by name. Think auto fire.
GetKeyDown	Returns true during the frame the user starts pressing down the key identified by name.
GetKeyUp	Returns true during the frame the user releases the key identified by name.
GetMouseButton	Returns whether the given mouse button is held down.
GetMouseButtonDown	Returns true during the frame the user pressed the given mouse button.
GetMouseButtonUp	Returns true during the frame the user releases the given mouse button.
GetTouch	Returns object representing status of a specific touch. (Does not allocate temporary variables).
IsJoystickPreconfigured	Determine whether a particular joystick model has been preconfigured by Unity. (Linux-only).
ResetInputAxes	Resets all input. After ResetInputAxes all axes return to 0 and all buttons return to 0 for one frame.

Fig. 7.16

Input Manager

Unity tiene unos botones virtuales, para acceder a ellos debemos ira al al menú principal de Unity y acceder a **Edit > Project Settings > Input,** verás que en la ventana Inspector se nos muestra el **InputManager.**

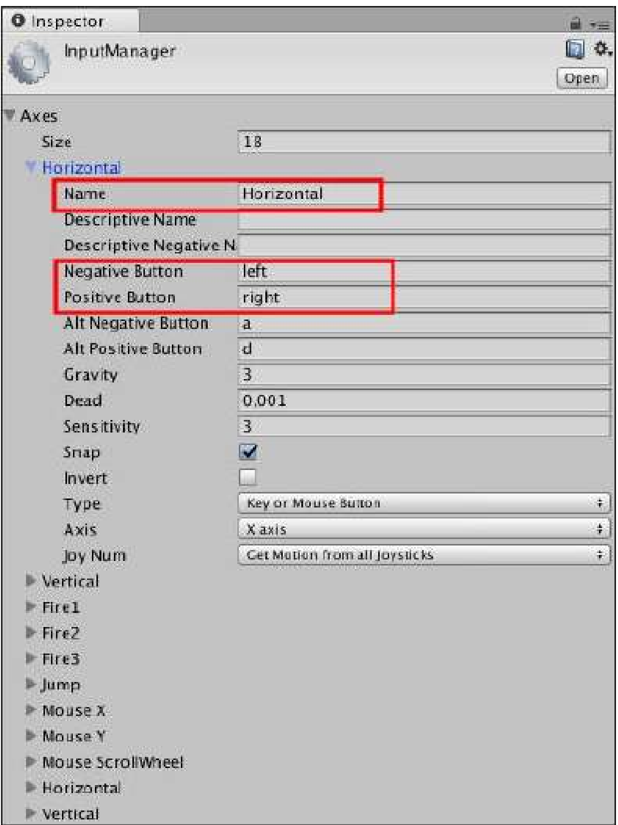


Fig. 7.17

Este apartado nos permite configurar un movimiento concreto y darle un nombre; por ejemplo, el nombre Horizontal tiene las teclas izquierda y derecha de las flechas del teclado, la flecha izquierda nos dará un valor negativo y la flecha derecha un valor positivo, como se muestra en la imagen anterior.

Para utilizar estos botones virtuales debemos utilizar el método GetAxis que podemos ver en la documentación de Unity.

https://docs.unity3d.com/2018.1/Documentation/ScriptReference/Input.html

A continuación vamos a ver cómo funciona el método GetAxis y como podemos utilizar el Input Manager.

Input GetAxis

Nos permite crear varios **Inputs** que pueden tener una o dos teclas. Otra característica es que no es necesario que este método este dentro de un **if** para poder ser ejecutado.

Para este nuevo ejemplo elimina el componente script del objeto **Player** como hemos hecho anteriormente accediendo a la ventana Inspector y dentro del apartado Scripts

haciendo clic en el icono con forma de engranaje y seleccionando la opción **Remove component.** Otra cosa importante es que pongas con valor 0 el componente Rotation en el eje Y.

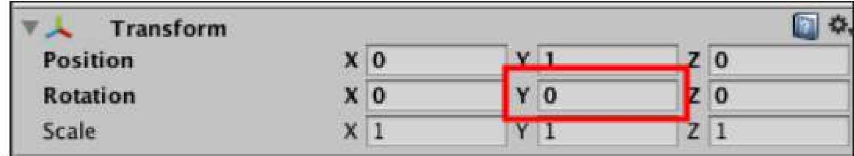


Fig. 7.18

Ahora crea un nuevo script con el nombre de **getAxisMetodo** y lo arrastramos al objeto Player.

Script: getAxisMetodo.cs

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

public class getAxisMetodo : MonoBehaviour
{
   public CharacterController controlador;
   public float speed;
   public float gravedad;
   public Vector3 direccion;

   void Start ()
   {
      this.controlador = gameObject.GetComponent<CharacterController> ();
      this.speed = 2f;
      this.gravedad = 9.8f;
   }

   void Update ()
   {
      this.direccion = Vector3.zero;
      this.direccion.y -= this.gravedad*Time.deltaTime;
      this.direccion.x = Input.GetAxis ("Horizontal") * speed;
      this.direccion.z = Input.GetAxis ("Vertical") * speed;
      this.controlador.Move(this.direccion * Time.deltaTime);
   }
}
```

Las variables son iguales al anterior script, la diferencia reside dentro de la función Update y en el valor de la gravedad dentro de la función Start() que en este script es positiva. Dentro hemos guardado en la variable dirección un vector3 en la posicion 0, para que se mantenga quieto.
Para la gravedad, que en este ejemplo es un valor positivo, simplemente le hemos aplicado una diferencia al valor de dirección.y, en realidad tiene los mismos efectos que en los ejemplos anteriores.

```
this.direccion.y -= this.gravedad*Time.deltaTime;
```

Ahora hemos guardado los inputs de GetAxis en las direcciones (x,z) multiplicados por el valor speed.

```
this.direccion.x = Input.GetAxis ("Vertical") * speed;
this.direccion.z = Input.GetAxis ("Horizontal") * speed;
```

Para finalizar aplicamos el método Move al charactercontroller multiplicado por el método Time.deltaTime.
Recuerda que hay que guardar el script desde Monodevelop para que se ejecuten los cambios.

Si todo está correcto debes poder mover el Player en todas direcciones mediante las flechas del teclado. Es posible que la dirección no sea del todo acertada pues la posición de nuestra cámara puede estar mal situada respecto la dirección local de nuestro objeto.

Una vez creado y editado el script vamos a resolver algunas cuestiones como qué son Vertical y Horizontal. El metodo GetAxis obtiene la información del InputManager que hemos visto al inicio del tema de Inputs. Si accedemos a él desde el menú principal **Edit > Project Settings > Input** en la ventana Inspector vamos a ver que existen dos opciones una llamada Vertical y la otra Horizontal.

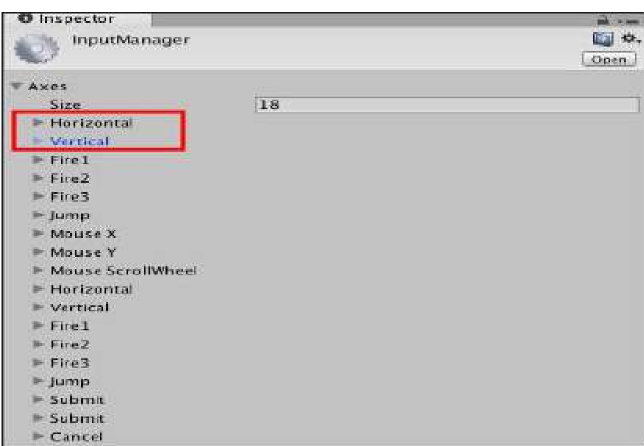


Fig. 7.19

Estas dos opciones están configuradas de forma que el objeto interpreta en qué ejes se debe mover y qué tipo **Input** debe tener en cuenta. En la siguiente imagen te muestro cómo interpreta el método **GetAxis** Horizontal y Vertical:

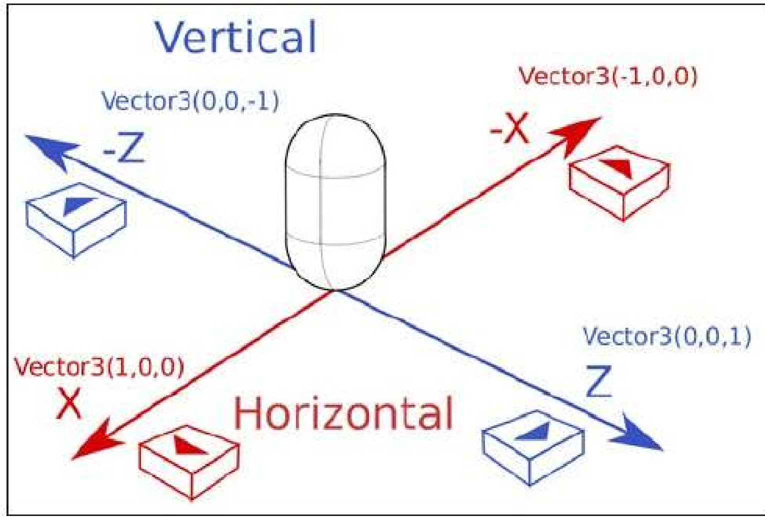


Fig. 7.20

La opción Horizontal toma como referencia el eje X y cuando pulsamos la tecla con la flecha a la izquierda nos devuelve un Vector3 con un valor de una unidad negativo y cuando pulsamos hacia la derecha nos devuelve un Vector3 con valor de una unidad positiva. En lo que se refiere a la opción Vertical ocurre lo mismo pero en el eje Z. Si desplegamos la opción Horizontal verás que te muestra varios parámetros que podemos configurar según nuestras necesidades.

▼ Horizontal

Name	Horizontal
Descriptive Name	
Descriptive Negative I	
Negative Button	left
Positive Button	right
Alt Negative Button	a
Alt Positive Button	d
Gravity	3
Dead	0,001
Sensitivity	3
Snap	☑
Invert	☐
Type	Key or Mouse Button
Axis	X axis
Joy Num	Get Motion from all Joysticks

Fig. 7.21

En este momento no vamos a tocar nada y lo dejaremos tal y como está por defecto. Pero si que me gustaría comentarte qué son estos parámetros.

- **Name:** este nombre es el que nos permite poder acceder desde un Script.
- **Descriptive Name:** podemos ponerle un nombre para el valor positivo que se muestra en la pestaña de entrada del cuadro de diálogo configuración para compilaciones independientes.
- **Descriptive Negative:** nombre del valor negativo que se muestra en la pestaña entrada del cuadro de diálogo configuración para compilaciones independientes.

- **Negative Buttton:** el botón que se utiliza para empujar el eje en la dirección negativa.
- **Positive Buttton:** el botón que se utiliza para empujar el eje en la dirección positiva.
- **Alt Negative Button:** el botón alternativo que se utiliza para empujar el eje en la dirección negativa.
- **Alt Positive Button:** el botón alternativo que se utiliza para empujar el eje en la dirección negativa.
- **Gravity:** es la velocidad en unidades por segundo en que el eje vuelve al estado neutral cuando no se presiona ningún botón.
- **Dead:** tamaño de la zona muerta análoga. Todos los valores de dispositivos analógicos dentro de este rango dan como resultado un mapa en neutral.
- **Sensitivity:** velocidad en unidades por segundo en que el eje se moverá hacia el valor objetivo. Esto es solo para dispositivos digitales.
- **Snap:** si está habilitado, el valor del eje se restablecerá a cero al presionar un botón de la dirección opuesta.
- **Invert:** si está habilitado, los botones negativos proporcionan un valor positivo, y viceversa.
- **Type:** nos permite escoger qué tipo de **Inputs** va a controlar este eje.
- **Axis:** determina el eje de un dispositivo conectado que controlará este eje.
- **Joy Num:** Joystick conectado que controlará este eje.

Input GetKey

Este método nos permite detectar teclas asignadas dentro del Input Manager, o crear nuestras propias. Recoge la información por el nombre de la tecla pulsada. Este método lo utilizaremos dentro de un If para controlar qué sucede cuando es pulsado.

A diferencia del método anterior, en este debemos decir que hace cada tecla. Para que entiendas mejor cómo funciona vamos a realizar un ejemplo en donde aparecerá un mensaje por consola cada vez que pulsemos en las flechas del teclado. Para este ejemplo elimina el objeto Player seleccionando el objeto y pulsando la tecla suprimir del teclado.

Ahora accede a la carpeta Prefabs selecciona el Prefab con nombre cube y arrástralo a la escena. La escena debería quedarte como te muestro en la siguiente imagen:

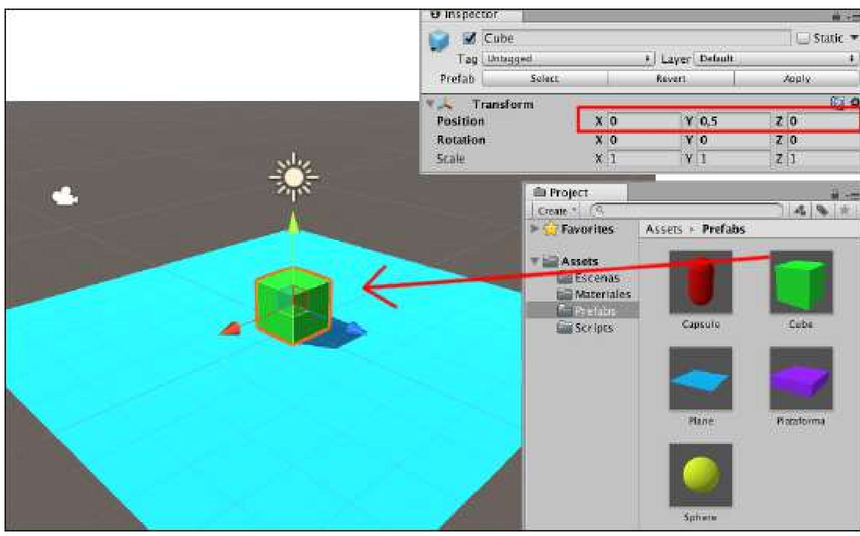


Fig. 7.22

Luego crearemos un nuevo script en la carpeta Scripts con el nombre getKeyMetodo. Después este script lo arrastramos al cubo para que quede agregado como componente de este.

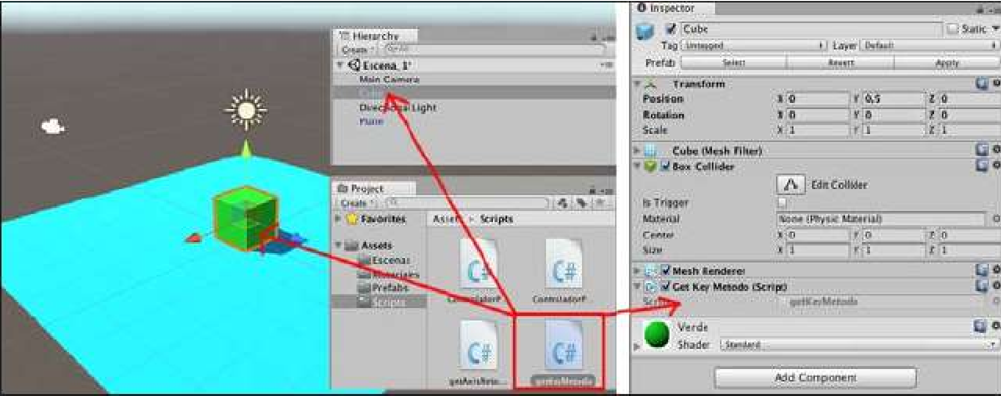


Fig. 7.23

Una vez hecho todo lo anterior hacemos doble clic encima del script y empezaremos a editar en Monodevelop de la siguiente manera:

Script: getKeyMetodo.cs

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

public class getKeyMetodo : MonoBehaviour
{

    void Update ()
    {
       if (Input.GetKey("up"))
       {
          Debug.Log("Estás pulsando la flecha del teclado hacia arriba");
       }
       if (Input.GetKey("down"))
       {
          Debug.Log("Estás pulsando la flecha del teclado hacia abajo");
       }
       if (Input.GetKey("right"))
       {
          Debug.Log("Estás pulsando la flecha del teclado hacia derecha");
```

```
        }
        if (Input.GetKey("left"))
        {
            Debug.Log("Estás pulsando la flecha del teclado hacia izquierda");
        }
    }
}
```

Este método es otra forma de introducir inputs en nuestros scripts para el ejemplo solamente he utilizado la función Update con una serie de inputs dentro de if.
Explico el primero porque los demás funcionan igual con distinto input.
El if nos dice que si pulsamos la tecla "up" hace referencia a la flecha del teclado que apunta hacia arriba, aparecerá un mensaje de consola en el que he puesta que tecla estás pulsando.
Recuerda que hay que guardar el script desde Monodevelop para que se ejecuten los cambios.

Si volvemos a Unity y ejecutamos el juego, cada vez que pulsemos en una de las flechas del teclado, el script nos devolverá un mensaje por consola como te muestro a continuación.

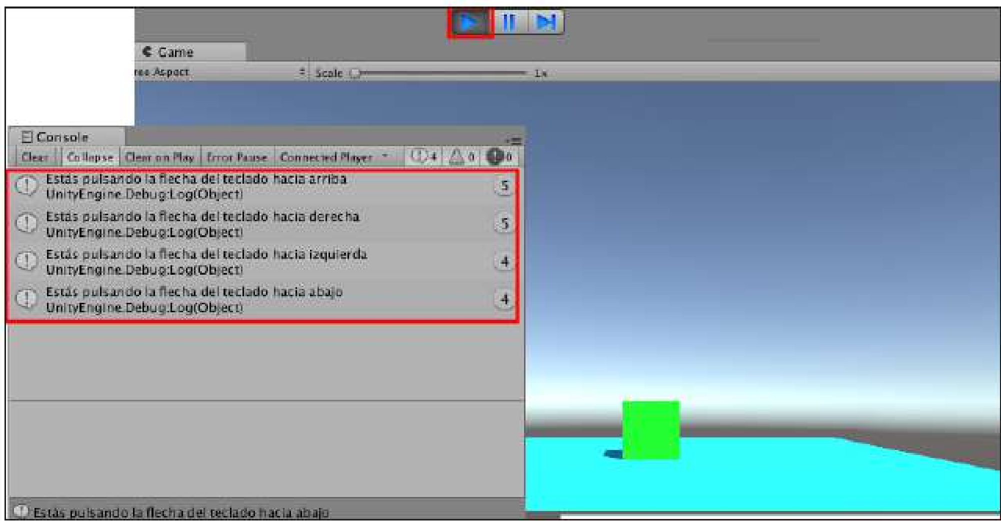


Fig. 7.24

Una de las preguntas que se suele tener es de dónde he sacado el nombre "up" de **Input.GetKey("up").** El nombre que va entre comillado es el nombre de la tecla a continuación te dejo una pequeña lista de los mas utilizados.

- **Keys Normales**: son las letras del teclado y son *"a", "b", "c" ...*
- **Keys numéricas:** son los números del teclado y son *"1", "2", "3", ...*
- **Keys de flecha o flechas de teclado** : se escriben en inglés y son *"up", "down", "left", "right"*
- **Keys de simbolo:** son los números que encontramos encima de las letras *"[1]", "[2]", "[3]", "[+]", "[equals]"*
- **Keys combinadas:** Es decir cuando utilizamos combinaciones de teclas como *"right shift", "left shift", "right ctrl", "left ctrl", "right alt", "left alt"*, (*"right cmd", "left cmd"*, cuando utilizamos un Mac)

- **Botones Ratón:** para referirnos a los botones del ratón utilizamos *"mouse 0", "mouse 1", "mouse 2".*
- **Botones Joystick** (para cualquier joystick): Los escribimos en inglés *"joystick button 0", "joystick button 1", "joystick button 2", ...*
- **Botones Joystick** (para joystick específicos): En este caso cuando tenemos más de un joystick *"joystick 1 button 0", "joystick 1 button 1", "joystick 2 button 0", ...*
- **Keys especiales:** normalmente son para referirnos a los botones de espacio, suprimir y se escriben de este modo; "backspace", "tab", "return", "escape", "space", "delete", "enter", "insert", "home", "end", "page up", "page down"
- **Keys de Función:** para referirnos a las teclas que tienen la letra F segida de un numero; "f1", "f2", "f3", ...

Input GetKey(KeyCode)

Otra forma de realizar el ejercicio anterior sin tener que saber cómo se llaman todas las teclas de nuestro teclado es utilizar un evento KeyCode.

En la documentación podemos buscar por KeyCode y aparecerán todos los nombres de los atributos que dan nombre a las teclas y que por supuesto podemos utilizar para crear nuestro set de configuración de teclado.

https://docs.unity3d.com/2018.1/Documentation/ScriptReference/KeyCode.html

Voy a reutilizar el ejemplo anterior pero esta vez utilizaré el nombre que viene vinculado a cada tecla que me muestra la documentación de Unity. Te aconsejo que accedas a ella y pruebes con otras teclas. La única variación que tendremos en el script es que en vez de escribir el nombre de la tecla directo **Input.GetKey ("up")**; deberemos utilizar el **KeyCode** entre paréntesis seguido del nombre de la tecla específica.

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

public class getKeyMetodo : MonoBehaviour
{
   void Update ()
   {
      if (Input.GetKey(KeyCode.UpArrow))
      {
       Debug.Log("Estás pulsando la flecha del teclado hacia arriba");
      }
      if (Input.GetKey(KeyCode.DownArrow))
      {
       Debug.Log("Estás pulsando la flecha del teclado hacia abajo");
      }
      if (Input.GetKey(KeyCode.RightArrow))
```

```
        {
          Debug.Log("Estás pulsando la flecha del teclado hacia derecha");
        }
        if (Input.GetKey(KeyCode.LeftArrow))
        {
          Debug.Log("Estás pulsando la flecha del teclado hacia izquierda");
        }
    }
}
```

El resultado tiene que ser el mismo que el del ejemplo anterior. A continuación pondremos en práctica todo lo anterior creando un **Player** controlado por teclado en el que podremos moverlo y rotarlo en la dirección que deseemos.

5. Mover y rotar nuestro Character Controller

Para el ejemplo que vamos a realizar crearemos una escena nueva accediendo al menú principal en **File > New Scene.** Te desaparecerán todos los objetos de la ventana Scene, ahora vamos una vez más al menú principal y seleccionamos **File > Save Scene** cuando te aparezca la ventana del navegador ponle el nombre de **Escena_2** y guárdala en la carpeta Escenas. Dentro de la carpeta Escenas de la ventana **Project** debe de aparecerte la escena.

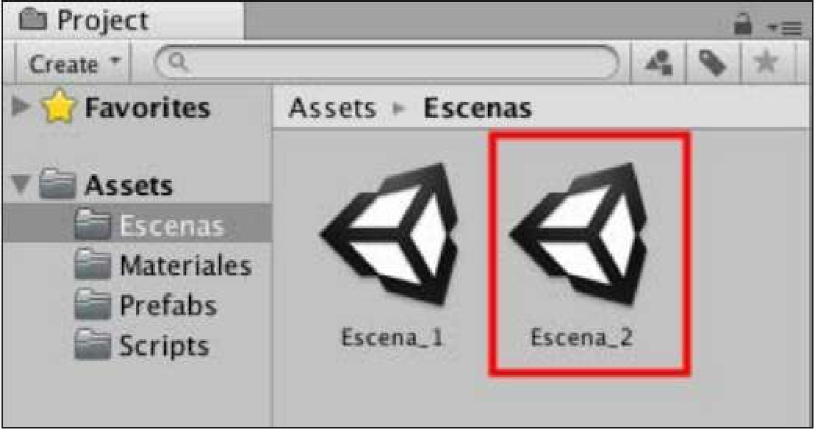


Fig. 7.25

Ya tenemos creada la escena 2, ahora vamos a colocar los objetos que necesitamos para la escena. En la carpeta que tenemos con el nombre Prefabs arrastraremos dentro de la ventana escena los Prefabs Capsule, Plane y un Cube. Para que la escena te quede como en el ejemplo vamos a renombrar la Capsule por Player, el Cube por Nariz y el Plane por Suelo. Para ponerles nombre selecciona el objeto en la escena o el nombre en la ventana Jerarquía y luego en la ventana inspector podemos cambiar el nombre de los objetos. También puedes hacer clic con el botón derecho del ratón encima del nombre

del objeto dentro de la ventana Jerarquía y seleccionar en el menú que te aparece la opción **rename**.

La escena debería quedarte como en la siguiente imagen.

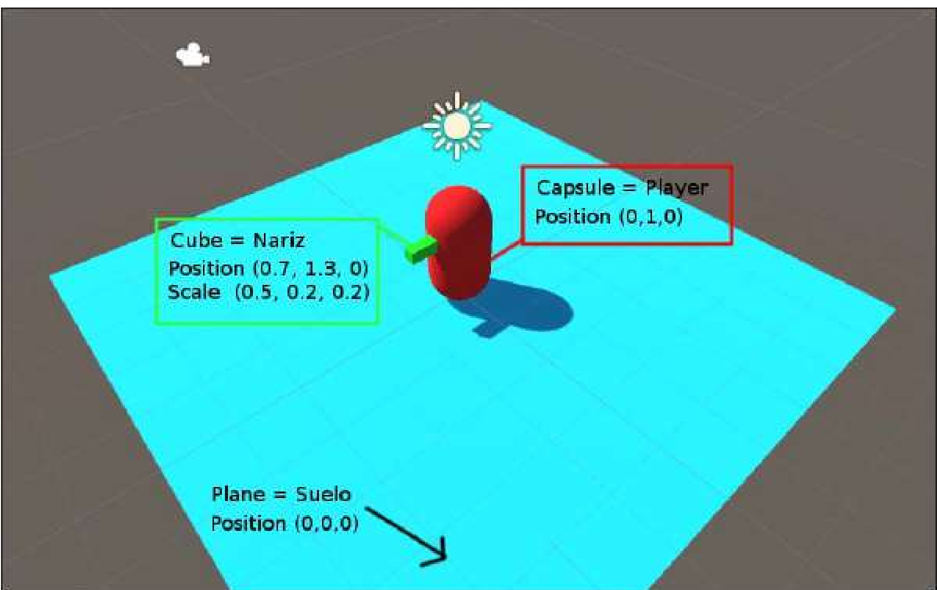


Fig. 7.26

En la imagen anterior dispones del nombre que tiene cada objeto y las variaciones en su transformación que debes introducir en la ventana inspector. Otro aspecto que debes tener en cuenta es que el objeto Nariz va a ser hijo de Player, para ello en la ventana Jerarquía seleccionamos el nombre Nariz y lo arrastramos encima de Player como te muestro a continuación:

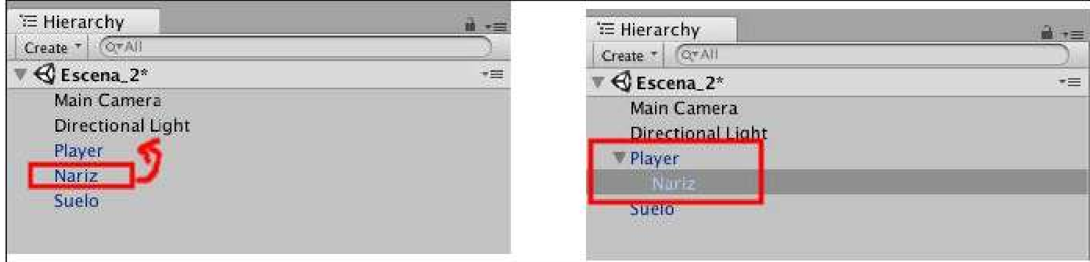


Fig. 7.27

Ahora ya tenemos nuestra escena creada y voy a pasar a explicar qué vamos a hacer. Hasta ahora hemos creado scripts para mover un objeto automáticamente ahora ya sabemos que podemos realizar acciones mediante los Inputs, es decir pulso una tecla y sucede algo. En este ejemplo vamos a crear un movimiento de desplazamiento hacia delante y hacia atrás controladas con las flechas del teclado Up , Down (arriba, abajo) y para la rotación utilizaremos las flechas del teclado Left y Right (izquierda, derecha).

Antes de crear el script nuestro objeto **Player** necesita un componente **Character Controller** o no servirá de nada lo que hagamos a continuación. Para añadir este componente selecciona el objeto **Player** y en la ventana inspector pulsamos en el botón Infe-

rior que pone **Add Component** en el menú accedemos a **Physics > Character Controller** y el componente se agregará.

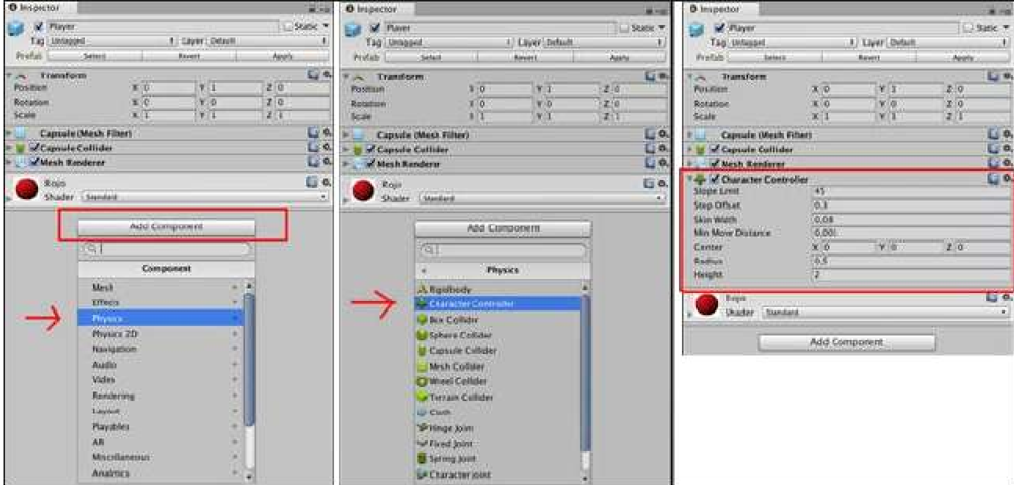


Fig. 7.28

La idea que queremos conseguir es que cuando pulsemos las flechas del teclado Izquierda y derecha (Horizontal) nuestro **Player** rote y cuando pulsemos las flechas Arriba y abajo (Vertical) nuestro **Player** avance o retroceda. En este caso les daremos un orden distinto al anterior.

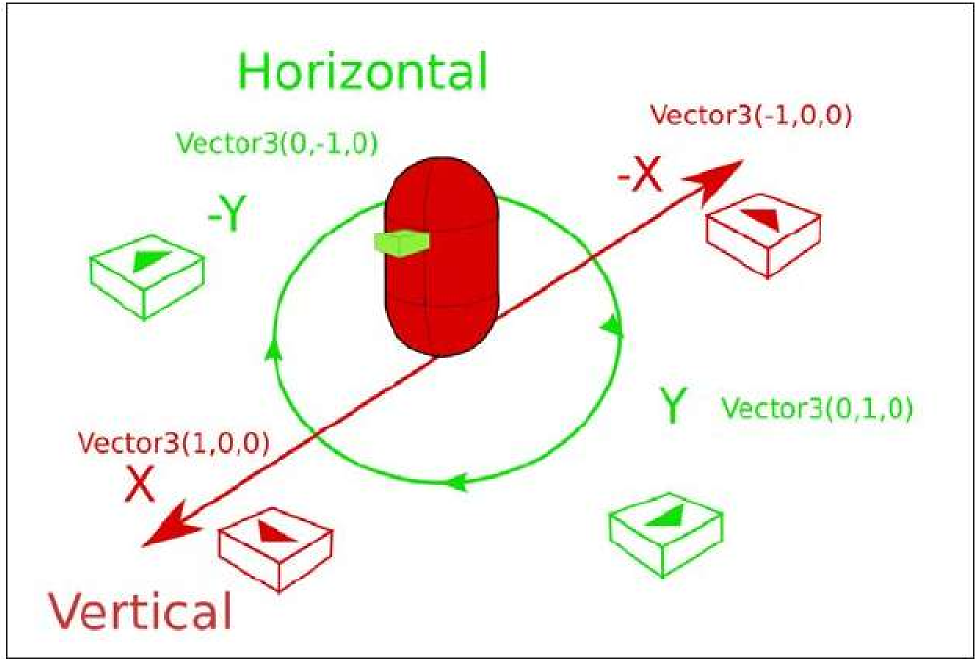


Fig. 7.29

Crearemos un nuevo script dentro de la carpeta Scripts, con el nombre de **MovimientoPlayer** y lo arrastramos encima del objeto Player. En esta nueva ocasión vamos a programar un script que utilice los inputs para mover el objeto y para rotar el objeto.

Script: MovimientoPlayer.cs

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

public class MovimientoPlayer : MonoBehaviour
{
    public CharacterController controlador;
    public Vector3 direccion;
    public float gravedad=9.8f;
    public float rotacion;
    public float speed = 5f;
    public float rotSpeed=5f;

    void Start ()
    {
        this.controlador = gameObject.GetComponent<CharacterController> ();
    }

    void Update ()
    {

        this.direccion = gameObject.transform.TransformDirection(new Vec-
tor3(Input.GetAxis        ("Vertical",0,0) * this.speed));
        this.rotacion = Input.GetAxis ("Horizontal") * this.rotSpeed;
        this.direccion -= new Vector3 (0, this.gravedad * Time.deltaTime,0);
        this.controlador.transform.Rotate(new Vector3(0f, this.rotacion, 0f));
        this.controlador.Move (this.direccion * Time.deltaTime);
    }
}
```

Las variables declaradas al principio son una variable de tipo CharacterController llamada controlador que servirá para almacenar la información de este componente. Una variable de tipo Vector3 llamada dirección, una variable de tipo float para darle un valor a la gravedad, otra variable tipo float para darle un valor a la rotación, para acelerar nuestro player una variable speed de tipo float y para acelerar la rotación una variable de tipo float llamada rotSpeed.

En este script he declarado las variables con su valor directamente.
En la función Start guardo la información del componente CharacterController dentro de la variable controlador, para poder utilizarlo.
En la función Update vamos a ir por partes:

A la variable dirección que es de tipo Vector3 guardaremos la información de la transformación local del objeto player (fijate que accedo a el desde un gameObject en minúscula). Dentro de la información de su transformación Local he aprovechado para indicarle un nuevo vector3 que tendrá en su eje x un Input de tipo GetAxis para poder mover el objeto en esa dirección y lo multiplico por la variable speed. Ya tenemos controlado un eje el x.
Para controlar la fuerza de la gravedad que empuja hacia abajo y no extender la información anterior le restaremos a la variable dirección un nuevo vector3 en que el eje (Y) contenga la gravedad multiplicado por Time.deltaTime, de este modo tenemos en el eje y la fuerza de gravedad empujando hacia abajo.
Para la rotación hemos creado una variable en la que guardaremos el valor Input de tipo GetAxis Horizontal multiplicado por la variable rotSpeed, Ahora hemos guardado la información de cuando pulsamos las teclas derecha e izquierda multiplicado por un valor para acelerarlo.
Ahora que tenemos toda la información necesaria guardada en nuestras variables vamos a acceder primero:
A la transformación de rotación del character controller que la tenemos guardada en la variable controlador. Accedemos a su transformación de rotación y creamos un nuevo vector3 en donde nos interesa rotar en el eje (Y) por medio de Inputs, como ya hemos guardado esta información anteriormente solo debemos poner la variable rotación en el lugar donde va el eje (Y). Bien ya tenemos controlada la rotación.
Para el movimiento utilizaremos el método Move que ya hemos visto anteriormente y ahora volvemos a Unity y si no hay errores pulsamos en ejecuta el juego.
Recuerda que hay que guardar el script desde Monodevelop para que se ejecuten los cambios.

6. Saltar obstáculos

Vamos a utilizar el ejemplo anterior y le vamos añadir una serie de variaciones en el script. Para que sea mas interesante puedes poner en la escena algunas plataformas de la carpeta Prefabs pero solo es una opción.

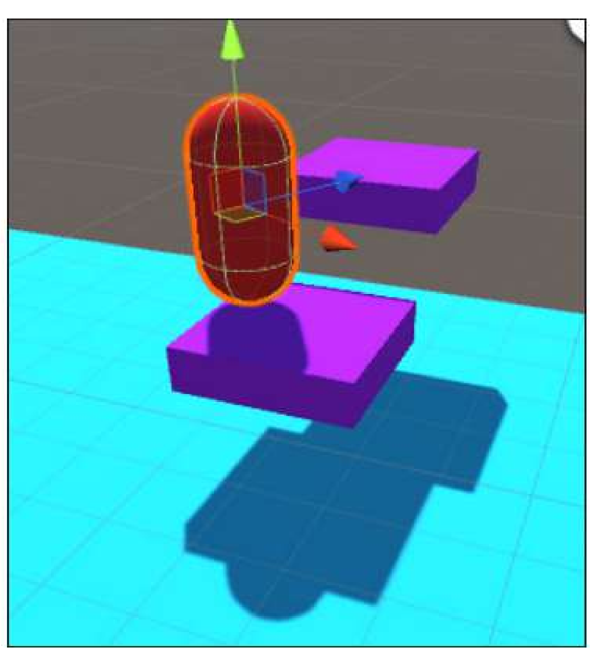


Fig. 7.30

Ahora vamos a añadir al anterior script un salto para que nuestro Player pueda realizar esta acción.

Script: MovimientoPlayer

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

public class MovimientoPlayer : MonoBehaviour
{
   public CharacterController controlador;
   public Vector3 direccion;
   public float gravedad=9.8f;
   public float rotacion;
   public float speed = 5f;
   public float rotSpeed=5f;
   public float salto= 50f;

   void Start ()
   {
      this.controlador = gameObject.GetComponent<CharacterController> ();
   }

   void Update ()
   {
          if (this.controlador.isGrounded)
          {
      direccion = gameObject.transform.TransformDirection(new Vector3(Input.
GetAxis ("Vertical",0,0) * this.speed));
      rotacion = Input.GetAxis ("Horizontal") * this.rotSpeed;
      if (Input.GetKeyDown(KeyCode.Z))
      {
         direccion.y += this.salto * Time.deltaTime * this.speed;
         this.controlador.transform.Rotate(Vector3.zero);
      }
     }

     this.direccion -= new Vector3 (0, this.gravedad * Time.deltaTime,0);
         this.controlador.transform.Rotate(new Vector3(0f, this.rotacion,
0f));
     this.controlador.Move (this.direccion * Time.deltaTime);
   }
}
```

Las variaciones empiezan por añadir una variable de tipo float para el salto, le he puesto un valor 50f, es un valor puesto a ojo, puedes utilizar cualquier otro valor.
En la función Start simplemente he vuelto a confirmar el valor de la variable salto, aunque no es necesario si ya le hemos proporcionado un valor anteriormente.

Luego en la función Update utilizaremos un atributo del CharacterController llamado isGrounded que es de tipo booleano y nos indica si toca el suelo o no. Por ese mismo motivo utilizaremos un If así comparamos si nuestro CharacterController que hemos guardado en la variable controlador toca el suelo, en ese caso tenemos la dirección y la rotación que controlamos con las teclas, pero tenemos otro if que indica que si estamos tocando el suelo y pulsamos la tecla Z de nuestro teclado, la dirección en el eje (y) se le añade el valor de salto por cada segundo y se le multiplica por el valor speed. Una vez este nuestro personaje en el aire no podremos rotar ni avanzar hasta que volvamos a tocar el suelo, de eso se encarga el vector3.zero.

Todo lo demás se queda igual, ahora puedes volver a Unity, comprobar que no hay ningún error y ejecutar el juego.
Recuerda que hay que guardar el script desde Monodevelop para que se ejecuten los cambios.

Uno de los aspectos nuevos que hemos encontrado en el script que acabamos de editar es el atributo isGrounded es de tipo booleano y nos permite saber cuando el **CharacterController** toca el suelo y cuando no. Si lo deseas te facilito un enlace para que puedas verlo en la documentación de Unity:

https://docs.unity3d.com/2018.1/Documentation/ScriptReference/CharacterController-isGrounded.html

Si has realizado bien el script podrás disfrutar de un Player que puedes mover, rotar y hacer saltar mediante Inputs de teclado. Recuerda ir guardando la escena y el proyecto.

En este apartado hemos visto el componente **CharacterController** que nos permite a partir de colisionadores y dos métodos, como podemos controlar cualquier objeto. Este ha sido una forma muy básica de cómo podemos configurar algunos juegos sin utilizar físicas. A continuación en el siguiente apartado vamos a empezar a crear un **Player** utilizando físicas.

7. Rigidbody

El **Rigidbody** nos permite controlar la posición de un objeto a través de las simulaciones físicas. Para explicarlo vamos a crear una nueva escena accediendo al menú principal y seleccionando **File > New Scene,** seguidamente vuelve a acceder al menú principal y guarda la escena accediendo a **File > Save Scene,** en la ventana navegador que se nos abrirá le ponemos el nombre de **Escena_3** y la guardamos dentro de nuestra carpeta Escenas.

Los objetos del escenario que vamos a utilizar son, un plano y una esfera. Primero nos dirigimos a la carpeta Prefabs dentro de la ventana Project y arrastramos la esfera dentro de la escena, realizamos la misma acción para el Plano. Al plano le cambiamos el nombre por el de Suelo y a la esfera le ponemos el nombre de Player. La escena debería quedar de la siguiente manera:

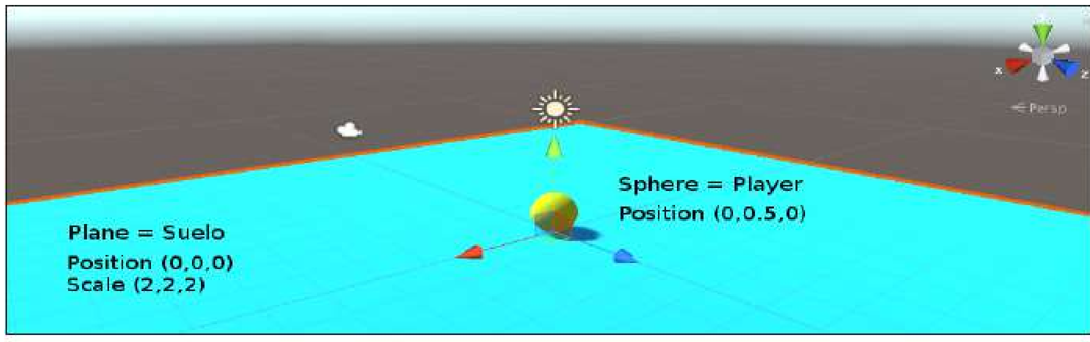


Fig. 7.31

Si te fijas en la imagen anterior veras que el Suelo esta escalado en 2 en todos sus ejes para tener más espacio de movilidad para el Player.

Ahora selecciona el Player (Esfera) y le añadimos un componente Rigidbody. Para ello con el Player seleccionado nos dirigimos a la ventana Inspector y accedemos al botón inferior Add Components > Physics > Rigidbody y se nos agregará el componente Rigidbody en nuestro Player.

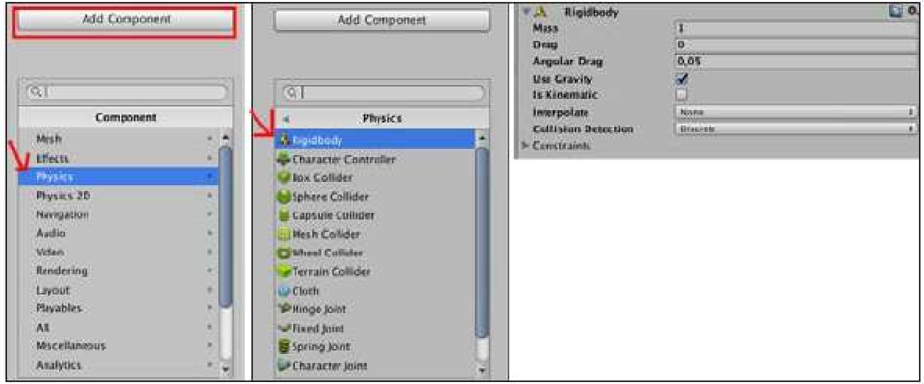


Fig. 7.32

Este es el componente que hemos añadido a nuestro Player, a continuación te explico que características tiene:

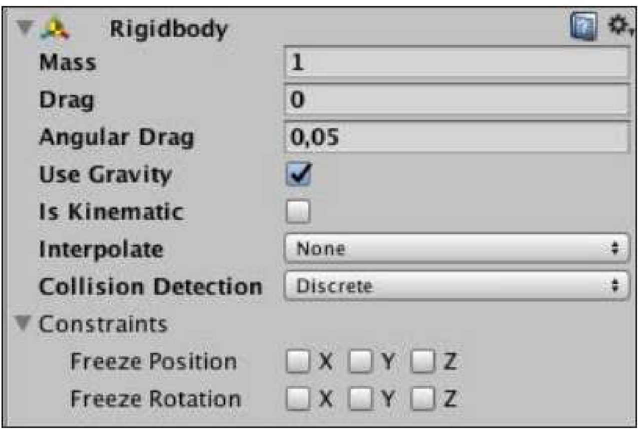


Fig. 7.33

Propiedades	Función
Mass	Nos permite darle un valor de masa al objeto que por defecto es 1 Kg.
Drag	Es la cantidad de resistencia al aire que afecta al objeto cuando se mueve por medio de fuerzas. Los valores son 0 para crear un objeto sin resistencia al aire y cuando el valor es mayor a 0 ofrece una resistencia que concluye con la disminución de movimiento del objeto.
Angular Drag	Es la cantidad de resistencia al aire que afecta al objeto cuando este gira. Igual que el parámetro anterior el valor 0 significa crear un objeto sin resistencia.
Use Gravity	Con esta opción podemos activar si al objeto le afecta la gravedad o no.
Is Kinematic	Esta opción permite activar o desactivar el uso de físicas
Interpolate	Este parámetro se puede activar si vemos que el movimiento de nuestros objetos va a trompicones o con animaciones poco nítidas. • **Interpolate:** las transformaciones son suavizadas basándose en el previo frame. • **Extrapolate:** las transformaciones son suavizadas basándose en la estimación del siguiente cuadro.
Collision Detection	Utilizado para prevenir que objetos que se estén moviendo rápido pasen a través de otros objetos sin detectar colisiones.
Constrain	Es una restricción en los ejes de posición o rotación

Este componente hemos dicho que funciona con Físicas de Unity, es decir que ahora el objeto es sensible a la gravedad y a toda clase de eventos físicos. En Unity podemos consultar y configurar las funciones de Físicas desde el **PhysicsManager**. Para acceder a el debemos ir al menú principal **Edit > Project Settings > Physics** y se nos aparecerá el **PhysicsManager** dentro de la ventana **Inspecto**.

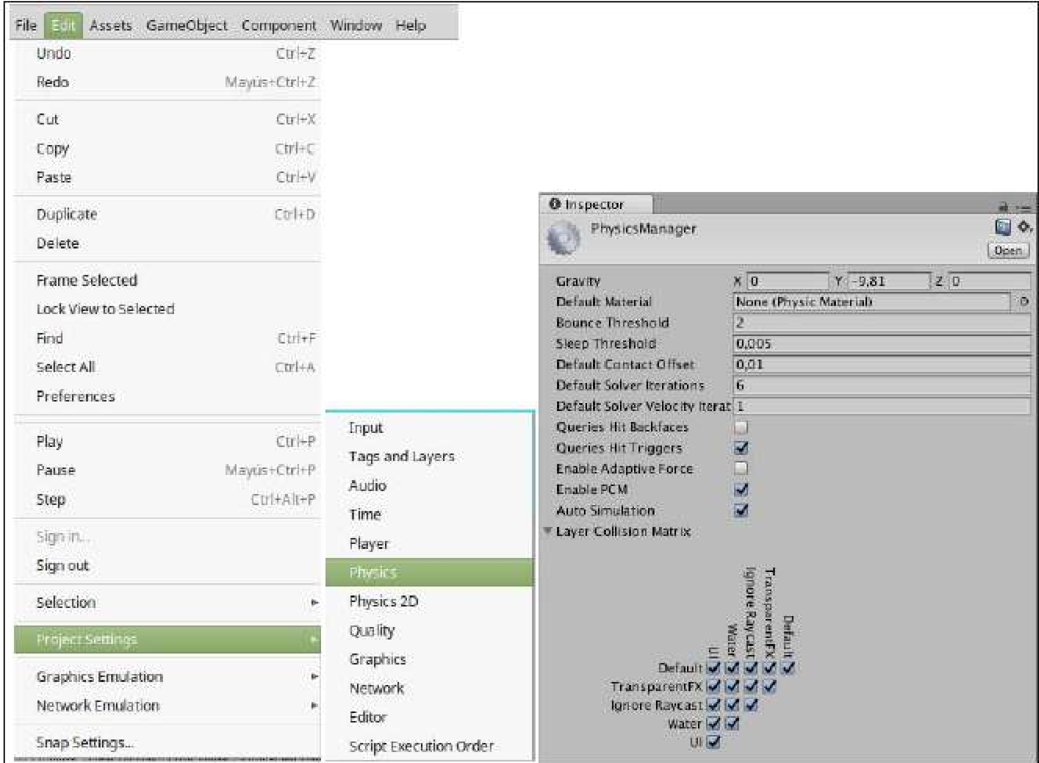


Fig. 7.34

PhysicsManager según define Unity es la configuración que define los límites en la precisión de simulación física. La precisión en las simulaciones requiere de una gran demanda de recursos del sistema y por lo general y basándome en la documentación de Unity la configuración que ofrece por defecto, sacrifica precisión frente a rendimiento. Para esta obra la configuración que nos ofrece es perfecta para realizar las actividades del libro y recomiendo no variar ningún parámetro.

A continuación te enuncio que función desarrolla cada parámetro a modo de información.

Propiedades	**Funciones**
Gravity	Utilizamos los ejes x,y,z para establecer el valor de gravedad aplicada a los objetos que contengan un componente **Rigidbody**. Se recomienda un valor negativo en el eje y para asemejarse a la gravedad real. También hay que tener en cuenta que si se aumenta la gravedad, se deberá aumentar el valor de **Default Solver Iterations**, para mantener contactos estables.
Default Material	Este parámetro no es un material como el que hemos visto hasta ahora, es un material de físicas que define la interacción de un objeto con el mundo físico. Este parámetro se utiliza si no se ha asignado ningún material de este tipo a un colisionador o collider individual.

Bounce Threshold	Este parámetro establece el valor de velocidad. Si dos objetos colisionan y tienen una velocidad relativa inferior a este valor , estos dos objetos no rebotarán entre sí. Este valor también reduce la inestabilidad, por eso se recomienda que el valor no sea muy bajo.
Sleep Threshold	Este parámetro tiene un valor que establece un umbral de energía global; cuando un objeto con Rigidbody sin cinemática (es decir que el objeto no está controlado por el sistema de físicas) se encuentra por debajo de este umbral puede (ir a dormir) desactivarse momentáneamente. Cuando un objeto con Rigidbody está "durmiendo" no se actualiza cada cuadro, lo que conseguimos que no se consuman tantos recursos. Otro aspecto a tener en cuenta es que la energía cinética de un cuerpo rígido dividido por su masa está por debajo de este umbral, también puede optar a "dormir".
Default Contact Offset	Este parámetro se encarga de establecer la distancia que usa el sistema de detección de colisiones para generar contactos de colisión. El valor debe ser positivo, en el caso de que el valor sea demasiado próximo a cero es posible que dé problemas. Los colisionadores solo generan colisión si su distancia es menor que la suma de sus valores de Contact offset.
Default Slover Iterations	Este parámetro define cuántos procesos de resolución se ejecutan en Unity cada cuadro de físicas. Por lo general se utiliza para reducir el jitter (temblores) en los contactos.
Default Solver Velocity Iterations	Este parámetro se utiliza para establecer cuántos procesos de velocidad realiza un solver (solucionador) en cada cuadro de física. A mayor valor de procesos mayor sera la precisión de la velocidad de salida resultante de un rebote.
Queries Hit Backface	Esta casilla no está marcada por defecto. Marcamos la casilla si queremos consultas de físicas para detectar impactos con los triángulos de las caras posteriores de MeshColliders.
Queries Hit Triggers	Marca esta casilla si deseas que las pruebas de impacto físico (como Raycasts, SphereCasts y SphereTests) devuelvan un golpe cuando se cruzan con un Colisionador marcado como Disparador.
Enable Adaptive Force	Este parámetro afecta la forma en que las fuerzas se transmiten a través de una pila o pila de objetos, para proporcionar un comportamiento más realista.
Enable PCM	Este parámetro está activo por defecto y esto significa que se regeneran menos contactos en cada cuadro de física y se comparten más datos de contacto entre los cuadros. Esto también genera más precisión y generalmente produce una mejor respuesta al colisionar dos objetos en la mayoría de los casos.

Auto Simulation	Ejecute la simulación física automáticamente o permita un control explícito sobre ella.
Auto Sync Transform	Sincronice automáticamente los cambios de transformación con el sistema de física siempre que cambie un componente de transformación.
Layer Collision Matrix	Se encarga de la detección de colisiones basada en capas. Es una forma de hacer que un objeto colisiones con otro objeto que está configurado para una capa o capas especificas. Te recomiendo por el momento todo activado.

Bien comentado los puntos anteriores a continuación vamos a ver como podemos acceder al RigidBody mediante un script y como podemos mover el objeto.

8. Mover un Rigidbody

Ahora es el momento de mover nuestro objeto accediendo a este componente para ello vamos a crear un script que añadiremos a nuestra esfera. Le vamos a cambiar el nombre si no lo has hecho todavía por el de Player. Como siempre antes de empezar un nuevo componente te invito a que te mires la documentación para ver que métodos y atributos podemos utilizar en Rigidbody.

https://docs.unity3d.com/2018.1/Documentation/ScriptReference/Rigidbody.html

Uno de los métodos de Rigidbody que vamos a utilizar es el método AddForce. Este método necesita dos argumentos para que funcione, uno es la fuerza que va argumentada con un Vector3 y el modo en el que debemos decirle el modo de fuerza que le vamos a otorgar.

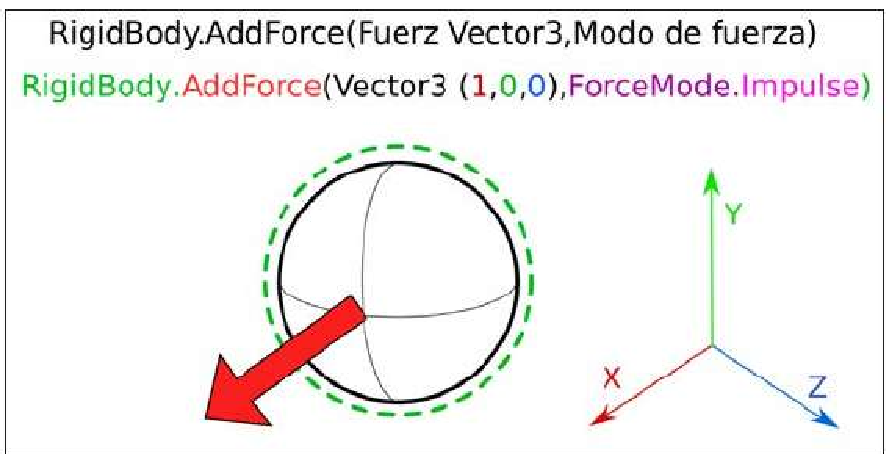


Fig. 7.35

Primero creamos un script dentro de la carpeta scripts con el nombre de **controlBola**, para diferenciarlo de los otros scripts y luego arrastramos el script encima de nuestro Player. Para empezar primero accederemos al RigidBody del Player.

Script: controlBola.cs

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

public class controlBola : MonoBehaviour
{
    public Rigidbody rb;

    void Awake()
    {
        this.rb = gameObject.GetComponent<Rigidbody>();
    }

    void FixedUpdate ()
    {
        this.rb.AddForce (Vector3.right, ForceMode.Impulse);
    }
}
```

En primer lugar vamos a crear una variable pública de tipo Rigidbody con nombre rb, en donde vamos a almacenar la información del componente Rigidbody.
Verás que en este caso la función utilizada es el void Awake() que permite que el componente se cargue antes de iniciar el juego.
Para finalizar utilizamos el FixedUpdate que se aconseja utilizar siempre que trabajemos con físicas. En este vamos acceder al componente Rigidbody al método AddForce que nos permite añadir una fuerza al objeto mediante un vector3, en este caso de tipo right que impulsará el objeto en el eje x positivo.
Verás que nuestro Player avanza rodando en dirección al eje x.
Recuerda que hay que guardar el script desde Monodevelop para que se ejecuten los cambios.

Una vez guardado el script y vuelta a Unity en el momento que ejecutes el script verás que nuestro **Player** es impulsado en el eje X. No te asustes todavía no hemos creado los Inputs para controlarlo.

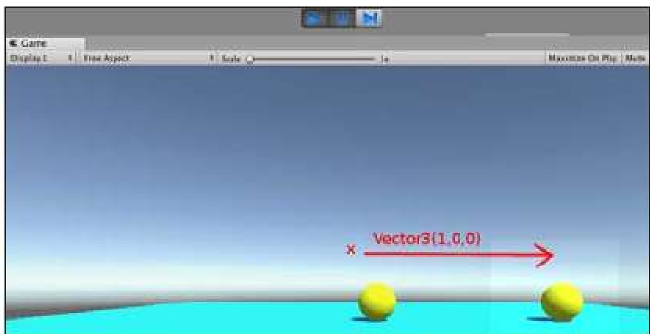


Fig. 7.36

9. Controlar el movimiento de un Rigidbody

En el ejemplo anterior hemos aplicado una fuerza a nuestro objeto. Esta fuerza si miras la documentación tiene dos parámetros, uno es la fuerza y el otro es el modo en que se aplica la fuerza. En el ejemplo anterior solamente le hemos dado la fuerza y un modo de fuerza llamado Impulse que nos impulsa la pelota. Ahora para controlarlo mediante las flechas del teclado vamos aprovechar el Vector3. Si recuerdas el apartado de **Inputs** el **Input.GetAxis** disponemos de una valor **Horizontal** y de un valor **Vertical**, estos dos parámetros nos resolverían los **Inputs** en los ejes X y Z . En el siguiente gráfico voy a intentar mostrar el ejemplo.

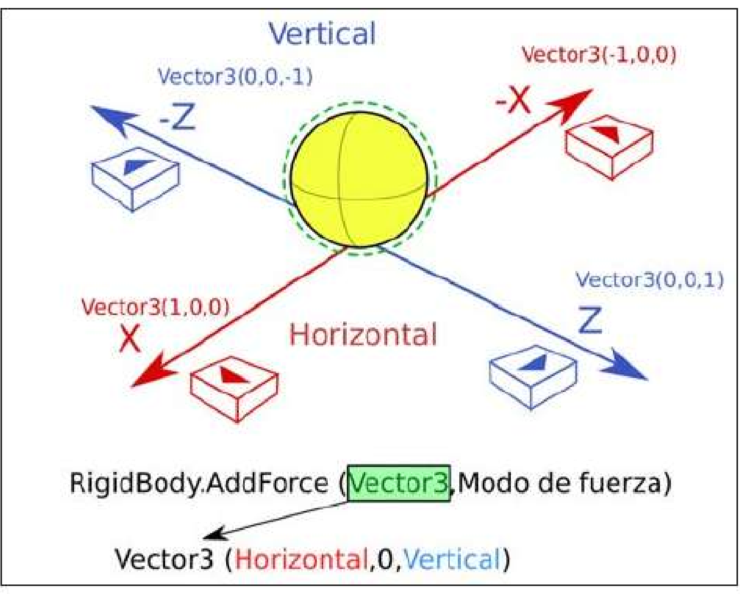


Fig. 7.37

A continuación vamos a seguir con el mismo script **controlBola** en donde vamos a crear los Inputs de teclado.

Script: controlBola.cs

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

public class controlBola : MonoBehaviour
{
   public Rigidbody rb;
   private Vector3 direccion;
   public float speed;

 void Awake()
```

```
 {
   this.rb = gameObject.GetComponent<Rigidbody>();
   this.direccion = Vector3.zero;
 }

 voidFixedUpdate ()
 {
   this.direccion = new Vector3 (Input.GetAxis("Horizontal"),0f,Input.GetA-
xis("Vertical"));
   this.rb.AddForce(this.direccion*this.speed,ForceMode.Impulse);
 }
}
```

Para controlar el Rigidbody necesitamos un vector3; así pues crearemos una variable de tipo Vector3 con nombre dirección y para poder mover el objeto necesitamos una velocidad, por eso mismo creamos una variable de tipo float para ponerle un valor a la velocidad y en este caso la hacemos pública para que podamos verla en el inspector de Unity.

En el Awake() le he dado un valor a la dirección de Vector3.zero para que cuando se inicie el juego la bola no se mueva.

Para darle movimiento mediante teclado dentro del FixedUpdate le damos valor a la variable direccion en donde crearemos un nuevo Vector3 y en el eje x le ponemos que tome el valor del Input Vertical que se encarga de los valores 1 y -1 de las teclas arriba y abajo, en el eje y le damos el valor 0f y en el eje z le ponemos el Input de tipo GetAxis Horizontal para que tome los valores 1 y -1 para las teclas izquierda y derecha.

Ahora solo tenemos que substituir dentro del método AddForce el vector por la variable dirección multiplicado por la variable speed y utilizaremos el argumento o parámetro que es el modo de fuerza, en este caso he utilizado el Impulse.
Recuerda que hay que guardar el script desde Monodevelop para que se ejecuten los cambios.

Para que el script anterior funciones en Unity debemos introducirle un valor a la variable speed dentro de la ventana inspector antes de ejecutar el juego. Te recomiendo utilizar valores bajos en este ejemplo en concreto he utilizado 0,1 como te muestro en la siguiente imagen.

Fig. 7.38

10. Añadir un salto al Rigidbody

Si ahora vamos a darle el comportamiento de salto a nuestro Player para que pueda superar obstáculos. Para ello debemos utilizar un Imput distinto del de las flechas ya que estas controlan la movilidad del Player. Una de las soluciones que podemos utilizar es un **Imput**.**GetAxis** ("Jump") que encontraremos en el **ImputManager que hemos visto en apartados anteriores** y que está preparado para este cometido. También me gustaría intentar explicar como podemos controlar el salto de forma que este salto no dure más de un segundo. Para eso voy a intentar explicarlo con la siguiente imagen.

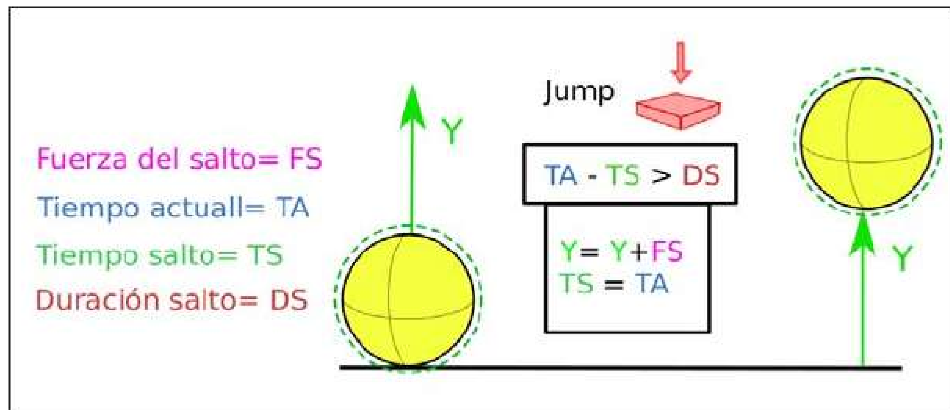


Fig 7.39

Para que nuestro **Player** pueda elevarse está claro que debemos darle un valor de fuerza en el salto dirigido al eje Y. Uno de los problemas que podemos encontrarnos es que si le decimos que al pulsar la tecla añada una fuerza al **Player**, cada vez que pulses la tecla **Jump** este ira incrementando el salto tantas infinitas veces. Es por ese motivo que utilizo tres valores para controlar el salto. El tiempo actual para poder saber el tiempo que pasa, el tiempo del salto para saber en que momento empiezo a ejercer esa fuerza para saltar y una duración de tiempo para el salto. La idea es que cuando pulsamos la tecla **Jump** comparemos la diferencia del momento actual menos el momento en que nuestro **Player** salta con el tiempo que queremos que se ejecute el salto. Esta pequeña comprobación nos permite controlar el salto en cierta medida.

Script: controlBola.cs

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

public class controlBola : MonoBehaviour
{
   Rigidbody rb;
   Vector3 direccion;
  public float speed;
  public float salto;
```

```
    private float saltoUltimo;
    public float saltoDuracion;

void Awake()
       {
      this.rb = gameObject.GetComponent<Rigidbody>();
      this.direccion = Vector3.zero;
      this.saltoUltimo = Time.time;
    }

    void FixedUpdate ()
    {
      this.direccion = new Vector3 (Input.GetAxis("Horizontal"), 0f,Input.
GetAxis("Vertical"));
      if (Input.GetButtonDown("Jump"))
      {
       if ((Time.time - this.saltoUltimo) > this.saltoDuracion )
       {
         this.direccion.y += this.salto;
         this.saltoUltimo = Time.time;
       }
      }
      this.rb.AddForce (this.direccion*this.speed,ForceMode.Impulse);
    }

}
```

En el grupo de variables creamos 3 variables nuevas. Una es una variable de tipo float para dar una fuerza al salto. La variables saltoUltimo la utilizaremos para almacenar el tiempo del ultimo salto y el saltoDuracion servirá para almacenar la carencia del salto.

Dentro de la función Awake() guardaremos dentro de la variable saltoUltimo, el tiempo que transcurre o el momento en el que se realiza el ultimo salto, para ello utilizamos Time.time.

Dentro de FixedUpdate crearemos un if para decir que si pulsamos el botón Jump que es la barra espaciadora, haremos la siguiente comprobación si restamos el tiempo con el momento del ultimo salto y es mayor que la duración del salto sucederá lo siguiente:
La dirección en el eje y sera la suma de esa dirección mas el valor del salto.
El ultimo salto sera igual al tiempo transcurrido.
Recuerda que hay que guardar el script desde Monodevelop para que se ejecuten los cambios.

Cuando vuelvas a Unity recuerda que debes agregar los valores a los parámetros **Salto , Salto Duracion**, de lo contrario si ejecutas la escena no vas a poder saltar. A continuación te muestro una imagen con los parámetros utilizados para este ejemplo.

Fig. 7.40

Ahora ya disponemos de un Objeto que podemos controlar perfectamente mediante teclado. El problema que vemos es que en la ventana **Game** es difícil controlar el **Player** porque la cámara no sigue a nuestro al objeto. A continuación vamos a ver como podemos configurar de una forma simple nuestra cámara para que este siga a nuestro **Player**.

11. Seguimiento simple de nuestra cámara

Ahora vamos a hacer que la cámara siga a nuestra pelota en este caso lo realizaremos de la siguiente manera. Crearemos un script dentro de la cámara de la escena y le daremos un Vector 3 de distancia para posicionar . En este caso no emparentamos la cámara al Player porque en el momento que empiece a rodar la pelota la cámara también rodaría. La solución será que las transformaciones de movimiento del Player sean iguales a las de la cámara como te muestro en el siguiente script que llamaremos **ControlCamara** y se lo añadiremos a la cámara de la escena.

Para ello crea en la carpeta Scripts un nuevo script llamado ControlCamara. Luego arrastra este script al Objeto **MainCamera** o dentro de los componentes de esta como te muestro en la siguiente imagen.

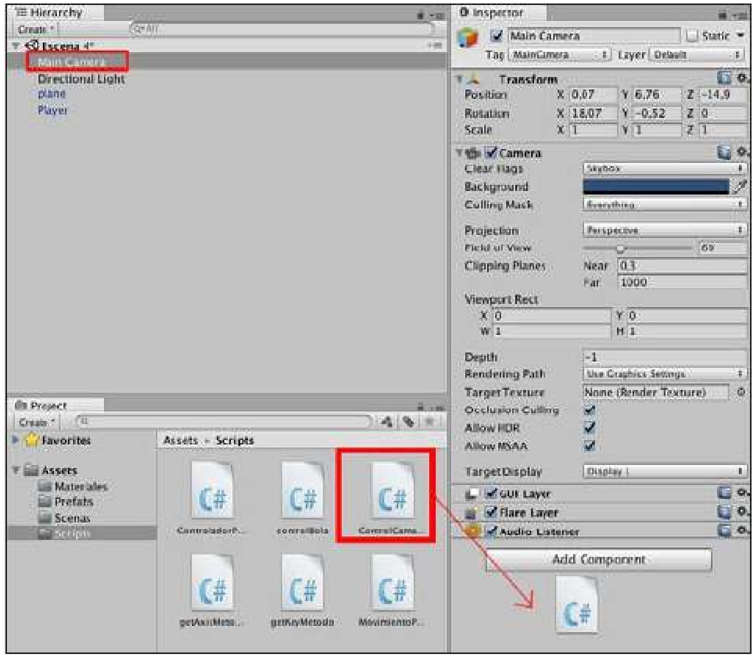


Fig. 7.41

Script: ControlCamara.cs

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

public class ControlCamara : MonoBehaviour
{
 private GameObject player;
 public Vector3 distancia;

 void Awake()
 {
   this.player = GameObject.Find("Player");
   this.distancia = new Vector3(0f,-1.5f,5f);
 }

 void LateUpdate ()
 {
   gameObject.transform.position=this.player.transform.position-this.distan-
cia;
 }
}
```

En este script primero crearemos una variable para almacenar el objeto Player y le damos el nombre de player. Queremos almacenar este objeto porque la cámara tiene que saber en qué lugar se encuentra la bola para seguirla. La otra variable es para almacenar mediante un Vector3 la distancia que se tiene que separar la cámara del objeto al que va ha seguir.
Dentro de la función Awake diremos que this.player es un GameObject con un nombre Player (Cuidado asegúrate de que la pelota de la escena tiene el nombre de Player dentro de Unity), de este modo ya tenemos localizado el objeto. La otra variable es un Vector3, los valores que ves en el ejemplo ya han sido buscados anteriormente si lo estás haciendo por primera vez pon los valores (0,0,0) y una vez ejecutes la escena mueve los parámetros hasta colocar la cámara correctamente.
Para finalizar en el LateUpdate (Se utiliza para cámaras) diremos que la posición del player es la misma que la posición de la cámara menos la distancia que es el vector3 que hemos creado.
Recuerda que hay que guardar el script desde Monodevelop para que se ejecuten los cambios.

Si todo está correcto y vamos a Unity, selecciona la cámara y mira en la ventana inspector como te aparece la variable Distancia que has creado.

Fig. 7.42

Una vez actives la escena , no te asustes porque si has puesto los valores dentro del script en la variable *this.distancia= new Vector3(of,of,of)* verás algo parecido a la imagen que te muestro a continuación.

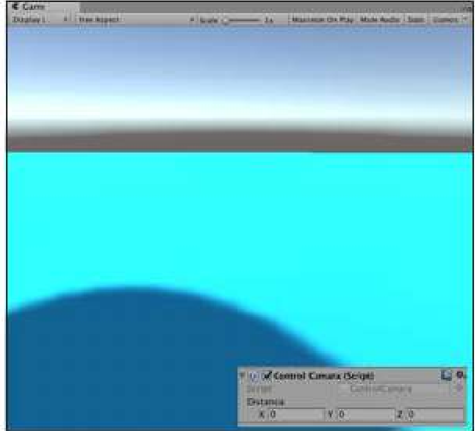


Fig. 7.43

Debes mover los parámetros de distancia para alejar la cámara del Player. Recuerda que si varias los parámetros con la escena ejecutándose cuando finalices el juego los valores vuelven por defecto al valor que tenían en el inicio, asegúrate de apuntarte los valores y luego escribe estos valores dentro de la función Awake en la declaración de valores de la variable distancia.

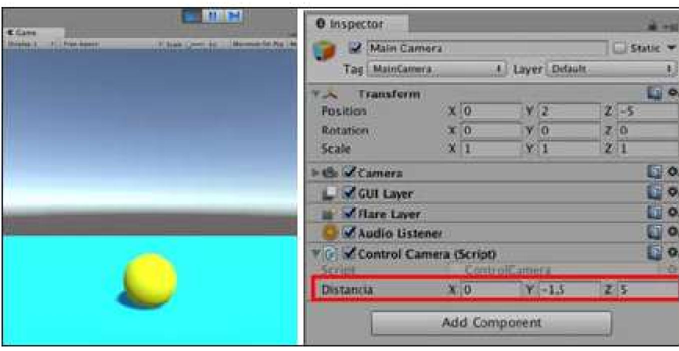


Fig. 7.44

Una vez encuentres la distancia correcta también puedes rotar la cámara manualmente para que enfoque mejor el objeto, pero eso depende de tu gusto. Ahora puedes disfrutar del control del este objeto.

12. Destruir objetos con colisiones

Para hacer una practica un poco más divertida o entretenida voy a explicarte como podemos destruir objetos. Cuando digo destruir objetos me refiero a que Unity los elimina de la escena. La idea es que nuestro **Player** cuando toque un objeto en concreto este se destruya es decir desaparezca dando la sensación de que nuestro **Player** recoge este objeto.

Primero de todo vamos a seguir en la misma escena (Escena3) y vamos a añadir de la carpeta Prefabs y arrastraremos el objeto **Cube** a la escena. A este cubo le cambiaremos la escala desde la ventana Inspector en el componente **Tranform Scale** como te muestro en la siguiente imagen.

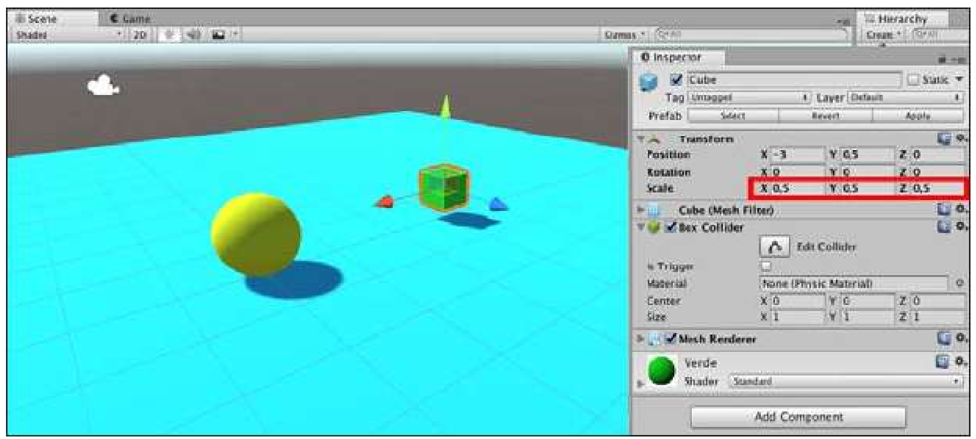


Fig 7.45

El cubo que muestra la imagen anterior tiene una escala de 0,5 para todos sus ejes, la posición de este es indiferente, puedes colocar este objeto donde quieras , siempre que este por encima del objeto suelo. Otro aspecto importante que debe tener el objeto **Cube** es un **Box Collider** , si has utilizado el Cube que hay en la carpeta Prefabs ya lleva uno.

Otra cosa importante es que le vamos a poner una etiqueta con el nombre Item. Si no recuerda como se ponía una etiqueta revisa el capitulo 2 el apartado de la Ventana Inspector, de todos modos a continuación te explico como debemos hacerlo.

Selecciona el cubo y en la ventana Inspector en el apartado **Tag** desplegamos el menú y seleccionamos la opción **Add Tag.** En la siguiente ventana que se nos aparecerá pulsamos en el símbolo de suma para agregar una **nueva Tag** (Etiqueta). Nos pedirá que le demos un nombre, escribimos el nombre **Item** y pulsamos en **Save** para guardar la nueva etiqueta.

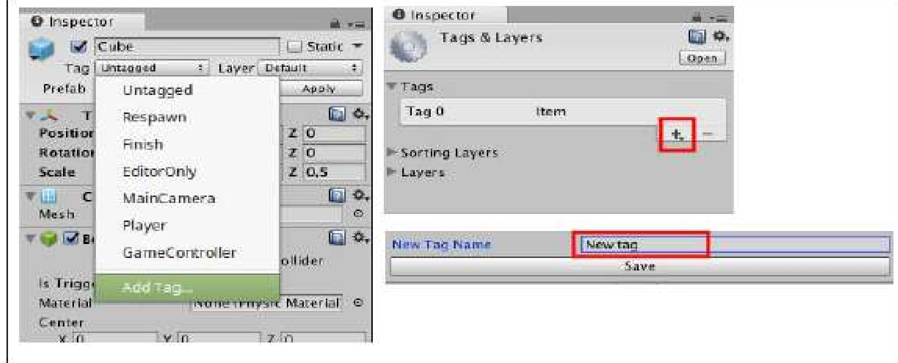


Fig. 7.46

Ahora ya hemos creado una nueva etiqueta con el nombre **Item** pero todavía no se la hemos agregado al cubo. Para agregar esta etiqueta, con el cubo seleccionado accedemos a la ventana Inspector y ahora en el menú **untagged** aparecerá la etiqueta que hemos creado, la seleccionamos y listo.

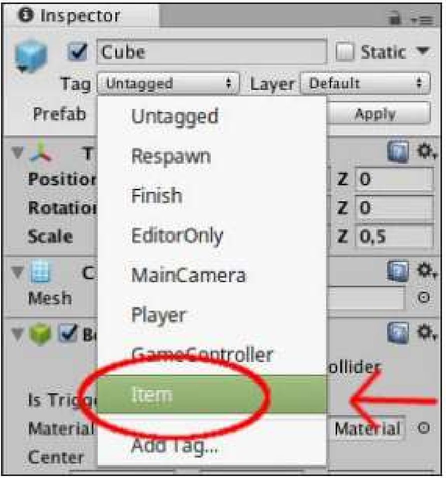


Fig. 7.47

Luego puedes duplicar el cubo unas 10 veces como se muestra en la siguiente imagen.

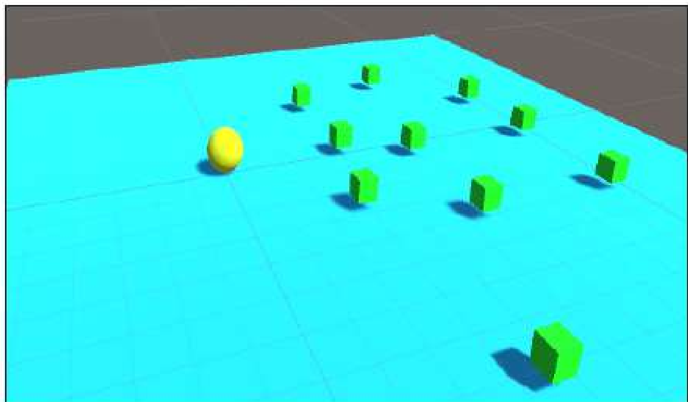


Fig. 7.48

Para duplicar este objeto accedemos a la ventana Jerarquía y seleccionando el objeto **Cube** hacemos clic encima con el botón derecho y seleccionamos en el menú que se nos despliega la opción **duplicate** y automáticamente se nos aparece un nuevo objeto con el mismo nombre y un numero entre paréntesis.

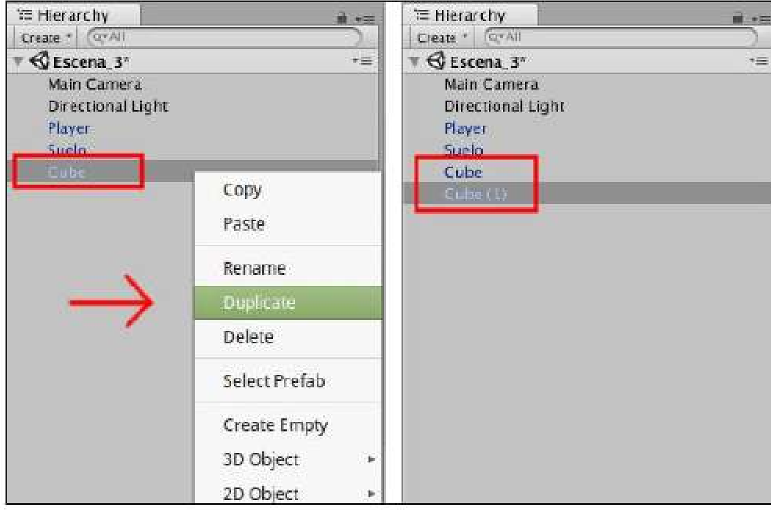


Fig 7.49

La copia que se crea, se encuentra posicionada encima del objeto original, debes posicionar el objeto copia manualmente selecciona dicho objeto, pulsar el botón mover objetos que encuentras encima de la ventana Scene y utilizando los ejes de coordenadas del objeto mueve el objeto donde quieras.

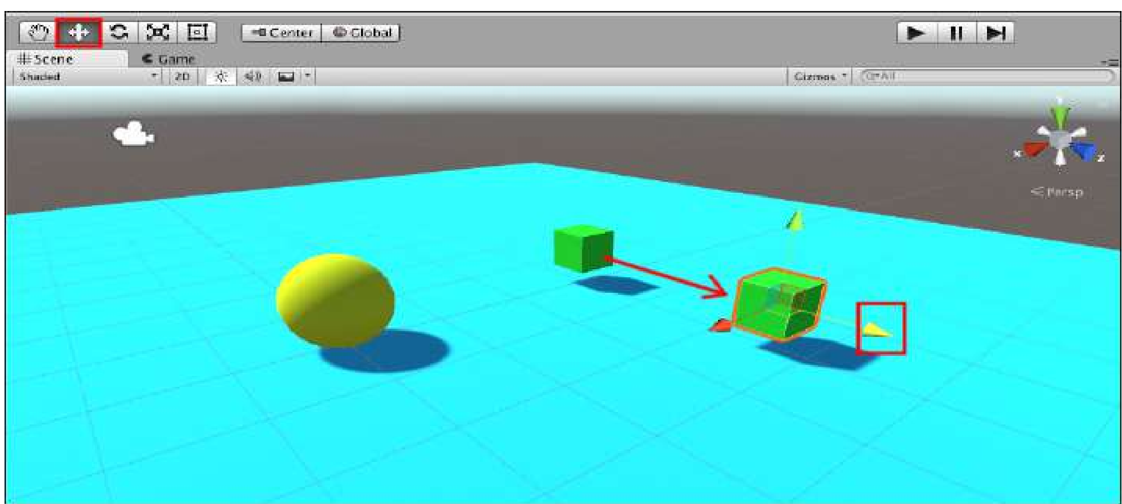


Fig. 7.50

Ahora para poder destruir un objeto que tiene un componente **Collider** utilizaremos un método llamado **OncollisionEnter.** Para entender mejor sus funciones debes mirar la documentación de Unity. Puedes acceder al enlace que te facilito.

https://docs.unity3d.com/2018.1/Documentation/ScriptReference/Collider.OnCollisionEnter.html

Este método se llama cuando un RigidBody empie za a tocar un objeto con un Collider. Esta acción nos permite detectar el RigidBody del Player. Este método tiene un parametro llamado other (otro) que son los datos de colisión asociados con el evento de colisión que sucede.

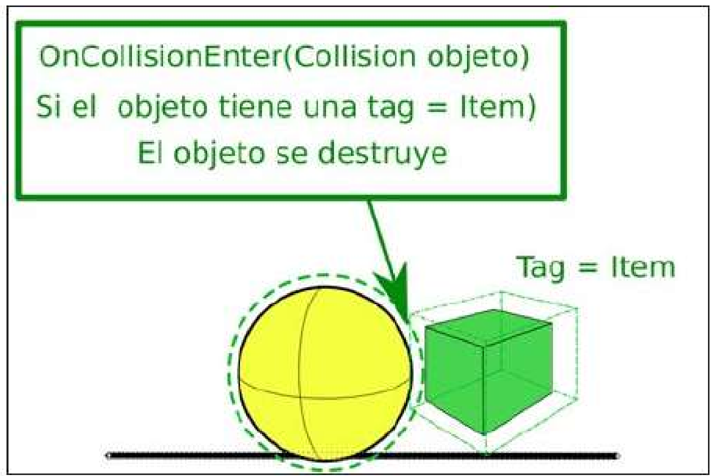


Fig. 7.51

En esta actividad vamos a añadir un nuevo script a nuestro Player para facilitar la explicación del método **OnCollisionEnter**. Dentro de la carpeta Scripts crea un nuevo script con el nombre **destroyItem** y arrastra el script encima del objeto **Player.** Para abrir el script hacemos doble clic encima del script y se nos abrirá con el editor **Monodevelop.** Este método nos permite detectar objetos mediante su collider y en este caso en concreto sabremos a que objeto nos referimos porque lleva una etiqueta.

Script: destroyItem.cs

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

public class destroyItem : MonoBehaviour
{
    void OnCollisionEnter(Collision objeto)
    {
   if (objeto.gameObject.tag == “Item”)
   {
      Destroy (objeto.gameObject);
   }
    }
}
```

Creamos una función OnCollisionEnter y entre paréntesis creamos una variable de tipo Collision y dentro de la función comparamos si la variable tiene la etiqueta Item le decimos que destruya el objeto. Recuerda que hay que guardar el script desde Monodevelop para que se ejecuten los cambios.

Una vez volvemos a Unity y ejecutamos la escena verás que cuando la bola colisione con los Los Cubos con etiqueta Items estos se destruirán y desaparecerán de la escena.

El método Destroy pertenece a la clase Object y nos permite eliminar objetos de distinta forma, te facilito un enlace a la documentación de Unity para más información sobre este método :

https://docs.unity3d.com/2018.1/Documentation/ScriptReference/Object.Destroy.html

Método	Explicación
Destroy(gameObject,tiempo)	Sirve para que el objeto se destruya una vez pase cierto tiempo. El valor debe ser un float por ejemplo 2,0f significaría que una vez pasen 2 segundos el objeto se destruye.
Destroy(this)	Destruye el gameObject al que se le ha aplicado el script. Sería como destruye este (refiriéndose a si mismo).
Destroy(componente)	En este caso podemos destruir un componente del objeto en concreto.

13. Tele-transportación con Triggers

Ya conocemos los colliders y ahora vamos a ver como utilizar la opción que tienen llamada **Trigger**. En principio el collider es como un campo de fuerza que nos permite colisionar con otros objetos, pero en el caso de que quisiéramos utilizar este collider para que cuando un objeto entre o salga suceda algo como por ejemplo tele-transportarse a otro lugar, debemos activar la opción **Trigger**. Esta opción permite que el campo de fuerza collider no ejerza ninguna fuerza de colisión y nos permita detectar objetos que entren, permanezcan o salgan del campo de fuerza. En la siguiente imagen te muestro dentro de un componente **BoxCollider** donde se encuentra la opción **Trigger**.

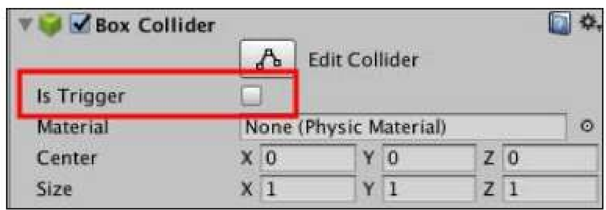


Fig. 7.52

Para el ejemplo que vamos a realizar vamos a duplicar la escena actual es decir la Escena_3. Primero guarda la Escena actual accediendo al menú principal **File > Save Scene** de este modo nos aseguramos que todo lo que hemos hecho en la escena actual queda guardado.

Hecho lo anterior volvemos al menú principal y accedemos a **File > Save Scene As...** se nos abrirá una ventana del navegador accedemos a la carpeta Escenas y guardamos nuestra escena con el nombre de Escena_4. Ahora a pesar de que la escena no ha mostrado ningún cambio todo lo que realicemos ahora no afectará a la escena anterior.

Para la Escena_4 vamos a eliminar todos los cubos que teníamos del ejemplo anterior, para ello selecciona el nombre dentro de la ventana Jerarquía y pulsa la tecla Suprimir (Supr) del teclado. En la escena solamente debemos tener el Player y el suelo, de momento.

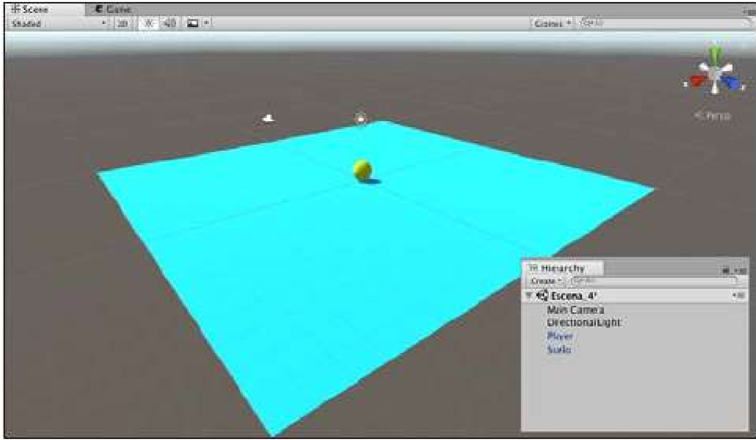


Fig. 7.53

Ahora vamos a crear un GameObject vació accediendo al menú principal y accediendo a **GameObject > Create Empty.** Accedemos a los componentes de este objeto desde la ventana Inspector y le cambiamos el nombre por el de Portal como te muestro en la siguiente imagen.

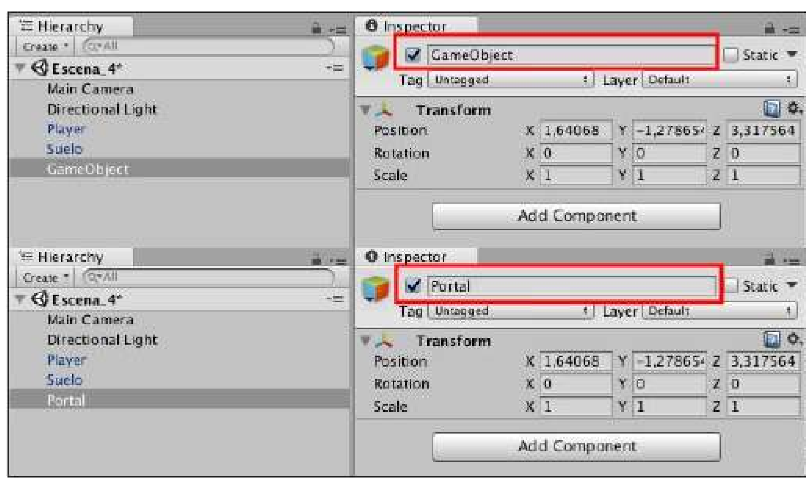


Fig. 7.54

Ahora vamos añadirle un componente **BoxCollider** accediendo al botón inferior de la ventana **Inspector** con nombre **Add Component** > **Physics** > **Box Collider** automáticamente se nos agregará este componente al que activaremos la opción **Is Trigger.** Otros aspectos que debemos cambiar para que el ejemplo sea igual que el que te explico es cambiar los siguientes parámetros que te muestro a continuación.

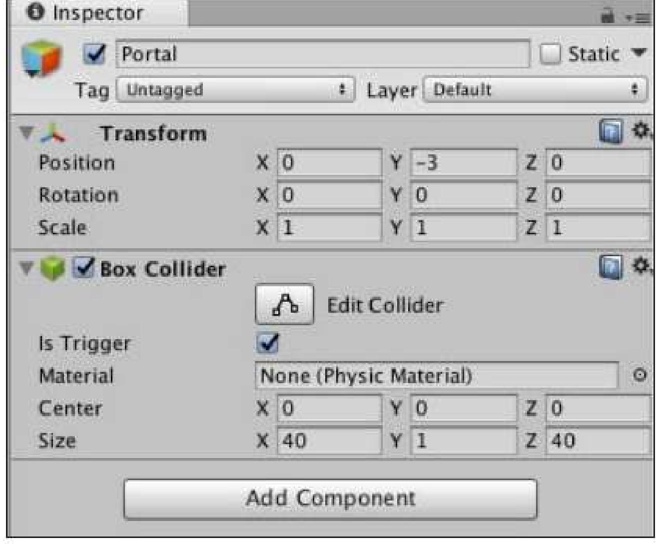


Fig. 7.55

En el componente **Transform** cambiamos los parámetros de posición en X=0, Y= -3, Z=0. En el componente **Box Collider** nos aseguramos de que la opción **IsTrigger** este activo y el parámetro **Size** tenga los siguientes valores X= 40, Y=1, Z=40. Si todo es correcto la escena debería quedarnos de la siguiente manera.

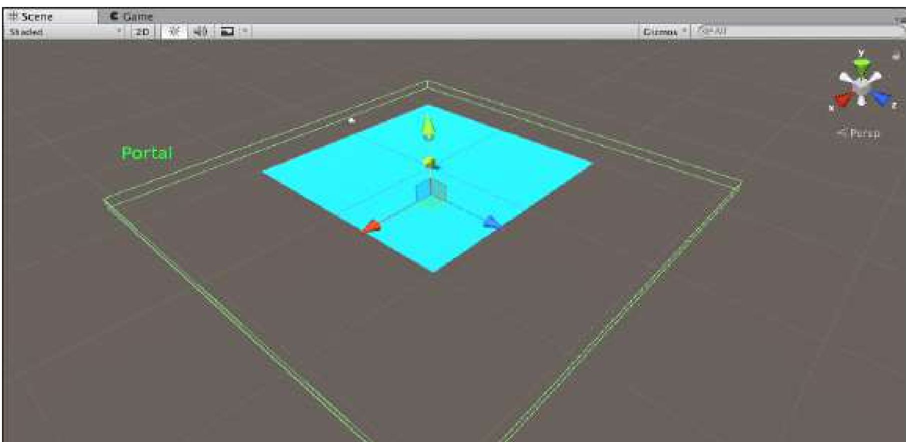


Fig. 7.56

El objeto portal al ser un objeto vacío no será visible en el momento que seleccionemos otro objeto, no debe preocuparte, la idea es que este colocado y escalado como se muestra en la imagen anterior. El ultimo objeto que vamos a añadir a nuestra escena es otro Objeto vacío que le vamos a dar de nombre **Posicion**.

Para crear este objeto realizamos la misma acción, accedemos al menú principal en el menú **GameObject > Create Empty** y accedemos a los componentes de este objeto desde la ventana Inspector y le cambiamos el nombre por el de **Posicion.** Como es un objeto vacío no se mostrara en la escena, por ese mismo motivo vamos a ponerle un Icono de color rojo y lo posicionamos como te muestro a continuación. ***Posición (x=0, y=3, z=0)***

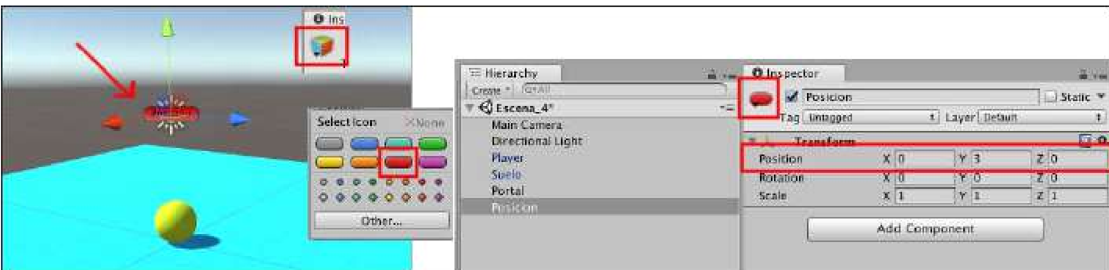


Fig. 7.57

Una vez tenemos la escena preparada como te he explicado guarda la escena. Ahora el ejemplo consiste en que nuestro **Player** en el momento que rebase el suelo, caiga al vacío y tome contacto con el objeto Portal, nuestro Player vuelva a la posición que marca el objeto **Posicion.** Esta sería una forma de utilizar la opción **Trigger.**

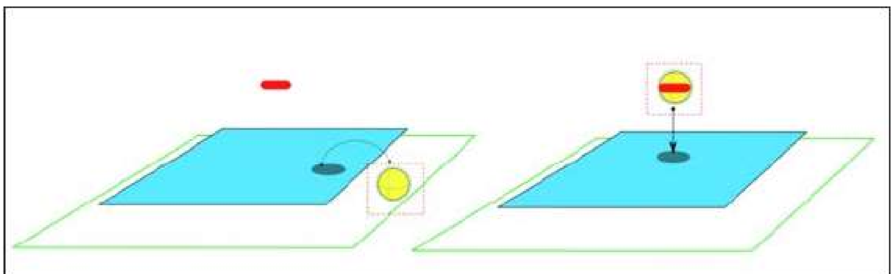


Fig. 7.58

Antes de continuar asegúrate de ponerle una etiqueta Tag con el nombre de Player a tu pelota. Para ello seleccionamos nuestro objeto **Player** y en la ventana Inspector accedemos al menú Tag y seleccionamos la etiqueta Player como te muestro en la siguiente imagen.

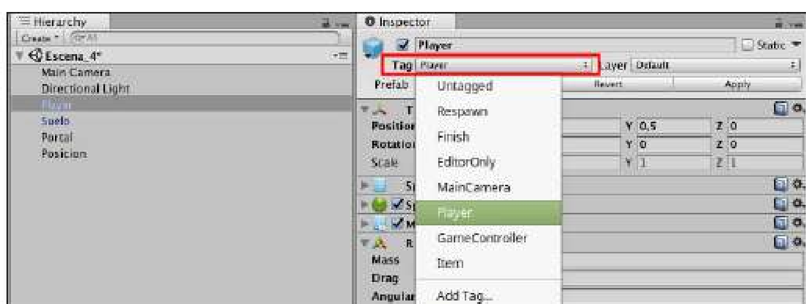


Fig. 7.59

Ahora dentro de la carpeta Scripts creamos un script con nombre **Teletransportacion** y lo arrastramos dentro de la ventana Jerarquía encima del nombre Portal que es el objeto que dispone del **Trigger** activado. Una vez agregado el script hacemos doble clic encima del Script y en Monodevelop empezamos a editar.

Script: Teletransportacion.cs

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

public class Teletransportacion : MonoBehaviour
{
   public GameObject miPlayer;
   public GameObject miPosicion;

   void Awake()
   {
      this.miPlayer = GameObject.FindWithTag(“Player”);
      this.miPosicion = GameObject.Find(“Posicion”);
   }

   void OnTriggerEnter(Collider otro)
   {
      if (otro.tag == “Player”)
      {
         Debug.Log(“Volvemos al principio”);
         this.miPlayer.transform.position = this.miPosicion.transform.posi-
tion;
      }
 }
```

Necesitamos encontrar dos objetos: el de la esfera (Player) y el del punto de origen (Posicion) para tele-transportar la esfera. Para ello crearemos dos variables de tipo GameObject una con nombre miPlayer para almacenar el objeto Player y la otra con nombre miPosicion para almacenar la información del GameObject vacío que hemos puesto como posición de inicio.
En la función Awake decimos que la variable miPlayer es el GameObject que tiene una etiqueta “Player” (Debes ponerle la etiqueta Player a la bola para que funcione). La variable miPosicion sera el GameObject que tiene el nombre Posicion.(“Cuidado si a este objeto le has puesto otro nombre no lo va a encontrar”).

Ahora vamos a utilizar la función OnTriggerEnter para decir que cuando un objeto entre en el collider que tiene activada la opción Trigger pasará algo. Entre paréntesis ponemos un atributo de tipo Collider llamado otro (Aquí puedes llamarle otro o el nombre que tú quieras lo importante es que sea de tipo Collider).

Entre llaves queremos que el objeto que se tele-transporte sea solamente la esfera Player, por eso creamos un if y decimos que cuando el collider que entre contenga la etiqueta Player(es decir sea la esfera), la posición del objeto Player que está guardada en la variable miPlayer se cambiará y tendrá el valor de la posición del objeto Posicion que está guardada en la variable miPosicion. También he añadido un mensaje para la consola cada vez que caigamos dentro del Trigger nos enviará un mensaje.
Recuerda que hay que guardar el script desde Monodevelop para que se ejecuten los cambios.

Una vez vuelvas a Unity lo primero que debes comprobar cuando ejecutes la escena es que en la ventana Inspector localiza los objetos del Script, para hacer esta comprobación debes seleccionar el objeto Portal desde la ventana Jerarquía y cuando ejecutes la escena mirar en la ventana Inspector que las variables localizan los objetos de la escena.

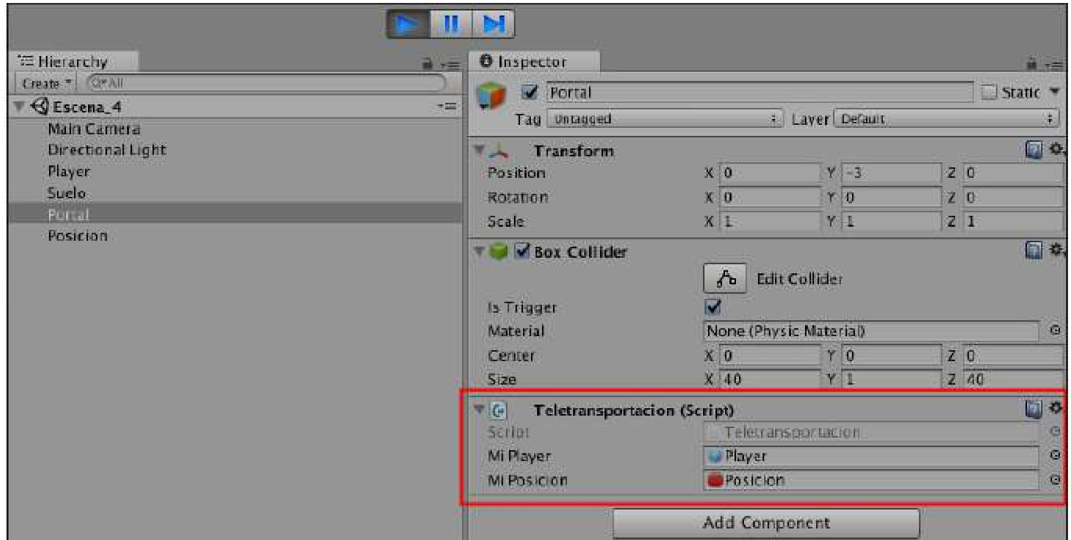


Fig. 7.60

Si todo es correcto y al ejecutar la escena localiza el objeto Player y el objeto Posicion solamente debes mover con las flechas del teclado el objeto Player y comprobar que cuando caes al vacío nuestro objeto Player se posiciona en el lugar donde tenemos el objeto Posicion.

Esta función OnCollision, OnTrigger pertenecen a la clase Collider el tema Triggers los seguiremos viendo en el siguiente capitulo. Si deseas más información puedes consultar en la documentación de Unity. Puedes acceder desde el enlace que te facilito a continuación.

https://docs.unity3d.com/2018.1/Documentation/ScriptReference/Collider.html

En resumen

En este capitulo has aprendido a crear controles para tus personajes, y hemos empezado a introducir comportamientos básicos con Colliders y Triggers, conforme vayas avanzando en los próximos capítulos iras viendo como podemos enriquecer todo los conceptos aprendidos hasta ahora. Por ultimo has visto como podemos destruir objetos de la

escena mediante una colisión. En el principio del capitulo te comento que tenemos dos paquetes de assets con materiales uno era para trabajar todos los conceptos que hemos visto hasta ahora, el segundo paquete es para que te pongas a prueba con todo lo que hemos aprendido en el capitulo.

En el siguiente apartado te propongo un proyecto a desarrollar. Si has seguido todo el capítulo no tendrás ningún problema.

14. Proyecto final

Para este proyecto vamos a guardar todo lo que hemos hecho anteriormente y creamos un nuevo proyecto 3d con nombre **Proyecto_Capitulo7.**

Una vez lo hayas creado importamos el paquete **Práctica_Capitulo_7.unitypackage** del material que acompaña la obra. Para importar accedemos al menú principal **Assets > Import Package > CustomPackage...** Se abrirá una ventana de tu sistema en donde debes buscar donde está guardado el material adicional de la obra. Una vez lo encuentres en la carpeta Proyecto_7 selecciona el paquete con el nombre **Práctica_Capitulo_7. unitypackage s**e nos aparece una nueva ventana en donde podemos ver el contenido del paquete y Unity nos da la posibilidad de seleccionar que queremos importar. Seleccionalo todo si no está seleccionado y para que ejecute la importación hacemos clic encima del botón **Import** que se encuentra en la parte inferior derecha.

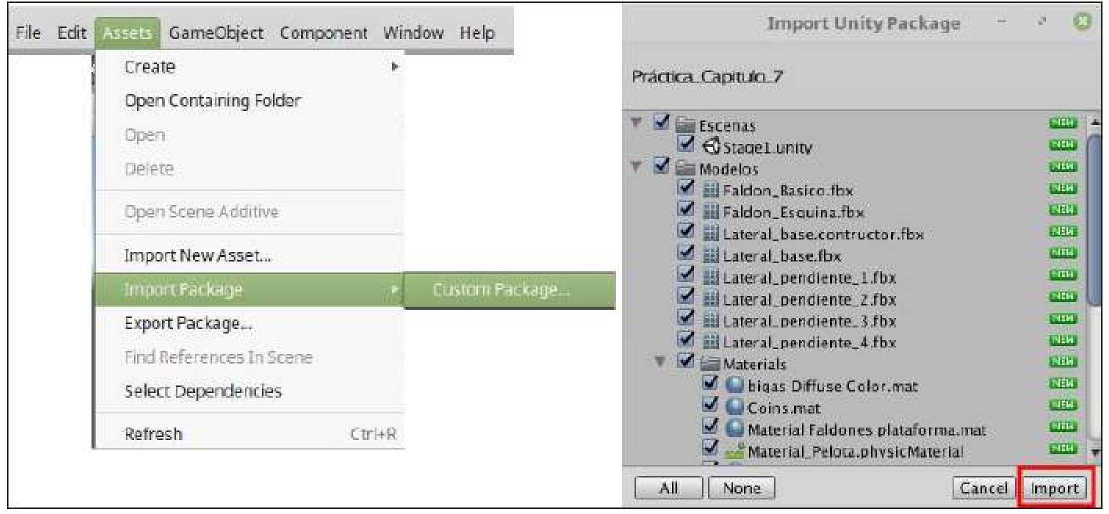


Fig 7.61

Este paquete que se importara dispone de varias carpetas con tres carpetas principales Escenas Modelos y Prefabs. Dentro de Modelos también encontramos otra carpeta llamada Materiales. Para que la práctica se enfoque en el contenido del temario y no tengas que crear un escenario desde cero, en la ventana Project dentro de la carpeta Escenas disponemos de una escena llamada Stage1.

Fig. 7.62

Si hacemos doble clic encima del icono Stage1 dentro de la ventana **Project** se nos aparece un escenario montado como el que te muestro en la siguiente imagen.

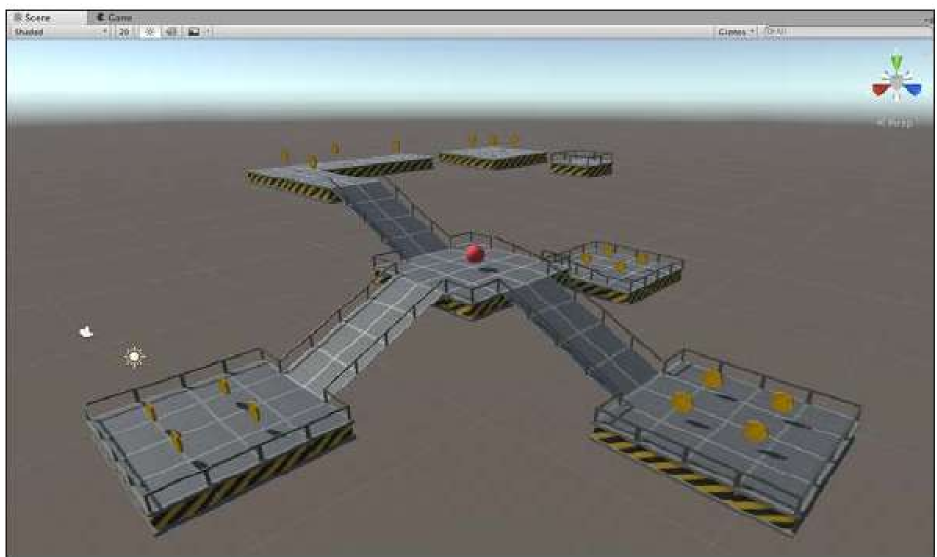


Fig. 7.63

Este escenario tiene muchos elementos pero el proyecto consiste en los siguientes enunciados:

*#1 Para el objeto Esfera con nombre Player un script que controle la pelota mediante RigidBody por medio de las teclas de flecha y un salto con la tecla espacio. Cuidado que el objeto Player por defecto no tiene ningún componente **Rigidbody**. También debemos crear un método para destruir monedas (Por defecto las monedas tienen un **Sphere Collider** con **Is Trigger** activado).*

*#2 Para el objeto **Main Camara** crear un script para que siga la posición del Player.*

*#3 Para los objetos Monedas crear un script que las haga rodar. Si tienes dudas de como hacerlo consulta el **capítulo 6** el apartado rotar objetos. Todas las monedas tienen una etiqueta "Monedas".*

*#4 Para finalizar deberás crear un objeto vacío con un **Box Collider** con la opción **Trigger** activada que detecte cuando la pelota **(Player)** cae al vacío y la tele-transporte a la posición de un segundo objeto que también debes crear y posicionar en el inicio de la pelota.*

Recuerda que en este paquete de Assets, debes crear una carpeta para los Scripts, es importante que te acostumbres a crear las carpetas tu mismo cuando las necesites y ser muy ordenado con tu trabajo.

Si has seguido el capitulo desde el principio y en orden, deberías poder realizar este proyecto sin problema, si tienes dudas puedes volver a los apartados anteriores y consultar los scripts. Este proceso es importante porque te va ayudar a leer código y poder extrapolar, interpretar y crear tus propios scripts.

Capítulo 8

Raycast y Decals

- Introducción
- Abrir puertas con Triggers
- Raycast
- Cómo obtener información con RaycastHit
- Comunicación con SendMessage
- Decals
- Instanciar Decals con Raycast
- Rotación de nuestros Decals
- Selección de objetos para disparar
- Crear un Array de decals
- Emparentar los Decals
- Destruir cajas

1. Introducción

En este capitulo vamos a trabajar la comunicación entre objetos, es decir como interactuar con algunos objetos. En el capitulo anterior vimos un poco como utilizar los colliders y tuvimos un primer contacto con los **triggers** en este capitulo vamos a seguir utilizando los **triggers** para animar una puerta, entenderás que es un **Raycast** y como podemos utilizarlo. Para finalizar veras como podemos detectar objetos con **Raycast,** utilizaremos **decals** para mostrar un agujero como si disparáramos un arma en distintos objetos de una escena y acabaremos destruyendo las cajas de la escena.

Para seguir el capitulo en el material que acompaña el libro tenemos dentro de la carpeta Proyecto_7 dos paquetes para importar uno es el **Assets_Capitulo_7.unitypackage** que es el que vamos a necesitar para el principio y el otro es el **Práctica_Capitulo_7.unitypackage** que es el que importaremos al final del capitulo.

Para empezar crea un nuevo proyecto 3d en Unity y ponle el nombre que quieras yo te propongo como nombre Capitulo_8. Una vez lo hayas creado importamos el primer paquete **Assets_Raycast_Capitulo_8.unitypackage** del material que acompaña la obra. Para importar accedemos al menú principal **Assets > Import Package > CustomPackage...** Se abrirá una ventana de tu sistema en donde debes buscar donde está guardado el material adicional de la obra. Una vez lo encuentres en la carpeta Proyecto_8 selecciona el paquete con el nombre **Assets_Raycast_Capitulo_8.unitypackage, s**e nos aparece una nueva ventana en donde podemos ver el contenido del paquete y Unity nos da la posibilidad de seleccionar que queremos importar. En este paquete disponemos de cuatro carpetas principales una con Escenas preparadas para realizar los ejemplos, Materiales, P**refabs** y una carpeta con el nombre **Standard Assets** que contiene un **FPSController** con el que vamos a trabajar. Déjalo todo seleccionado. Para que ejecute la importación hacemos clic encima del botón **Import** que se encuentra en la parte inferior derecha.

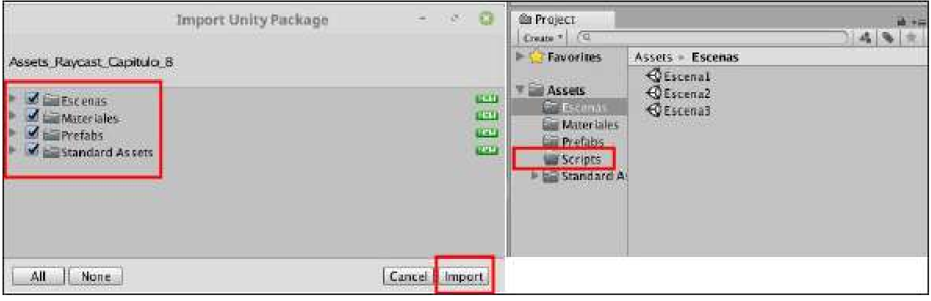


Fig. 8.1

En la ventana Project podrás ver como se han importado las carpetas con todo su contenido, en la imagen anterior verás que he creado una carpeta con el nombre Scripts para guardar todos los scripts, esta carpeta tienes que crearla una vez hayas importado el paquete, porque no viene incluido en los assets del paquete.

2. Abrir puertas con triggers

Una forma en la que aprendí a entender el funcionamiento de los triggers y las transformaciones de los objetos es intentando abrir puertas mediante scripts. En este apartado

vamos a ver como animar una puerta que esta cerrada a una posición abierta, cuando detecta un objeto, en este caso nuestro **FPSController**.

En la ventana **Project** en la carpeta **Assets > Escenas** seleccionaremos haciendo doble clic encima de la Escena1. Automáticamente se nos aparecerá una escena que se compone de un **FPSController** que sera nuestro **Player** y una puerta en el centro de la escena que también encontraras dentro de la carpeta Prefabs.

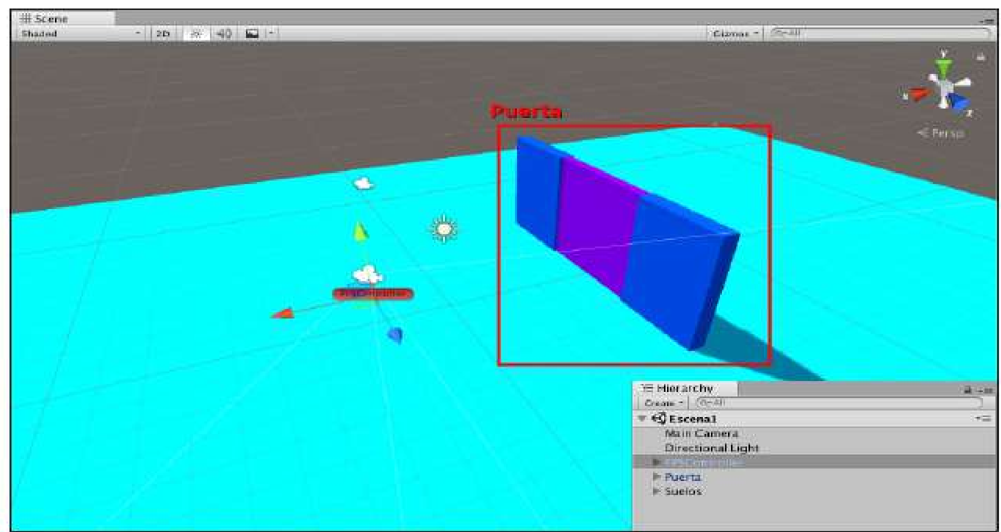


Fig. 8.2

Primero vamos a ver de que se compone esta puerta que encontramos en la escena.

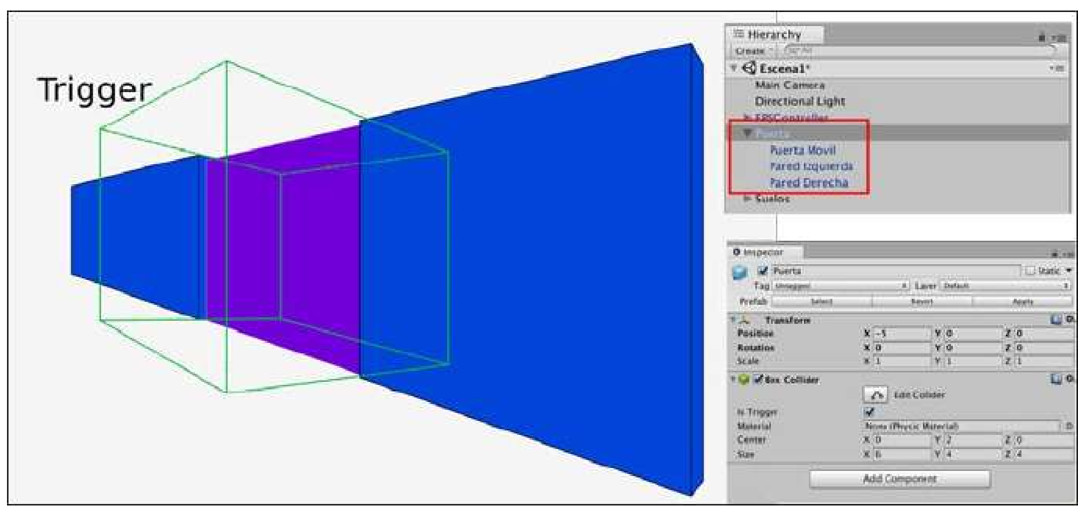


Fig. 8.3

En la imagen anterior vemos en la ventana Jerarquía de Unity un **GameObject** con nombre Puerta que es padre de tres elementos : Puerta Móvil, Pared Izquierda y Pared Derecha.

Si seleccionamos el objeto Puerta que es el padre de todos los demás objetos y miramos sus componentes en la ventana Inspector veremos que tiene un **Box Collider** en

donde se le ha activado la opción **Trigger**. El **Trigger** va a ser el responsable de detectar el objeto Player para dar la orden de que se abra la puerta y también va a ser el responsable de detectar cuando el Player sale y dar la orden de que la puerta se cierre.

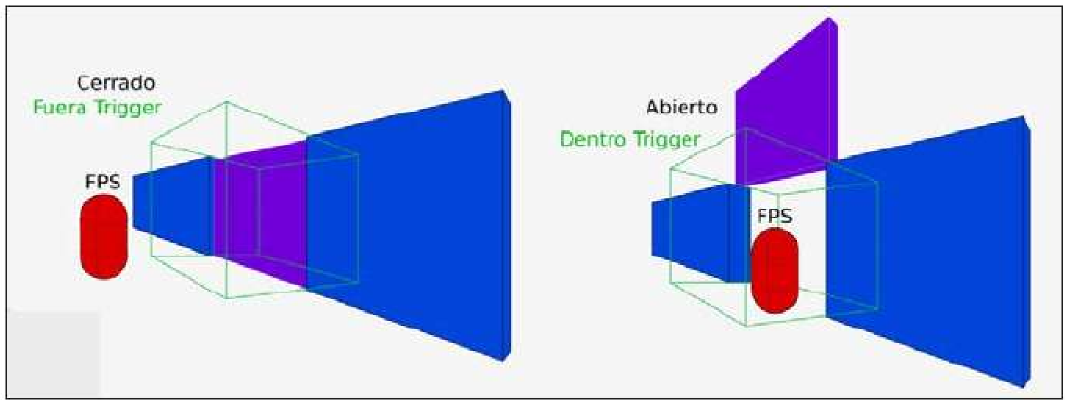


Fig. 8.4

Los otros objetos que son hijos del Objeto Puerta llevan un box collider sin activar la opción Trigger de este modo nuestro **FPS** no podrá atravesar las paredes ni la puerta móvil. Ahora vamos a crear un script que añadiremos al Objeto Puerta. Para ello podemos hacer lo siguiente:

Selecciona el objeto Puerta desde la ventana Jerarquía (Hierarchy), en la ventana inspector vamos a la parte infreior y accedemos al botón **Add Component > NewScript** le ponemos el nombre **ControlPuerta.cs**

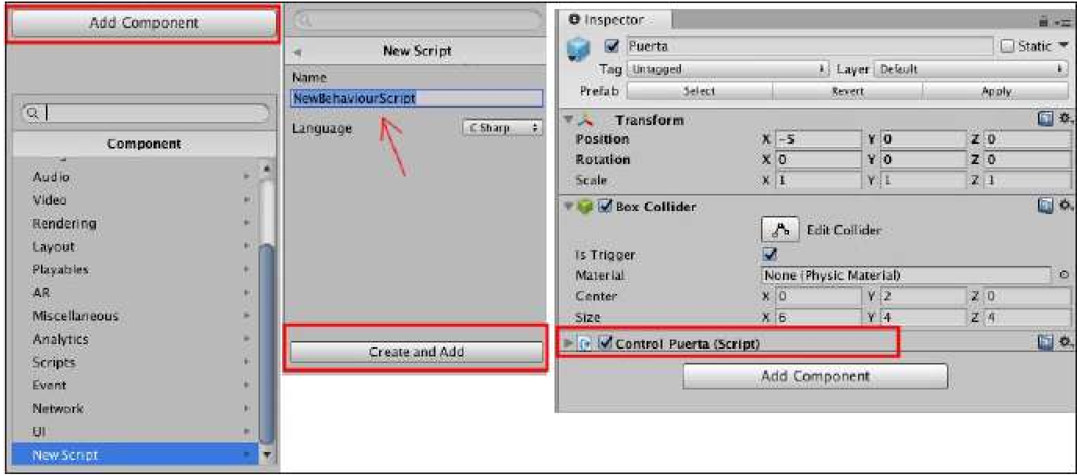


Fig. 8.5

La ventaja de crear un script de este modo es, que el script queda directamente aplicado al objeto que queremos, el inconveniente es que el script se ha creado fuera de la carpeta Scripts, por ese mismo motivo debemos de ir a la ventana **Project** y seleccionando la carpeta principal assets seleccionar el Script y arrastrarlo dentro de la carpeta Scripts.

Fig. 8.6

Para finalizar, guarda el proyecto accediendo al menú principal **File> Save Project** y guardar la escena accediendo al menú principal **File> Save Scene** , ahora accedemos a la carpeta scripts y hacemos doble clic encima del script que hemos creado para que se nos abra el editor Monodevelop y podamos editarlo.

Script: ControlPuerta.cs

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

public class ControlPuerta : MonoBehaviour
{
    public bool abriendo= false;
    public bool cerrando= false;
    public bool abierta= false;
    public Transform puertaPosicion;
    public Vector3 posicionAbierta;
    public Vector3 posicionCerrada;
    public Vector3 posicionFinal;

    public float recorrido;
    public float tiempoInicio;
    public float tiempoApertura;

    void Start ()
    {
        this.puertaPosicion = gameObject.transform.GetChild (0);
        this.posicionCerrada = this.puertaPosicion.transform.localPosition;
```

```
        this.posicionFinal = new Vector3 (0f, 4f, 0f);
        this.posicionAbierta = this.posicionCerrada + this.posicionFinal;

    }

    void OnTriggerEnter (Collider colisionador)
    {
        this.tiempoInicio = Time.time;
        this.abriendo = true;

    }

    void OnTriggerExit (Collider colisionador)
    {
        this.tiempoInicio = Time.time;
        this.cerrando = true;

    }

    void Update ()
    {
        if (this.abriendo)
        {
          this.recorrido = (Time.time - this.tiempoInicio) / tiempoApertura;
          this.puertaPosicion.transform.localPosition = new Vector3(0f,Mathf.
Lerp(this.posicionCerrada.y,this.posicionAbierta.y,this.recorrido),0f);
          if (this.puertaPosicion.localPosition.y == this.posicionAbierta.y)
          {
                  this.abriendo = false;
          }
        }
        if (this.cerrando)
        {
          this.recorrido=(Time.time - this.tiempoInicio) / tiempoApertura;
          this.puertaPosicion.transform.localPosition = new Vector3(0f,Mathf.
Lerp(this.posicionAbierta.y,this.posicionCerrada.y,this.recorrido),0f);
          if (this.puertaPosicion.localPosition.y == this.posicionCerrada.y)
          {
                  this.cerrando = false;
           }
        }
     }
}
```

Declaración de variables
He creado tres variables de tipo boolean en donde guardaré el estado de cuando se está abriendo, de cuando se está cerrando y de cuando la puerta está abierta.
También he creado una variable de tipo Transform para acceder a la posición de la puerta por ese motivo la he llamado puertaPosicion.

Después tenemos tres variables de tipo Vector3 para almacenar la posición de la puerta abierta, la posición de la puerta cerrada y un posición final que me servirá para encontrar la posición abierta que ahora veremos.
También he creado tres variables de tipo float para que almacenar el tiempo de inicio, el tiempo de apertura y con ello crear un valor para guardar el recorrido y poder modificar la velocidad de la puerta.

La función Start
Aquí vamos a buscar las referencias y a almacenar la información que necesitamos al ejecutar la escena. Primero utilizo la variable de tipo Transform con el nombre puertaPosicion para acceder al las transformaciones de la puerta. Si recordamos estamos en el objeto Puerta que es el padre de la puerta móvil. Para acceder a la puerta móvil primero nos referiremos al propio gameObject en minúscula y utilizamos el método transform.GetChild para acceder a su hijo, después entre paréntesis debemos decirle en que posición se encuentra el hijo, en este caso tenemos la puerta móvil en primer lugar eso le corresponde el valor 0.

Ahora en la siguiente línea necesitamos la referencia de la puerta cerrada para la variable posicionCerrada, para ello utilizamos la variable puertaPosicion porque ya dispone de las transformaciones de la puerta y le decimos que posición cerrada será igual a la posición de la puerta en modo local.
La siguiente línea es la posición final, en este caso lo que he hecho es mover la puerta dentro de Unity en la ventana escena en el eje (y) ver el valor de este eje con la puerta abierta y en el script he creado un nuevo Vector3 con el valor de la posición final que quiero que tenga la puerta.
Para encontrar la información de la siguiente variable que es la posiciónAbierta he hecho una simple operación sumar la posición cerrada + la posición final.

Los Triggers
En esta sección del script he utilizado dos métodos OnTriggerEnter para cuando detecte un objeto con colisionador como nuestro player y reaccione. Dentro de esta función quiero que la puerta se abra, así pues a la variable de Tiempo inicio le he dado el valor del Tiempo con Time.time y a la variable de tipo booleana abriendo le puesto el estado de true, porque la puerta se abre.

Para el método OnTriggerExit quiero que reacciones cuando el player salga del trigger y para eso también guardo el momento en la variable tiempo Inicio con Time.time y en este caso la variable booleana cerrando será verdadera porque la puerta se tiene que cerrar.

Función Update
En esta función vamos a crear dos condiciones una si se cumple que la variable abriendo es verdadera y otra si se cumple que la variable cerrando es verdadera.

Si abriendo es verdadera primero calcularemos el recorrido que será una formula muy básica que es el tiempo actual menos el tiempo inicial dividido por el tiempo de la apertura (el tiempo de la apertura deberás introducirlo en la ventana inspector de Unity).
A continuación crearemos una animación primero diciendo que la posición local de la puerta es igual a un nuevo vector 3 (x,y,z) este vector 3 tiene en (x) y en (z) los valores 0, y el que nos interesa que es el eje (y) utilizaremos una funcion Mathf.Lerp que nos pide una posición inicial en este caso le damos la cerrada, una posición final en este caso la abierta y una distancia que en este caso es el recorrido.
Para terminar creamos otra condición más, mientras se está abriendo, si la posición de la puerta en el eje (y) es el mismo que la posición final es decir, que la posición de la puerta está abierta, se cumplirá que ya no se está abriendo.
Recuerda que hay que guardar el script desde Monodevelop para que se ejecuten los cambios.

Este ejemplo es perfecto para practicar las transformaciones que vimos en el capitulo 6, así que si no recuerdas muy bien como mover objetos revisar otra vez dicho capitulo. Una vez tengamos editado el script volvemos a Unity y debemos mirar primero que no tenga errores y luego con el objeto Puerta seleccionado accedemos a la ventana Inspector, para ver el componente Script. Luego le ponemos un valor al parámetro Tiempo Apertura. En este caso en concreto le he puesto el valor 3, pero puedes ponerle el valor que creas más conveniente.

Fig. 8.7

Una vez ya tenemos el valor en el parámetro Tiempo Apertura ejecutamos la escena y puedes comprobar moviendo el FPSController hacia la puerta como esta se abre cuando te acercas y se cierra cuando te alejas.

3. Raycast

El Raycast es un rayo invisible que colisiona con los otros colisionadores (Colliders) y nos devuelve una información. Por ese mismo motivo es tan importante los colliders.

Para los Raycast necesitamos siempre:

- un punto de origen
- un punto final o distancia

El rayo es disparado desde el punto origen y este atraviesa la escena hasta encontrar el punto final o recorrer la distancia. Durante el trayecto el rayo puede colisionar con otros objetos que lleven colisionadores, cuando sucede esto te devuelve información del objeto con el que ha colisionado.

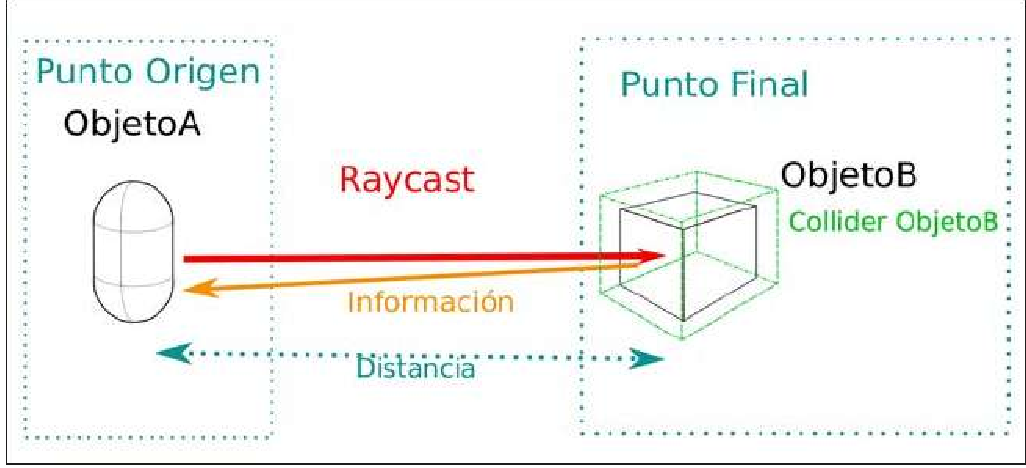


Fig. 8.8

Necesitamos un **Raycast** cuando queremos saber a que distancia estamos de un objeto o que distancia hay entre dos objetos etc..

Para utilizar **Raycast** necesitamos colliders y físicas.

Tenemos varios tipos de **Raycast**:

- **Physic.Raycast** que es el que vamos a utilizar
- **Plane.Raycast** que es cuando el rayo intercede un plano.
- **NavMesh.Raycast** para personajes que controla la maquina.
- **Collider.Raycast** que son creados a partir de colisionadores.

Hay algunos más pero no debes preocuparte porque todos funcionan de un modo similar, así que si entiendes bien el temario que vas ha ver, no tendrás problema alguno en desarrollar funcionalidad con los otros métodos.

Para empezar vamos a la documentación de Unity para ver sus métodos y atributos que podemos utilizar. Es de extrema importancia que vayas viendo la documentación para aprender a utilizar bien Unity.

https://docs.unity3d.com/2018.1/Documentation/ScriptReference/Physics.Raycast.html

En la documentación verás que la información que nos devuelve un Raycast de tipo booleano si toca un objeto nos devuelve true y si no toca ningún objeto nos devolverá un false.

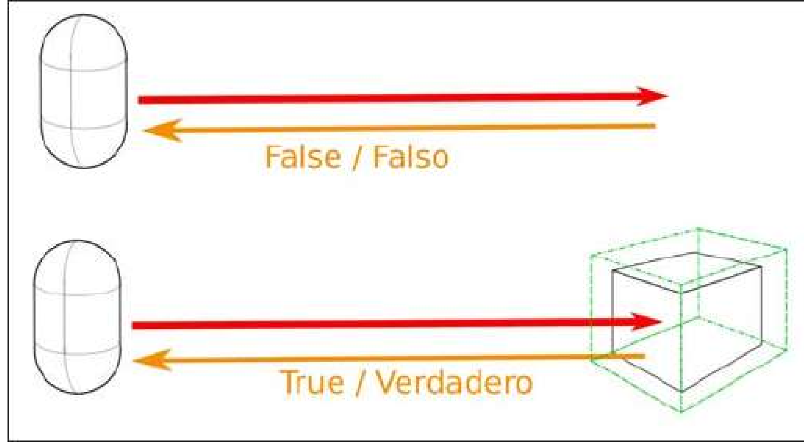


Fig. 8.9

Cómo Visualizar el Raycast

El rayo de Raycast es invisible por defecto, pero podemos utilizar un método para dibujarlo en la ventana escena. El método que vamos a utilizar se llama Debug.DrawRay, esto nos permite dibujar el Rayo pero no nos va a proporcionar información sobre los objetos con los que colisiona el rayo. Como siempre te invito a que accedas a la documentación de Unity para mayor información.

https://docs.unity3d.com/2018.1/Documentation/ScriptReference/Debug.DrawRay.html

Para seguir el temario sin perder tiempo si te diriges a la ventana **Project** y hacemos doble clic encima del icono con el nombre Escena2 que encontrarás dentro de la carpeta Escenas, dispondrás de una escena preparada para la tarea que vamos a realizar a continuación.

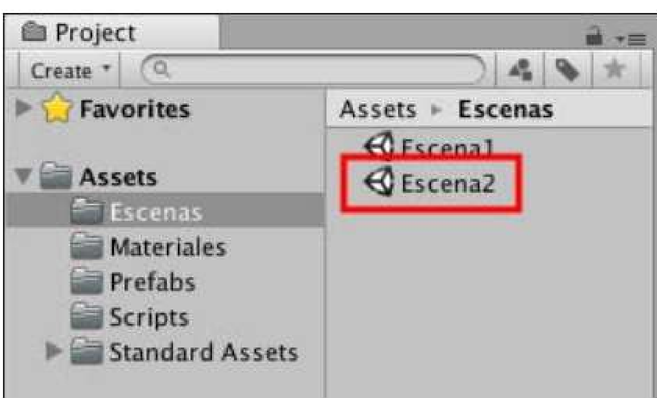


Fig. 8.10

En esta Escena2 disponemos de el **FPSController** varios planos que realizan la función de suelo y tres objetos; un cubo, una esfera y una capsula. Estos objetos se les ha añadido una etiqueta con el nombre del color del objeto que veremos más adelante para que las utilizaremos.

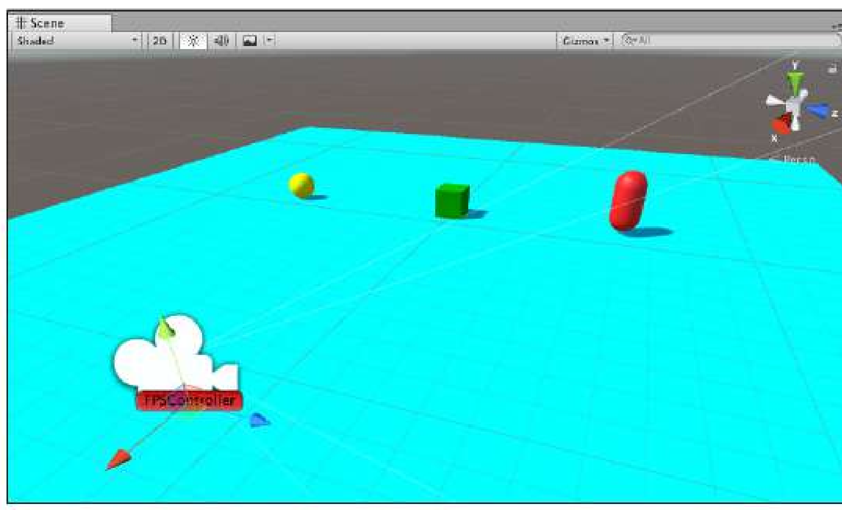


Fig. 8.11

El objetivo de este apartado es poder ver el **Raycast** en la ventana escena, para ello te recomiendo que la disposición de tu interfaz tenga la ventana escena y la ventana **Game** una al lado del otro para poder ver el **Raycast**. Este aspecto es importante porque cuando ejecutamos la escena moveremos el FPS desde la ventana Game, pero el Rayo del Raycast sera dibujado en la ventana escena.

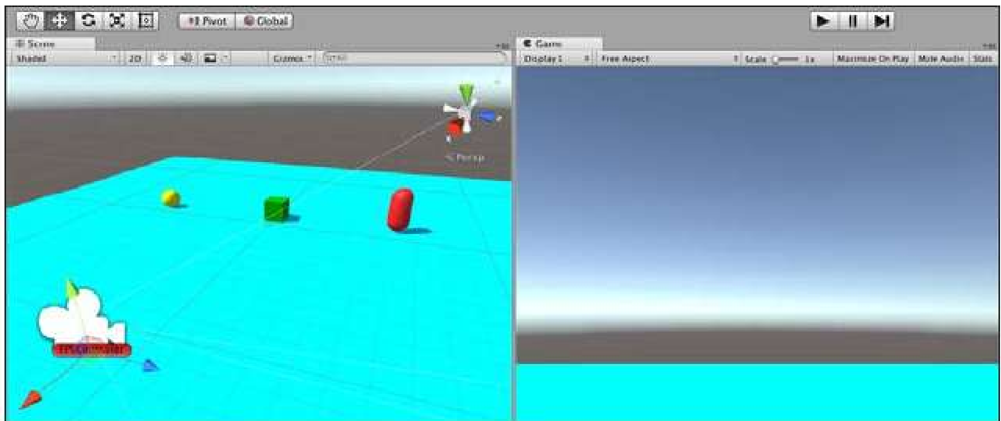


Fig. 8.12

Ahora vamos a crear un script dentro de la carpeta Scripts con el nombre **ComportamientoRay.cs** y lo arrastraremos encima del **FPSController**. Una vez hemos añadido el script al **FPSController** hacemos doble clic encima del script dentro de la carpeta Scripts de la ventana **Project** y se nos abrirá el editor **Monodevelop** para empezar a editarlo.

Script: ComportamientoRay.cs

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

```
public class ComportamientoRay : MonoBehaviour
{

    void Update ()
    {
        Debug.DrawRay(transform.position,transform.TransformDirection(Vector3.
forward)*5, Color.blue);
    }
}
```

Si seguimos las indicaciones de la documentación de Unity , dentro de la función Update pondremos Debug.DrawRay y nos pide dos argumentos es la posición de origen del FpsController, el segundo argumento es una dirección en este caso le he dado un Vector 3 de tipo forward que es los mismo que dar la posición (0,0,1) y aunque no hiciera falta, Debug.DrawRay nos permite poner otro argumento que es el color al cual le he puesto azul.
Dentro del argumento de dirección le he multiplicado por el valor 5 al Vector3.forward para que la distancia de la dirección tome el valor de 5 puesto que por defecto solo toma el valor de 1 unidad.
Recuerda que hay que guardar el script desde Monodevelop para que se ejecuten los cambios.

Una vez editado nuestro script al volver a Unity y comprobar que no hay errores, cuando ejecutamos la escena veremos como en la ventana **Escene** se dibuja una linea que sale de nuestro **FPSController** como te muestro en la siguiente imagen.

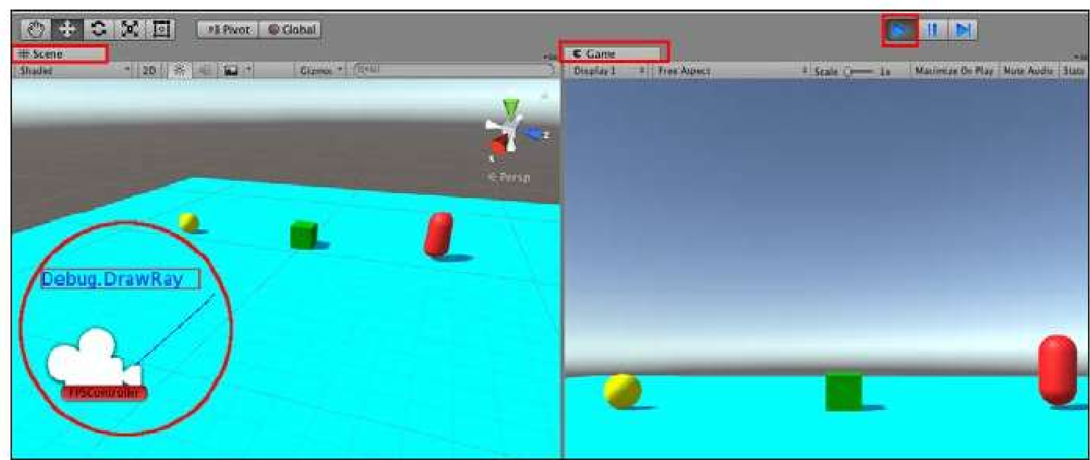


Fig. 8.13

En principio este método es simplemente para dibujar el Rayo y no nos aporta nada mas que poder visualizar el Rayo para tener una idea más visual del alcance del **Raycast**. A continuación te muestro una imagen explicando los argumentos que necesitamos para dibujar un **Raycast**.

Fig. 8.14

Entendiendo Raycast de físicas

Cuando hablamos de Physics.Raycast tenemos que tener muy claro que el rayo va dirigido a los colliders. Es decir si un objeto en escena no tiene collider este no será detectado por el rayo.

Ahora vamos a hacer que nuestro FpsController detecte los objetos con colliders y nos envíe un mensaje en la consola de Unity. Para ver como funciona vamos a crear en el mismo script que hemos hecho en el apartado anterior el siguiente código.

Script: ComportamientoRay.cs

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

public class ComportamientoRay : MonoBehaviour
{

  void Update ()
  {
    Debug.DrawRay(transform.position,transform.TransformDirection(Vector3.
forward)*5, Color.blue);

    if (Physics.Raycast (transform.position, transform.TransformDirection
(Vector3.forward), 5.0f))
    {
   Debug.Log (“Estas tocando un objeto con Collider”);
    }

  }
}
```

> Dentro de la función Update y debajo de nuestro Debug.DrawRay crearemos un if en donde utilizaremos el método Physics.Raycast, los dos primeros argumentos son exactamente iguales a Debug.DrawRay a excepción de que no multiplicamos el vector 3 por ningún valor , la distancia máxima es el tercer argumento que en este caso le he dado un valor 5 de tipo float.
> Si se cumple que el rayo detecta algo nos mostrará por consola el mensaje.
> Recuerda que hay que guardar el script desde Monodevelop para que se ejecuten los cambios.

Este ejemplo es el principio y de como funciona un **Raycast** para detectar objetos en escena. Si todo es correcto cuando ejecutes la escena y te acerques a menos de 5 unidades de distancia se nos aparecerá un mensaje en la ventana de consola, como te muestro en la siguiente imagen.

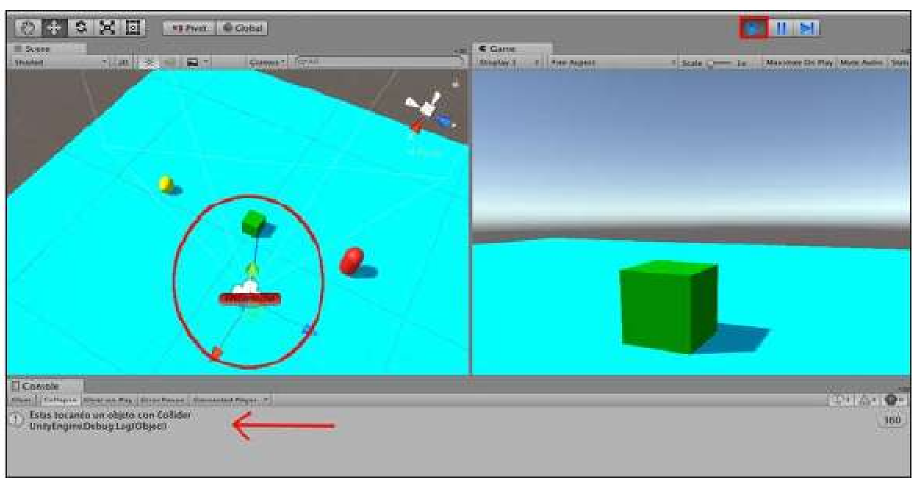


Fig. 8.15

La clase Ray

Es una forma simplificada de lo que hemos hecho anteriormente. Ray es una linea infinita que empieza en un origen y se dirige hacia alguna dirección. Para mejorar la comprensión de lo que te estoy explicando consulta en la documentación.

https://docs.unity3d.com/2018.1/Documentation/ScriptReference/Ray.html

Como puedes ver en la documentación dispone de dos atributos que son la dirección y el origen y podemos utilizar uno de sus métodos GetPoint para encontrar un punto distante.

Esto nos facilita mucho el trabajo a la hora de crear nuestros Scripts puesto que podemos utilizar los atributos de Ray.origen y Ray.direction.

Para utilizar los atributos de Ray vamos a hacer los siguiente con el mismo script

Script: ComportamientoRay.cs

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

public class ComportamientoRay : MonoBehaviour
{
   private Ray rayo;

   void Update ()
   {
      this.rayo.origin = transform.position;
      this.rayo.direction = transform.TransformDirection(Vector3.forward);

      Debug.DrawRay(rayo.origin,rayo.direction*5,Color.blue);

       if (Physics.Raycast ( rayo, 5.0f))
      {
       Debug.Log ("Estas tocando un objeto con Collider");
       }
   }
}
```

Primero creamos una variable de tipo Ray que llamaremos rayo, dentro de la función update accederemos a los atributos de la clase Ray que son origin y direction. A origin le daremos la posición del objeto en si, y a direction le daremos un Vector3 de tipo forward.
Ahora solamente debemos substituir los atributos de rayo dentro del Debug.DrawRay y en la condición if dentro del Physics.Raycast, solamente debemos utilizar la variable rayo porque esta ya almacena el origen y la dirección.
El resultado tiene que ser igual que en el ejemplo anterior.

Recuerda que hay que guardar el script desde Monodevelop para que se ejecuten los cambios.

Al utilizar los atributos de Ray podemos simplificar el script, en realidad este script tiene que realizar la misma función que en el ejemplo anterior, con la diferencia de que aprovechamos, gracias a la documentación de Unity, la funcionalidad de la clase Ray.

4. Cómo obtener información con RaycastHit

Hemos visto que el rayo de momento nos devuelve un valor verdadero o falso según si toca un colisionador o no , en Unity podemos utilizar una estructura llamada **RaycastHit** que nos proporcionará información variada. Como siempre te invito a que mires la documentación de Unity accediendo a la siguiente dirección.

https://docs.unity3d.com/2018.1/Documentation/ScriptReference/RaycastHit.html

Como puedes ver esta estructura tiene muchos atributos, por comentar algunos diría que nos puede dar información de un collider, la distancia que hay entre el origen del rayo y el collider al que impacta o un punto en el espacio global.

Seguimos con el mismo Script, ahora vamos a realizar algunas modificaciones para capturar la información de la distancia a la que se encuentra un objeto de nuestro FPSController.

Script: ComportamientoRay.cs

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

public class ComportamientoRay : MonoBehaviour
{
   private Ray rayo;
   RaycastHit infoColision;

   void Update ()
   {
        this.rayo.origin = transform.position;
      this.rayo.direction = transform.TransformDirection(Vector3.forward);

      Debug.DrawRay(rayo.origin,rayo.direction*5,Color.blue);

       if (Physics.Raycast (rayo, out infoColision, 5.0f))
      {
       Debug.Log (“Objeto tocado: “+ this.infoColision.collider.tag
       +”\r\n  Objeto distancia” + this.infoColision.distance.ToString());
      }
   }
}
```

Primero creamos una variable de tipo RaycastHit que llamaremos infoColision, dentro de la función update en la condición if dentro del Physics.Raycast, a parte de su origen y su dirección contenidos en la variable rayo debemos poner otro argumento que es out infoColision y la distancia.

El out es una asignación a una variable existente que proporciona un valor por defecto, es la forma que en que ha resuelto Unity el que la función que en principio solo devuelve verdadero o falso, pueda proporcionar más información.

En el mensaje por consola DebugLog he utilizado la variable de RaycastHit para acceder a los atributos collider.tag para que me muestre el nombre de la etiqueta de los objetos y el atributo distance que al ser un valor float he utilizado el método ToString() para convertirlo en cadena de texto.
Recuerda que hay que guardar el script desde Monodevelop para que se ejecuten los cambios.

Con este script conseguimos detectar el nombre de la etiqueta del objeto y la distancia a la que se encuentra nuestro **FPSController**. El resultado que te muestra por consola es el siguiente.

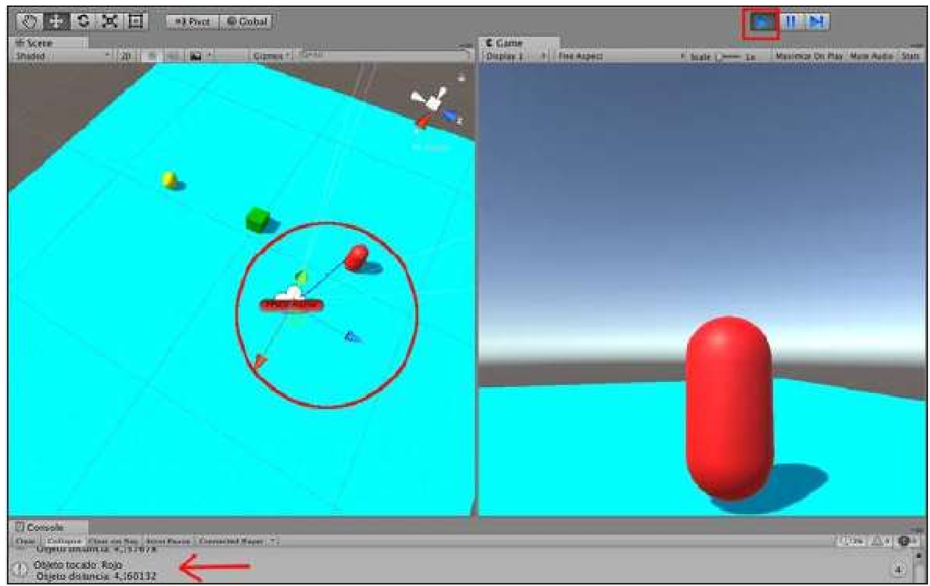


Fig. 8.16

Supongo que si has ejecutado la escena te habrás dado cuenta de que no podemos detectar la esfera amarilla. Esta hecho a propósito porque de momento hemos creado un Raycast que va en linea recta a una distancia de 5 unidades, en el próximo apartado veremos como podemos solucionar este pequeño problema.

Raycast y la posición del ratón ScreenPointToRay

Hasta ahora el Raycast solo mira hacia a delate en el caso de que hubiera un objeto en el suelo este no lo detectaría porque el rayo mira hacia adelante. Nuestro FPSController tiene una cámara que se mueve con el movimiento del ratón, en este componente cámara existe un método llamado **ScreenPointToRay** que nos resuelve este problema. Como siempre mira la documentación desde el enlace que te facilito.

https://docs.unity3d.com/2018.1/Documentation/ScriptReference/Camera.ScreenPointToRay.html

Devuelve un rayo que va desde la cámara a través de un punto de pantalla. El rayo resultante está en el espacio del mundo, comenzando en el plano cercano de la cámara y pasando por las coordenadas de posición (x, y) a pesar que nos pida un Vector3 en la pantalla.

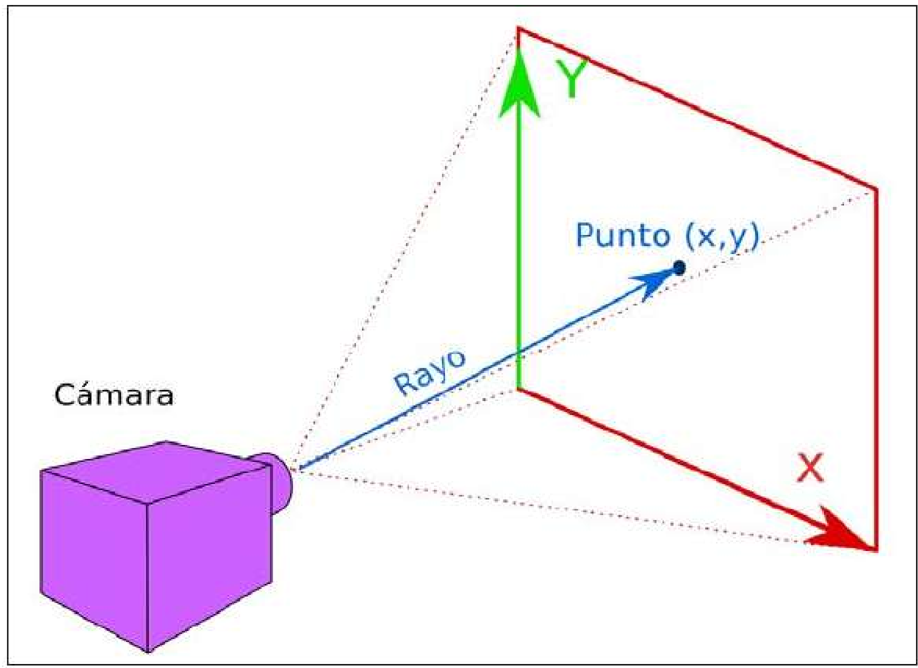


Fig. 8.17

Bien esto nos soluciona parte del problema porque en este caso proyectaremos el rayo desde la cámara pero queremos que el control del rayo sea con el cursor del ratón. Para eso utilizaremos un **Imput.mousePosition** que nos devuelve un Vector3 de la posición del ratón en la pantalla y esto nos va a venir muy bien para poder proyectar el rayo hacia la dirección que deseemos.

Dispones de la información en la documentación de Unity te facilito el siguiente enlace con el objetivo de consulta.

https://docs.unity3d.com/2018.1/Documentation/ScriptReference/Input-mousePosition.html

Primero que quede claro que vamos a utilizar la cámara de nuestro FpsController que tiene una etiqueta tag llamada mainCamera, como te muestro en la siguiente imagen.

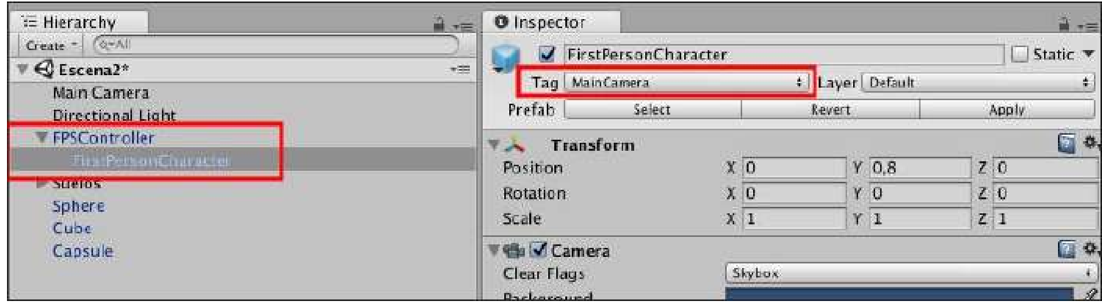


Fig. 8.18

Por ese mismo motivo y para que no haya ninguna duda vamos a eliminar la cámara general, es decir la que tiene la escena por defecto. Selecciona la cámara con nombre **Main Camera** desde la ventana **Jerarquía (Hierarchy)** y pulsa la tecla Suprimir del teclado para eliminar el objeto.

Fig. 8.19

Si volvemos a nuestro script llamado **ComportamientoRay** que puedes encontrar en la carpeta Scripts dentro de la ventana **Project**. Para acceder si todavía no está abierto con el editor haz doble clic encima del icono Script. Ahora vamos a rescribir el script para tomar un punto desde la cámara de nuestro **FPSController** con el que apuntara nuestro **Raycast** que sera guiado por el puntero del ratón.

Script: ComportamientoRay.cs

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

public class ComportamientoRay : MonoBehaviour
{
    private Ray rayo;
    RaycastHit infoColision;
    private Camera miCamara;


    void Awake()
    {
       this.miCamara = GameObject.FindGameObjectWithTag ("MainCamera").Get-
Component<Camera> ();
    }


     void Update ()
    {
       this.rayo.origin= transform.position;
                this.rayo.direction = transform.TransformDirection(Vector3.
forward)
       Debug.DrawRay(rayo.origin,rayo.direction*5,Color.blue);
                this.rayo = this.miCamara.ScreenPointToRay (Input.
mousePosition);
```

```
        if (Physics.Raycast (rayo, out infoColision, 5.0f))
        {
          Debug.Log ("Objeto tocado: "+ this.infoColision.collider.tag
          +" \r\n Objeto distancia: " + this.infoColision.distance.ToString());
        }
    }
}
```

Primero creamos una variable de tipo Camera para guardar la información de nuestra cámara. Después crearemos una función Awake para que cargue la información de la variable miCamara antes de que ejecutemos el juego.

Dentro de Awake en la variable miCamara he buscado el objeto cámara del FpsCotroller mediante la etiqueta y luego accedo a sus componentes, todo en una misma linea (Es otra forma que hasta ahora no hemos visto y que puedes utilizar).

En la función Update primero vamos a eliminar las lineas this.rayo.origin y la linea de this.rayo.direction y vamos a crear un rayo directamente para ello accedemos a la cámara y utilizamos el método ScreenPointToRay y entre paréntesis nos pide la posición y es aquí donde utilizaremos el input.mousePosition para que detecte la posición del puntero de nuestro ratón.
Todo lo demás queda igual.

Recuerda que hay que guardar el script desde Monodevelop para que se ejecuten los cambios.

Si volvemos a Unity y ejecutamos la escena podemos hacer la comprobación acercándonos al objeto esfera para que nos muestre por consola el nombre de su etiqueta y la distancia.

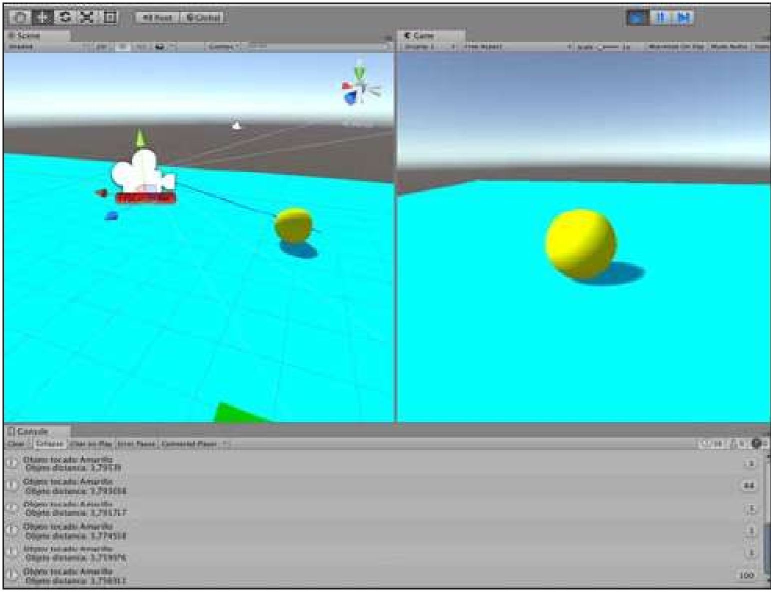


Fig. 8.20

5. Comunicación con SendMessage

Ahora que ya podemos apuntar con el rayo hacia todos los lados podemos utilizarlo para resaltar el material de algunos objetos de escena enviando un mensaje desde el player al objeto en concreto. Es una forma de llamar la atención del jugador hacia un lugar del escenario.

SendMessage llama a al método denominado **methodName** en cada **MonoBehaviour** en este objeto de juego. El método de recepción puede elegir ignorar el argumento teniendo cero parámetros. Si las opciones se configuran en **SendMessageOptions**.**RequireReceiver**, se imprime un error si el componente no selecciona el mensaje. En otras palabras lo que hace este método es enviar un mensaje a otro objeto. Para ver de que se compone vamos a la documentación de Unity.

https://docs.unity3d.com/2018.1/Documentation/ScriptReference/GameObject.SendMessage.html

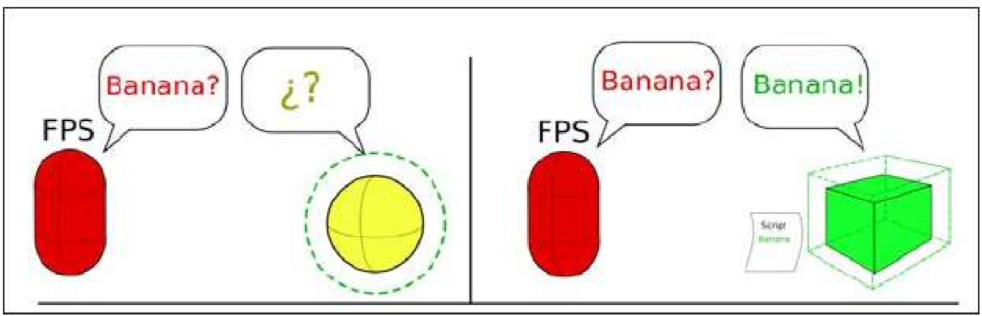


Fig. 8.21

Este método nos puede plantear algunos problemas porque si existen dos métodos que se llamen igual en dos scripts diferentes del mismo **GameObject** llamaría a los dos y este método llama a todos los objetos.

Para el siguiente ejemplo vamos a la carpeta escenas que encontramos dentro de la ventana **Project** y hacemos doble clic encima de la escena con nombre **Escena3**. La nueva escena que se carga no es muy diferente de la escena anterior, tenemos Nuestro **FPSController** y un cubo con el nombre Caja, como te muestro en la siguiente imagen.

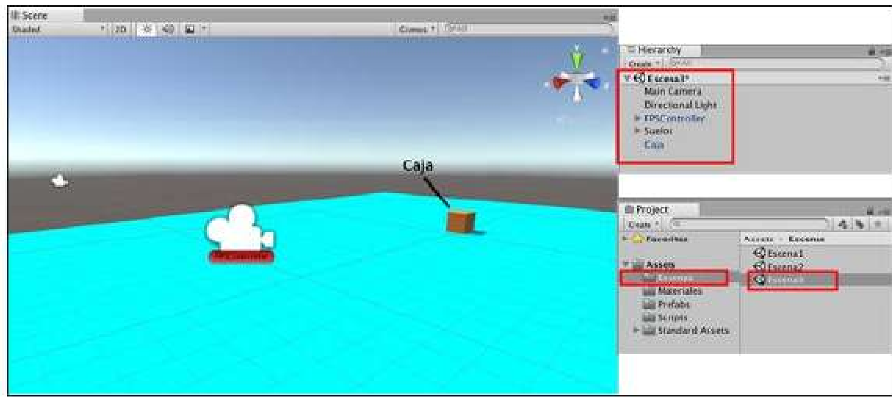


Fig. 8.22

En esta escena nueva debes saber que nuestro **FPSController** no tiene agregado el script **ComportamientoRay.cs** por el echo de que esta escena fue creada para facilitar las actividades y que no tuvieras que crear las escena una y otra vez. Para solucionar este pequeño inconveniente debes acceder a la carpeta **scripts** de la ventana **Project** y arrastrar el script con el nombre **ComportamientoRay.cs** al **FPSController** de este modo volverá a tener el **Raycast** que hemos creado anteriormente.

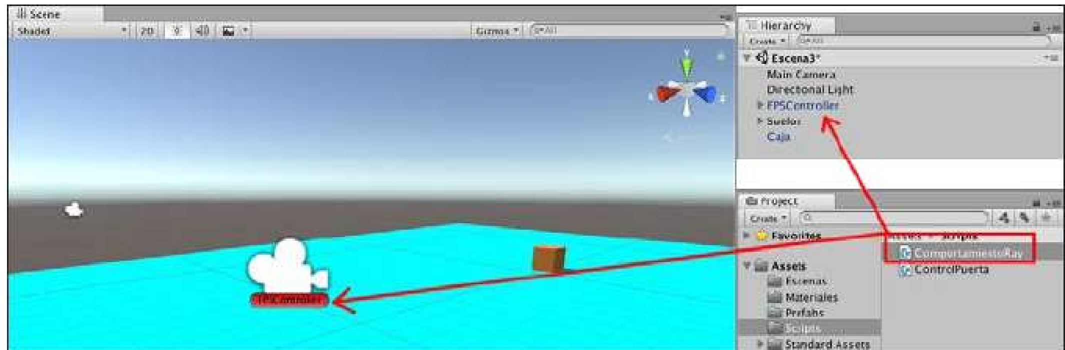


Fig. 8.23

Primero crearemos un script nuevo dentro de la carpeta Scripts con nombre **Caja-Mensaje.cs** y lo arrastramos al objeto llamado Caja. Esto nos permite crear un método dentro del script de la Caja, al que podemos llamar desde nuestro **FPSController**.

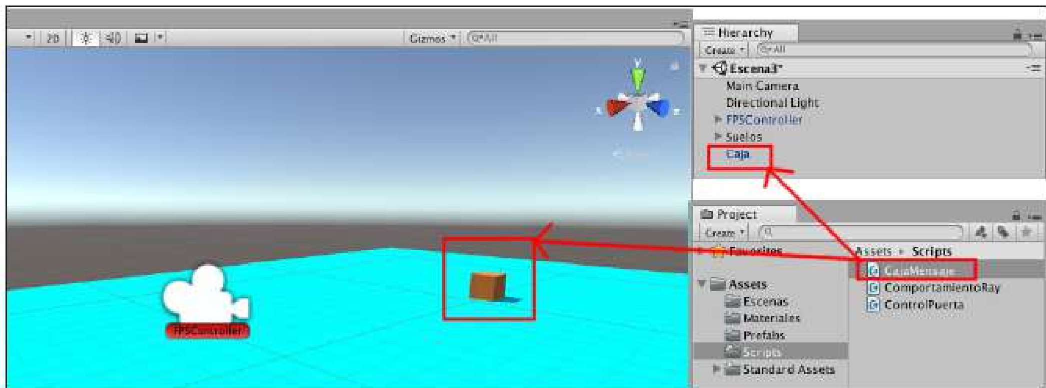


Fig. 8.24

Lo que queremos hacer en el script es que el cubo cambie de color, pero si cambiamos el color del material se cambiará el color de todas las cajas que hubiera en la escena en el caso de que fuera un Prefab duplicado (en este caso solo tenemos una caja), por ese mismo motivo lo que vamos a realizar es un cambio de material, el que tiene por defecto la caja, por otro material con otro color. En el ejemplo que vamos a realizar vamos a utilizar el material con nombre **Caja1** por defecto y el material **Caja2** como material de cambio.

Abrimos el script haciendo doble clic encima para editarlo en **Monodevelop.** Este script vamos a comentarlo por partes. Primero vamos a crear las variables que van a contener los materiales.

Script: CajaMensaje.cs

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

public class CajaMensaje : MonoBehaviour
{
    public Material defCaja;
    public Material colorCaja;

    void Start()
    {
    }

     void Update ()
    {
    }
}
```

Estas variables las haremos publicas para poder arrastrar los materiales dentro del componente script. Estas variables son de tipo Material porque tienen que contener materiales que utilizaremos, en el script. Recuerda que hay que guardar el script desde Monodevelop para que se ejecuten los cambios en Unity.

Ahora al volver a Unity si seleccionamos la caja y miramos en la ventana Inspector sus componentes veremos que se nos muestra en el componente Script las variables **DefCaja** y **ColorCaja**. Ahora debemos agregar a cada una de estas variables los materiales correspondientes, es decir para ***DefCaja = Material Cajas*** y para ***ColorCaja = Material Caja2***, como te muestro en la siguiente imagen.

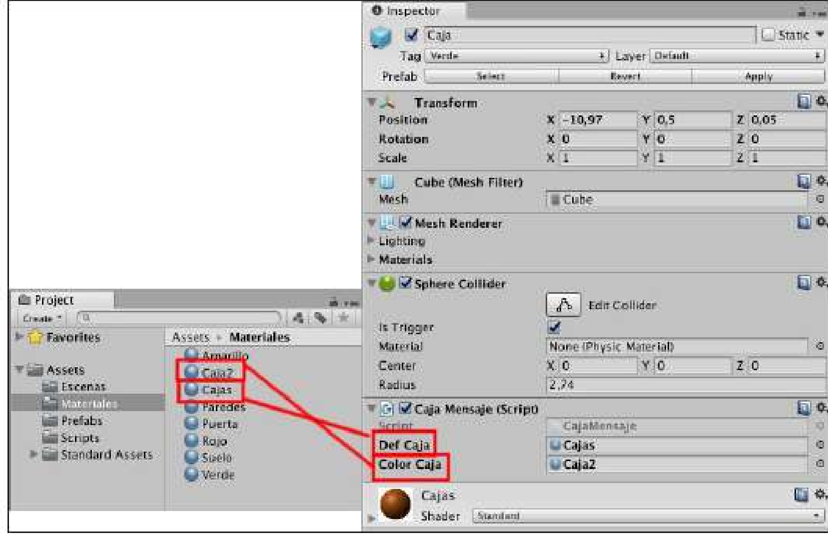


Fig. 8.25

Ahora que estamos viendo los componentes del objeto Caja veras que tiene un componente llamado **Mesh Renderer**, dentro de este componente tenemos una sección llamada **Materials** en donde podemos ver el material que está utilizando el objeto. Este es el componente que vamos a acceder en nuestro script para cambiar el material del objeto. Volvemos al script **CajaMensaje** y seguimos de la siguiente manera.

Script: CajaMensaje.cs

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

public class CajaMensaje : MonoBehaviour
{
   public Material defCaja;
   public Material colorCaja;
   private Renderer materiales;

   void Start()
   {
      this.materiales = gameObject.GetComponent<Renderer>();
      this.materiales.material= this.colorCaja;
   }

    void Update ()
   {
   }
}
```

Creamos una nueva variable esta vez de tipo Renderer para poder acceder al componente del objeto, le he llamado materiales.
En la función Start() (Puedes utilizar la función Awake() si lo prefieres), primero utilizamos la variable materiales para acceder al componente Renderer. Una vez que hemos accedido mediante GetComponent, vamos a realizar una prueba que consiste en acceder desde la variable this.materiales, si ponemos un punto después accederemos a sus métodos y atributos. Veremos que nos aparece la opción material que es el nombre que hemos visto en los componentes de MeshRenderer de la Caja, en la ventana Inspector. Ahora le decimos que el material que queremos sea el que contiene la variable colorCaja que es en realidad el material Caja2.
Recuerda que hay que guardar el script desde Monodevelop para que se ejecuten los cambios en Unity.

Si volvemos a Unity para comprobar si la caja cambia de color, al ejecutar la escena veremos que efectivamente la caja toma el material con nombre Caja2.

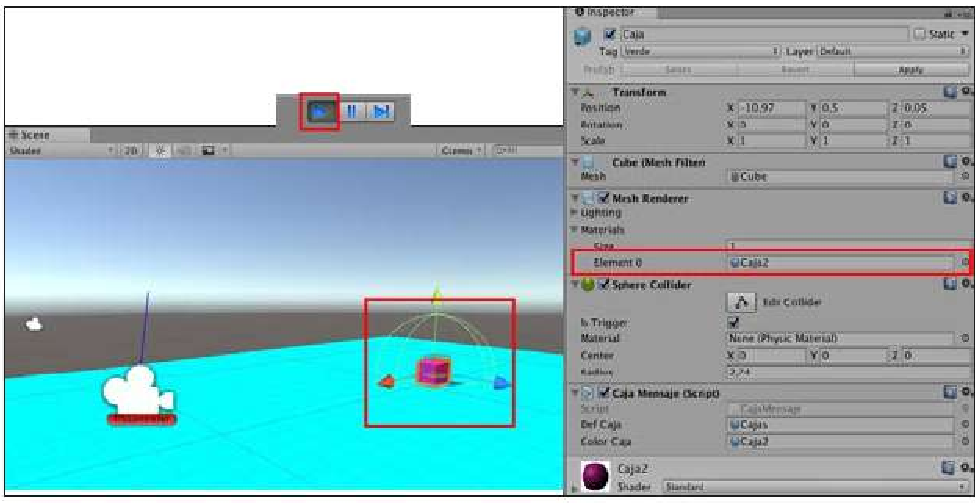


Fig. 8.26

Volvemos al script **CajaMensaje** y ahora crearemos un método para que podamos cambiar de material cuando nuestro **FPSController** lo llame. Te pongo un tachón en el script para que borres esta linea que no nos va a hacer falta.

Script: CajaMensaje.cs

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

public class CajaMensaje : MonoBehaviour
{
   public Material defCaja;
   public Material colorCaja;
   private Renderer materiales;

   void Start()
   {
      this.materiales = gameObject.GetComponent<Renderer>();
      this.materiales.material= this.colorCaja;
   }
   void Resaltar(bool tocado)
   {
      if(tocado)
      {
       this.materiales.material = this.colorCaja;
      }else{
       this.materiales.material = this.defCaja;
      }
}
```

Vamos a crear una función llamada Resaltar en donde le añadimos un atributo tocado de tipo booleano y pondremos una condición si tocado es verdadero el material del objeto tomará el material de la variable colorCaja y si es falso el objeto del material tomara el de la variable defCaja.
Recuerda que hay que guardar el script desde Monodevelop para que se ejecuten los cambios en Unity.

Ahora tenemos preparado el método dentro de la caja y ahora necesitamos que nuestro **FPSController** envíe un mensaje llamando a este método para que se produzca el cambio de material. Para ello accede a la carpeta scripts y haz doble clic encima del script **ComportamientoRay.cs,** si es que no lo tienes abierto, para editarlo en **Monodevelop**. El script debe quedarnos de la siguiente, manera para enviar un mensaje.

Script: ComportamientoRay.cs

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

public class ComportamientoRay : MonoBehaviour
{
    private Ray rayo;
    RaycastHit infoColision;
    private Camera miCamara;

    void Awake()
    {
       this.miCamara = GameObject.FindGameObjectWithTag (“MainCamera”).Get-
Component<Camera> ();
    }

     void Update ()
    {
      Debug.DrawRay(rayo.origin,rayo.direction*5,Color.blue);
               this.rayo = this.miCamara.ScreenPointToRay (Input.
mousePosition);
      if (Physics.Raycast (rayo, out infoColision, 5.0f))
      {
        Debug.Log (“Objeto tocado: “+ this.infoColision.collider.tag
        +” \r\n Objeto distancia: “ + this.infoColision.distance.ToString());
        this.infoColision.collider.SendMessage(“Resaltar”,true,
        SendMessageOptions.DontRequireReceiver);
        }
    }
}
```

El cambio que debemos hacerle a este script es añadirle dentro de la función Update() dentro de la condición del Physics.Raycast, que cuando el rayo impacte con un collider envíe un mensaje y entre paréntesis debemos indicarle el método que buscamos, en este caso la Caja tiene un método llamado Resaltar, también podemos darle el argumento que necesita el método, como queremos que cambie el material y el método dispone de un argumento booleano true o false le daremos un valor true.

El último argumento que necesitamos para enviar el mensaje es darle una opción al envío, en este caso en concreto he utilizado DontRequireReceiver, esto significa que cuando el rayo impacta con un collider cualquiera envía este mensaje, pero en el caso de que el objeto que tiene el collider no tenga el método Resaltar ignorará el mensaje.

Recuerda que hay que guardar el script desde Monodevelop para que se ejecuten los cambios.

Al volver a Unity y ejecutar la escena, en un primer momento la caja no va ha cambiar de material, en el momento en que nuestro **FPSController** con su **Raycast** impacte el con el collider del Objeto caja este cambiará de material.

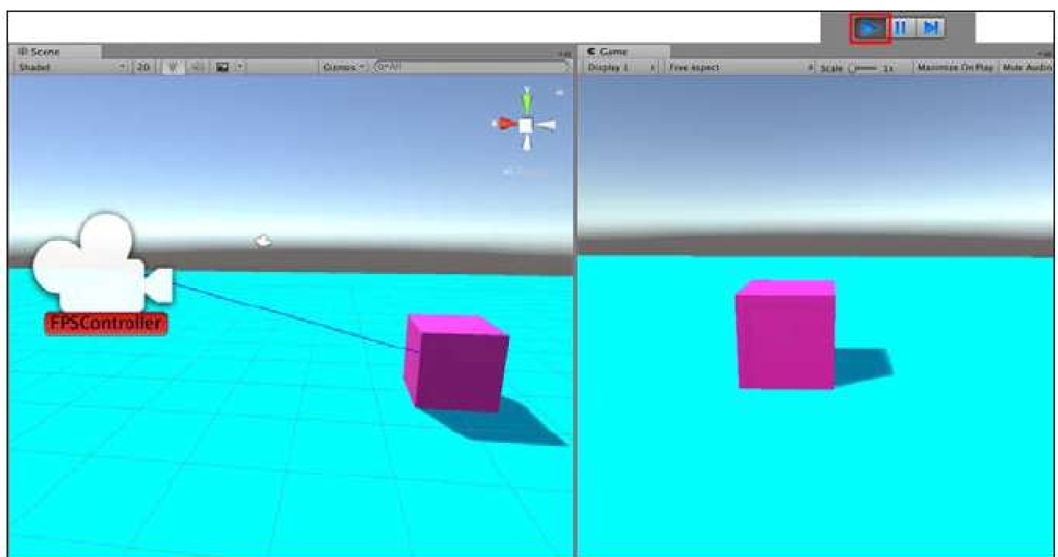


Fig. 8.27

Ahora tenemos el problema de que la caja se queda con el material para resaltar permanente y nos gustaría que cuando dejáramos de golpear el Objeto Caja este volviera a su material por defecto. Una solución muy sencilla para no complicar el tema es dentro del script **CajaMensaje** vamos a poner dentro de la función Update que la función Resaltar sea falsa, de este modo siempre es falsa hasta que el **FPSController envía** un mensaje.

Script: CajaMensaje.cs

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

public class CajaMensaje : MonoBehaviour
{
    public Material defCaja;
    public Material colorCaja;
    private Renderer materiales;

    void Start()
    {
        this.materiales = gameObject.GetComponent<Renderer>();
    }

    void Update()
    {
        this.Resaltar(false);
    }

    void Resaltar(bool tocado)
    {
        if(tocado)
        {
          this.materiales.material = this.colorCaja;
        }else{
          this.materiales.material = this.defCaja;
        }
}
```

Dentro de la función Update llamamos al método Resaltar con el argumento false para que el material siempre sea this.defCaja.
Recuerda que hay que guardar el script desde Monodevelop para que se ejecuten los cambios.

6. Decals

Cuando queramos tener marcas en objetos, por ejemplo de agujeros de balas en un escenario. En realidad es una impresión de una textura en un objeto. Estas impresiones se realizan en los Collider de los objetos y uno de los problemas que podemos tener es la colocación de estas texturas. Los decals que vamos a mostrar en los próximos ejemplos va a ser el de crear agujeros de balas.

Para trabajar cómodamente deberás crear un nuevo proyecto 3D e importar el segundo paquete de assets con nombre **Assets_Decals_Capitulo_8.unitypackage** que

disponemos en este capitulo. Para importar accedemos al menú principal **Assets > Import Package > CustomPackage...** Se abrirá una ventana de tu sistema en donde debes buscar donde está guardado el material adicional de la obra. Una vez lo encuentres en la carpeta Proyecto_8 selecciona el paquete con el nombre **Assets_Decals_Capitulo_8.unitypackage s**e nos aparece una nueva ventana en donde podemos ver el contenido del paquete y Unity nos da la posibilidad de seleccionar que queremos importar. Para seguir el capitulo deja todas las carpetas seleccionadas y para que ejecute la importación hacemos clic encima del botón I**mport** que se encuentra en la parte inferior derecha.

Si todo se ha realizado correctamente deberás ver en la ventana **Project** las siguientes carpetas. A continuación te muestro como debería de quedar.

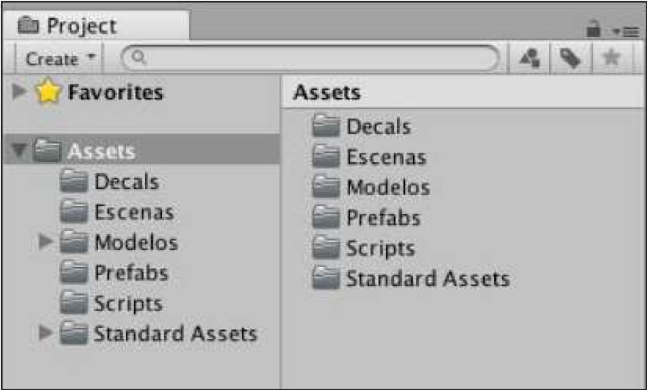


Fig. 8.28

A continuación dispones de una carpeta Escenas, accede a ella y realiza doble clic encima de la escena con nombre Escena 1.

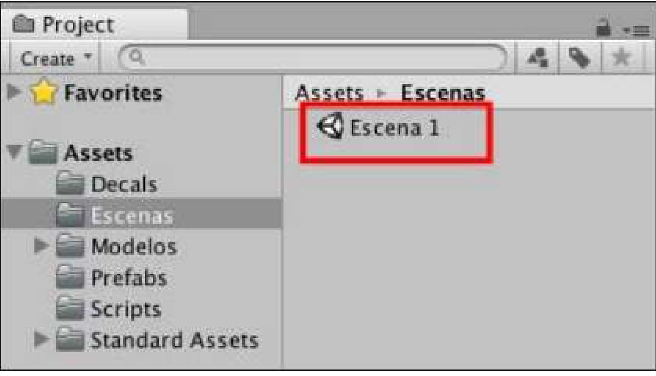


Fig. 8.29

En el escenario que hemos cargado tenemos todo lo necesario para empezar a trabajar en el tema de los decals. Tenemos barriles, cajas y una puerta móvil que ya hemos visto como hacer que se abra y se cierre al principio del capítulo. En esta escena está todo preparado para que abordemos el tema de los decals. Si quieres puedes ejecutar la escena y mover el **FPSController** por la escena. De momento no podemos hacer gran cosa pero si puedes ver como se abre y se cierra la puerta.

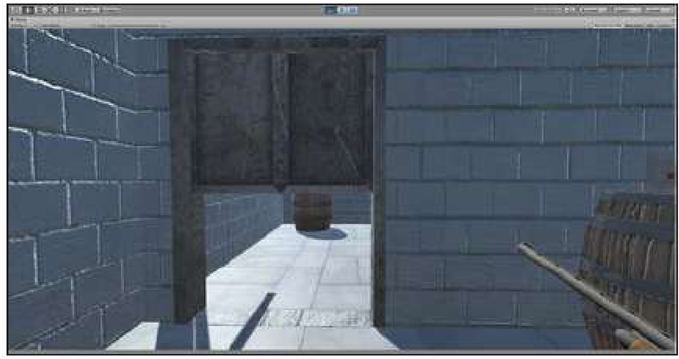


Fig. 8.30

Preparar nuestro Prefab Decal

Los decals añaden una marca a nuestra escena en objetos determinados, si te fijas en la ventana **Project** tenemos una carpeta en donde disponemos de varios decals. En esta carpeta encontramos tres imágenes con un tamaño en potencias de dos y en formato **png.** Estas imágenes las vamos a utilizar para crear la ilusión de disparar en la escena y que se marque el agujero en el objeto que disparemos.

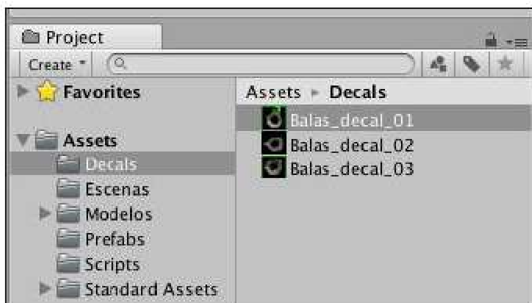


Fig. 8.31

Primero vamos a crear un nuevo Prefab, para ello accedemos a la ventana **Project** seleccionamos la carpeta **Prefabs** y accedemos al menú **Create > Prefab** y nos creara un Prefab nuevo en donde le pondremos el nombre de **Decal_Base** como te muestro en la siguiente imagen.

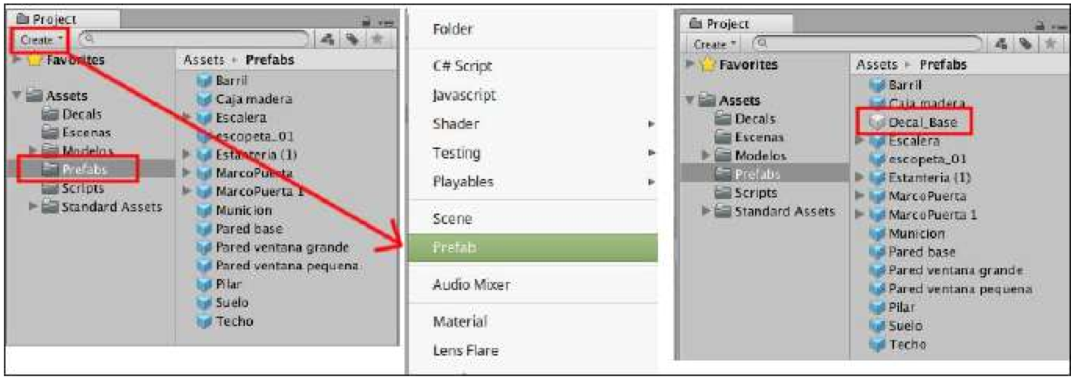


Fig. 8.32

Ahora tenemos un Prefab vacío y vamos a rellenarlo con un objeto Quad. Para ello Creamos un plano en la escena accediendo a la ventana **Hierarchy** (Jerarquía) y accedemos al botón **Create > 3D Object> Quad.** Una vez creado el **Quad** escalamos su componente **Scale** a 0,5 en todos sus ejes y rotamos en 90 grados en el componente **Rotation** en el eje X, todo desde la ventana **Inspector** para que se nos quede un plano pequeño.

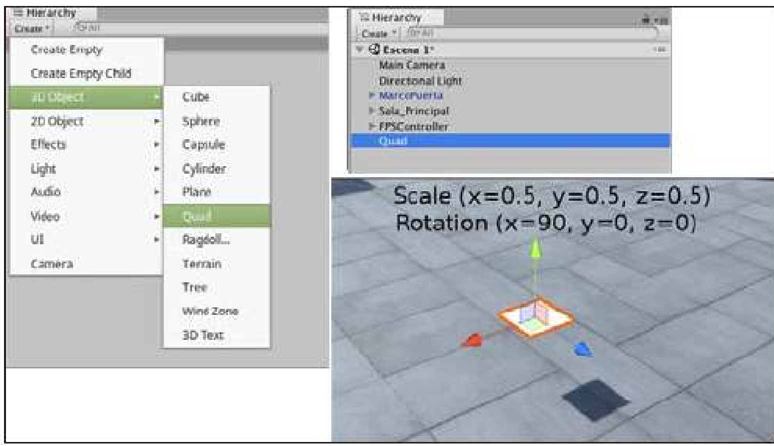


Fig. 8.33

El siguiente paso es ponerle un material, para ello dentro de la ventana **Project** accedemos a la carpeta **Modelos > Materiales** y con esta carpeta seleccionada pulsamos en el botón **Create** en la parte superior de la ventana **Project** y seleccionamos en el menú que se despliega la opción Material. Se creara un nuevo material dentro de esta carpeta Materiales al que le daremos el nombre de **Decals**. El nombre de este material normalmente suele ser **Decal_01, Decal_02**, porque se suelen utilizar varios materiales para los decals, pero en este ejemplo solo vamos a utilizar uno.

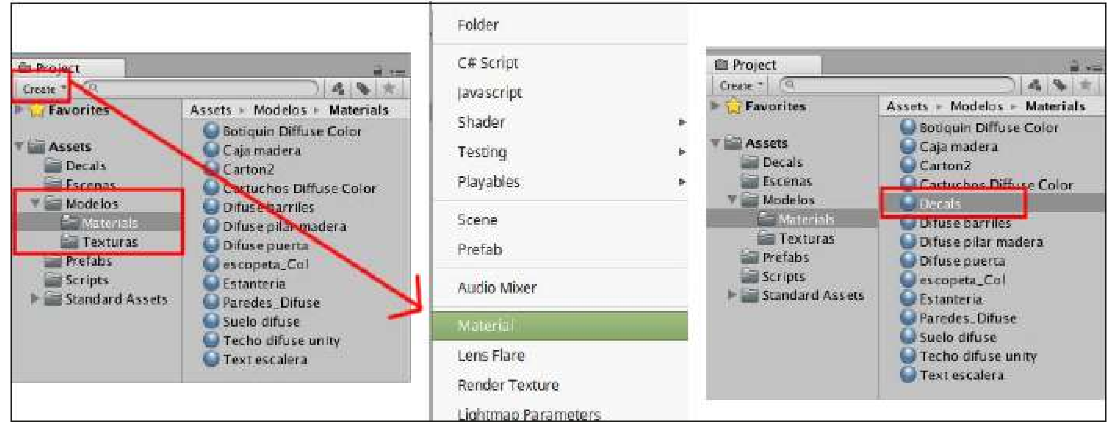


Fig. 8.34

Con el material seleccionado podemos ver las propiedades del material en la ventana Inspector. En esta ventana debemos ponerle la textura del **decal** que queramos. Para añadir la textura hacemos clic encima de la marca con forma de circulo al lado del parámetro **Albedo** y se nos aparecerá una ventana en donde se nos muestra todas las texturas. En esta ventana debemos seleccionar una de las tres texturas con nombre **Ba-**

las_decal seguido de un numero, en este caso yo he seleccionado el primero con nombre **Balas_decal_01**.

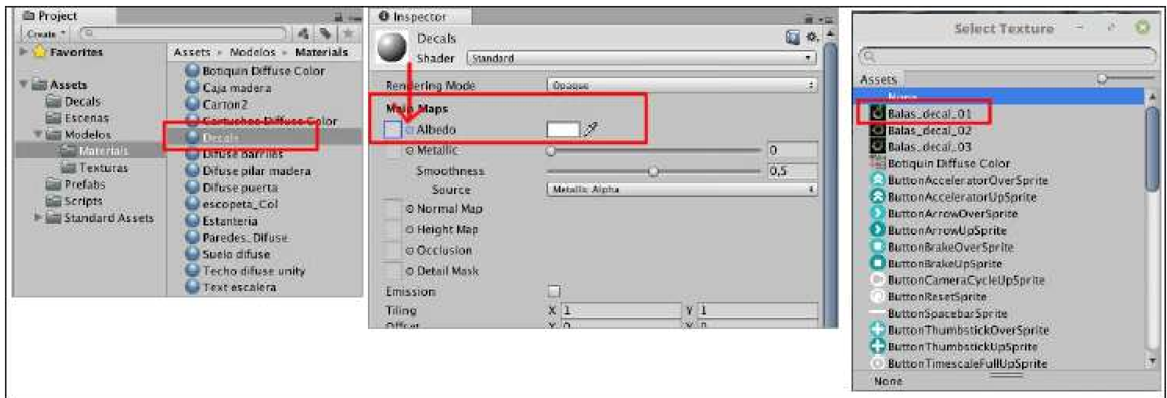


Fig. 8.35

Una vez has aplicado la textura en la ventana inspector en la parte inferior puedes ver una esfera en donde se te muestra la textura aplicada. Primero vamos a arrastrar el material al Quad que hemos creado, para ver que tal queda.

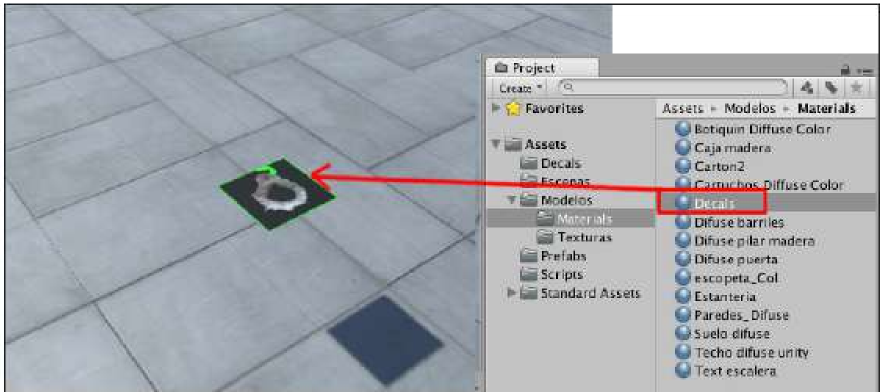


Fig. 8.36

Si te fijas bien veras que la textura no se transparenta y se ve un color verde y negro de fondo. Este problema es debido al tipo de **Shader** que utiliza el material. En este caso vamos a cambiar el tipo de **Shader Standar** por uno de tipo **Sprites** que nos proporcionará mejores resultados. Para cambiar el shader del material, selecciona primero el material **Decals** desde la ventana **Project > Modelos> Materiales** y luego desde la ventana Inspector accedemos al menú que encontramos al lado de **Shaders** y seleccionamos **Sprites >Default**.

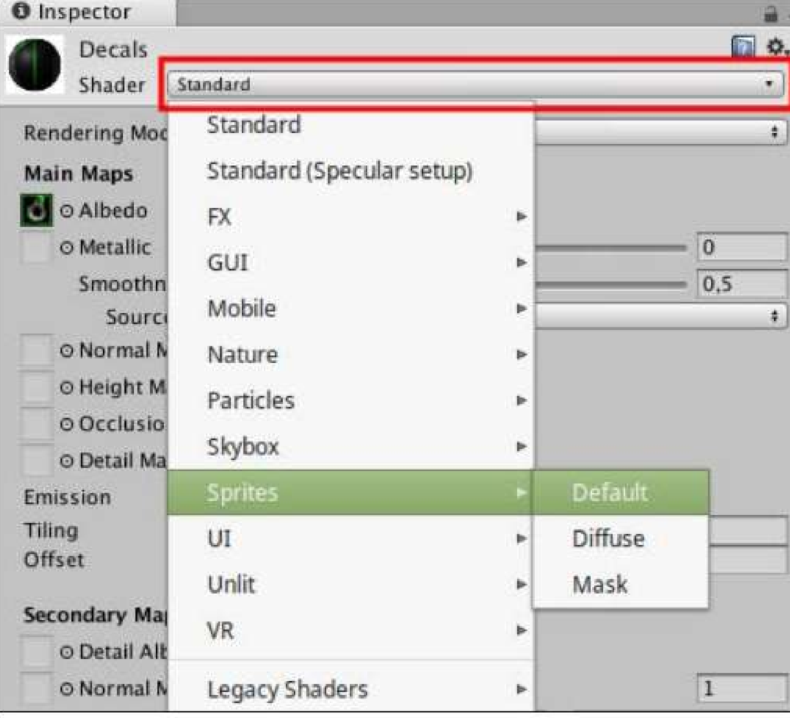


Fig. 8.37

El resultado final es una imagen con transparencia en donde podemos ver tanto en la imagen inferior de la ventana **Inspector**, como en la ventana **Scene** disponemos de un plano con un agujero que podemos mimetizar perfectamente en el suelo o paredes.

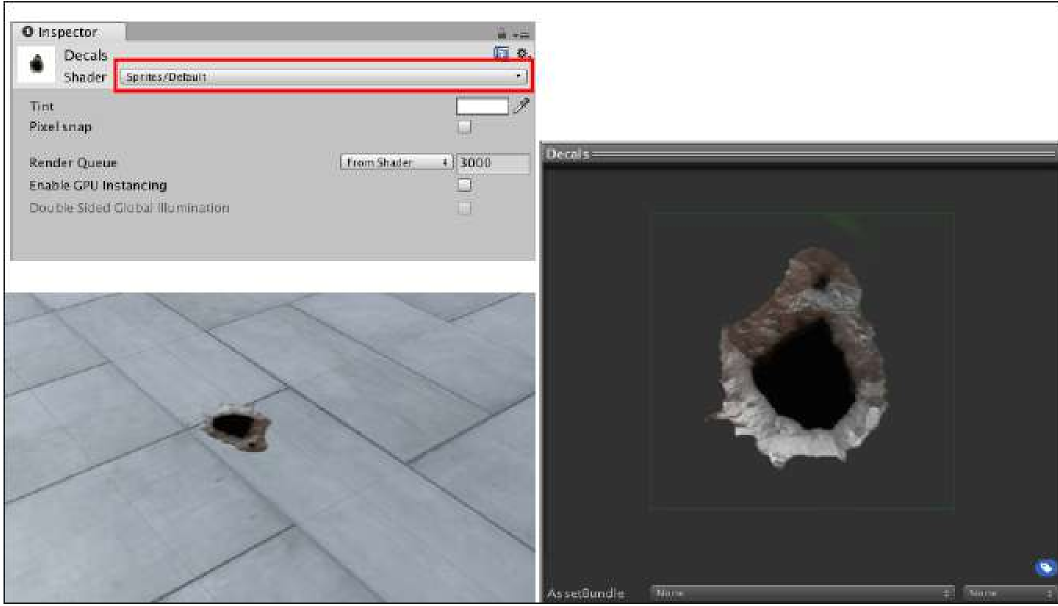


Fig. 8.38

El tema de los **Shaders** de los materiales es un tema muy extenso con el que se podría escribir un libro entero, no quiero entrar en profundidad pero si tienes curiosidad por

conocer más sobre este tema, te invito a que accedas al enlace que te facilito y mires en la documentación de Unity.

https://docs.unity3d.com/2018.1/Documentation/Manual/Shaders.html

Ahora que tenemos nuestro objeto **Quad** con el material vamos a crear el Prefab, pero antes nos vamos a asegurar de que el **Quad** esta situado en la posición 0,0,0 en la escena. Si no es el caso seleccionamos el objeto **Quad** y en la ventana Inspector en el componente **Transform** introducimos el valor 0 en todos los ejes del parámetro **Position**.

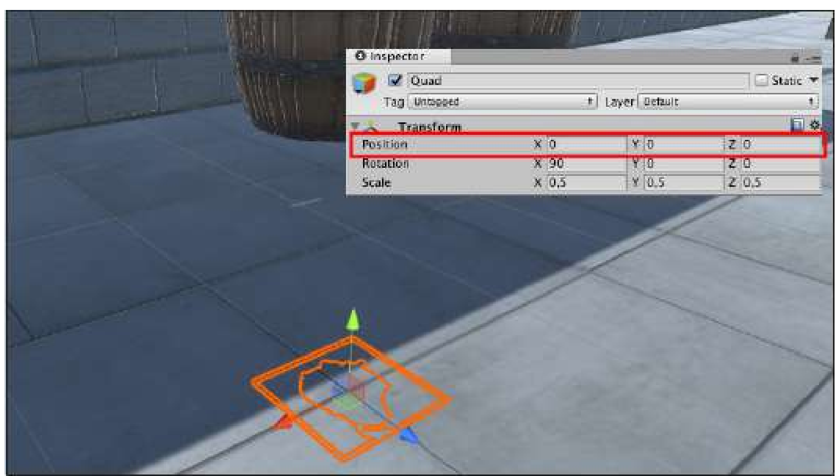


Fig. 8.39

Ahora accedemos desde la ventana **Project** a la carpeta Prefabs y seleccionando el objeto **Quad** desde la ventana **Hierarchy** lo arrastramos encima del Prefab **Decal_Base** que hemos creado anteriormente. Si todo es correcto el icono del Prefab **Decal_Base** pasara de ser de un color gris a un color azul, esto significa que nuestro Prefab ya no está vacío.

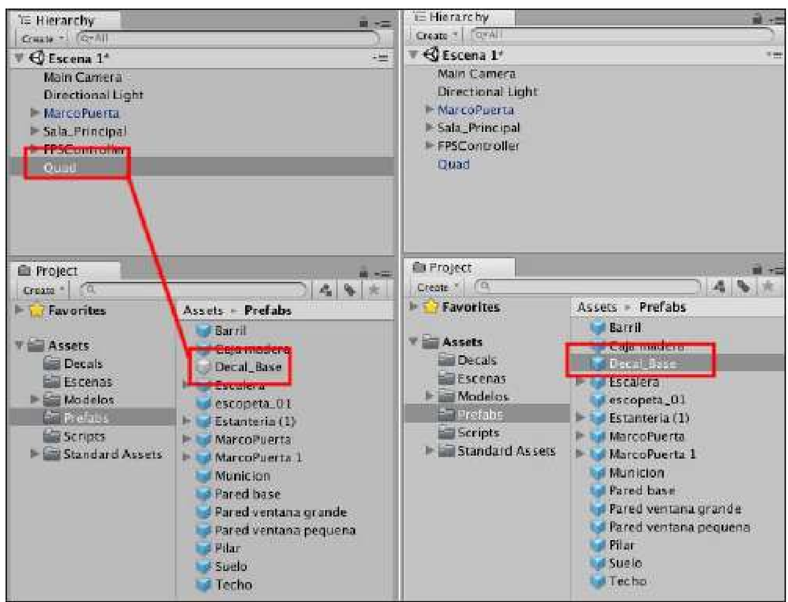


Fig. 8.40

Ahora ya puedes eliminar el objeto **Quad** de la escena, porque ya disponemos de un Prefabs que es el que utilizaremos para los siguientes ejemplos. Te recomiendo que vayas guardando la escena y el proyecto accediendo al menú principal y seleccionando **Save Scene** y luego la misma acción para guardar el proyecto seleccionando **Save Project**.

7. Instanciar los Decals con Raycast

Es el momento de crear un nuevo script que añadiremos a nuestro **FPSController** de la escena para que vaya creando estos decals y se enganchen a los distintos objetos. Para ello accede a la carpeta Scripts de la ventana **Project** y crea un nuevo script que llamaremos **Shooter.cs**, para crear este nuevo script primero asegurate de tener la carpeta Scripts seleccionada y luego accede al botón **Create > C# Script** de la ventana **Project**. Se creara un nuevo script le pones el nombre de **Shooter** y arrastras el script encima del **FPSController.**

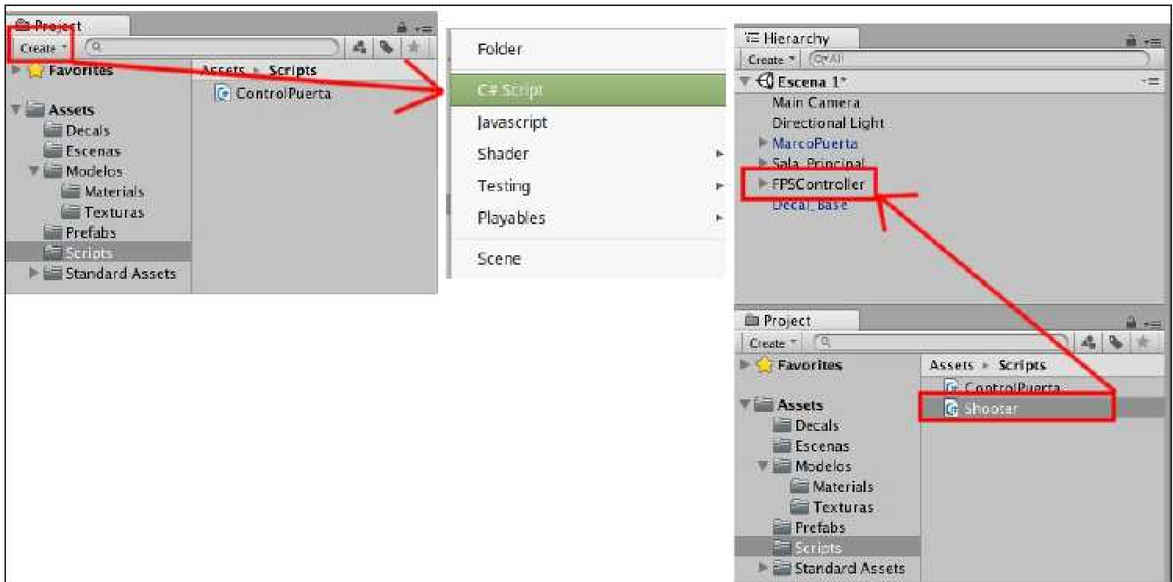


Fig. 8.41

Antes de empezar a editar el script vamos a borrar la cámara de la escena con nombre Main Camera, en este caso lo recomiendo puesto que el **FPSController** ya lleva una incorporada y es posible que con el script que vamos a editar nos de algún problema. En resumen accede a la ventana **Hierarchy** selecciona la **Main Camera** que por defecto debe ser el primer objeto de la lista, y la eliminamos pulsando la tecla **supr** del teclado.

El primer objetivo que tenemos con este script es la de posicionar un **Decal** en un punto marcado por la cámara del **FPSController.** En es siguiente gráfico se ve representado lo que queremos conseguir.

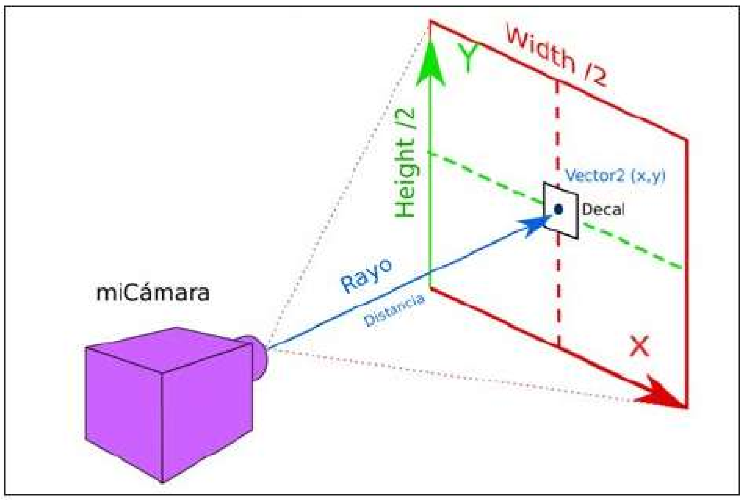


Fig. 8.42

Vamos a utilizar la cámara para que nos posiciones un punto en el centro de la pantalla que en realidad es un Vector2 que contienen el eje Y para la altura y el eje X para la anchura. En este caso para calcular el centro de la pantalla dividiremos el total de la anchura entre 2 y haremos lo mismo para la altura. Si miramos en la documentación de Unity en el enlace que te facilito veras en la clase **Screen (Pantalla)**, que disponemos de dos propiedades Height y Widht para localizar la el centro de nuestro pantalla.

https://docs.unity3d.com/2018.1/Documentation/ScriptReference/Screen.html

Una vez tengamos un punto central crearemos un **Input** para disparar nuestro **Decal** que posicionaremos mediante un Rayo. Recuerda que este Rayo sera de tipo Physics y con la información que nos proporcione instanciaremos nuestro **Decal**. Cuando hablamos de Instanciar nos referimos a clonar un Objeto o Prefab. Te invito a que te mires la documentación de Unity antes de pasar a la edición del script, en el enlace que te facilito.

https://docs.unity3d.com/2018.1/Documentation/ScriptReference/Object.Instantiate.html

Ahora hacemos doble clic encima del script **Shooter** para que se nos abra **Monodevelop** en el caso de que no esté abierto y vamos a editar el Script.

Script: Shooter.cs

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

public class Shooter : MonoBehaviour
{
  private Ray rayo;
  private RaycastHit hit;

  private Camera miCamara;
```

```
  private Vector2 centroCamara;
  public float distanciaDisparo;

  public GameObject decalPrefab;

  void Awake()
  {
  this.miCamara = gameObject.transform.GetChild (0).GetComponent<Camera>();
  this.centroCamara.x= Screen.width/2;
  this.centroCamara.y= Screen.height/2;
  }

  void Update()
  {
  if(Input.GetButtonDown("Fire"))
  {
    this.rayo= this.miCamara.ScreenPointToRay(this.centroCamara);
    if(Physics.Raycast (this.rayo,out this.hit,this.distanciaDisparo))
    {
        GameObject.Instantiate(this.decalPrefab, this.hit.point,this.decal-
Prefab.transform.rotation);
      }
    }
  }
}
```

Primero debemos declarar las variables que necesitamos por el momento. Creamos una variable de tipo Ray que llamamos rayo que lo utilizaremos para apuntar al centro de la pantalla y otra variable de tipo RaycastHit con nombre hit, para que cuando golpee un collider guarde la información. Las siguientes variables son miCamara que es de tipo Camera en donde guardaremos la información del componente Camera del FPS, otra variable de tipo Vector2 llamado centroCamera en donde vamos a guardar las posiciones del centro de la cámara. También vamos a crear dos variables más en este caso públicas, una de tipo float con nombre distancia, para poderle dar una distancia al Rayo y otra variable de tipo GameObject con nombre decalPrefab en donde guardaremos el PrefabDecal que utilizaremos para clonar. He creado una función Awake(), puedes utilizar la funcion Start() si lo prefieres. Primero vamos a guardar dentro de la variable miCamara donde esta dicha cámara, en este caso está en el propio objeto al que le aplicamos el script por ese motivo se utiliza gameObject en minúscula y en este caso accedemos a su tranform y con el método GetChild accedemos al primer hijo que es el valor 0 y es donde tenemos ubicada la cámara. Seguidamente y mediante un punto accedemos al componente Camera con el método GetComponent. Ahora en la línea de debajo vamos a encontrar el centro de la pantalla y utilizamos la variable centroCamara en sus ejes como hemos visto en el gráfico anterior y utilizamos la clase Screen para acceder a su anchura en (x) y dividirla entre 2 para calcular la mitad de esta y para la altura utilizamos el mismo método. Dentro de la función Update utilizamos mediante un if el Input con el método GetButtonDown y el nombre Fire que esta predefinido para disparar con el botón Control o el botón Izquierdo del ratón. Dentro de la condición diremos que el rayo debe apuntar utilizando this.miCamara que es nuestra cámara con el método ScreenPointToRay que nos pide el argumento de una posición de la pantalla, que ya tenemos preparado dentro de la variable this.centroCamara.

A continuación vamos a crear otra condición dentro del disparo que es cuando nuestro Physics.Raycast que tiene un rayo y nos da información cuando este impacta con hit en un collider a cierta distancia que le daremos con la variable this.distanciaDisparo, crearemos un Objeto instanciado y le daremos primero el objeto a clonar, luego una posición en este caso le he dicho que utilice la información del punto en que golpea y por último una rotación a la que queremos que se coloque nuestro objeto; de momento le pondremos la del propio prefab .
Recuerda que hay que guardar el script desde Monodevelop para que se ejecuten los cambios.

Una vez volvemos a Unity primero debemos seleccionar nuestro **FPSController** y en la ventana Inspector arrastrar nuestro Prefab **Decal_Base** dentro de la variable **Decal Prefab**, poner un valor a la Distancia Disparo y muy importante por si no lo has echo todavía eliminar el objeto MainCamera.

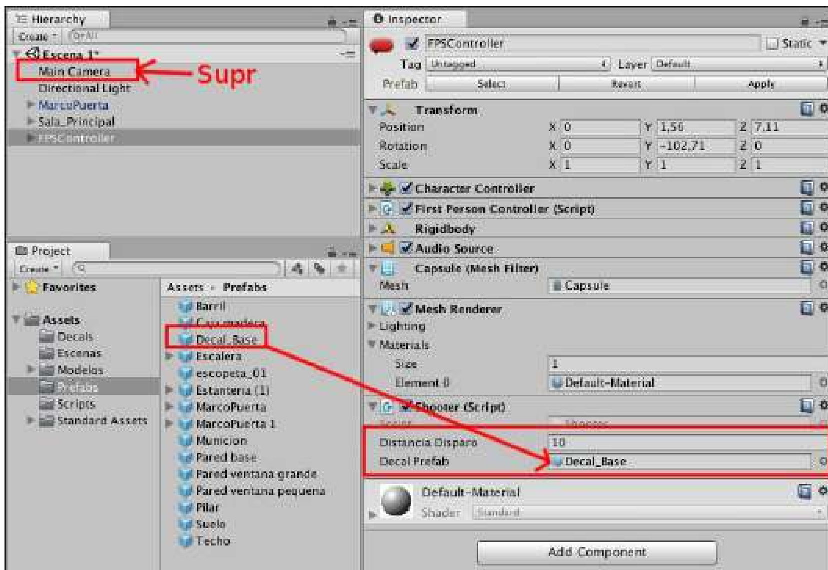


Fig. 8.43

Si ejecutamos la escena y pulsamos o bien la tecla **Ctrl** o el botón izquierdo del ratón veremos que se nos aparece el **Decal**, que se engancha a los objetos que tienen colliders, pero la rotación en es del todo correcta. También si mantienes pulsado el botón de disparar veras que se crean muchos Prefabs seguidos. Este problema lo resolveremos en el próximo apartado.

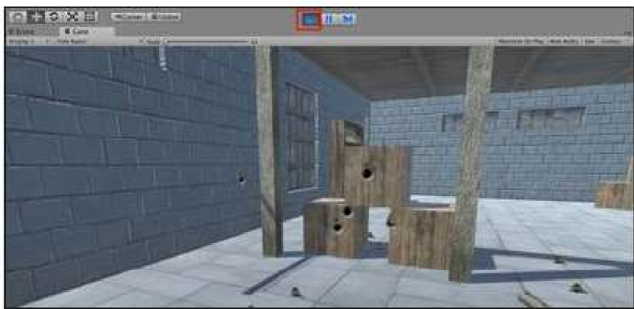


Fig. 8.44

Definir el tiempo del disparo

Primero vamos a solucionar el problema que tenemos con el disparo continuo. Para solucionarlo podemos definir un tiempo entre el primer disparo y el último disparo. Para ello en el script Shooter añadiremos las siguientes lineas de código.

Script: Shooter.cs

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

public class Shooter : MonoBehaviour
{
  private Ray rayo;
  private RaycastHit hit;
  private Camera miCamara;
  private Vector2 centroCamara;
  public float distanciaDisparo;
  public GameObject decalPrefab;

  public float tempoDisparo;
  private float tempoUltimoDisparo;

  void Awake()
  {
  this.miCamara = gameObject.transform.GetChild (0).GetComponent<Camera>();
  this.centroCamara.x= Screen.width/2;
  this.centroCamara.y= Screen.height/2;
  this.tempoUltimoDisparo = Time.time;
  }

  void Update()
  {
  if(Input.GetButtonDown("Fire"))
  {
    if((Time.time-this.tempoUltimoDisparo)>this.tempoDisparo)
    {
      this.rayo= this.miCamara.ScreenPointToRay(this.centroCamara);
      this.tempoUltimoDisparo = Time.time;

      if(Physics.Raycast (this.rayo,out this.hit,this.distanciaDisparo))
      {
```

```
        GameObject.Instantiate(this.decalPrefab, this.hit.point,this.decal-
Prefab.transform.rotation);
       }
     }
   }
 }
}
```

Primero crearemos una variable pública de tipo float con nombre tempoDisparo para darle un valor de tiempo entre disparo y disparo y luego creamos otra variable esta privada de tipo float con nombre tempoUltimoDisparo, en la que almacenaremos el momento en que se disparo por ultima vez.

Dentro de la función Awake() utilizo la variables TempoUltimoDisparo para guardar el tiempo.

Dentro de la función Update cuando se cumple la condición de disparar crearemos otra condición que compare el tiempo menos el tempoUltimoDisparo y que el valor resultante si es mayor que el valor de tempoDisparo se ejecute todo lo que hemos hecho anteriormente. Dentro de esta condición debemos volver a igualar el tempoUltimoDisparo al tiempo para poder renovar la condición.
Recuerda que hay que guardar el script desde Monodevelop para que se ejecuten los cambios.

Cuando volvemos a Unity primero de todo debemos ponerle un valor en la ventana Inspector en el apartado del Script Shooter en la variable Tempo Disparo, en el caso del ejemplo he puesto el valor de 0,5.

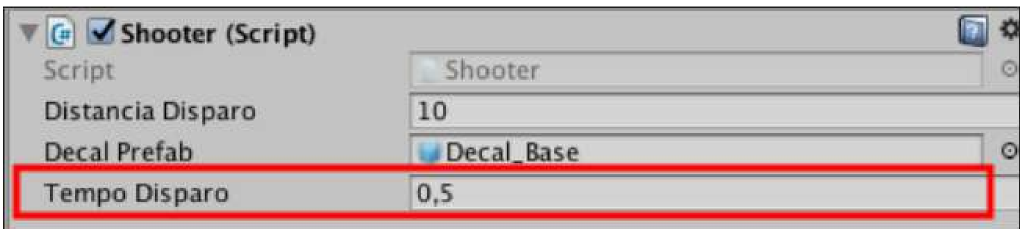


Fig. 8.45

También me he fijado que nuestro Prefab **Decal_Base** es muy grande para el tipo de arma que tenemos en la escena. Para cambiar el tamaño del Prefab accedemos a la ventana **Project** y dentro de la carpeta Prefabs seleccionamos el prefab con nombre **Decal_Base**. Una vez seleccionado accedemos a los componentes de este, en la ventana Inspector. En el componente Transform nos aseguramos de que el prefab esta en la posición (0,0,0) , en la rotación (90,0,0) y en la escala (0.2,0.2,0.2). Te muestro una imagen por si no se entiende del todo bien.

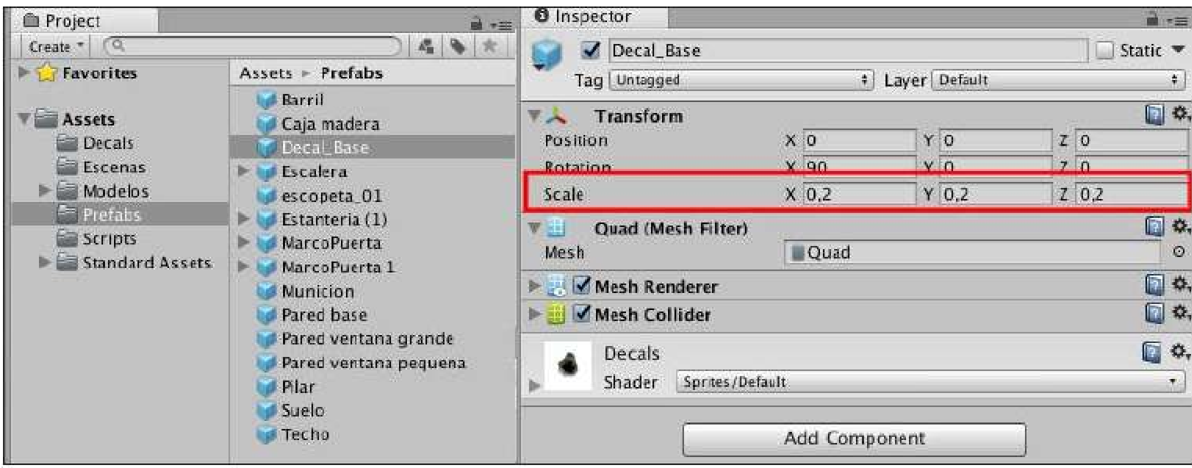


Fig. 8.46

Antes de continuar me he dado cuenta de que el Decal contiene un **MeshCollider** que no nos hace falta y podemos eliminarlo para ello accede al componente **Mesh Collider** y pulsa en el icono con forma de engranaje y selecciona **Remove Component**.

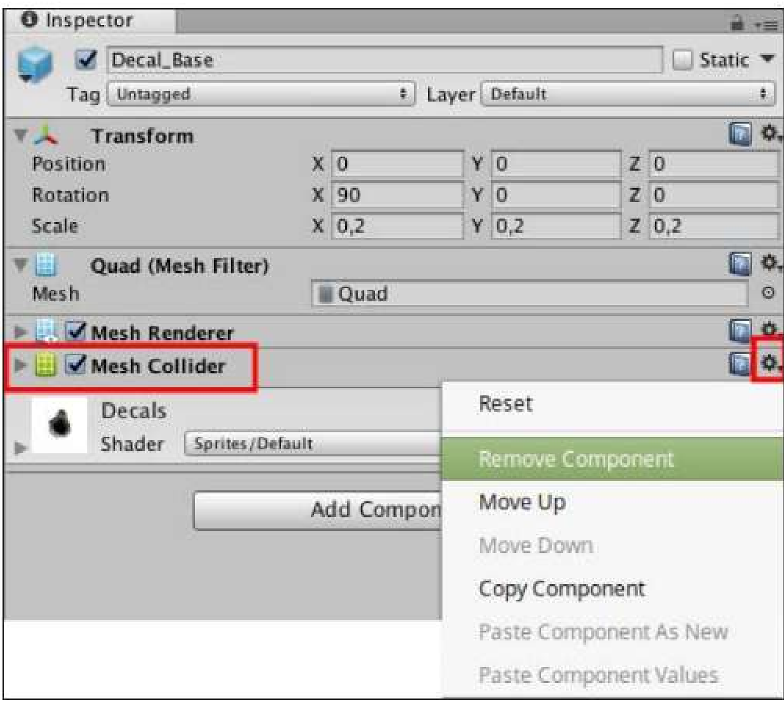


Fig. 8.47

Ahora ya podemos ejecutar la escena y probar como hay un cierto tiempo entre disparo y disparo, pero todavía nuestro **Decal** no acaba de colocarse bien los objetos. El próximo paso será colocar bien nuestro Decals.

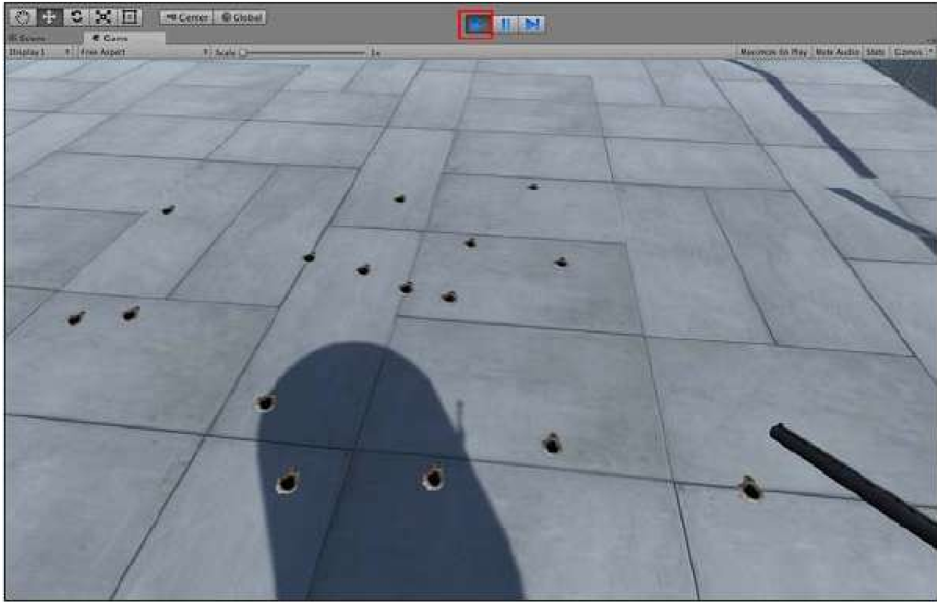


Fig. 8.48

8. Rotación de nuestros Decals

Para rotar nuestro **Decal** necesitamos hacer una visita a la documentación de Unity para utilizar los **Quaternion** para localizar las normales de los objetos. Las normales son los vectores que nos indica hacia donde apunta una cara de un objeto.

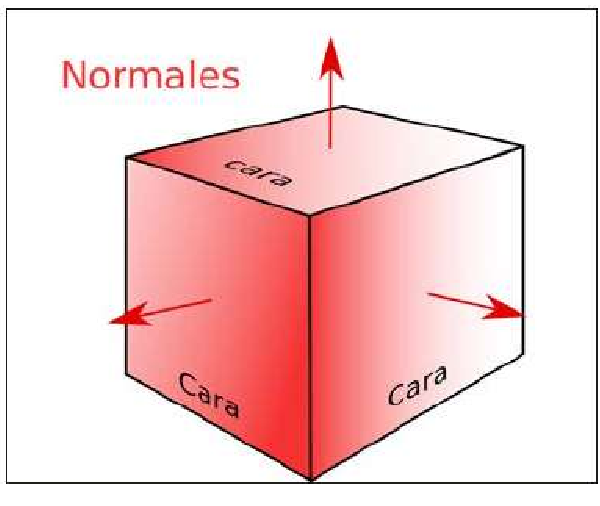


Fig. 8.49

Para ver las rotaciones podemos acceder a la documentación de Unity desde el enlace que te facilito a continuación.

https://docs.unity3d.com/2018.1/Documentation/ScriptReference/Quaternion.html

Dentro de **Quaternion** vamos existen muchos métodos pero el que vamos a utilizar es el **FromToRotation**, al que puedes acceder desde el siguiente enlace que te facilito para mayor comodidad.

https://docs.unity3d.com/2018.1/Documentation/ScriptReference/Quaternion.FromTo-Rotation.html

Lo que realiza este método es una rotación desde una dirección hasta otra dirección, es decir nuestro rayo tiene una dirección en donde va a instanciar nuestro Prefab, el rayo impacta contra un objeto y este rayo revota en dirección a la normal del objeto, Es en ese mismo momento la dirección que queremos que rote nuestro **Decal** para que se posicione correctamente.

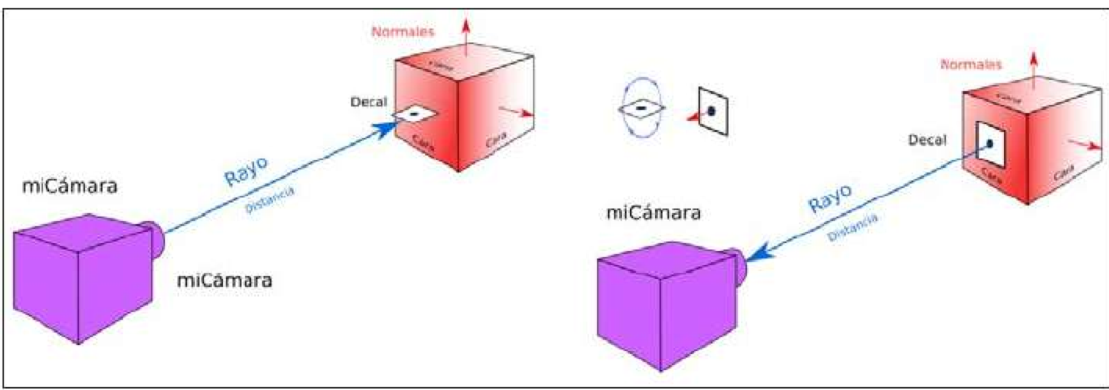


Fig. 8.50

Si volvemos a nuestro **Script Shooter** para editarlo, debemos añadir las siguientes lineas para realizar la rotación.

Script: Shooter.cs

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

public class Shooter : MonoBehaviour
{
  private Ray rayo;
  private RaycastHit hit;
  private Camera miCamara;
  private Vector2 centroCamara;
  public float distanciaDisparo;
  public GameObject decalPrefab;
  public float tempoDisparo;
  private float tempoUltimoDisparo;

  private Quaternion rotDecal;

  void Awake()
  {
    this.miCamara = gameObject.transform.GetChild (0).GetComponent<Camera>();
```

```
      this.centroCamara.x= Screen.width/2;
      this.centroCamara.y= Screen.height/2;
      this.tempoUltimoDisparo = Time.time;
    }

    void Update()
    {
     if(Input.GetButtonDown("Fire"))
     {
       if((Time.time-this.tempoUltimoDisparo)>this.tempoDisparo)
       {
         this.rayo= this.miCamara.ScreenPointToRay(this.centroCamara);
         this.tempoUltimoDisparo = Time.time;

         if(Physics.Raycast (this.rayo,out this.hit,this.distanciaDisparo))
         {
          this.rotDecal= Quaternion.FromToRotation(Vector3.forward,this.hit.
normal);
          GameObject.Instantiate(this.decalPrefab, this.hit.point,this.
rotDecal);
         }
       }
     }
    }
}
```

Primero crearemos una variable privada de tipo Quaternion con nombre rotDecal

Dentro de la función Update vamos a la condición del Phsics.Raycast porque queremos rotar el Decal cuando el rayo golpee el objeto y tomar la normal de la cara del objeto que golpea. Dentro de la condición diremos que this.rotDecal es igual a Quaternion y utilizaremos el método FromToRotation que necesita dos argumentos una dirección de ida y otra de vuelta. Para la ida utilizo un Vector3.forward y para la vuelta vamos a utilizar el RaycastHit con la propiedad normal que es hacia donde retorna.

Por último en el argumento de la instancia que hace referencia a la rotación substituimos la rotación del prefab en sí this.decalPrefab.transform.rotation por la rotación que hemos guardado en this.rotDecal. Recuerda que hay que guardar el script desde Monodevelop para que se ejecuten los cambios.

Ahora al volver a Unity y ejecutar la escena verás como los Decals se rotan correctamente según las normales de los objetos.

Fig. 8.51

Siguiendo con el desarrollo de los Decals si te fijas bien en la escena, la instancia del Decal muchas veces no acaban de verse del todo bien, eso es debido a que deberían posicionarse con un mínimo de separación entre el objeto para poder apreciarse bien el **Decal**, para ello vamos a crear una posición nueva para el **Decal** que le sumaremos a la normal del **RaycastHit**. Accedemos al script **Shooter** y añadimos las siguientes lineas.

Script: Shooter.cs

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

public class Shooter : MonoBehaviour
{
  private Ray rayo;
  private RaycastHit hit;
  private Camera miCamara;
  private Vector2 centroCamara;
  public float distanciaDisparo;
  public GameObject decalPrefab;
  public float tempoDisparo;
  private float tempoUltimoDisparo;
  private Quaternion rotDecal;
  private Vector3 posDecal;

  void Awake()
```

```
  {
    this.miCamara = gameObject.transform.GetChild (0).GetComponent<Camera>();
    this.centroCamara.x= Screen.width/2;
    this.centroCamara.y= Screen.height/2;
    this.tempoUltimoDisparo = Time.time;
  }

  void Update()
  {
   if(Input.GetButtonDown("Fire"))
   {
     if((Time.time-this.tempoUltimoDisparo)>this.tempoDisparo)
     {
       this.rayo= this.miCamara.ScreenPointToRay(this.centroCamara);
       this.tempoUltimoDisparo = Time.time;

       if(Physics.Raycast (this.rayo,out this.hit,this.distanciaDisparo))
       {
          this.rotDecal= Quaternion.FromToRotation(Vector3.forward,this.hit.
normal);
          this.posDecal= this.hit.point+this.hit.normal*0,01f;
          GameObject.Instantiate(this.decalPrefab, this.posDecal,this.
rotDecal);
        }
      }
    }
  }
}
```

Primero crearemos una variable privada de tipo Vector3 con nombre posDecal

Dentro de la función Update vamos a la condición del Phsics.Raycast porque queremos posicionar el Decal poniendo una cierta distancia entre la cara del objeto y el Decal. Para ellos decimos que this.posDecal es la suma entre el punto que golpea el rayo y la normal multiplicado por una distancia mínima que en este caso le he dado un valor de 0,01f (Recuerda que los valores decimales terminan con una f de float).
Por último en el argumento de la instancia que hace referencia a la posición substituimos el this.hit.point por la variable posDecal que contiene la nueva posición.
Recuerda que hay que guardar el script desde Monodevelop para que se ejecuten los cambios.

Una vez volvemos a Unity, ejecuta la escena y cuando dispares veras que los Decals se ven mejor que antes. Ahora seguimos teniendo un problema y es que cuando disparamos a la puerta los decals quedan flotando a mucha distancia de la puerta. Puedes verlo en la siguiente imagen.

Fig. 8.52

9. Selección de objetos para disparar

Como ves en la imagen anterior actualmente el **Raycast** detecta el **Trigger** de la puerta y por ese mismo motivo imprime los decals en el. Si miramos en la documentación de **Physics.Raycast**, disponemos de un parámetro llamado **layerMask** puedes acceder a la documentación desde el enlace que te facilito a continuación.

https://docs.unity3d.com/2018.1/Documentation/ScriptReference/Physics.Raycast.html

Este parámetro se utiliza para especificar que colliders son detectados por el Rayo y cuales son ignorado. Así que tendremos que crear una capa o **Layer** para determinar que queremos que detecte el rayo. Para ello seleccionamos dentro de la ventana **Hierarchy** el objeto con nombre **Sala_Principal** y luego accedemos a la ventana inspector en donde haremos clic encima del botón que se encuentra al lado de Layer y seleccionamos la opción **Add Layer**, como te muestro en la siguiente imagen.

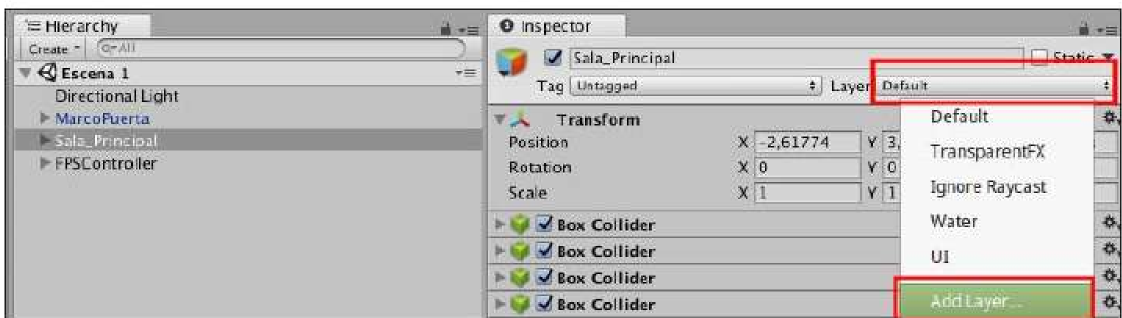


Fig. 8.53

En el inspector se nos abrirá un apartado llamado **Tags & Layers** (Etiquetas y capas). En este apartado vemos que disponemos de 31 **Layers** de las cuales las siete primeras

están desactivadas. Ahora en la **User Layer 8** introducimos el nombre de **ColisionDecals** para crear esta nueva capa. Una vez escribas el nombre pulsa **Enter** y ya tenemos creada una nueva **Layer**.

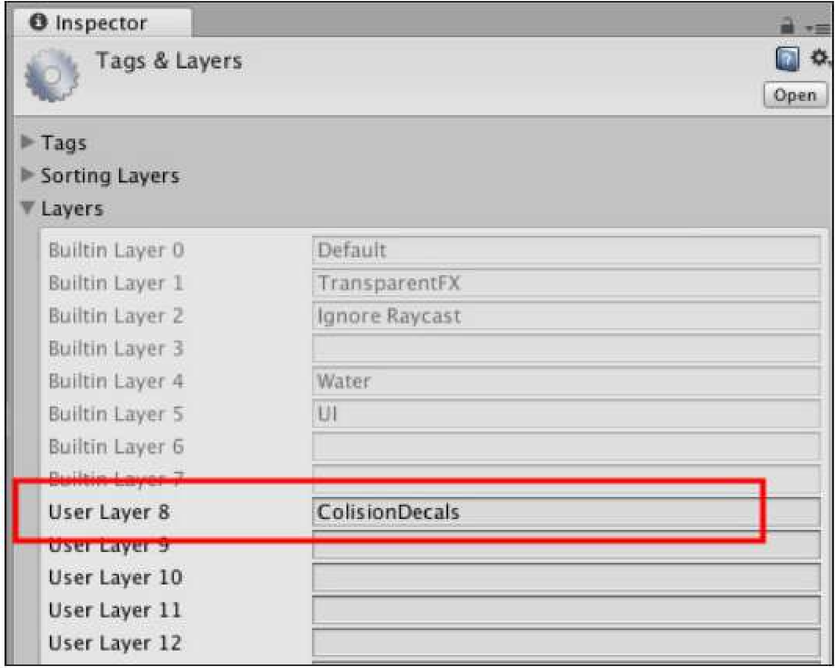


Fig. 8.54

Ahora debemos poner esta capa en los objetos que queremos que detecten el Rayo, en el ejemplo vamos a seleccionar desde la ventana **Hierarchy** el objeto **Sala_Principal** que contiene todos los Colisionadores de paredes y suelo y en la ventana Inspector accedemos otra vez al apartado **Layers** y en el menú seleccionamos la opción **ColisionDecals**.

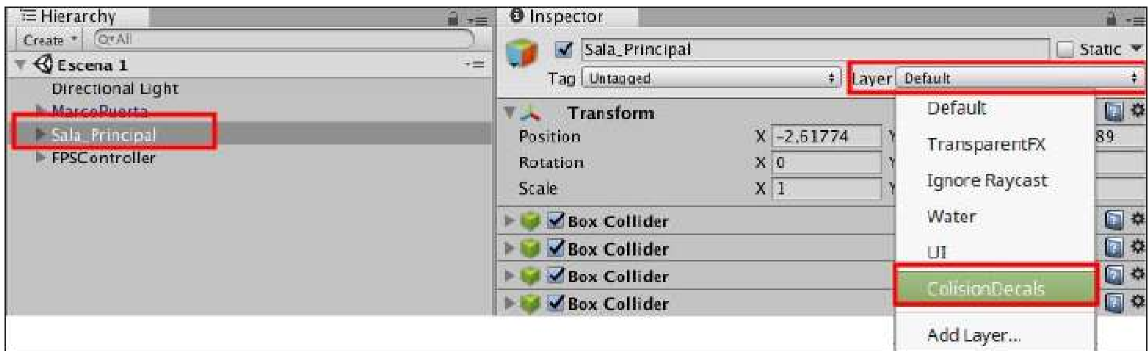


Fig. 8.55

Como el objeto Sala_Principal es Padre de muchos objetos de la escena Unity te preguntara si quieres que todos los hijos de este objeto también pertenezcan a la misma capa, en este ejemplo le decimos que solo queremos cambiar el objeto seleccionado es decir solo el objeto padre como te muestro en la siguiente imagen.

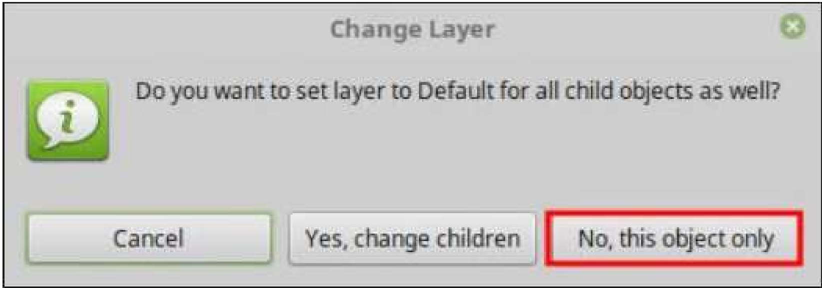


Fig. 8.56

Para seleccionar los demás objetos vamos a la ventana Project y accedemos a la carpeta Prefabs en esta carpeta seleccionamos manteniendo pulsada la tecla **Ctrl** el **Barril**, la **Caja madera**, dentro del Marco Puerta seleccionamos solamente la **Puerta** y el **Pilar**. Una vez seleccionado estos prefabs vamos a la ventana Inspector y en el botón de **Layers** seleccionamos la opción **ColisionDecals**

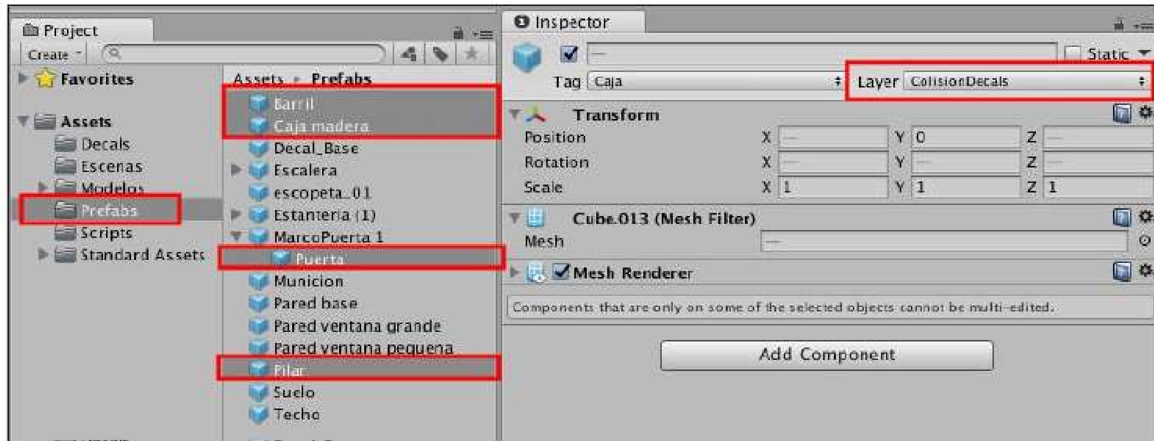


Fig. 8.57

Para que todo esto tenga efecto debemos añadir las siguiente lineas en nuestro Script Shooter.

Script: Shooter.cs

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

public class Shooter : MonoBehaviour
{
  private Ray rayo;
  private RaycastHit hit;
  private Camera miCamara;
 private Vector2 centroCamara;
```

```
  public float distanciaDisparo;
  public GameObject decalPrefab;
  public float tempoDisparo;
  private float tempoUltimoDisparo;
  private Quaternion rotDecal;
  private Vector3 posDecal;
  public LayerMask decalLayerMask;

  void Awake()
  {
    this.miCamara = gameObject.transform.GetChild (0).GetComponent<Camera>();
    this.centroCamara.x= Screen.width/2;
    this.centroCamara.y= Screen.height/2;
    this.tempoUltimoDisparo = Time.time;
  }

  void Update()
  {
    if(Input.GetButtonDown("Fire"))
   {
     if((Time.time-this.tempoUltimoDisparo)>this.tempoDisparo)
     {
       this.rayo= this.miCamara.ScreenPointToRay(this.centroCamara);
       this.tempoUltimoDisparo = Time.time;

       if(Physics.Raycast (this.rayo,out this.hit,this.distanciaDisparo,this.
decalLayerMask))
       {
           this.rotDecal= Quaternion.FromToRotation(Vector3.forward,this.hit.
normal);
           this.posDecal= this.hit.point+this.hit.normal*0,01f;
          GameObject.Instantiate(this.decalPrefab, this.posDecal,this.
rotDecal);
        }
      }
    }
 }
}
```

Primero creamos una variable pública de tipo LayerMask a la que he llamado decalLayerMask.

Dentro de la función Update vamos a la condición del Phsics.Raycast y añadimos este argumento this.decalLayerMask.
Recuerda que hay que guardar el script desde Monodevelop para que se ejecuten los cambios.

Una vez hemos realizado los cambios en el script al volver a Unity debemos seleccionar nuestro FPSController y ver en el componente Script Shooter que se nos aparece la

variable Decal Layer Mask en donde debemos desplegar el menu y seleccionar la opción **ColisionDecals**.

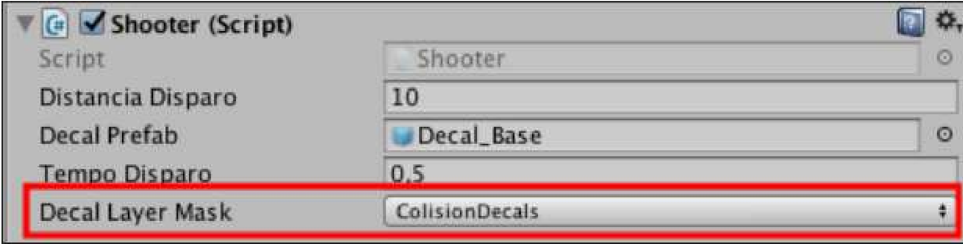


Fig. 8.58

Ahora puedes ejecutar la escena y comprobar como al disparar en las estanterías los decals ignoran completamente estas y se imprimen en la pared.

10. Crear un Array de Decals

Hemos visto como funciona el **Raycast** y como podemos instanciar un prefab en forma de Decal en los ejemplos anteriores. Uno de los inconvenientes que podemos tener es que rellenemos toda la escena de Prefabs y esto nos puede dar problemas ya que sobrecargamos la escena indiscriminadamente. En este apartado vamos a crear un Array que contenga un numero determinado de Prefabs y que cuando terminemos de recorrer el Array volvamos a reutilizar el primero que imprimimos, también vamos a resolver el problema de los Decals que se quedan flotando en la puerta.

Para este ejemplo vamos a variar el script **Shooter** de nuestro **FPSController** como te muestro a continuación. Vamos a crear un sistema de disparo que imprima el **Prefab Decal_Base** dentro de un **Array** para tener un numero máximo de **Prefabs** en la escena.

Script: Shooter.cs

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

public class Shooter: MonoBehaviour
{
  private Ray rayo;
  private RaycastHit hit;
 public float distanciaDisparo;
  private Camera camara;
  private Vector2 centroCamara;

  public GameObject[] decalsPrefabs; //Array de los prefabs
```

```
public GameObject[] createdDecals; //Array para crear los Decals
public int decalIndex;

public float tempoDisparo;
private float tempoUltimoDisparo;
private Quaternion rotDecal;
private Vector3 posDecal;
public LayerMask decalLayerMask;

void Awake()
{
 this.camara = gameObject.transform.GetChild (0).GetComponent<Camera>();
 this.centroCamara.x= Screen.width/2;
 this.centroCamara.y= Screen.height/2;
 this.tempoUltimoDisparo = Time.time;

 for(int decalNum=0; decalNum<this.createdDecals.Length; decalNum++)
 {
      this.createdDecals[decalNum] = GameObject.Instantiate(this.de-
calsPrefabs[0], Vector3.Zero,                    Quaternion.identity)as
GameObject;
      this.createdDecals[decalNum].GetComponent<Renderer>().enabled=false;
 }
  this.decalIndex=0
}

void Update()
{
 if(Input.GetButtonDown("Fire"))
 {
   if((Time.time-this.tempoUltimoDisparo)>this.tempoDisparo)
   {
     this.rayo= this.miCamara.ScreenPointToRay(this.centroCamara);
     this.tempoUltimoDisparo = Time.time;

     if(Physics.Raycast (this.rayo,out this.hit,this.distanciaDisparo,
this.decalLayerMask))
     {
         this.rotDecal= Quaternion.FromToRotation(Vector3.forward,this.
hit.normal);
        this.posDecal= this.hit.point+this.hit.normal*0,01f;
       this.createdDecals[this.decalIndex].transform.position=this.
posDecal;
        this.createdDecals[this.decalIndex].transform.rotation=this.
rotDecal;
```

```
            this.createdDecals[this.decalIndex].GetComponent<Renderer>().
enabled=true;
            this.decalIndex++;
            if(this.decalIndex>9)
            {
         this.decalIndex=0;
            }
          }
       }
     }
 }
}
```

Primero vamos a crear dos variables de tipo GameObject que son arrays: una para guardar el prefab que contiene el decal que he llamado decalsPrefabs y la otra variable de tipo GameObject que es otro array para guardar el numero de Decals que quiero que me imprima en escena que he llamado createdDecals. Para poder localizar cada elemento en el array he creado una variable de tipo Integer con nombre decalIndex.

Dentro de la función Awake he creado un loop for en donde me permitirá recorrer el array de los Decals que quiero crear, dentro de este loop voy a instanciar el numero de decals que quiero para la escena en su posición y rotación inicial. Luego accedo a su componente Renderer y lo desactivo para que no se muestre en escena. Fuera del loop le doy el valor de 0 a la variable decalIndex.

Dentro de la función Update vamos a la condición del Phsics.Raycast y vamos a guardar la tranformacion de la posición del array createdDecals en la variable posDecal y hacemos lo mismo con la rotación. Ahora en la siguiente linea accedemos a su componente <Renderer> y lo activamos para que muestre los decals. En la siguiente linea diremos que la variable decalIndex vaya incrementando en una unidad. Para finalizar y poder reutilizar un máximo de decals creamos una condición que nos dice que si la variable decalIndex es mayor del valor 9 volveremos a reiniciar a decalIndex=0.
Recuerda que hay que guardar el script desde Monodevelop para que se ejecuten los cambios.

Al volver a Unity y antes de ejecutar la escena debemos introducir unos parámetros en el componente Script de nuestro **FPSController**. En **DecalsPrefabs** el tamaño le damos el valor de 1 puesto que solo tenemos un prefab con un **Decal**. En el **CreateDecals** le damos el tamaño de 10 en el parámetro **Size**.

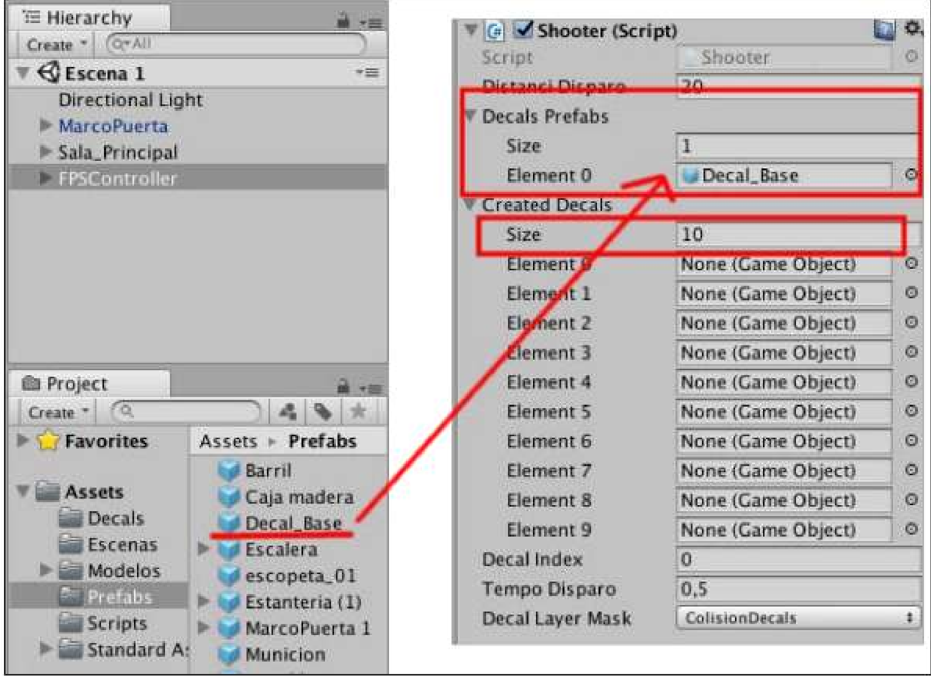


Fig. 8.59

Ahora cada vez que lleguemos a 10 disparos se reutilizara los decals para el siguiente disparo.

11. Emparentar los Decals

Para solucionar el problema de los decals que quedan flotando en la puerta debemos hacer que nuestro **Decal** se emparente con la puerta. Para ello si continuamos con el Script Shooter debemos añadir las siguientes lineas.

Script: Shooter.cs

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

public class Shooter: MonoBehaviour
{
  private Ray rayo;
  private RaycastHit hit;
  public float distanciaDisparo;
  private Camera camara;
  private Vector2 centroCamara;
 public GameObject[] decalsPrefabs; //Array de los prefabs
 public GameObject[] createdDecals; //Array para crear los Decals
```

```
  public int decalIndex;
  public float tempoDisparo;
  private float tempoUltimoDisparo;
  private Quaternion rotDecal;
  private Vector3 posDecal;
  public LayerMask decalLayerMask;

  void Awake()
  {
    this.camara = gameObject.transform.GetChild (0).GetComponent<Camera>();
    this.centroCamara.x= Screen.width/2;
    this.centroCamara.y= Screen.height/2;
    this.tempoUltimoDisparo = Time.time;

    for(int decalNum=0; decalNum<this.createdDecals.Length; decalNum++)
    {
        this.createdDecals[decalNum] = GameObject.Instantiate(this.de-
calsPrefabs[0], Vector3.Zero,                    Quaternion.identity)as
GameObject;
        this.createdDecals[decalNum].GetComponent<Renderer>().enabled=false;
    }
     this.decalIndex=0
  }

  void Update()
  {
    if(Input.GetButtonDown("Fire"))
   {
     if((Time.time-this.tempoUltimoDisparo)>this.tempoDisparo)
     {
       this.rayo= this.miCamara.ScreenPointToRay(this.centroCamara);
       this.tempoUltimoDisparo = Time.time;

       if(Physics.Raycast (this.rayo,out this.hit,this.distanciaDisparo,
this.decalLayerMask))
       {
            this.rotDecal= Quaternion.FromToRotation(Vector3.forward,this.
hit.normal);
           this.posDecal= this.hit.point+this.hit.normal*0,01f;
          this.createdDecals[this.decalIndex].transform.position=this.
posDecal;
           this.createdDecals[this.decalIndex].transform.rotation=this.
rotDecal;
           this.createdDecals[this.decalIndex].transform.parent=null;
           this.createdDecals[this.decalIndex].GetComponent<Renderer>().
enabled=true;
         if(this.decal.hit.collider.tag=="Puerta")
```

```
            {
                  this.createdDecals[this.decalIndex].transform.parent= this.
hit.collider.gameObject.tranform;
            }
            this.decalIndex++;
            if(this.decalIndex>9)
            {
                  this.decalIndex=0;
              }
           }
       }
      }
   }
}
```

Dentro de la función Update vamos a la condición del Phsics.Raycast y vamos a decirle al array createdDecals que su transformación parent de momento sea null, es decir que esté vacía.
Después crearemos una condición en la que cuando el rayo golpee un collider con una etiqueta con nombre Puerta (debemos asegurarnos que la puerta tiene esta etiqueta para que funcione), entonces le diremos que el array de createdDecals se emparente con el objeto que a golpeado el rayo.
Recuerda que hay que guardar el script desde Monodevelop para que se ejecuten los cambios.

Antes de volver a Unity quiero que entiendas que lo que hemos hecho es cambiar de padre un objeto el cual adquiere las transformaciones del nuevo padre. Si quieres puedes ver el ejemplo que te muestra la documentación de Unity accediendo al enlace que te facilito a continuación.

https://docs.unity3d.com/2018.1/Documentation/ScriptReference/Transform-parent.html

Si volvemos a Unity deberemos comprobar que la puerta tiene una etiqueta con el nombre Puerta como hemos puesto en el script, de lo contrario no va a funcionar. Si no tiene una etiqueta selecciona la puerta y crea una etiqueta en la ventana Inspector en la opción **tag** y escribe el nombre Puerta. Luego aplicarle la etiqueta a la puerta y ya podemos ejecutar la escena. Podrás comprobar como al disparar en la puerta y esta abrirse los Decals se quedan incrustados en la puerta siguiendo el movimiento de esta.

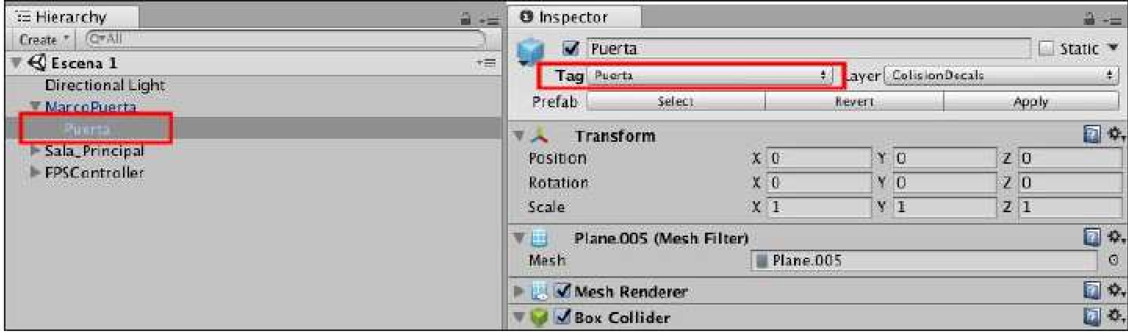


Fig. 8.60

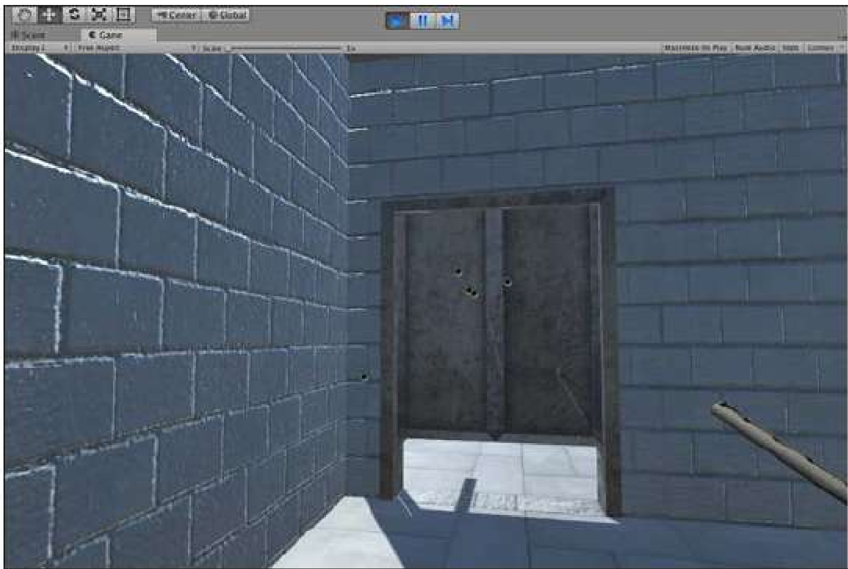


Fig. 8.61

12. Destruir las Cajas

En el escenario tenemos varias cajas a las que podemos añadirle un script que determine un cierto daño y aprovechar los decals y el **Raycast** para hacerlas desaparecer de la escena. La idea es que creemos la sensación de que disparamos a la caja y que esta, al tercer impacto desaparezca, por supuesto también podemos aplicarle una fuerza a las cajas con cada impacto para que esta se mueva cuando el rayo impacta con ella.

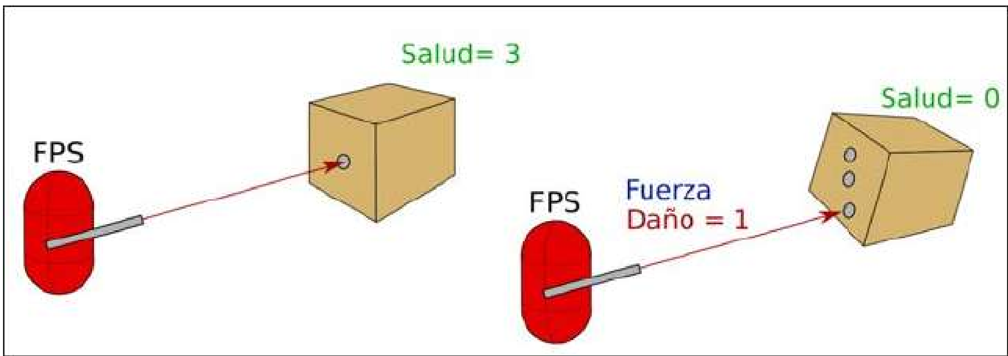


Fig. 8.62

Para empezar vamos a ver que componentes etiquetas y capas deben tener las cajas para que funcione todo cuando creemos el script. Como Tag (Etiqueta) debe tener una con el nombre Caja que servirá para identificarla con el Raycast, también debe de tener la Layer (Capa) ColisionDecals para que el Rayo no la ignore e impacte con la caja. Como componentes un BoxCollider y para poder crear una fuerza que empuje la caja le pondremos un Rigidbody.

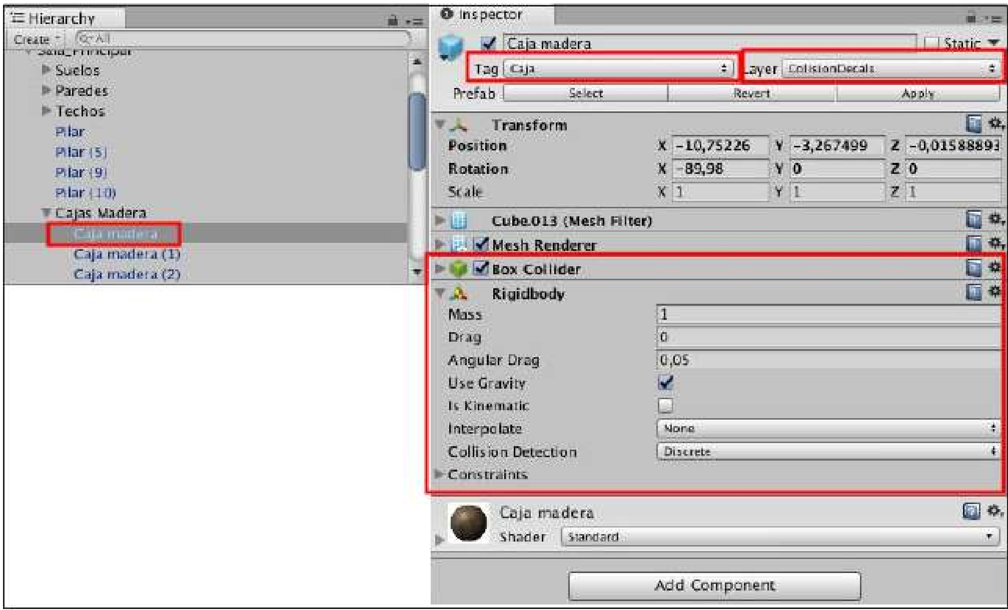


Fig. 8.63

Una vez te asegures de que las cajas contienen todo lo nombrado anteriormente vamos a la ventana **Project** y vamos a crear un script dentro de la carpeta Scripts con el nombre **DestroyCajas.** Este script que hemos creado lo tienen que llevar todas las cajas de la escena y por ese motivo lo podemos aplicar al **Prefab** directamente, arrastrar el script a cada una de las cajas o otra forma que va a darnos el mismo resultado es arrastrarlo encima de una caja de la escena y en la ventana Inspector te aparecerán tres botones uno de lo cuales es **Apply** que al pulsarse realiza el cambio en todas las otras cajas. Eso es debido a que todas las cajas de la escena son **Prefabs** duplicados.

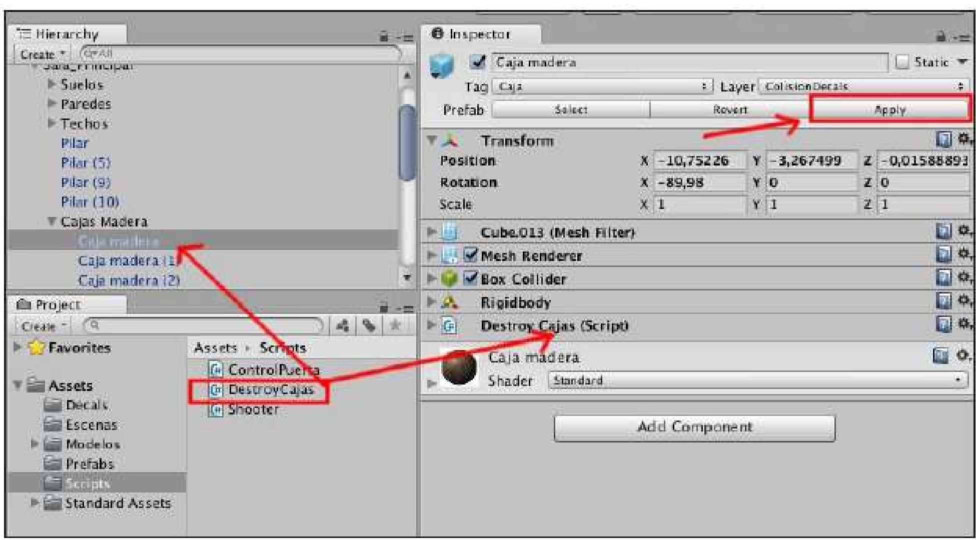


Fig. 8.64

Si hacemos doble clic encima del script se nos abrirá el editor **Monodevelop** si no lo tienes abierto. El siguiente script vamos a crear una propiedad Salud y un método para poder proporcionar "daño" a la caja.

Script: DestroyCajas.cs

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

public class DestroyCajas: MonoBehaviour
{
  public int saludActual=3;

  public void Daño(int cantidadDaño)
  {
    this.saludActual -= cantidadDaño;
   if(this.saludActual <=0)
   {
       gameObject.SetActive(false);
   }
  }
}
```

Crearemos una variable pública de tipo int con el nombre saludActural, para proporcionarle un valor entero de salud a la caja.
Despues crearemos un método o función con nombre Daño y un argumento de tipo int con un nombre de cantidadDaño para poder valorar el daño que recibe. Dentro de la función diremos que saludActual sera el valor de la resta entre saludActual y la cantidadDaño. Después ponemos la condición de que cuando la salud actual sea menor o igual a cero, el gameObject en minúscula que hace referencia a la propia caja se desactive diciéndole que su componente Active se ponga en falso es decir se desactive.
Recuerda que hay que guardar el script desde Monodevelop para que se ejecuten los cambios.

Si volvemos a Unity no va a suceder nada de momento porque debemos modificar el Script **Shooter** de nuestro **FPSController** para que interactúe con la Caja, pero si podemos ver como se han creado en el componente script de las cajas una propiedad Salud Actual con un valor de 3.

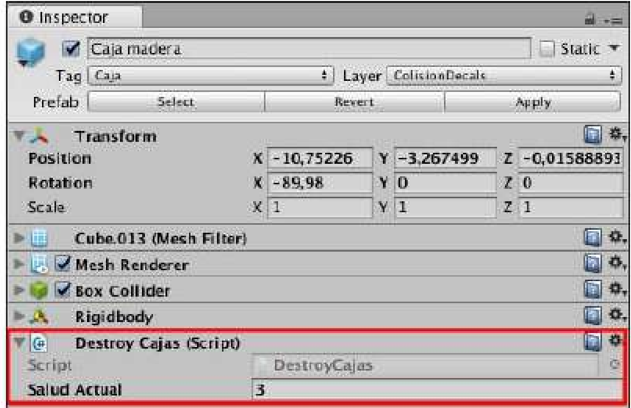


Fig. 8.65

Ahora el siguiente y ultimo paso es acceder al script Shooter para hacer las siguientes modificaciones. Queremos hacer daño a la caja y que se destruya (en este ejemplo la desactivamos), también que al recibir el impacto de nuestros Decals reciba una fuerza que la empuje y por ultimo no queremos que los decals queden flotando cuando la caja desaparezca.

Script: Shooter.cs

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

public class Shooter: MonoBehaviour
{
  private Ray rayo;
  private RaycastHit hit;
  public float distanciaDisparo;
  private Camera camara;
  private Vector2 centroCamara;
  public GameObject[] decalsPrefabs; //Array de los prefabs
  public GameObject[] createdDecals; //Array para crear los Decals
  public int decalIndex;
  public float tempoDisparo;
  private float tempoUltimoDisparo;
  private Quaternion rotDecal;
  private Vector3 posDecal;
  public LayerMask decalLayerMask;

  public float fuerzahit = 150f;
  public int escopetaDaño =1;

  void Awake()
  {
    this.camara = gameObject.transform.GetChild (0).GetComponent<Camera>();
    this.centroCamara.x= Screen.width/2;
    this.centroCamara.y= Screen.height/2;
    this.tempoUltimoDisparo = Time.time;

    for(int decalNum=0; decalNum<this.createdDecals.Length; decalNum++)
    {
        this.createdDecals[decalNum] = GameObject.Instantiate(this.de-
calsPrefabs[0], Vector3.Zero,                    Quaternion.identity)as
GameObject;
        this.createdDecals[decalNum].GetComponent<Renderer>().enabled=false;
    }
```

```
    this.decalIndex=0
  }

  void Update()
  {
    if(Input.GetButtonDown("Fire"))
    {
      if((Time.time-this.tempoUltimoDisparo)>this.tempoDisparo)
      {
        this.rayo= this.miCamara.ScreenPointToRay(this.centroCamara);
        this.tempoUltimoDisparo = Time.time;

        if(Physics.Raycast (this.rayo,out this.hit,this.distanciaDisparo,
this.decalLayerMask))
        {
          this.rotDecal= Quaternion.FromToRotation(Vector3.forward,this.hit.
normal);
          this.posDecal= this.hit.point+this.hit.normal*0,01f;
          this.createdDecals[this.decalIndex].transform.position=this.
posDecal;
          this.createdDecals[this.decalIndex].transform.rotation=this.
rotDecal;
          this.createdDecals[this.decalIndex].transform.parent=null;
          this.createdDecals[this.decalIndex].GetComponent<Renderer>().
enabled=true;

          if(this.decal.hit.collider.tag=="Puerta" || this.hit.collider.
tag=="Caja")
          {
            this.createdDecals[this.decalIndex].transform.parent= this.hit.
collider.gameObject.tranform;
          }
          this.decalIndex++;

          if(this.decalIndex>9)
          {
            this.decalIndex=0;
          }
        DestroyCajas salud = hit.collider.GetComponent<DestroyCajas>();

      if(salud != null)
      {
        salud.Daño(escopetaDaño);
      }
      if(hit.rigidbody != null)
```

```
            {
                  hit.rigidbody.AddForce (-hit.normal*fuerzahit);
            }
          }
        }
      }
    }
}
```

Primero crearemos una variable pública de tipo float con nombre fuerzahit con un valor de 150f que sera el valor que empujará a la caja. La otra variable también pública de tipo int con nombre escopeta-Daño con un valor de 1 que es el valor que irá restando la salud de la caja.
Dentro de la función Update nos vamos a dirigir en la condición donde el rayo hit detecta la puerta y emparenta los decals en la puerta, bien pues he añadido otra condición con el operador logico OR || porque en este caso quiero que los decals se emparenten cuando detecta un objeto con etiqueta Puerta o los objetos con etiqueta Caja.
Luego al final de todo he creado una variable dentro del Raycast hit de la clase DestroyCajas con el nombre salud al que voy a poder acceder a los componentes de la clase DestroyCajas. Después creo dos condiciones más, una que si la salud es distinto de nada es por que ha recibido daño por eso mismo dentro de la condición accedemos desde salud al método Daño al que le damos el valor de escopeta-Daño que es 1.La otra condición es que si el rayo golpea el rigidbody le proporcionaremos una fuerza negativa en la dirección a la normal multiplicada por el valor que le hemos dado a la variable fuerzahit.
Recuerda que hay que guardar el script desde Monodevelop para que se ejecuten los cambios.

Una vez más volvemos a Unity y comprobamos que todo está correcto. Al ejecutar la escena deberíamos poder disparar a las cajas y estas primero deberían desplazarse hacia atrás a cada impacto de **Decal** y luego al tercer impacto la caja y los decals tienen que desaparecer. En la siguiente imagen veras como han desaparecido dos cajas.

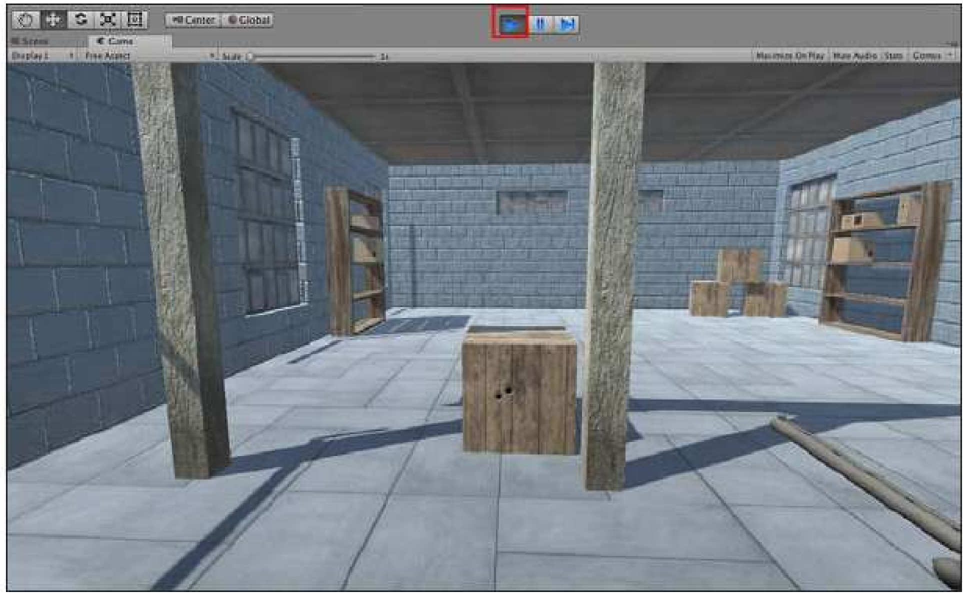


Fig. 8.66

Capítulo 9

UI (Interfaz de Usuario)

- Introducción
- Entendiendo el Canvas (lienzo)
- Entendiendo el Rec Transform
- Image
- ImageRaw
- Text
- Button
- Slider
- Creación de una mirilla
- Creación de un contador de coins
- Creación de una barra de vida
- Hacer daño a nuestro FPSController
- Cómo curar a nuestro FPSController
- Cómo limitar el número de munición
- Pantalla de fallecimiento

1. Introducción

Las UI (User interfaces) o interfaz de usuario son una serie de elementos que nos proporcionan información o interactividad dentro del juego como puede ser una barra de vida o algún menú para acceder a las propiedades de un player.

Para este capitulo vamos ha ir desglosando algunos de los componentes que podemos encontrar en una Interfaz de Usuario. Para empezar a ver algunos de los componentes y explicar para que sirve cada cosa podemos empezar por crea un nuevo proyecto 3d con el nombre que quieras; por ejemplo con el nombre de Introducción UI . Este nuevo proyecto lo vamos a utilizar con el fin de hacer todas las pruebas necesarias para explicar el contenido, no es un proyecto en si.

Para el objetivo principal de este capitulo dispondrás del escenario del capítulo anterior ya montado con varios elementos nuevos, en donde empezaremos a crear una pequeña interfaz de Usuario. Esta interfaz constará de una barra de vida, una mirilla para ver donde apuntamos, un contador de coins y un contador de balas. Esta pequeña interfaz interactuará con nuestro player según las acciones que tengamos en el escenario. Los scripts van a ser rudimentarios de manera que se entienda el objetivo de estos. Para disponer de este escenario debes acceder al material que acompaña el libro y importar en un nuevo proyecto el paquete con el nombre Proyecto_Capitulo_9.unitypackage. De todos modos este proyecto se te pedirá más adelante por el momento se explicará que elementos vamos a utilizar.

2. Entendiendo el Canvas (lienzo)

El canvas es el área o espacio que se utiliza para poner todos los componentes de una interfaz, es decir el canvas es el componente principal de una UI y a partir de este componente se introducen los demás.

Para ver de que estamos hablando accede al menú principal y selecciona la opción GameObject> UI> Image. Ahora veras que en la ventana Hierarchy aparece un nuevo objeto Canvas y dentro del Canvas otro objeto llamado Image. También tiene otro objeto llamado EventSystem que se encarga del sistema de mensajería. Como has comprobado hemos creado una imagen Ui pero Unity nos ha creado el canvas por defecto, eso es debido a que el canvas es la base para crear nuestra interfaz de usuario.

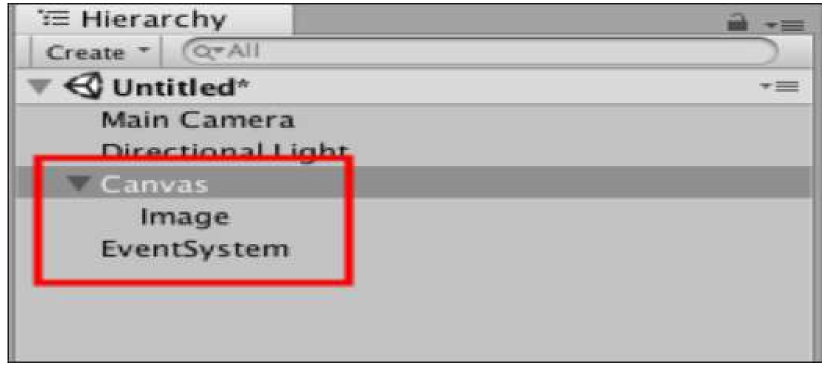


Fig. 9.1

En la vista de escena veras como se nos muestra un rectángulo transparente, esto es el canvas. Este espacio delimita el área que veremos en la ventana Game y nos facilita enormemente la colocación de los elementos de la interfaz de usuario sin necesidad de tener la vista de juego visible en todo momento. En la siguiente imagen te muestro como se ve el Canvas con el componente imagen en la ventana Escene y debajo como se ve en la ventana Game. Si lo pruebas tu mismo seguramente deberás hacer zoom en la ventana Escena para poder ver todo el canvas.

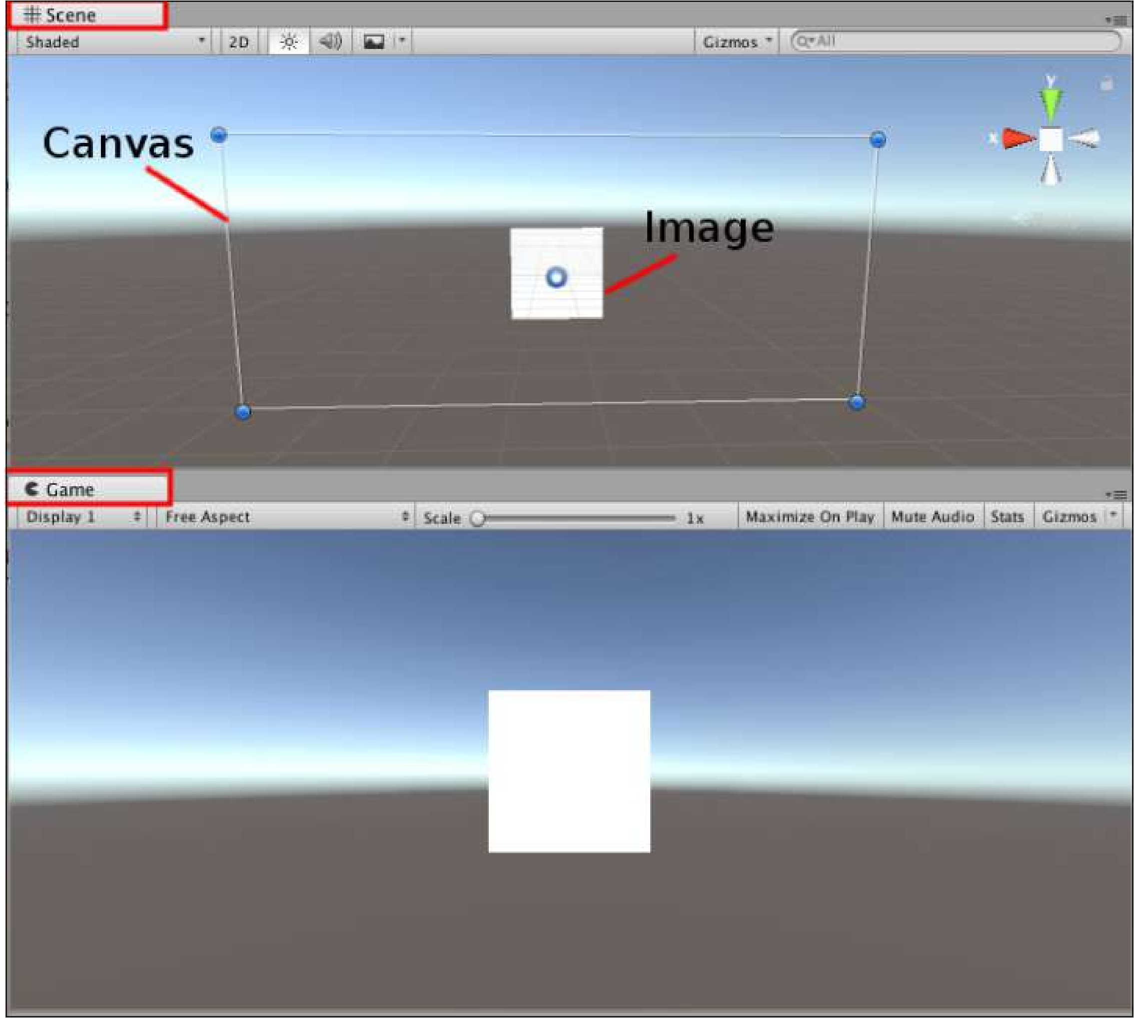


Fig. 9.2

Ahora vamos a crear un componente botón. Accedemos a GameObject> UI> Button. Unity nos crea un objeto botón, pero lo hace dentro del Canvas que ya habíamos creado. Para visualizar mejor el Canvas en la ventana Escena pulsa el botón 2D como te muestro en la siguiente imagen.

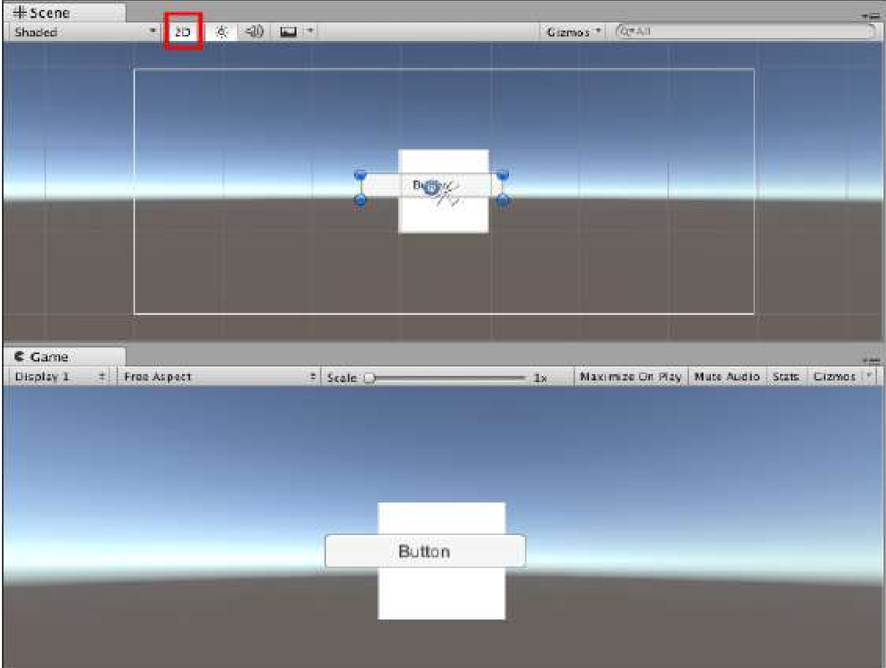


Fig. 9.3

Si miras en la ventana Escena y en la ventana Game, verás que el objeto Button pasa a verse por delante del objeto Image, eso es debido a que los elementos de la UI en el canvas se dibujan en el mismo orden en que aparecen en la jerarquía. Siempre el ultimo en dibujarse se mostrara por encima de los demás.

Si quisiéramos cambiar que elemento se muestra por delante, simplemente tenemos que seleccionar el objeto que queremos que este por delante y desde la ventana jerarquía la arrastramos hasta el final de la lista dentro de canvas. Te muestro como quedaría el poner el elemento Image delante de Button.

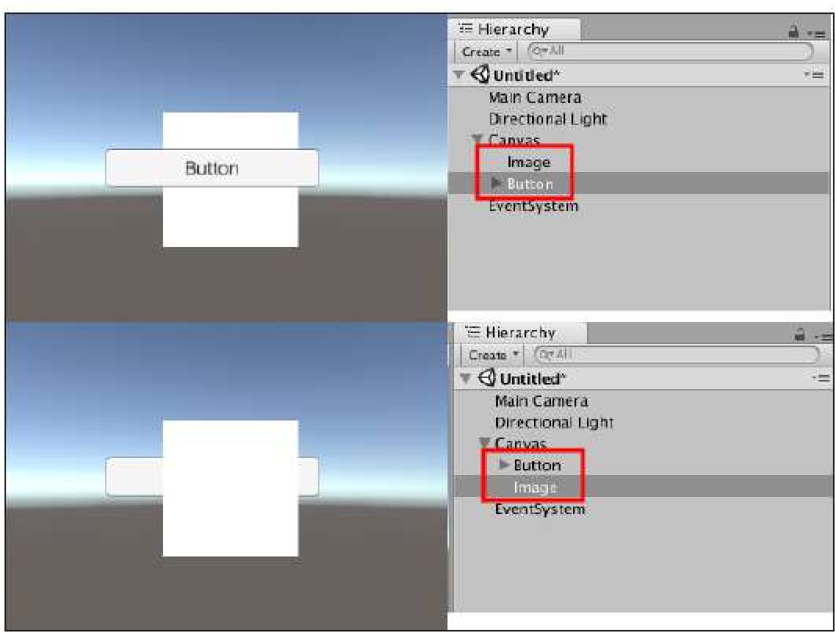


Fig. 9.4

Componentes del Canvas

Como ya sabes todos los GameObjects en Unity tienen un componente llamado Transform que nos permite posicionar el objeto, rotarlo y escalarlo. Todos los objetos de la UI tienen un componente en común que se llama Rect Transform que seria el equivalente al componente Transform de un GameObject.

Si seleccionamos desde la ventana Jerarquía el objeto Canvas y miramos sus componentes en la ventana Inspector veremos que su Rect Transform está desactivado, eso es debido a que el primer componente de Canvas es el componente Canvas que vamos a ver a continuación. En la siguiente imagen te muestro los distintos componentes que encontrarás en la ventana Inspector cuando selecciones el objeto Canvas.

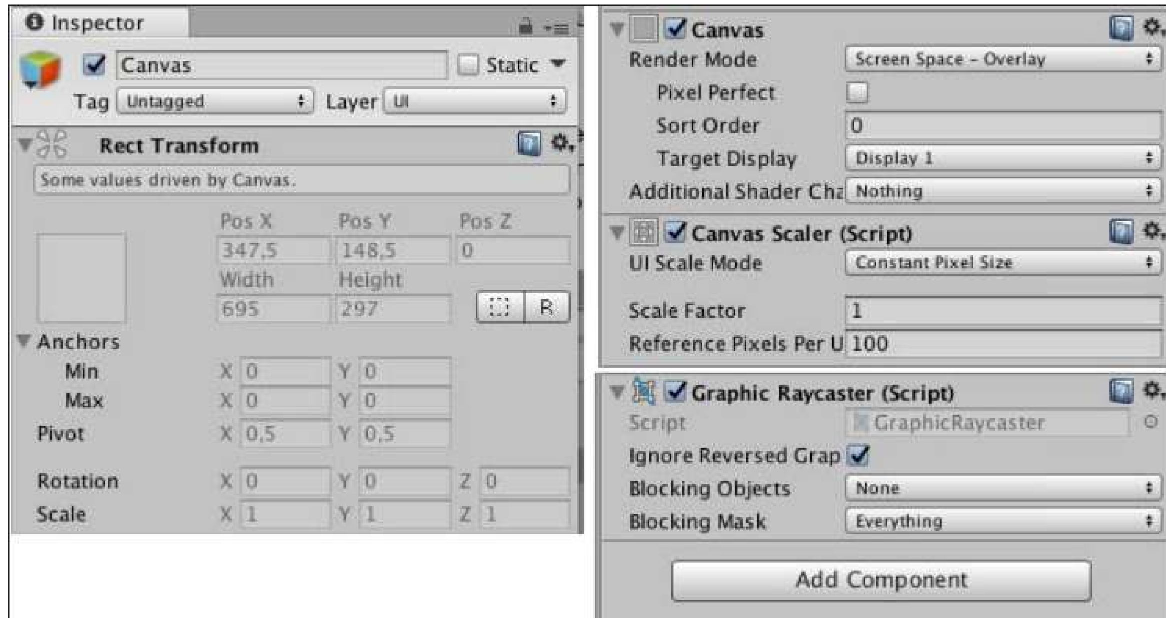


Fig. 9.5

Canvas

Dentro del componente Canvas que se llama igual que el objeto en si tiene un parámetro llamado Render Mode , esto nos permite poder seleccionar que tipo de visualización queremos para nuestro espacio.

Disponemos de dos tipos de modo de renderizado el Screen Space y el World Space:

- ScreenSpace: Este modo dispone de dos opciones.
- ScreenSpace – Overlay: Este modo nos coloca los elementos de la interfaz UI delante de todo y si la pantalla cambia de tamaño o cambia de resolución, el Canvas cambia automáticamente. Tiene un parámetro llamado Pixel Perfect que al activarlo debería mostrar la interfaz de Usuario sin antialising para mayor precisión, es decir va a tratar de dar la mayor calidad posible a la UI.

Fig. 9.6

- ScreenSpace – Camera: Este modo es similar al anterior pero en este caso el canvas se coloca a una determinada distancia, en frente de una cámara. En este modo los elementos se ven representados por la cámara, es decir la configuración de la cámara afectara en la apariencia de la UI. Uno de los parámetros que encontramos a parte del Pixel Perfect es el Render Camera en donde podemos seleccionar la cámara que deseemos.

Fig. 9.7

Si arrastramos la cámara MainCamera de la ventana Jerarquía hacia el parámetro Render Camera veremos que se nos aparece otro parámetro llamado Plane Distance en donde podemos configurar la distancia de la cámara respecto al plano del Canvas.

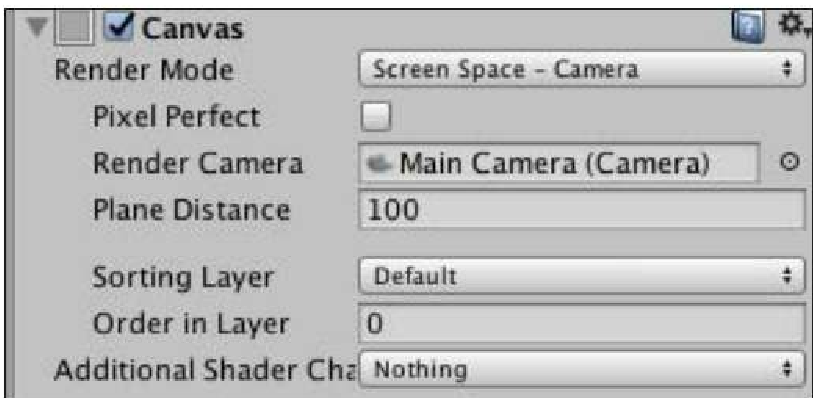


Fig. 9.8

- World Space: Este modo permite que el canvas tenga el mismo comportamiento que cualquier objeto en escena. Cuando activamos este modo de render se nos activa el componente Rect Transform que por defecto hemos visto desactivado. Otro aspecto importante es que en este modo todos los elementos de la UI se pueden mostrar delante o detrás de los objetos que tengamos en escena. Este modo es utilizado cuando queremos tener una UI que forme parte de la escena.

Fig. 9.9

Canvas Scaler

Se utiliza para controlar la escala general y la densidad de píxeles de los elementos de la interfaz de usuario en el Canvas. Esta escala afecta a todos los elementos por debajo del canvas, incluidos los tamaños de fuente y los bordes de la imagen.

Dependiendo del tipo de UI Scale Mode que tengamos seleccionado podemos encontrar distintos parámetros. De modo de Scalado tenemos tres opciones para escoger y es importante saber para que sirve cada una:

Constant Pixel Size: Hace que los elementos de la interfaz de usuario conserven el mismo tamaño en píxeles independientemente del tamaño de la pantalla.

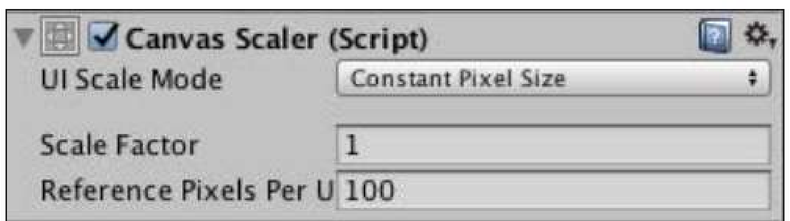


Fig. 9.10

- Scale Factor: Es el factor escala de todos los elementos de la UI en el canvas.
- Reference Pixels Per Unit: Es el pixel de referencia por unidad, es decir un sprite (imagen que se utiliza en las UI) tiene esta configuración, entonces un pixel del sprite cubrirá una unidad en la UI.

Scale With Screen Size: Hace que los elementos de la interfaz de usuario sean más grandes cuanto más grande sea la pantalla.

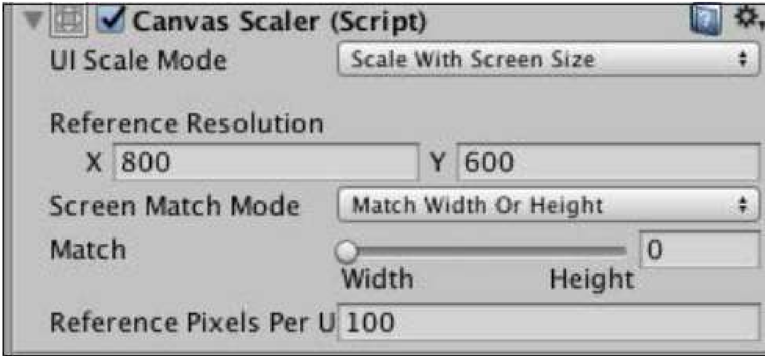


Fig. 9.11

- Reference Resolution: La resolución para la interfaz de usuario. Si la resolución de la pantalla es mayor, la interfaz de usuario se ampliará, y si es más pequeña, la interfaz de usuario se reducirá.
- Screen Match Mode: Un modo utilizado para escalar el área del canvas si la relación de aspecto de la resolución actual no se ajusta a la resolución de referencia.
    - Match Width or Height: Sirve para escalar el área del canvas con el ancho como referencia, la altura como referencia o algo intermedio.
    - Expand: Expande el área del canvas horizontal o verticalmente, por lo que el tamaño nunca será más pequeño que la referencia.
    - Shrink: Recorta el área del canvas horizontal o verticalmente, por lo que el tamaño del canvas nunca será mayor que la referencia.
- Match: Determina si la escala está usando el ancho o la altura como referencia, o una mezcla intermedia.
- Reference Pixels Per Unit: Es el pixel de referencia por unidad, es decir un sprite (imagen que se utiliza en las UI) tiene esta configuración, entonces un pixel del sprite cubrirá una unidad en la UI.

Constant Physical Size: Hace que los elementos de la interfaz de usuario conserven el mismo tamaño físico independientemente del tamaño y la resolución de la pantalla.

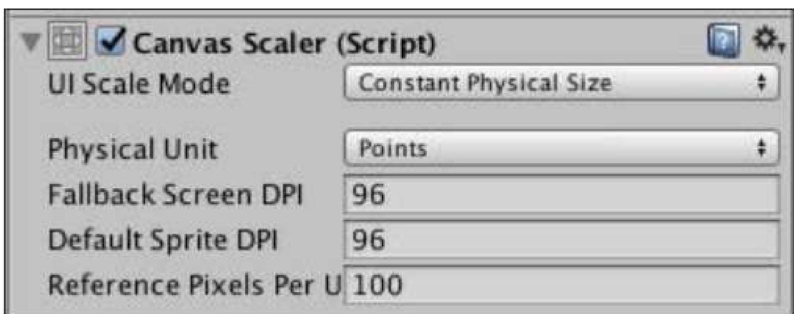


Fig. 9.12

- Physical Unit: Determinamos la unidad física para especificar posiciones y tamaños.
- Fallback Screen DPI: Para determinar si la pantalla DPI (dots por inch - puntos por pulgada) no es conocida. El DPI determina la calidad de impresión en el canvas.
- Default Sprite DPI: Los píxeles por pulgada que se usan para los sprites que tienen una configuración de 'Pixeles por unidad' que coincide con la configuración Reference Pixels Per Unit.

- Reference Pixels Per Unit: Es el pixel de referencia por unidad, es decir un sprite (imagen que se utiliza en las UI) tiene esta configuración, entonces un pixel del sprite cubrirá una unidad en la UI.

Estos serian para empezar los componentes mas importantes de Canvas. Ahora nos hace falta entender como funciona el Rect Transform para poder posicionar correctamente distintos elementos dentro del canvas y luego veremos como configurar los distintos tipos de elementos que tenemos para crear una UI en Unity. Quiero comentar que podemos tener más de un canvas en escena, uno principal y otro especifico para el juego. A continuación vamos a ver el Rect Transform.

3. Entendiendo el Rect Transform

Para utilizar este componente vamos a utilizar el elemento Image que tenemos en escena dentro del canvas. En la siguiente imagen te indico el elemento al que me refiero.

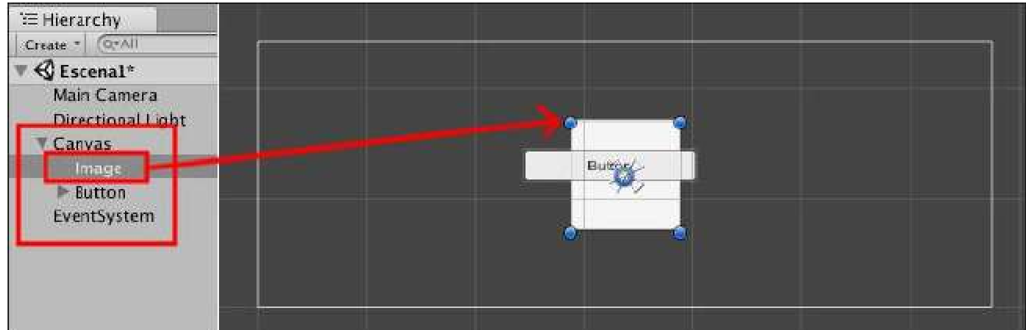


Fig. 9.13

Una vez hemos seleccionado desde la ventana Jerarquía el elemento Image en la ventana Inspector podremos ver su componente Rect Transform que a diferencia del Canvas este si que está activo.

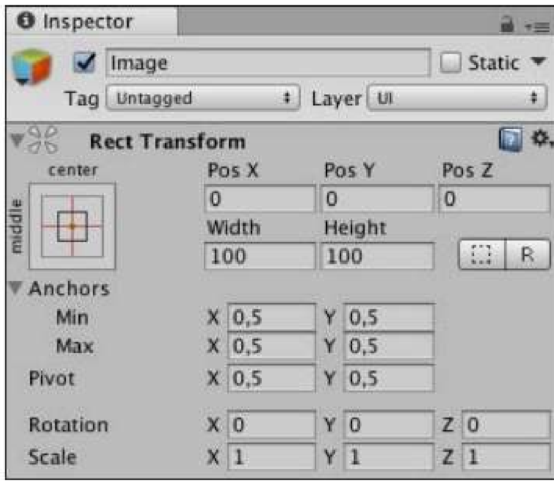


Fig. 9.14

Para mejorar la explicación de las imágenes voy a borrar el elemento Button del canvas, al ser un elemento que no vamos a utilizar por el momento. Cada elemento de UI se representa como un rectángulo en la ventana escena, este se puede manipular de distintas formas una muy intuitiva y manual seria utilizando la opciones de transformación que disponemos en la parte superior de la ventana Escene y utilizar los botones para mover, rotar y escalar como lo hacemos con los objetos 3d. Pero en este caso se suele utilizar mayoritariamente la opción Rect Tool como te muestro a continuación.

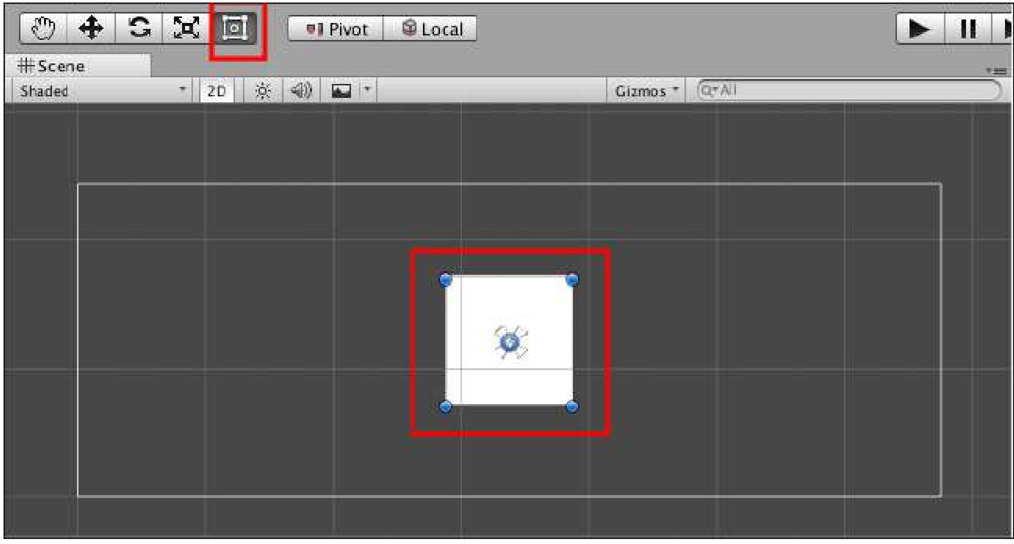


Fig. 9.15

Esta herramienta se compone de unos puntos azules, cuatro en cada esquina y un circulo azul en el centro del elemento llamado Pivot. El funcionamiento es muy sencillo si acercamos el cursor en distintos puntos externos del elemento nos permite mover, escalar o rotar.

Para rotar el elemento image, una vez que haya seleccionado el elemento, puedes mover-lo haciendo clic en cualquier lugar dentro del rectángulo y arrastrándolo.

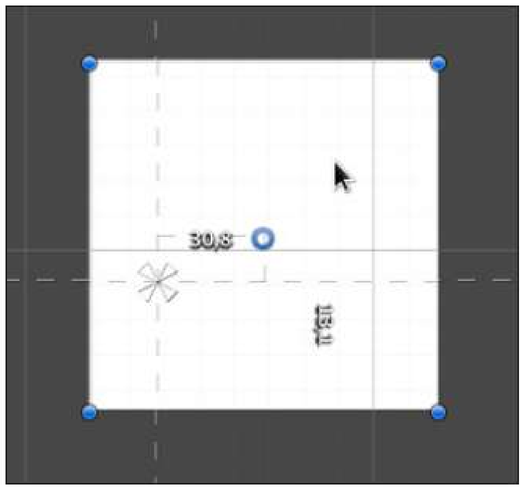


Fig. 9.16

Para cambiar el tamaño haciendo clic en los bordes o las esquinas y arrastrando.

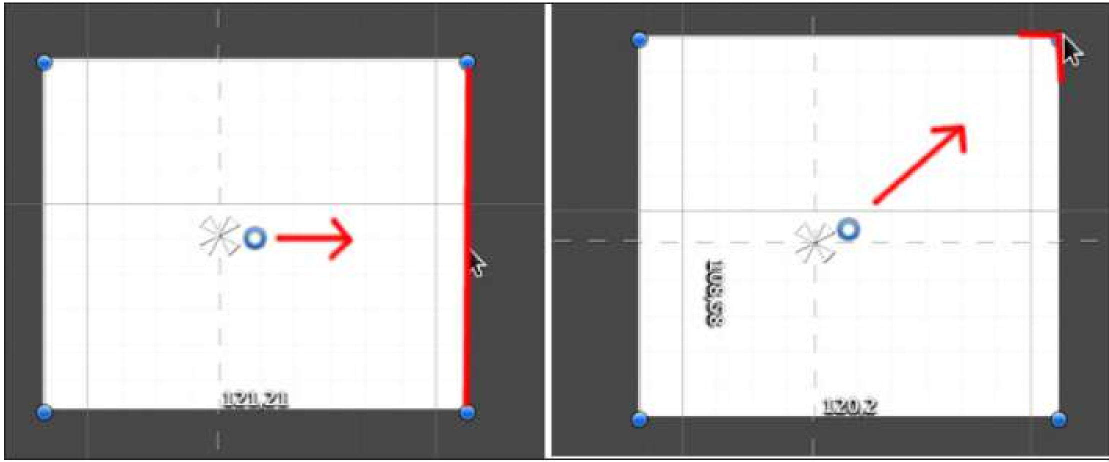


Fig. 9.17

El elemento se puede girar colocando el cursor ligeramente lejos de las esquinas hasta que el cursor del ratón se vea como un símbolo de rotación. A continuación, podemos hacer clic y arrastrar en cualquier dirección para girar.

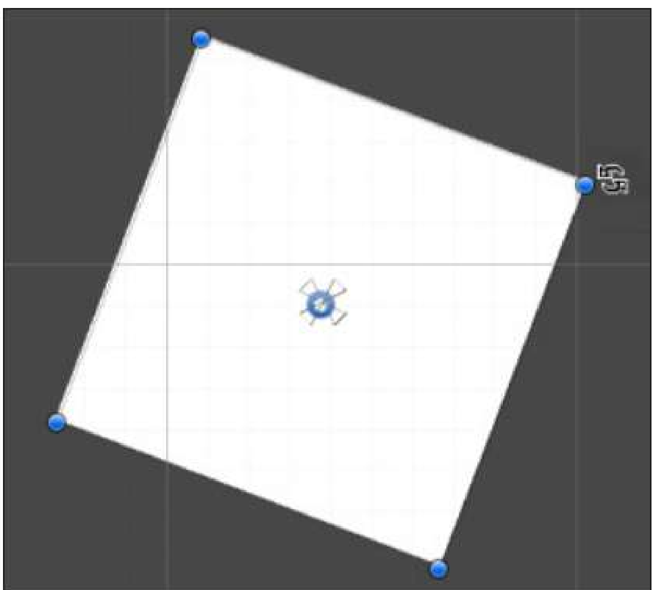


Fig. 9.18

Una vez que conocemos como se puede comportar un elemento dentro del canvas vamos a ver como podemos utilizar los parámetros del componente Rect Transform.

En primer lugar podemos realizar cambios de posición utilizando las cajas de texto que te indico en la siguiente imagen.

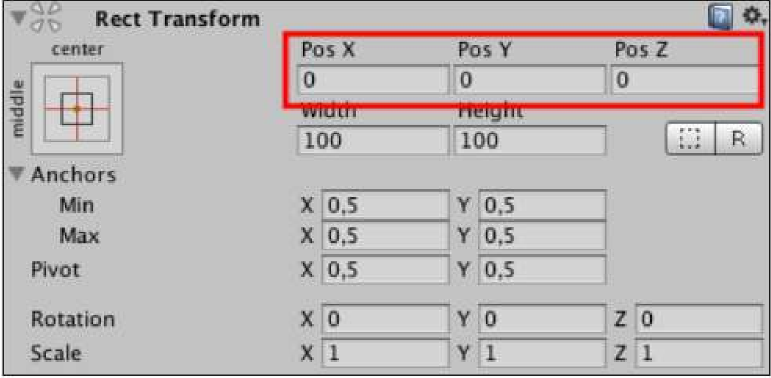


Fig. 9.19

El parámetro que controla la rotación es el que te muestro en la siguiente imagen, pero debes entender como funciona

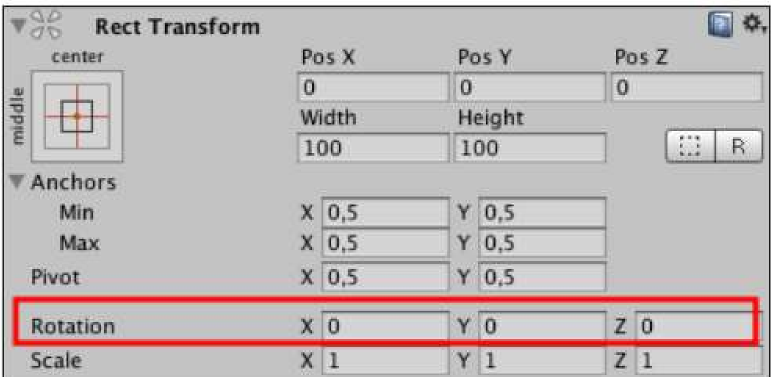


Fig. 9.20

Veras que en la rotación tenemos los ejes de coordenadas x y z y precisamente toma como referencia estas coordenadas. Es decir si estamos utilizando una imagen para una Interfaz de Usuario que se va a ver en 2D y queremos que la imagen quede un poco rotada como te muestro a continuación, veras que he utilizado el eje de coordenadas z.

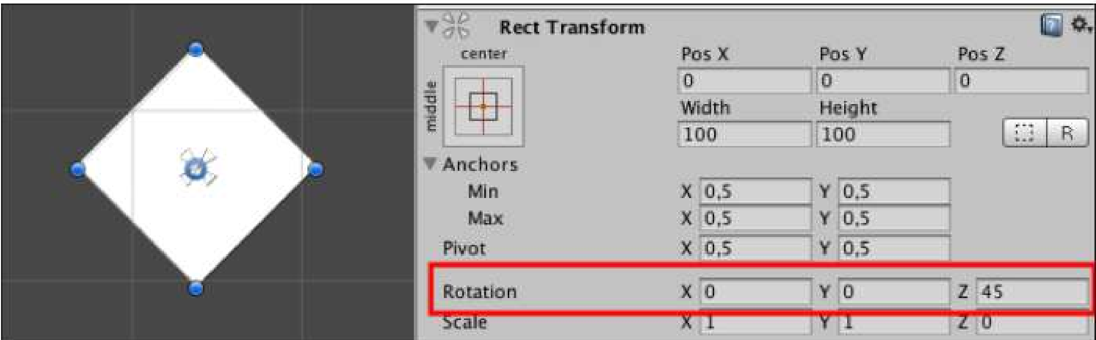


Fig. 9.21

Eso es debido a que utiliza el espacio 3D de la escena y los ejes de coordenadas están dispuestos de la siguiente manera.

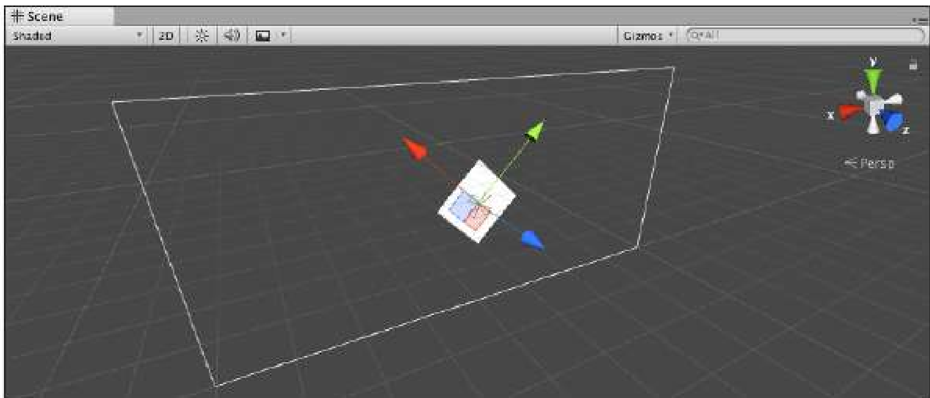


Fig. 9.22

Referente al parámetro de escalado es el que te marco a continuación y funciona siguiendo las mismas normas que en la rotación y la posición. También podemos dejar la escala que tienen los objetos por defecto y variar su anchura (Width) y su altura(Height) como te muestro en la siguiente imagen.

Rect Transform			
center	Pos X	Pos Y	Pos Z
middle	0	0	0
	Width	Height	
	100	100	R
Anchors			
Min	X 0,5	Y 0,5	
Max	X 0,5	Y 0,5	
Pivot	X 0,5	Y 0,5	
Rotation	X 0	Y 0	Z 0
Scale	X 1	Y 1	Z 1

Fig. 9.23

Un elemento importante que disponen los elementos del canvas en el Rect Transform es el parámetro Pivot. Este parámetro puede afectar en la rotación y escalado del elemento según su posición. En la ventana Inspector dispone de dos ejes para posicionarse en X e Y. También podemos posicionar-lo manualmente desde la ventana Escene y está representado por un circulo azul que por defecto se encuentra en el centro del elemento.

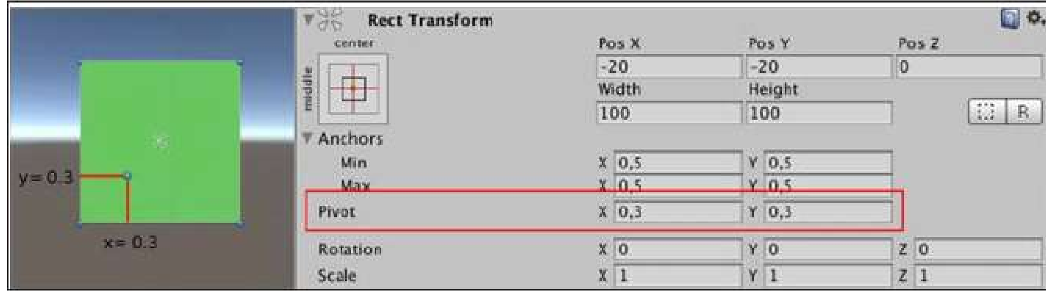


Fig. 9.24

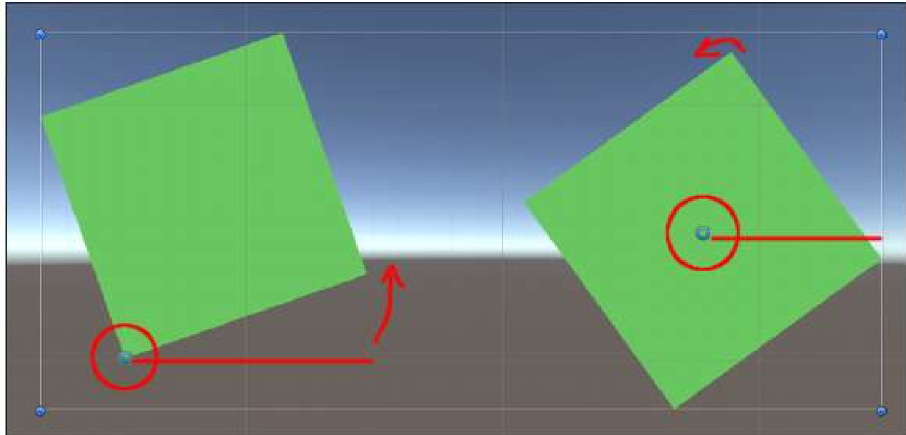


Fig. 9.25

Ahora que hemos identificado las transformaciones básica vamos a ver una herramienta muy útil el Anchor (Anclajes). Los anclajes se muestran como cuatro pequeños mangos triangulares en la vista de escena y la información de este anclaje se muestra en el Inspector.

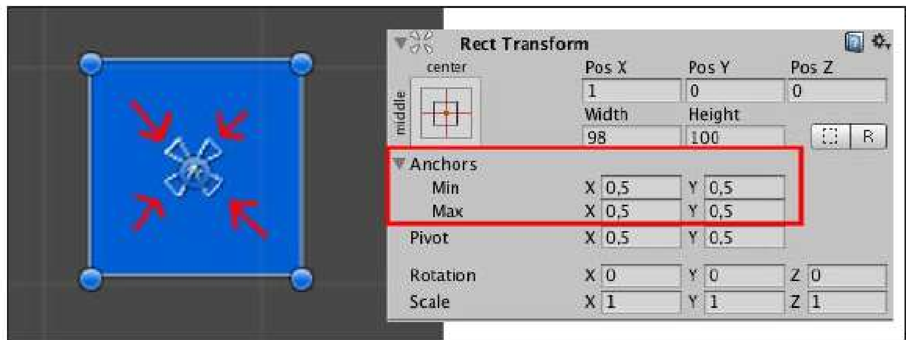


Fig. 9.26

Es un concepto que hay que entender, por ejemplo voy a crear otra imagen dentro del Canvas y la voy a llamar Image Padre. La Image que tengo por defecto la arrastro dentro de la Image Padre.

Ahora si selecciono la Image y le pongo el anclaje en el extremo inferior derecho lo que consigo es que cuando escale el padre se mantenga la distancia.

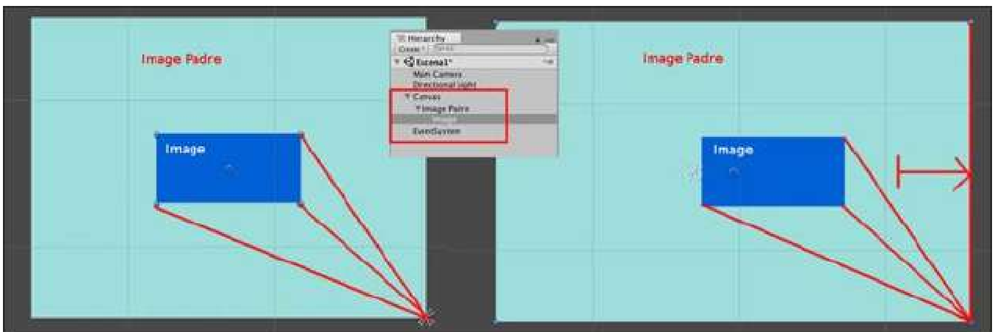


Fig. 9.27

El ejemplo anterior no es necesario que lo realices es solamente para explicar el concepto de anclajes. Ahora que hemos visto que es vamos a ver como podemos colocar estos anclajes en el canvas. Si seleccionamos el elemento del canvas Image y accedemos a la ventana Inspector, en su componente Rect Transform disponemos de una herramienta que nos permite colocar los anclajes y el centro Pivot del elemento donde necesitemos. Si haces clic encima del recuadro que encontraras dentro del Rect Transform en la parte superior izquierda, se desplegara un panel con varios recuadros, como te muestro a continuación.

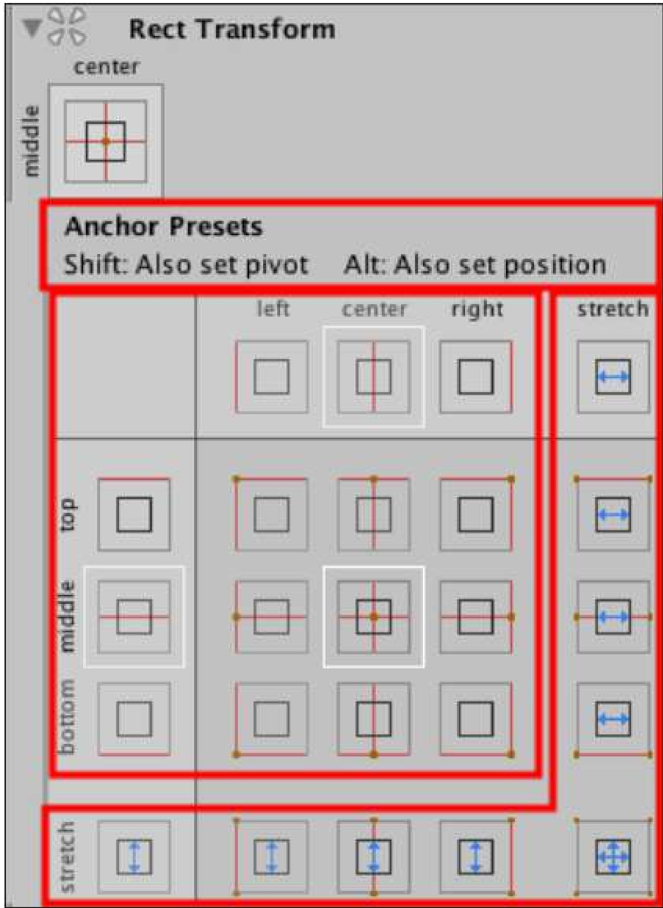


Fig. 9.28

En este panel he marcado tres secciones que debes tener en cuenta la primera en la parte superior tenemos información que son atajos de teclado. Nos dice que cuando hacemos un clic encima de cualquier recuadro de la parte inferior manteniendo pulsada la tecla Shift posicionamos el parámetro pívot y si lo hacemos con la tecla Alt posicionamos el elemento.

En la parte inferior he destacado una parte central y otra que esta en la parte exterior los recuadros de la parte central simbolizan las posiciones izquierda derecha superior e inferior tanto de los extremos como de la partes intermedias. Por otro lado los recuadros con el nombre Stretch hacen referencia al escalado superior inferior.

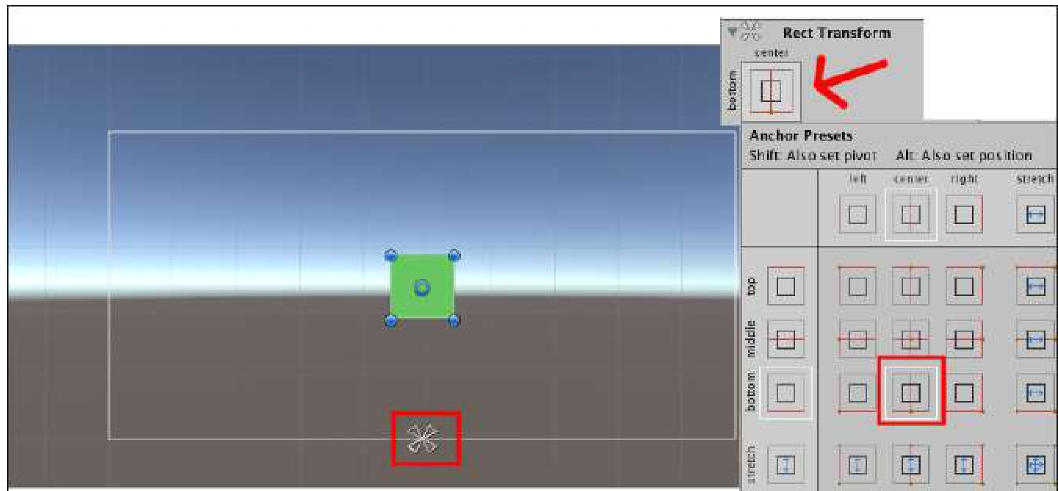


Fig. 9.29

A continuación vamos a ver algunos de los objetos UI que vamos a utilizar para el ejemplo de este capitulo.

4. Image

Como hemos visto anteriormente muestra una imagen no interactiva para el usuario. Este elemento lo podemos utilizar para crear partes de nuestro Interfaz de Usuario con decoración, iconos, etc., y la imagen también se puede cambiar a partir de una secuencia de imágenes con comandos para reflejar los cambios en otros controles. Este tipo de objeto requiere que su textura sea un Sprite.

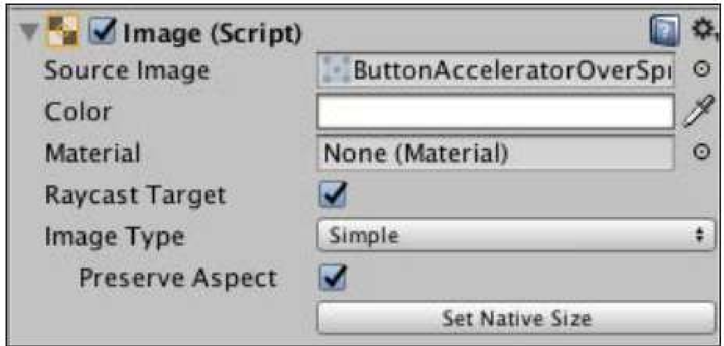


Fig. 9.30

- Source Image : Es donde debemos poner la textura (en este caso siempre ha de ser una imagen de tipo Sprite).
- Color : Este parámetro proporciona un color a la Imagen.
- Material: Podemos ponerle un material para renderizar la imagen.
- Raycast: En el caso de que quisiéramos utilizar Raycast.
- Image Type: Nos permite seleccionar que tipo de imagen queremos,

- Preserve Aspect: Nos permite asegurar el aspecto permitiendo que la imagen conserve la dimensión existente.
- Set Native Size: Este botón sirve para volver a establecer los valores originales de la imagen antes de aplicarle la textura.

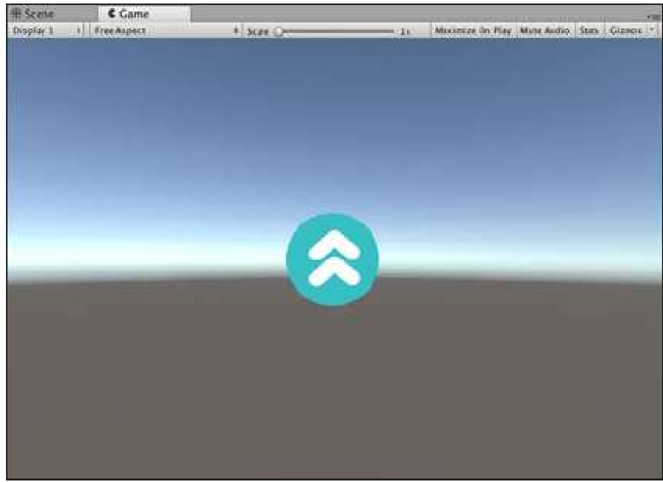


Fig. 9.31

5. Image Raw

Es similar al objeto Image, con algunas diferencias con los parámetros , otro aspecto importante es que Image Raw puede mostrar cualquier textura mientras que Image solo puede mostrar Texturas Sprite.

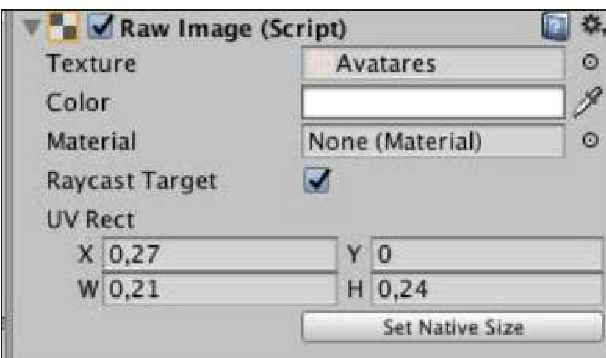


Fig. 9.32

- Texture: En este parámetro ponemos la textura que queremos que se muestre.
- Color: El color en el caso de que queramos poner uno.
- Material: Podemos utilizar un material para la representación de la imagen.
- Raycast Target: En el caso de que quisiéramos utilizar Raycast.
- UV Rect: Son un conjunto de coordenadas una es de posición X e Y. Los valores para W (width) Anchura y Altura H (height). Este parámetro nos permite mostrar una porción en concreto de una textura mayor

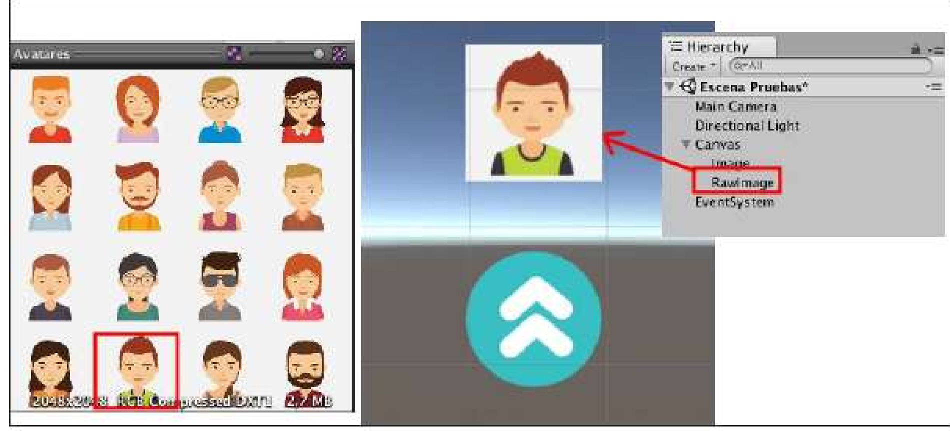


Fig. 9.33

6. Text

El objeto Texto, nos muestra un fragmento de texto no interactivo que nos permite proporcionar títulos etiquetas para otros objetos de GUI o para mostrar las instrucciones o información de nuestro juego.

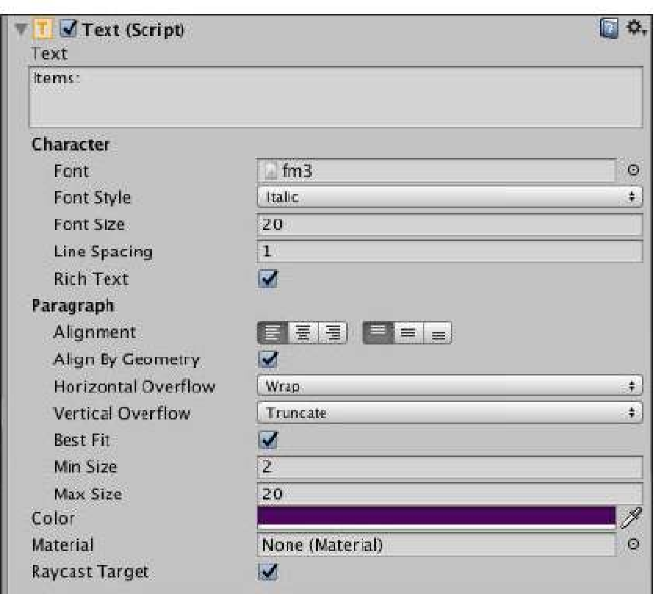


Fig. 9.34

- Text : Aquí disponemos de una caja en donde podemos escribir lo que queremos que se muestre
- Font: La fuente utilizada para mostrar el texto.
- Font Style: El estilo aplicado al texto. Las opciones son Normal, Negrita, Cursiva y Negrita Y Cursiva.
- Font Size: El tamaño del texto que se muestra.
- Line Spacing : La separación vertical entre líneas de texto.
- Rich Text: Interpretar los elementos de marcado en el texto como estilo de texto enriquecido
- Alignment : La alineación horizontal y vertical del texto.

- Align by Geometry : Use las extensiones de la geometría del glifo (Signo o grabado) para realizar la alineación horizontal en lugar de las métricas de glifo.
- Horizontal Overflow: El método utilizado para manejar la situación donde el texto es demasiado ancho para caber en el rectángulo. Las opciones son Wrap y Overflow.
- Vertical Overflow :El método utilizado para manejar la situación donde el texto envuelto es demasiado alto para caber en el rectángulo. Las opciones son Truncate y Overflow.
- Best Fit: Ignora las propiedades de tamaño y simplemente tratar de ajustar el texto al rectángulo del control
- Color: El color utilizado para representar el texto.
- Material: El material utilizado para representar el texto.

El objeto de UI Text es posible que algunos de sus parametros las encontremos dentro de otros tipos de elementos como son los Botones y Toggles. Mas adelante veremos como podemos utilizar un texto para que nos de información del Player mediante Scripts.

7. Button

El objeto Button, responde a un clic del usuario y se usa para iniciar o confirmar una acción. Los ejemplos conocidos incluyen los botones Enviar y Cancelar utilizados en los formularios web.

El botón está diseñado para iniciar una acción cuando el usuario hace clic y la lanza. Si el mouse se quita del control del botón antes de que se suelte, la acción no se lleva a cabo. El botón tiene un solo evento llamado On Click que responde cuando el usuario completa un clic. Los casos de uso típicos incluyen:

- Confirmar una decisión (por ejemplo, iniciar el juego o guardar un juego).
- Moverse a un submenú en una GUI Cancelando una acción en progreso (por ejemplo, descargando una nueva escena)

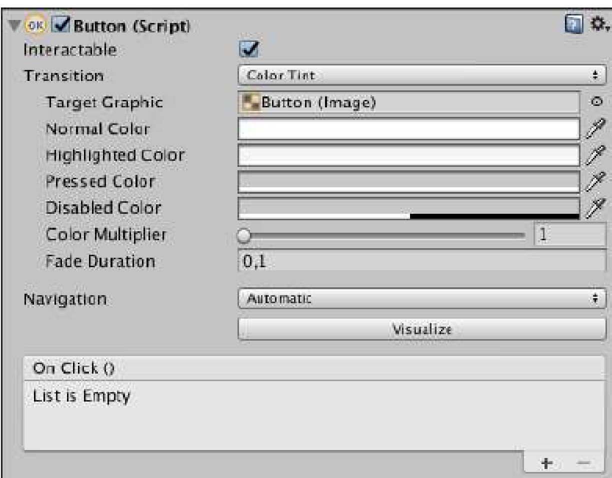


Fig. 9.35

Interactable: Esto determina si este componente aceptará entrada. Cuando se establece en interacción falsa, se deshabilita y el estado de transición se establece en el estado deshabilitado.

Transition: Dentro de un componente seleccionable hay varias opciones de transición dependiendo de en qué estado se encuentre actualmente la selección. Los diferentes estados son: normal, resaltado, presionado y deshabilitado.

- **None:** Esta opción es para que el botón no tenga ningún efecto de estado.
- **Color Tint:** Cambia el color del botón dependiendo del estado en que se encuentre. Es posible seleccionar el color para cada estado individual. También es posible configurar la duración del fundido entre los diferentes estados. Cuanto mayor sea el número, más lento será el desvanecimiento entre los colores.
  - **Target Graphic:** es la imagen o gráfico que se utiliza para el componente de interacción.
  - **Normal Color:** el color normal del botón.
  - **Highlighted Color:** el color del botón resaltado;
  - **Pressed Color:** el color de botón cuando se presiona.
  - **Disabled Color:** el color del botón cuando se desabilita.
  - **Color Multiplier:** multiplica el color del tinte para cada transición por su valor.
  - **Fade Duration:** el tiempo en segundos que tarda en desvanecerse de un estado a otro.
- **Swite Swap:** permite la visualización de diferentes sprites dependiendo de en qué estado se encuentre el botón actualmente, muy útil si queremos botones personalizados.

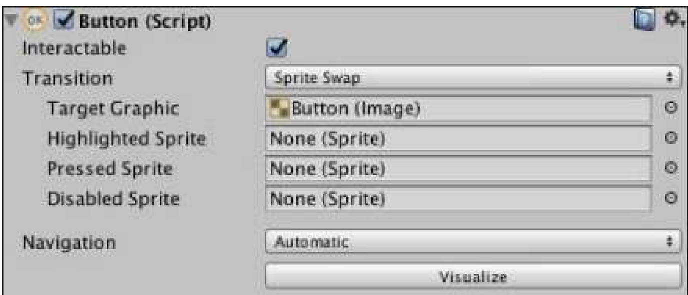


Fig. 9.36

  - **Target Graphic**: es el sprite normal para usar
  - **Highlighted Sprite :** Sprite resaltado para usar cuando el control está resaltado
  - **Highlighted Color:** el color del botón resaltado;
  - **Pressed Sprite:** Sprite presionado para usar cuando se presiona el botón.
  - **Disabled Sprite:** Sprite desactivado para usar cuando el botón está deshabilitado.
- **Animation:** permite que ocurran animaciones dependiendo del estado del botón, debe existir un componente animador (que todavía no hemos visto) para usar la transición de animación.

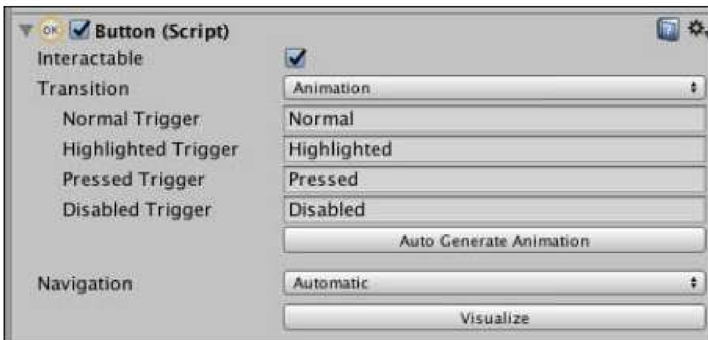


Fig. 9.37

Navigation: Las opciones de navegación se refieren a cómo se controlará la navegación de los elementos de la interfaz de usuario en el modo de reproducción.

- **Horizontally:** navegación Horizontal
- **Vertical :** navegación vertical
- **Automatic:** automática
- **Explicit:** explícito, se puede especificar explícitamente a dónde se dirige el control para diferentes teclas de flecha.
- **Visualize:** seleccionar visualizar nos da una representación visual de la navegación que ha configurado en la ventana de escena.

8. Slider

El control deslizante nos permite seleccionar un valor numérico de un rango predeterminado arrastrando el ratón. Los ejemplos conocidos incluyen ajustes de dificultad en juegos y configuraciones de brillo en editores de imágenes, pero también podemos utilizarlos como una barra de energía.

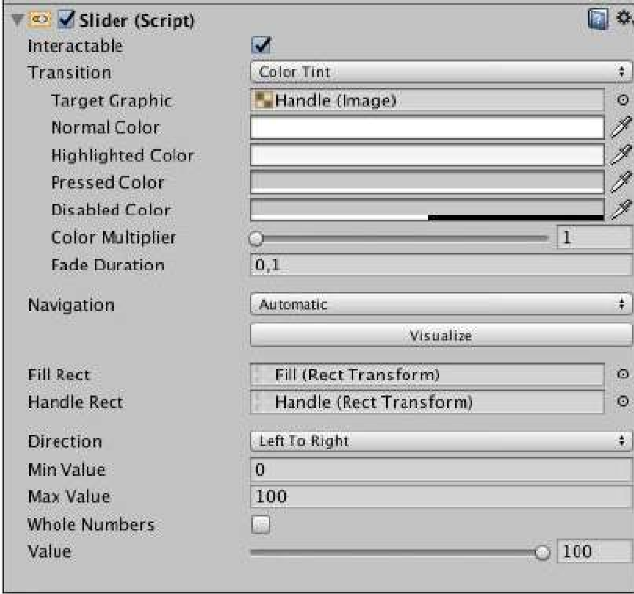


Fig. 9.38

Interactable: esto determina si este componente aceptará entrada. Cuando se establece en interacción falsa, se deshabilita y el estado de transición se establece en el estado deshabilitado.

Transition: propiedades de transición que determinan la forma en que el control responde visualmente a las acciones del usuario. Estos parámetros son iguales a los del elemento Button.

Navigation: propiedades que determinan la secuencia de controles.

Fill Rect: el gráfico utilizado para el área de relleno del control.

Handle Rect: el gráfico utilizado para la parte deslizante del "handle" del control.

Direction: dirección en la que el valor del control deslizante aumentará cuando se arrastre el control. Las opciones son de izquierda a derecha, de derecha a izquierda, de abajo a arriba y de arriba a abajo.

Min Value: el valor del control deslizante cuando el control está en su extremo más bajo (determinado por la propiedad Dirección).

Max Value: el valor del control deslizante cuando el control está en su extremo superior (determinado por la propiedad Dirección).

Whole Numbers: restringe los valores a números enteros cuando esta activo.

Value: es el valor numérico actual del control deslizante. Si el valor se establece en el inspector, se usará como el valor inicial, pero esto cambiará en el tiempo de ejecución cuando cambie el valor.

Proyecto del Capítulo

Estos elementos del canvas son ,de momento, suficientes para realizar las siguientes actividades. En el capitulo anterior (Capitulo 8) creamos un sistema para disparar en un **FPSController** y se explicó como instanciar Decals, a continuación vamos a crear un nuevo proyecto 3D con el nombre de **Proyecto_Capitulo_9**. En este nuevo proyecto vamos a importar el material que necesitamos para realizar las siguientes actividades, para ello accedemos al material que acompaña la obra y buscamos el paquete con nombre **Proyecto_Capitulo_9.unitypackage** como te muestro a continuación.

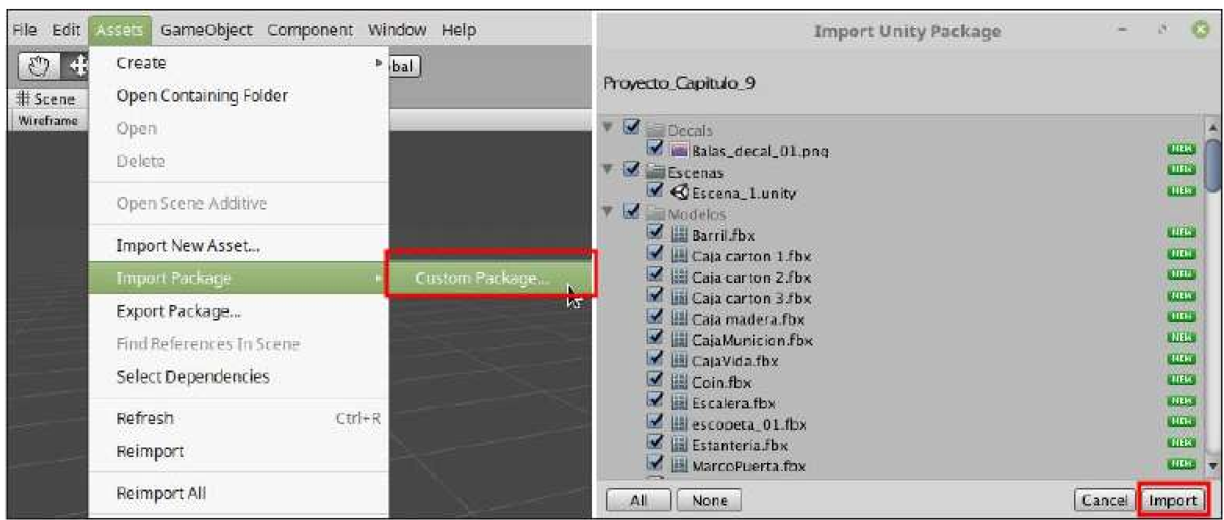


Fig. 9.39

Una vez tengas descargado el paquete veras que dispones de una serie de carpetas dentro de la ventana **Project**, pero no aparece nada en la ventana Escena, eso es debido a que debemos cargar la escena que he preparado y que encontraras accediendo a la carpeta Escenas y haciendo doble clic encima del archivo **Escena_1**.

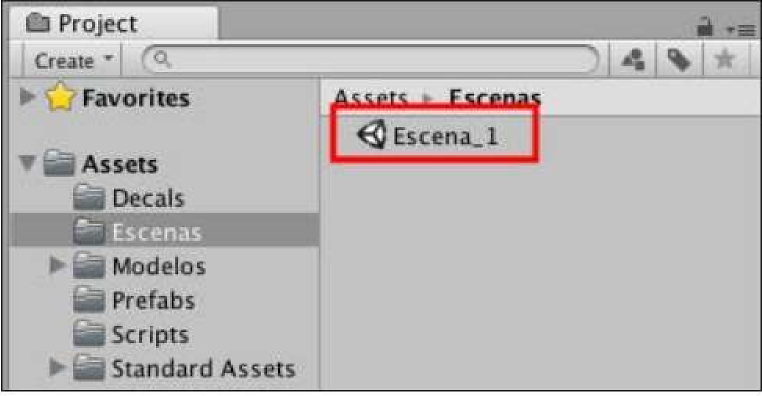


Fig. 9.40

Al cargar la escena veras que es el mismo escenario que el del capitulo anterior (Capítulo 8), pero con algunos elementos nuevos, que quiero comentar antes de empezar a crear código. Es posible que al cargar el escenario se visualice en modo **Wireframe**, si ese es tu caso puedes cambiar el modo de visualización accediendo al menú de debajo de la ventana **Scene** y seleccionar la opción **Shaded**.

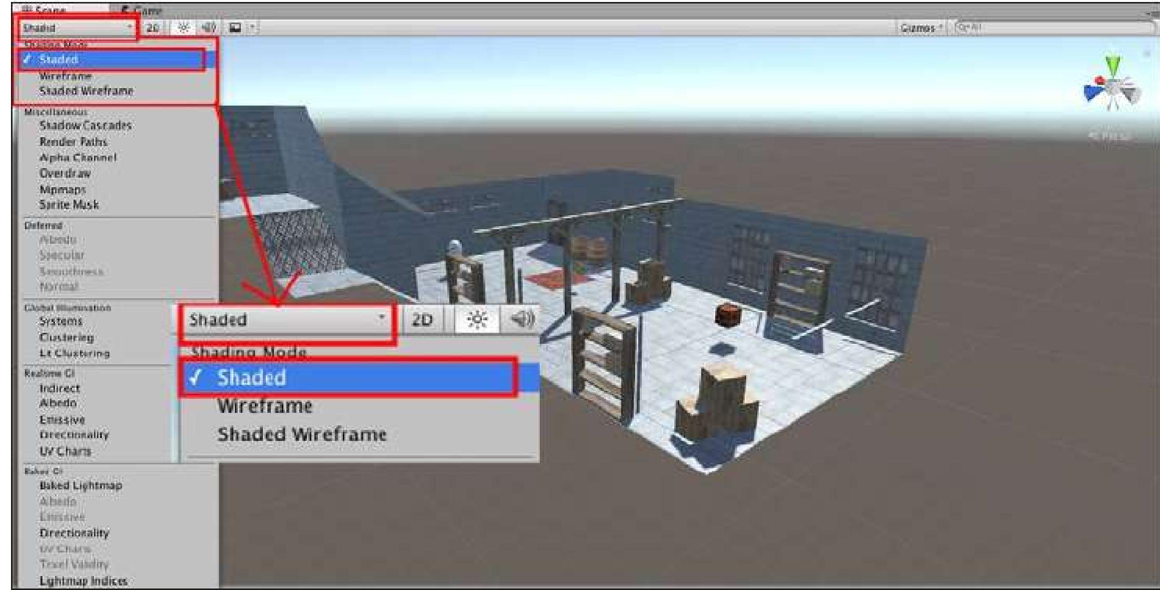


Fig. 9.41

Si ejecutas la escena podrás ver algunos objetos nuevos en la escena que vamos a utilizar para crear la interfaz del Player. En la siguiente imagen veras que tenemos una porción de suelo con una textura de lava, que vamos a utilizar para que nos reste vida, después tenemos una moneda al que tiene un script que le da movimiento que utilizaremos como elemento para recolectar (de momento solo tenemos una moneda). Si miramos al fondo también veremos una caja con un corazón, que utilizaremos para curar a nuestro Player, del daño que produzca la lava.

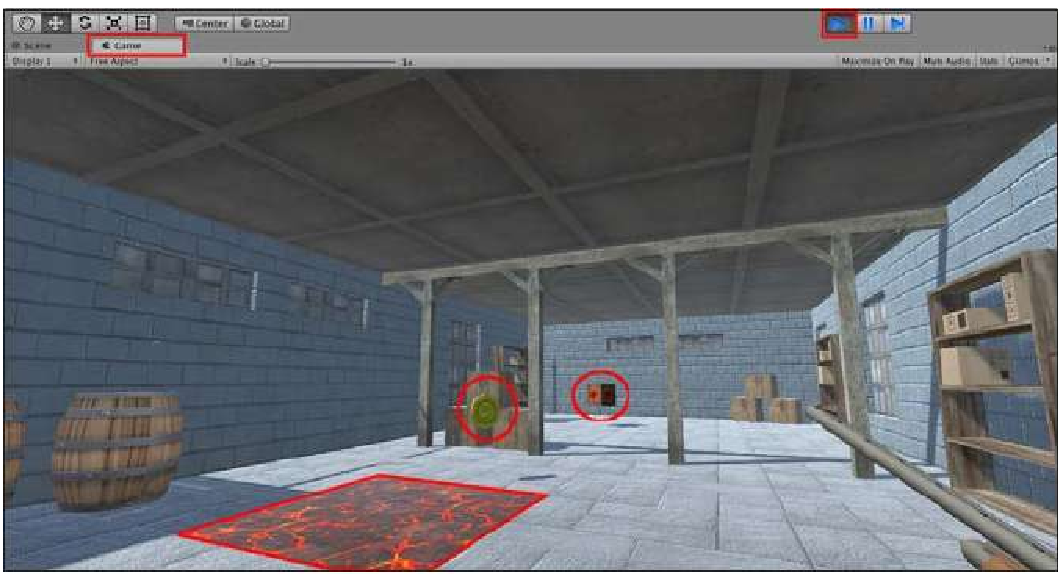


Fig. 9.42

También vamos a tener otro objeto (caja de municiones), este objeto lo utilizaremos para cargar las municiones de nuestro player, actualmente tenemos infinitas balas, pero en este proyecto vamos a crear un sistema que no nos permita disparar cuando nuestra munición llegue a 0.

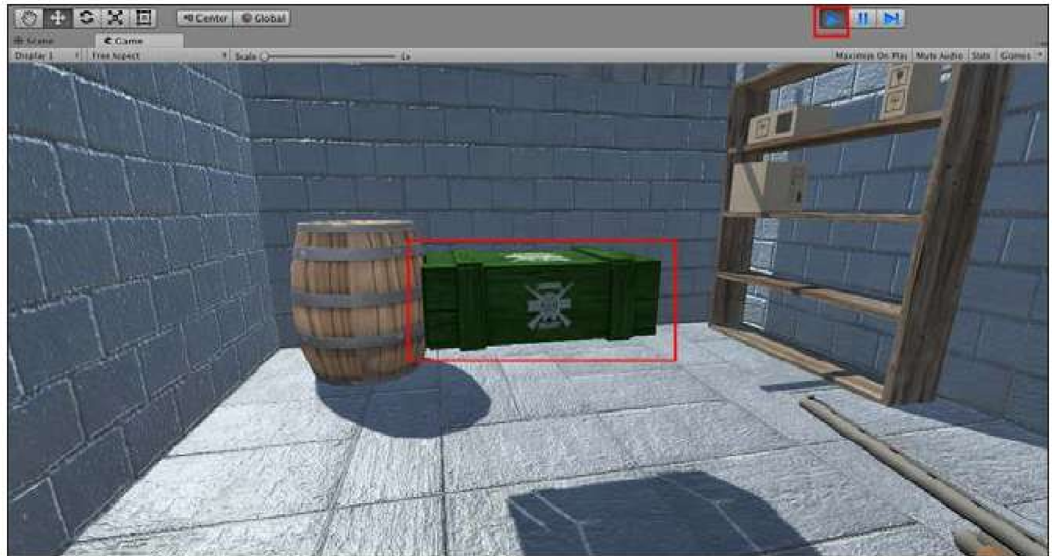


Fig. 9.43

9. Creación de una mirilla

Primero de todo vamos a crear nuestra mirilla para poder apuntar bien con nuestro FPSController, si miramos en la ventana **Project** dentro de la carpeta Decals disponemos de varias imágenes, entre ella una llamada mirilla_pistola.

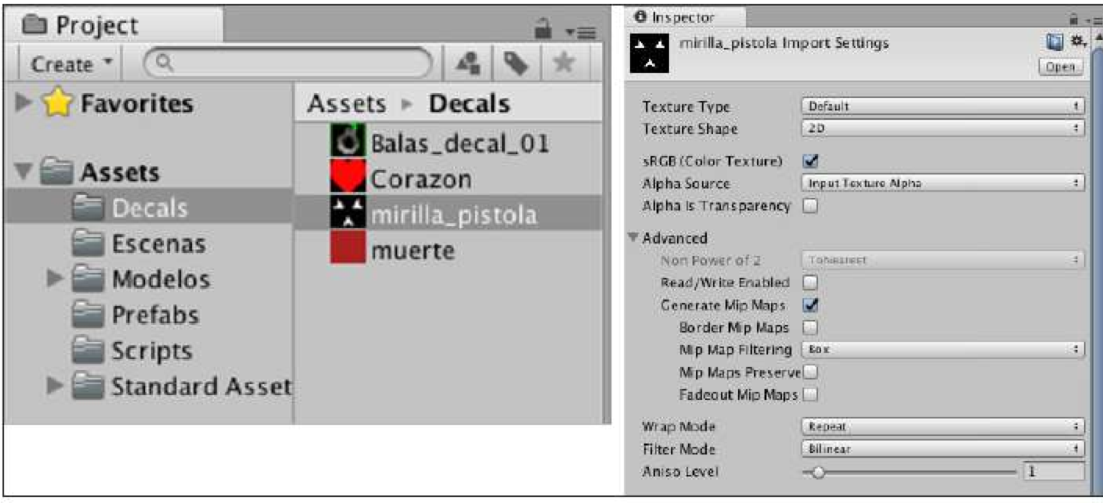


Fig. 9.44

Todas estas imágenes menos la de **Balas_ decal_01** tenemos que convertirlas en Sprites para poder utilizarlas correctamente como imágenes para la **UI** (User Interface). En la imagen anterior izquierda al seleccionar la mirilla_pistola se muestran las características de esta textura en la ventana inspector. Desde esta ventana podemos convertir las imágenes a **Sprites**. Primero selecciona la imagen que deseas convertir a Sprite por ejemplo mirilla_pistola y luego en la ventana Inspector nos dirigimos al primer parámetro **Texture Type** y seleccionamos la opción Sprite (2D and UI).

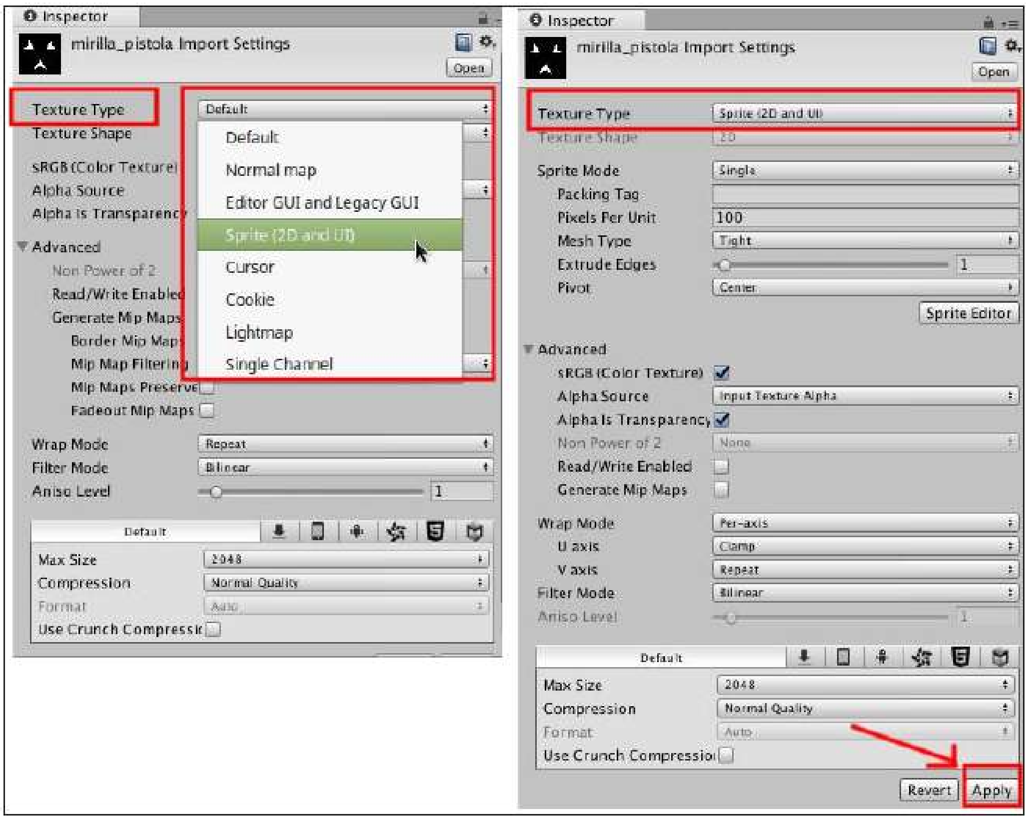


Fig. 9.45

En la imagen anterior no te lo he mostrado pero cuando seleccionamos una imagen como **mirilla_pistola** en las características de la ventana Inspector, abajo del todo tenemos una representación de la imagen que en un principio tenia fondo negro y al convertir la imagen a Sprite ,el fondo negro desaparece y se vuelve transparente. Es transparente cuando el fondo se ve como un tablero de ajedrez como te muestro en la siguiente imagen.

Fig. 9.46

Tengo que decir que estas imágenes suelo crearlas con formato **png** que es un formato que guarda el canal alfa de la textura y si es posible siempre recomiendo que estas imágenes sean potencias de dos. Ahora vamos a proceder a convertir las otras dos imágenes a **Sprites**, en Unity podemos seleccionar las dos imágenes y convertirlas a la vez. Para seleccionar las dos imágenes primero hacemos clic en la imagen **Corazon** y manteniendo pulsada la tecla Control seccionamos la imagen **muerte. Con las dos imagenes seleccionadas nos dirigimos a la ventana propiedades y en el parámetro Texture Type seleccionamos la opción Sprite (2D and UI) y pulsamos el botón Apply al final de las características.**

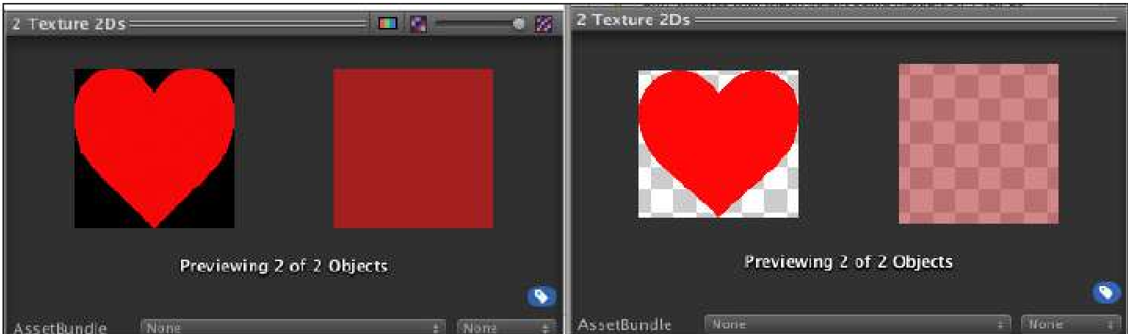


Fig. 9.47

Ahora que tenemos convertida la imagen mirilla convertida a Sprite vamos a crear un canvas para poder poner nuestra mirilla. Para ello primero accedemos al menú principal **GameObject > UI > Canvas** como te muestro en la siguiente imagen.

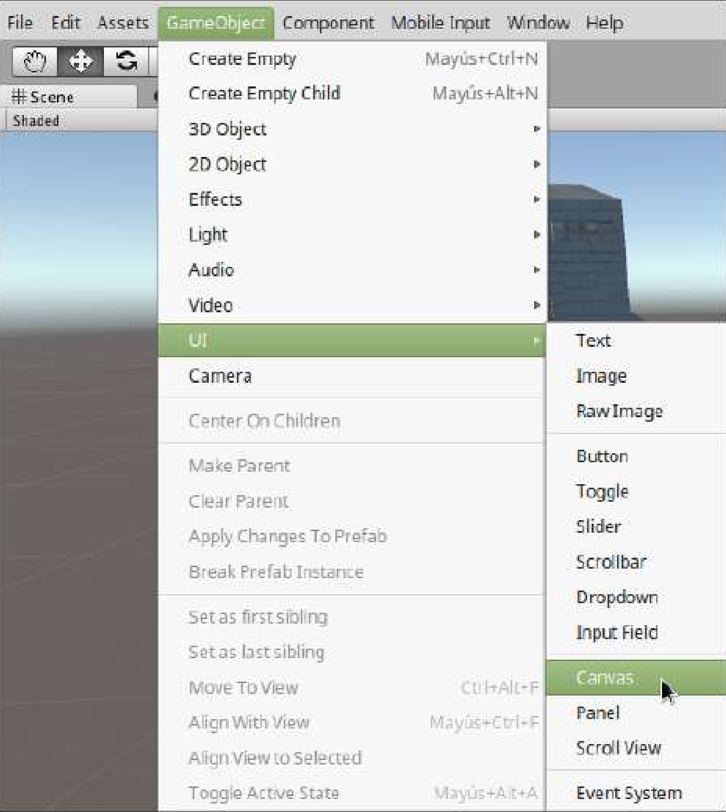


Fig. 9.48

En la ventana **Hierarchy** se nos creara el objeto Canvas y el objeto EventSystem. Seleccionamos el objeto Canvas y le vamos a cambiar el nombre desde la ventana inspector por el nombre HUB.

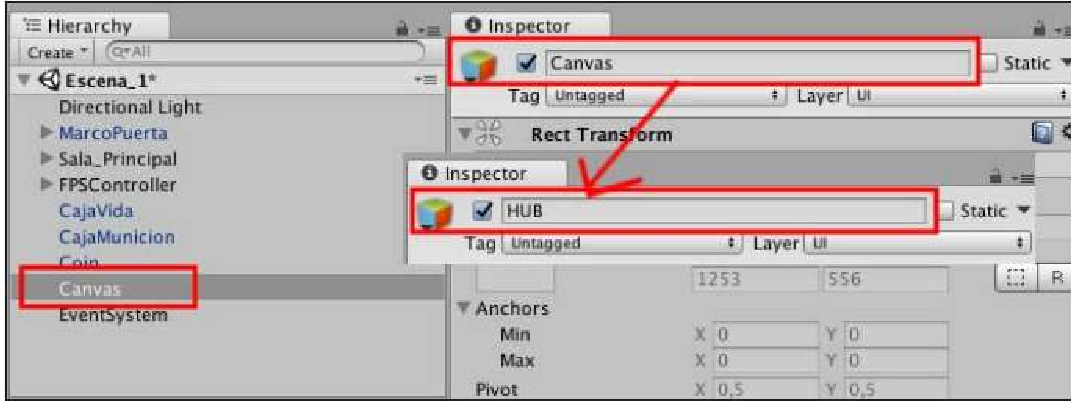


Fig. 9.49

Otro aspecto que vamos a cambiar del canvas que ahora se llama HUB es en la ventana Inspector en el apartado Canvas Scaler cambiamos la opción **Constant Pixel Size** por el de **Scale With Screen Size**, con una resolución de 800 x 600.

Ahora que tenemos el **HUB** (canvas) podemos añadir un elemento imagen para poner la textura mirilla_pistola. Para poner una imagen dentro del canvas podemos hacerlo desde la misma ventana **Hierarchy desde el botón Create > UI > Image**

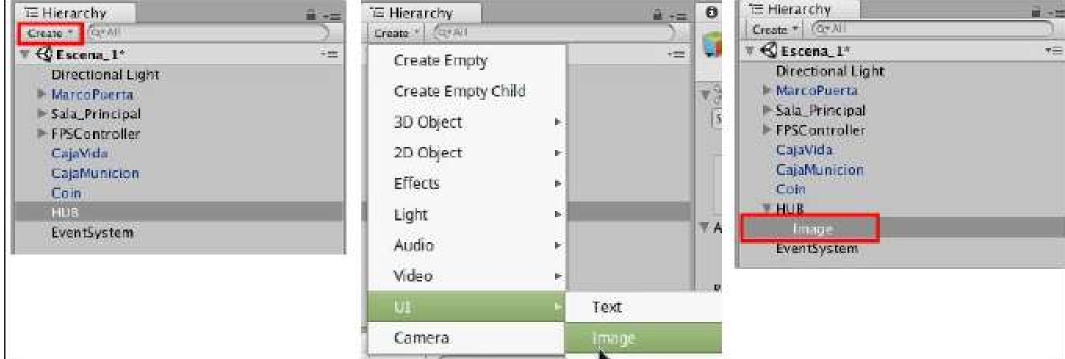


Fig. 9.50

Si has seguido las indicaciones que te he descrito deberías de poder ver un cuadro blanco en medio de la ventana **Game**. En la siguientes dos imágenes te muestro la ventana **Scene** en modo 2d para poder mostrar todo el canvas con la imagen en su interior y la ventana **Game** para que puedas ver como se muestra la imagen.

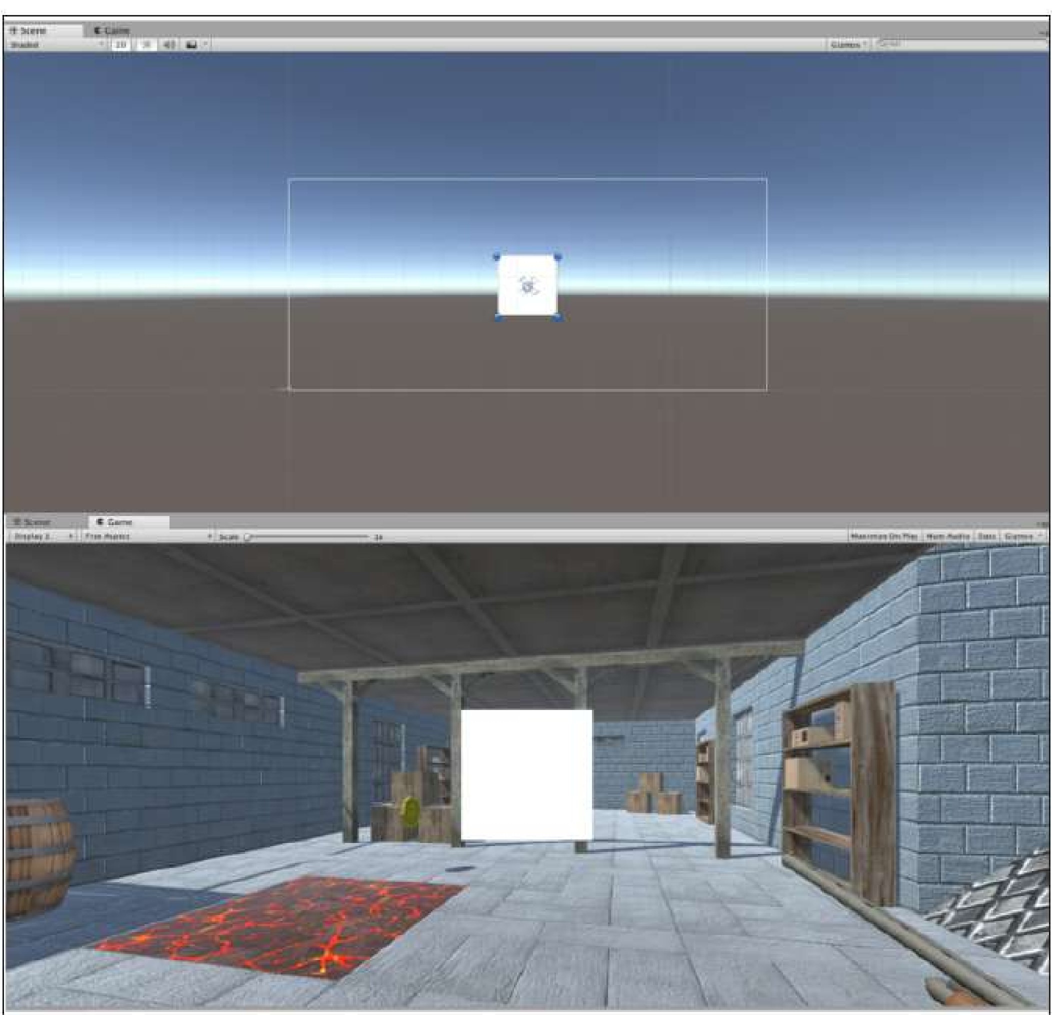


Fig. 9.51

Ahora seleccionamos la imagen y nos dirigimos a la ventana Inspector en las propiedades de **Image (Script)**. En el parámetro **Source Image** seleccionamos el Sprite mirilla_pistola accediendo desde el símbolo con forma de punto como te muestro en la siguiente imagen.

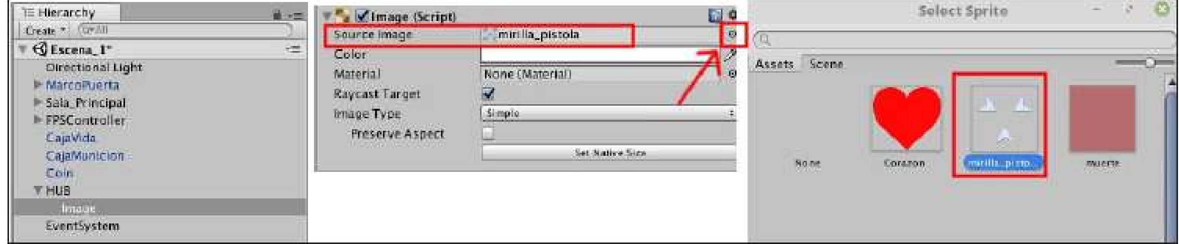


Fig. 9.52

Ahora si nos fijamos tenemos una mirilla en vez de un cuadro blanco, pero es demasiado grande para su función. Para cambiar el tamaño podemos utilizar las opciones del componente **Rect Transform** de la imagen. Primero te recomiendo que hagas clic encima del cuadro de anclas y pongas el anclaje en el centro como te muestro a continuación.

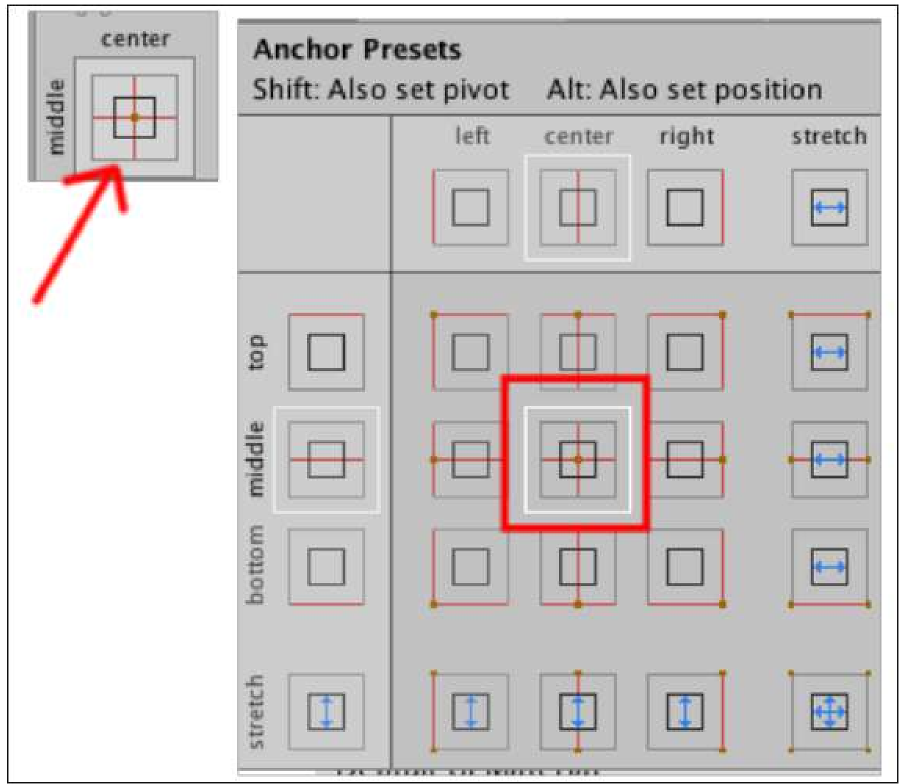


Fig. 9.53

Luego en las opciones del componente **Rect Transform** nos aseguramos de que el parámetro posición tiene el valor 0 en todos sus ejes, de este modo la mirilla estará perfectamente en el centro de la pantalla. Para escalar la mirilla vamos a utilizar los parámetros Width y Height a los que les daremos un valor de 32.

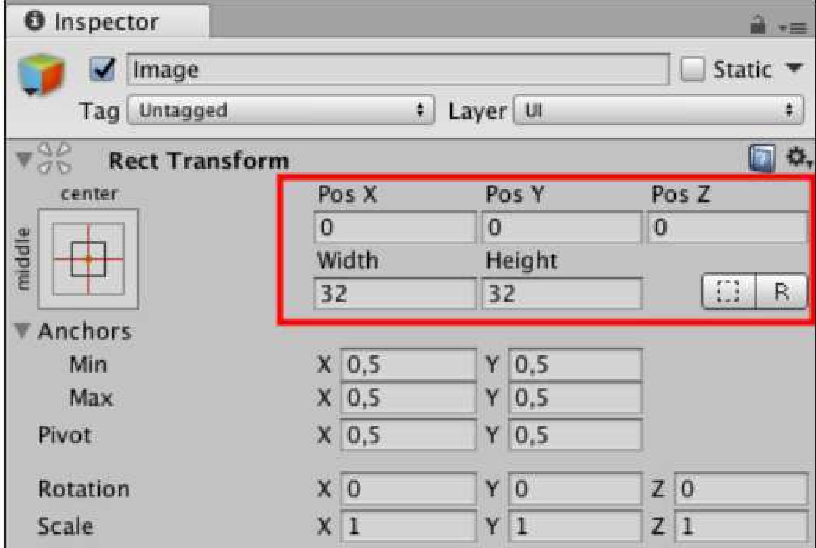


Fig. 9.54

Ahora si ejecutamos la escena podremos disfrutar de una mirilla que nos va a permitir apuntar mejor a los objetivos. Es posible que encuentres que la mirilla en color blanco no se acabe de percibir del todo bien, en ese caso si lo deseas (es opcional) podemos cambiar el color de la mirilla accediendo a la ventana Inspector en las propiedades de **Image (Script) e**n el parámetro color si haces clic encima se te abrirá un selector de color en donde puedes seleccionar el color que quieras. Una vez tengas el color seleccionado la mirilla tomara ese color.

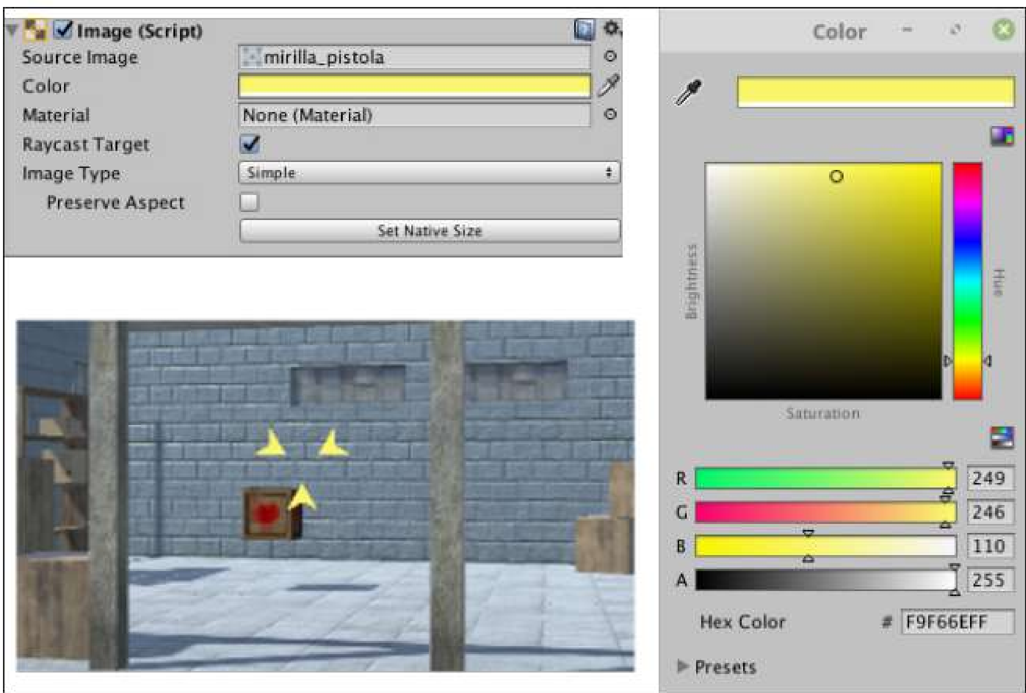


Fig. 9.55

Ahora para finalizar solamente debemos cambiarle el nombre a la Imagen por Mirilla. Lo podemos hacer desde la ventana Inspector o desde la ventana Hierarchy. También recuerda en ir guardando la escena y el proyecto conforme vayamos avanzando.

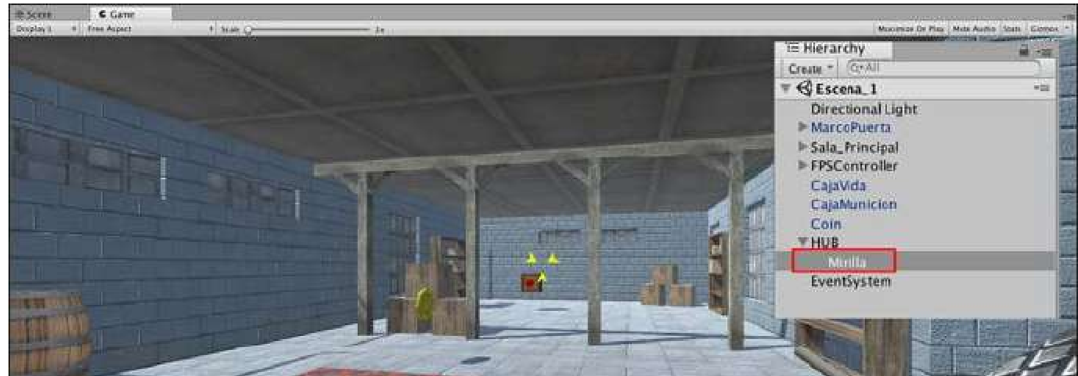


Fig. 9.56

10. Creación de un contador de coins

Ahora que disponemos de una mirilla vamos a crear mediante un objeto UI de tipo Texto un contador que nos de información de las monedas que vamos recopilando. Antes de empezar debemos tener claro que es lo que queremos.

1. Cuando nuestro FPSController toque una moneda esta desaparezca
2. Guarde la información de cuantas veces tocamos una moneda.
3. Un elemento en el canvas que nos muestre la cantidad de monedas.

Primero vamos a crear el objeto de tipo Texto accediendo a la barra de menús principal **GameObject > UI > Text**. Automáticamente veremos como se añade un elemento de tipo texto debajo de Mirilla dentro del HUB (Canvas). También veremos que en la ventana **Game** se muestra un texto con un color "negro".

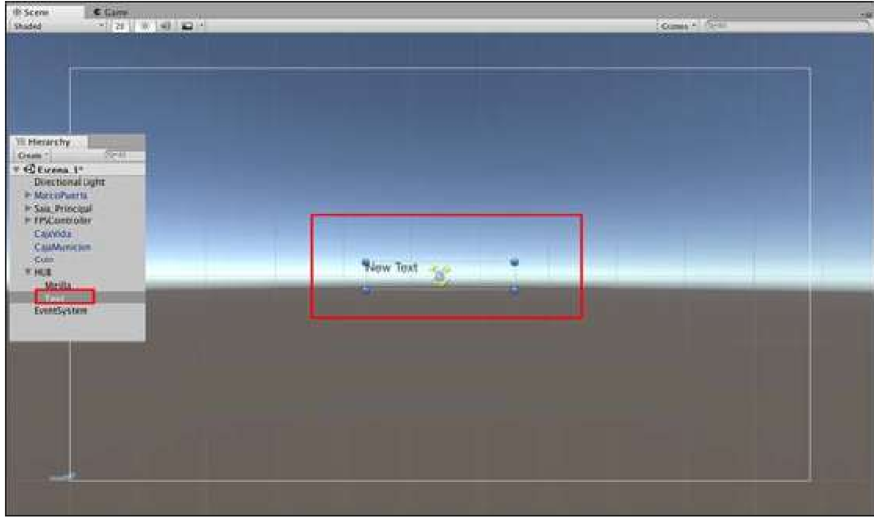


Fig. 9.57

Ahora vamos a posicionar lo en la parte superior derecha, para ello nos dirigimos a la ventana Inspector y en el cuadro de anclajes hacemos doble clic ara que se despliegue y seleccionamos precisamente el recuadro superior izquierdo como te muestro en la siguiente imagen.

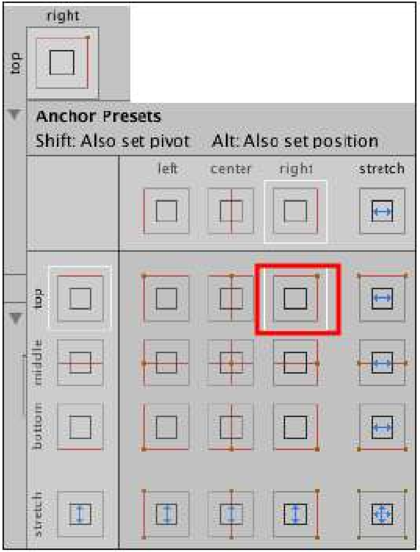


Fig. 9.58

Una vez que hemos realizado la acción anterior vamos a cambiar los siguientes parámetros del **Rect Transform** de nuestro texto. En el apartado de Posiciones en **X= -80, Y=-30, Z=0.** En **Width** le daremos el valor de 150 y en **Height** el valor de 30.

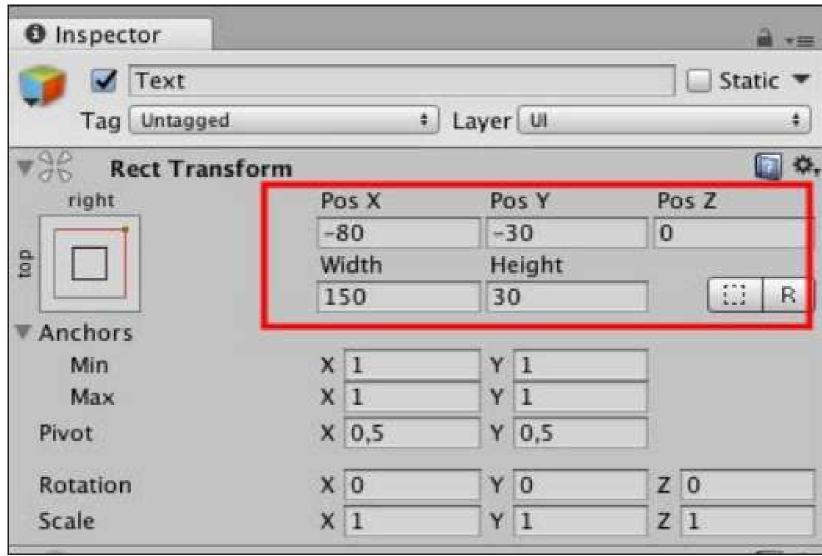


Fig. 9.59

Ahora vamos a cambiar el aspecto del texto desde el componente **Text(Script)** que se encuentra un poco más abajo del **Rect Transform**.

En la caja **Text** escribiremos **COINS:** en mayúscula. La fuente la dejaremos en **Arial** porque no disponemos de otra. El **Font Style** escogeremos **Bold**, en **Font Size** (Tamaño) le daremos un valor de 24. Por ultimo le pondremos un color Amarillo o el que a ti te guste.

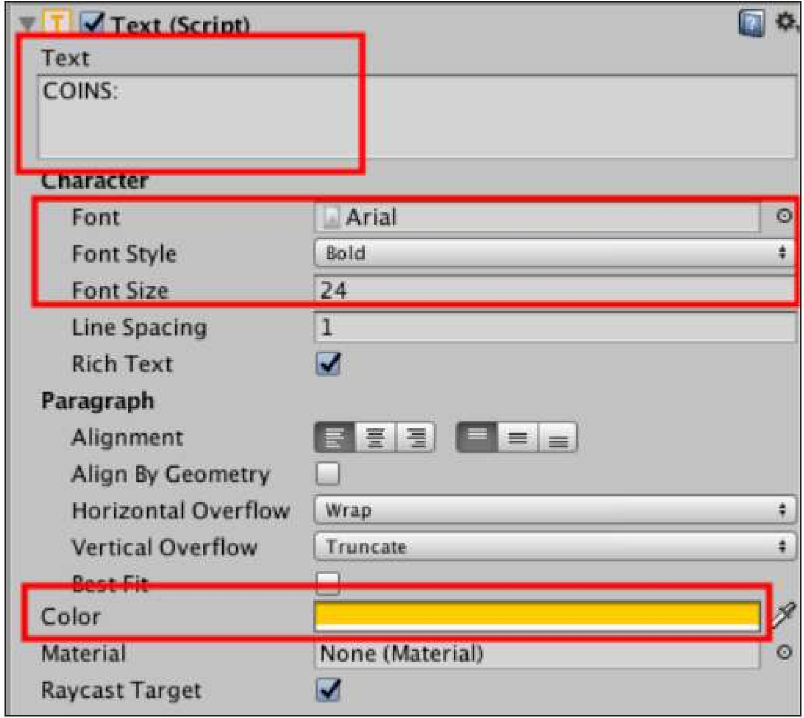


Fig. 9.60

Si miras en la ventana **Game** deberías poder ver un texto parecido al que te muestro a continuación.

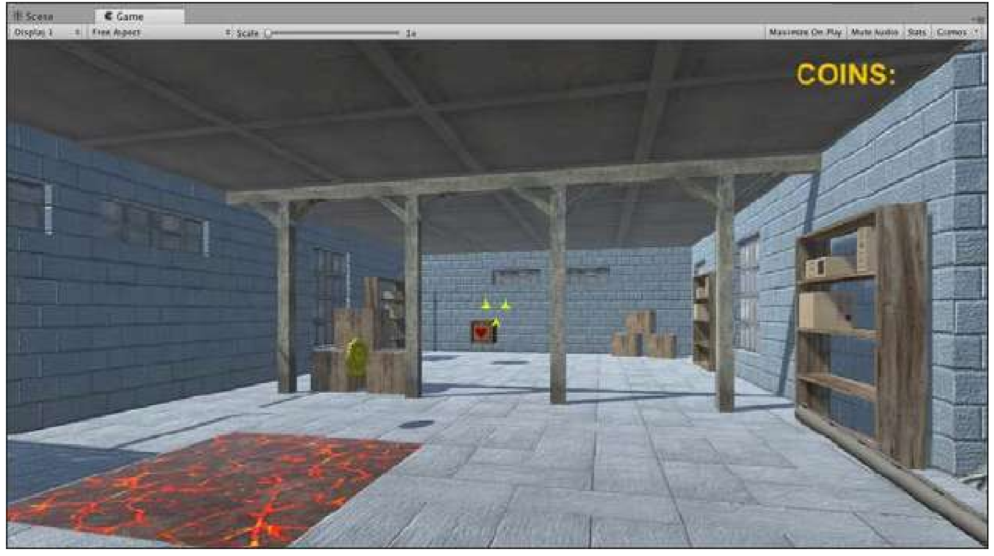


Fig. 9.61

Ahora vamos a ver como creamos un sistema para recopilar las monedas y que se muestre en el texto. Para ello vamos a crear un Script, nos dirigimos a la ventana **Project** seleccionamos la carpeta Scripts le damos al botón **Create > C# Scripts** y le ponemos como nombre **ControlPlayer.cs**. Como podrás adivinar por el nombre que le hemos puesto al Script este va dirigido al player. Asi que seleccionamos el scritpt **ControlPlayer** y lo arrastramos encima del **FPSController** de la ventana **Hierarchy**.

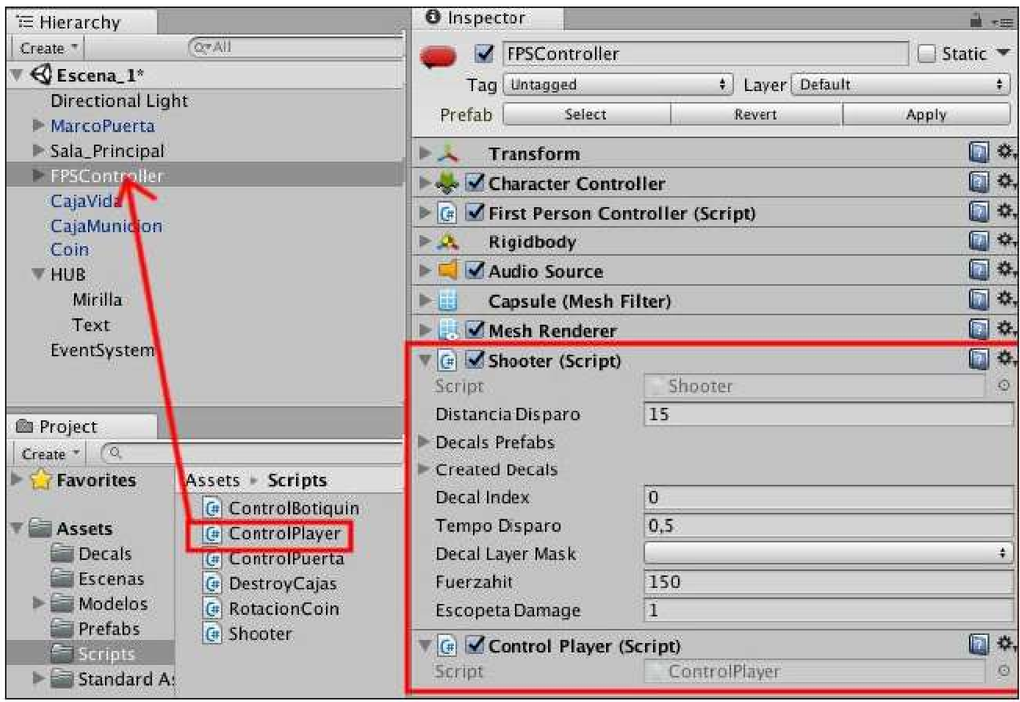


Fig. 9.62

Como verás nuestro **FPSController** ya tiene otro script añadido con el nombre **Shooter**, este script es el que aprendimos a crear en el capitulo 8 para poder crear un sistema **Raycast** que pudiera instancia decals, durante el capitulo lo mejoraremos, pero ahora vamos a centrarnos en el **ControPlayer.**

Antes de seguir con el script vamos a cambiar el nombre de **Text** por el de **ContradorCoins** que encontraremos dentro de la ventana **Hierarchy** debajo de **Mirilla,** como te muestro en la siguiente imagen.

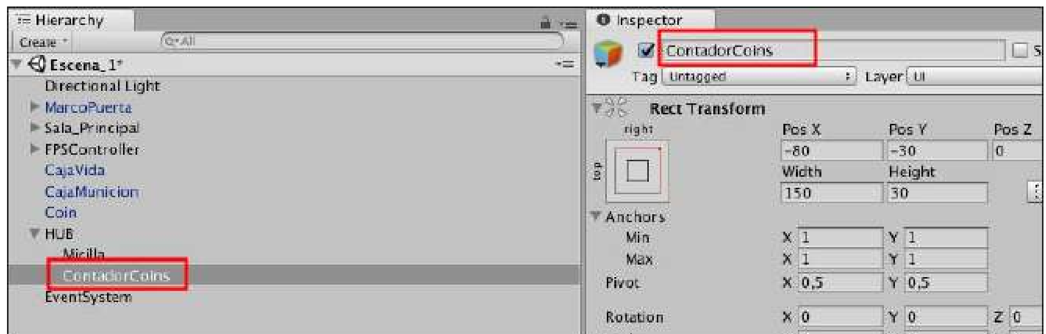


Fig. 9.63

El objetivo del script es que detectemos el trigger de la moneda, que este al ser detectado se desactive y sume una unidad. Otra cosa importante es que para detectar el trigger correcto es decir que no se confunda con otro objeto y lo desactive la moneda (Coin) tiene que tener una tag con el nombre Item. Si has descargado el paquete que acompaña el libro no debes preocuparte porque ya lleva la etiqueta.

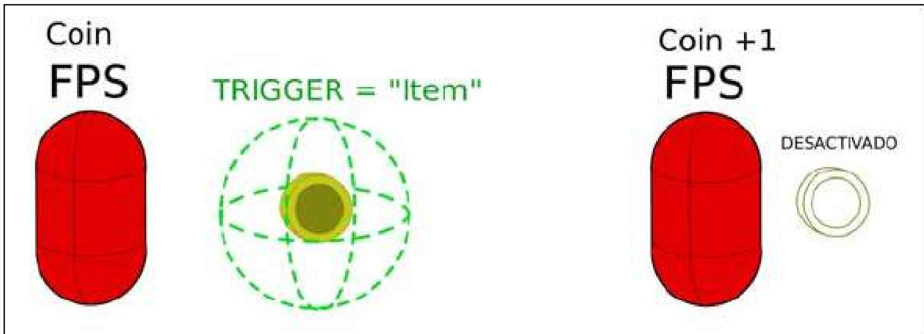


Fig. 9.64

Una vez aclarado lo que queremos hacer, hacemos doble clic encima del script **ControlPlayer** y se nos abrirá Monodevelop para empezar a escribir el código.

Script:ControlPlayer.cs

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.UI;
public class ControlPlayer : MonoBehaviour
{
       private int contador;
   public Text contadorMonedas;

   void Awake ()
   {
      this.contador = 0;
   }

   void OnTriggerEnter(Collider otro)
   {
      if (otro.gameObject.CompareTag ("Item"))
      {
       otro.gameObject.SetActive (false);
       this.contador += 1;
       this.ContadorTexto ();
      }
   }
```

```
    public void ContadorTexto()
    {
        this.contadorMonedas.text = "COINS: " + this.contador.ToString ();
    }
}
```

En este script primero de todo debemos añadir una de las librerías de Unity escribiendo en la parte de arriba using UnityEngine.UI.
Primero de todo vamos a crear dos variables, la primera es de tipo int privada con el nombre de contador, que se encargará de almacenar el numero de veces que recogemos monedas. La otra variable es pública porque quiero que me aparezca en la ventana Inspector y es de tipo Text con el nombre de contadorMonedas,esta variable nos va a permitir acceder al texto que tenemos en el HUD (Canvas).
Dentro de la función Awake() diremos que la variable contador empieza a contar desde 0 por eso la igualamos a 0.
Ahora vamos a crear una función OnTriggerEnter con un el argumento de collider con nombre otro para hacer referencia de cuando detectamos otro. Dentro de la función ponemos la condición de que el otro gameObject que detectemos si tiene una etiqueta tag con nombre "Item" (Hay que asegurarse de que se escribe exactamente igual), sucede que este objeto Item debe desactivarse, luego la variable contador sumará una unidad y en este caso he creado una función con nombre ContradorTexto() que llamo dentro de la función OnTriggerEnter.

Esta función ContradorTexto la he creado debajo y lo que hace es acceder a la variable contadorMonedas que es de tipo texto y accedo a su atributo text que es la cajita donde hemos escrito antes COIN: y le digo entre comillas que me escriba "COIN: " y luego lo anido con el valor de la variable contador. Como contador es un valor entero utilizo el método ToString() para que lo convierta en texto.
Guarda el script en Monodevelop para que tenga efecto en Unity.

Antes de probar la escena debemos arrastrar el objeto **ContadorCoins** que encontraremos en el HUB (Canvas) en la ventana **Hierarchy**, dentro del Contador Monedas que hemos creado en el Script Control Player. Te muestro como hacerlo en la siguiente imagen.

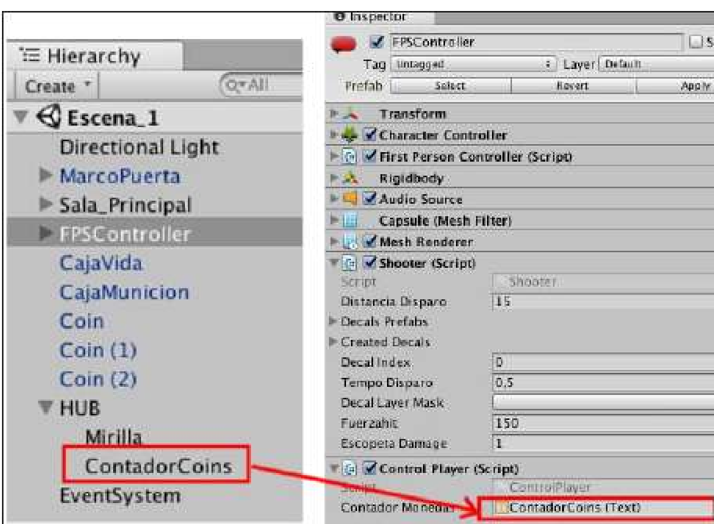


Fig. 9.65

Bien si todo esta escrito correctamente podemos realizar la prueba ejecutando la escena. Recuerda que solo podrás recopilar una moneda. En el caso de que quieras hacer la prueba con más monedas, solamente tienes que ir a la ventana **Hierarchy** seleccionar el objeto con nombre **Coin** realizar un clic encima con el botón derecho del ratón y selecciona la opción **duplicate**. Recuerda que cuando creas un duplicado de este modo se crea en la misma posición que el original por eso mismo debes acceder a la ventana **Scene** y mover la nueva copia y colocarla donde quieras. A continuación yo he creado dos monedas a partir de la original y he cogido una para ver si el texto me contaba la adquisición.

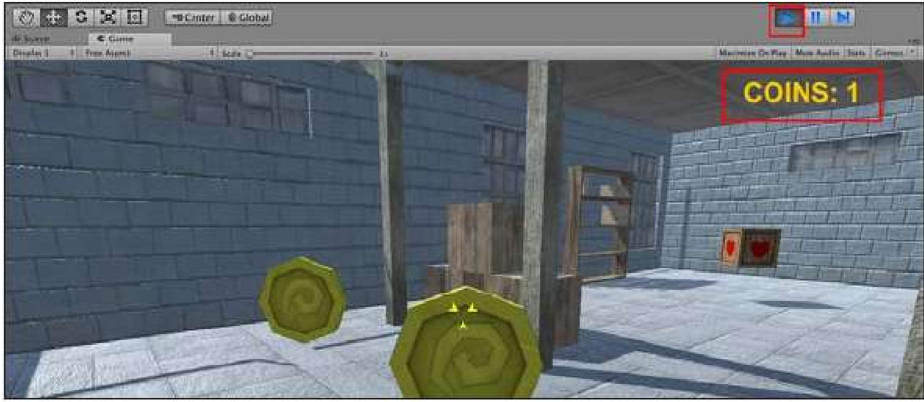


Fig. 9.66

11. Creación de una barra de vida

Ahora vamos a crear una barra de vida de una forma muy sencilla utilizando un Slider. Primero vamos a crear el objeto **Slider** accediendo a la barra de menús principal **GameObject > UI > Slider**. Automáticamente veremos como se añade un elemento de tipo Slider debajo de **ContadorCoins** dentro del HUB (Canvas). También veremos que en la ventana **Game** se muestra un circulo blanco con una barra en horizontal de color gris. En mi caso en concreto se me muestro en la parte inferior izquierda de la ventana Escena .

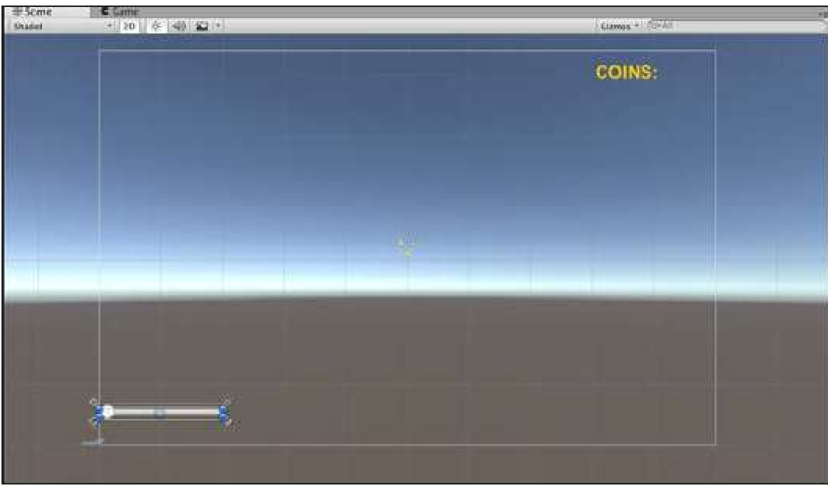


Fig. 9.67

El primer paso es colocar el Slider en la parte superior izquierda, para ello vamos a su componente **RectTransform** y accedemos al cuadro de los anclajes para seleccionar el cuadro que está en la parte superior izquierda. Luego ponemos la posición en **X=105, Y=-25, Z=0.** Por último dejamos por defecto los valores **Widht = 160** y **Height = 20**.

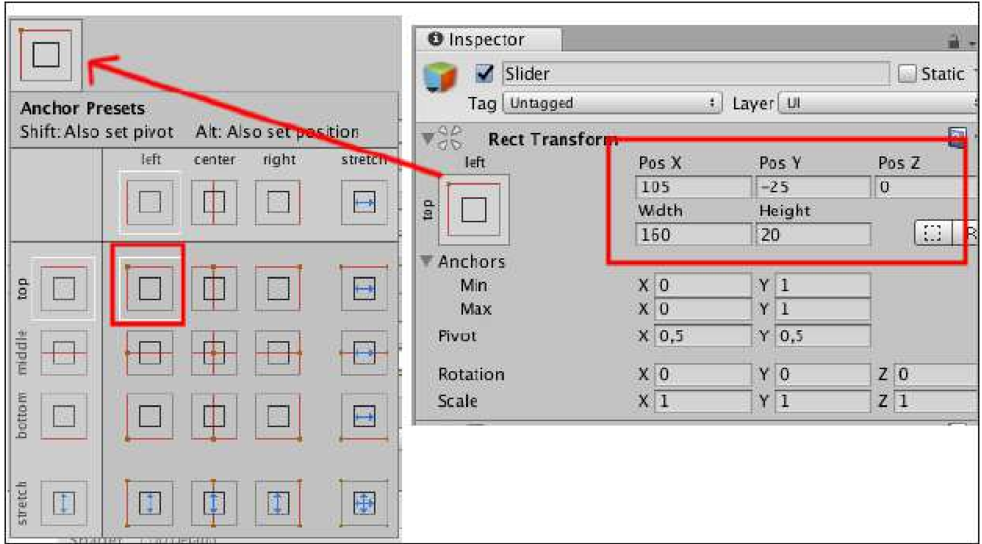


Fig. 9.68

Ahora deberíamos de tener la barra en la parte superior izquierda mas o menos centrada, si lo deseas puedes posicionar el **Slider** manualmente en el lugar que más te guste. Si la posicionas igual que te indico debería verse como te muestro en la siguiente imagen.

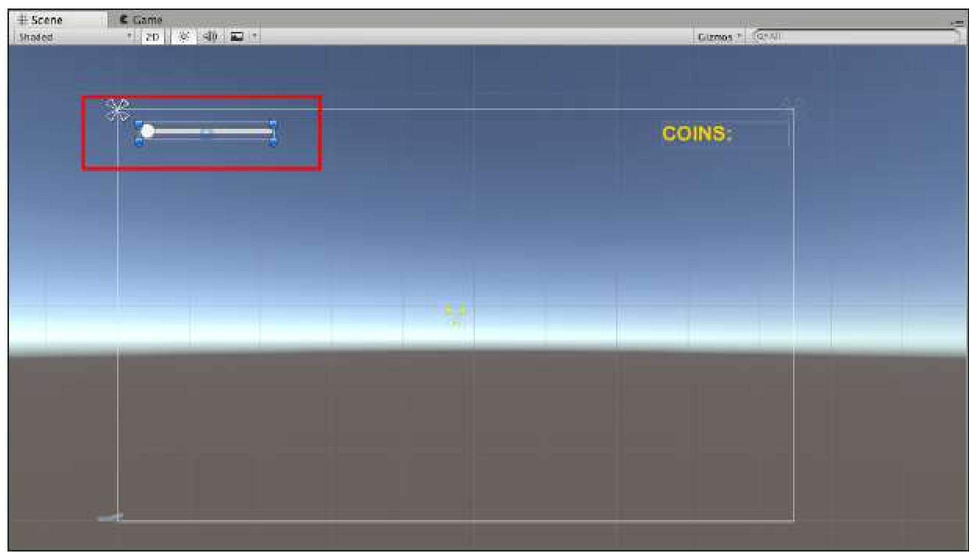


Fig. 9.69

Dentro de la ventana Hierarchy el Slider dispone de subobjetos que nos ayudarán a mejorar el aspecto de este. El primero que encontramos es el Background y es la ba-

rra que se muestra de fondo en el Slider. Seleccionamos el Background, accedemos a su componente Image y le cambiamos el color a rojo.

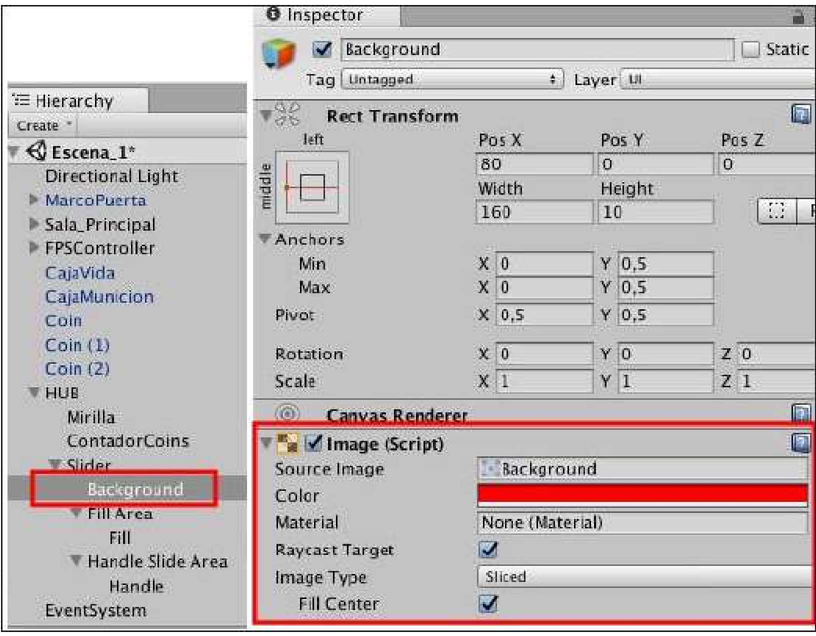


Fig. 9.70

El segundo que encontramos es **Fill Area** y es la barra que se muestra delante del fondo **es decir seria el área de relleno que nos permitirá ver cuando tenemos la vida al máximo**. Seleccionamos el elemento **Fill** que es hijo de **Fill Area**, accedemos a su componente Image y le cambiamos el color a verde.

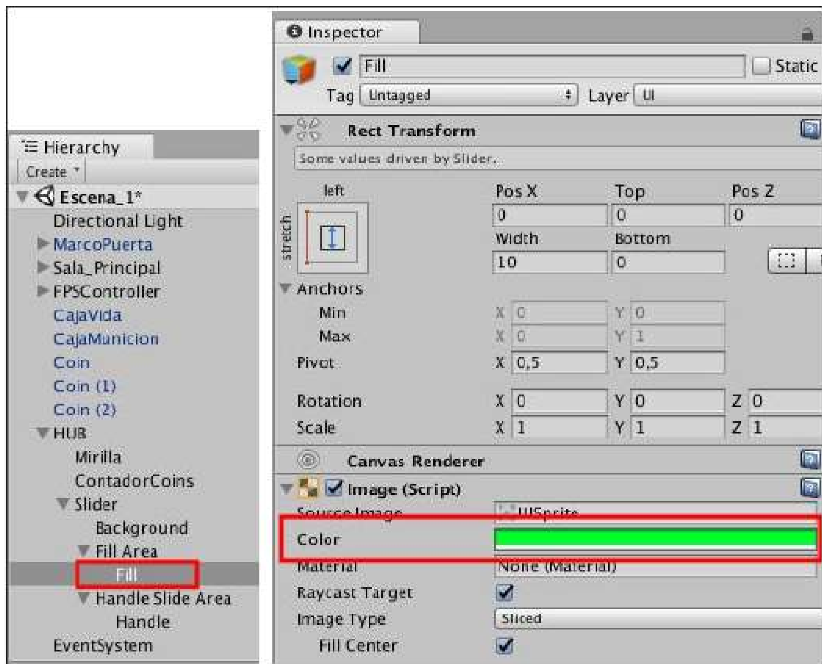


Fig. 9.71

El tercer elemento que encontramos es **Handle Slide Area** y es el icono en forma de circulo blanco que vamos a substituir por el Sprite corazón. Para ello seleccionamos el elemento **Handle** que es hijo de **Handle Slide Area**, accedemos a su componente Image y esta vez vamos a cambiar la imagen de Source Image que tiene por defecto por el Sprite Corazon.

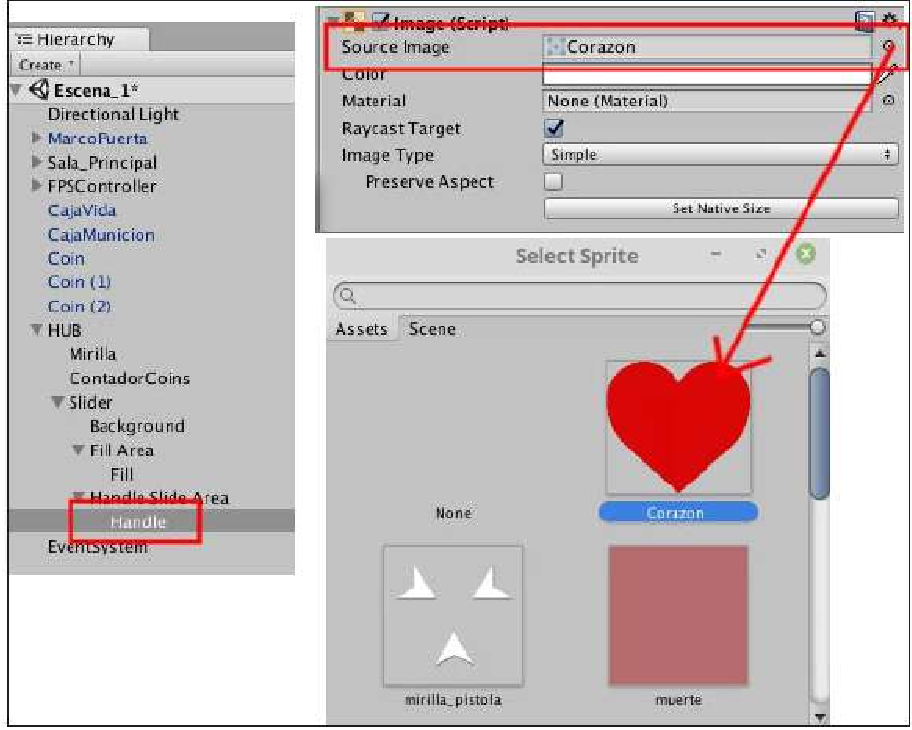


Fig. 9.72

Ahora si miramos la ventana **Game** veremos que el Slider está completamente rojo y eso es debido a por que el valor que tiene asignado por defecto es 0.

Fig. 9.73

Para llenar el Sider debemos seleccionar el elemento **Slider** y en la ventana Inspector acceder al parámetro **Max Value** que por defecto esta en 1 podemos darle el valor de 100 y luego podemos subir el valor a 100 del parámetro **Value**. De este modo nuestro **Slider** se llenara con la imagen de color verde.

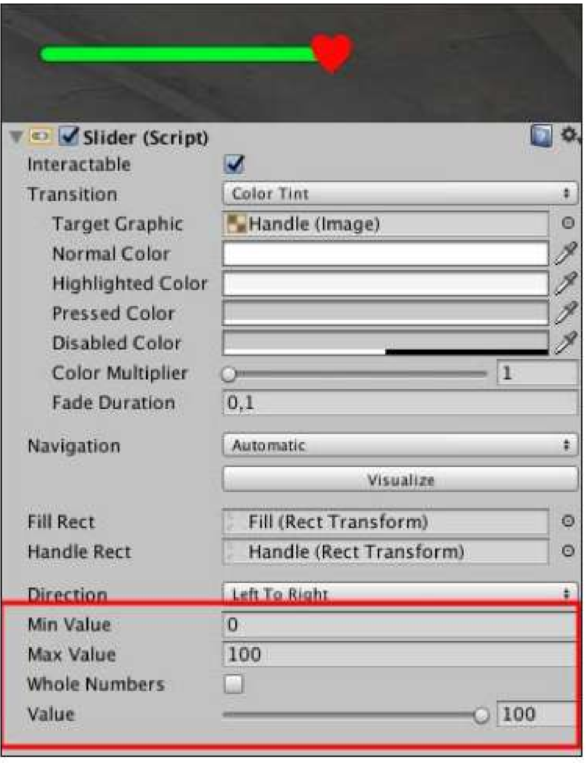


Fig. 9.74

Una vez tenemos preparado el slider con sus colores y posicionado en el lugar que queremos, vamos a empezar a editar el comportamiento de este para ello seguiremos con el script **ControlPlayer.cs**, hacemos doble clic encima del script si es que no lo tenemos abierto y se nos abrirá **Monodevelop** para empezar a escribir el código.

Script: ControlPlayer.cs

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

public class ControlPlayer : MonoBehaviour
{
    private int contador;
    public Text contadorMonedas;
    public int vidaInicial;
    public int vidaActual;
```

```
    public Slider barraVida;

    void Awake ()
    {
        this.contador = 0;
        this.vidaInicial = 100;
        this.vidaActual = this.vidaInicial;
    }

    public void SentirDolor(int cantidad)
    {
        this.vidaActual -= cantidad;
        barraVida.value = this.vidaActual;
    }

    void OnTriggerEnter(Collider otro)
    {
        if (otro.gameObject.CompareTag ("Item"))
        {
                otro.gameObject.SetActive (false);
                this.contador += 1;
                this.ContadorTexto ();
        }
    }

    public void ContadorTexto()
    {
        this.contadorMonedas.text = "COINS: " + this.contador.ToString ();
    }
```

Primero vamos a crear tres variables dos de tipo int una con nombre vidaInicial para saber con que valor de vida empieza mi player y otra con nombre vidaActual que me va a contener en cada momento cuanta vida tiene el player. La otra variable es de tipo Slider con nombre barraVida que la haremos pública para poder arrastrar después el objeto en si dentro de la ventana Inspector.
Dentro de la función Awake()decimos que la vidaInicial va a empezar con el valor 100, y que la vidaActual sera igual a la vidaInicial para empezar.
Ahora voy a crear una función para que mi personaje tenga un método para determinar el dolor que recibe. Esta función la he llamado SentirDolor() uno de los argumentos que va a contener entre paréntesis es una variable de tipo int con nombre cantidad. Dentro de la función utilizaremos la variable vidaActual para restarle la cantidad. En la siguiente linea de abajo utilizaremos la variable barraVida que contiene el objeto Slider y podemos acceder a su atributo value que es el que determina el relleno de la barra y lo igualamos al valor que tenga la variable vidaActual.
Guarda el script antes de volver a Unity para que tenga efecto el Script.

Bien antes de continuar con la ejecución de la escena necesitamos hacer varias cosas una de ellas es arrastrar el elemento **Slider** dentro del atributo Barra Vida que encontra-

remos en el componente script del objeto **FPSController** en la ventana Inspector. Si no lo tienes muy claro te lo indico en la siguiente imagen.

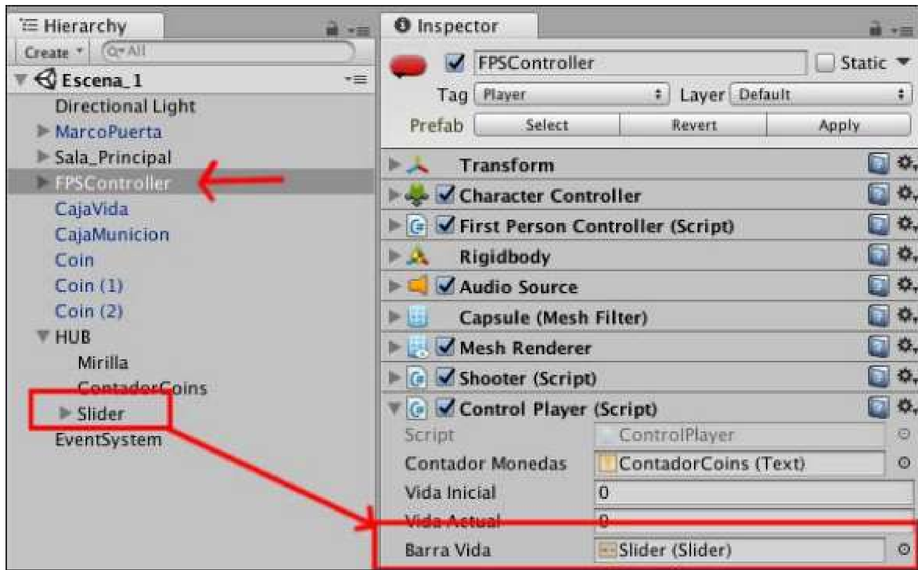


Fig. 9.75

Ahora nuestro script ya sabe donde está el **Slider,** pero si ejecutamos nuestra escena no va ha suceder nada porque necesitamos algún elemento que dañe a nuestro **FPSController** y es por eso que en la escena he creado una porción de suelo con otro material. A continuación configuraremos el suelo de lava.

12. Hacer daño a nuestro FPSController

Ahora vamos a configurar el objeto **Suelo_Damage**, para que no tengas que rebuscar entre la escena como este objeto es un prefab, si configuramos el prefab al actualizar el prefab todos los objetos de la escena sufren el cambio. Pero antes de empezar la configuración quiero que entiendas que queremos conseguir.

Para intentar explicar que quiero que pase me voy a ayudar con una imagen representativa de lo que quiero que suceda.

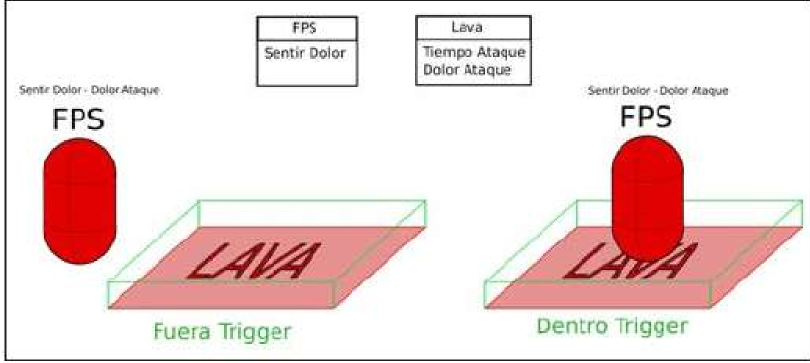


Fig. 9.76

Primero tenemos a nuestro Player al que le hemos dotado de un método para sentir dolor, este método lo utilizaremos para que el objeto Lava pueda comunicarse con nuestro player y hacerle saber que le vamos hacer daño. Otro punto es que aprovecharemos una vez más el **Trigger** del objeto lava para determinar cuando nuestro **FPS** entra o sale y crear un sistema para que pueda infligir una cantidad de dolor, cada (X) segundos mientras el **FPS** este dentro del **Trigger**.

Como he dicho anteriormente vamos a realizar los cambios al Prefab **Suelo_Damage q**ue repercutirá en el objeto de la escena. Primero vamos a la carpeta scripts le damos al botón **Create > C# Scripts,** para que nos creer el nuevo script y le ponemos como nombre Lava.

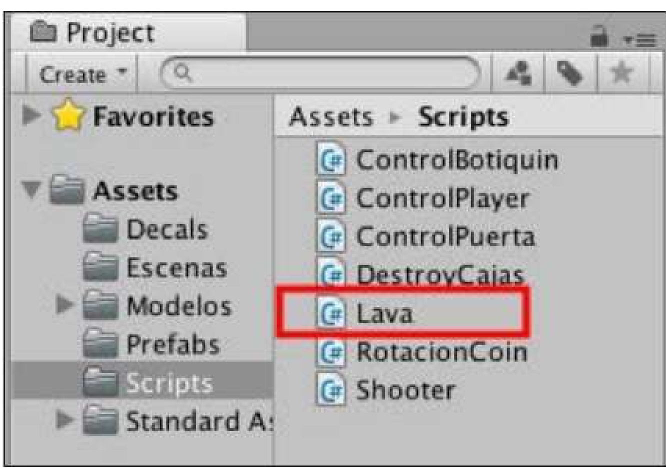


Fig. 9.77

A continuación accedemos a la carpeta **Prefabs** y seleccionamos el **prefab Suelo_Damage** y en la ventana **Inspector** le damos al botón **Add Component** y en el menú accederemos a seleccionar la opción **Scripts** y en el nuevo apartado podemos seleccionar el **Script** que hemos creado con nombre Lava, lo seleccionamos y se aplica el script automáticamente.

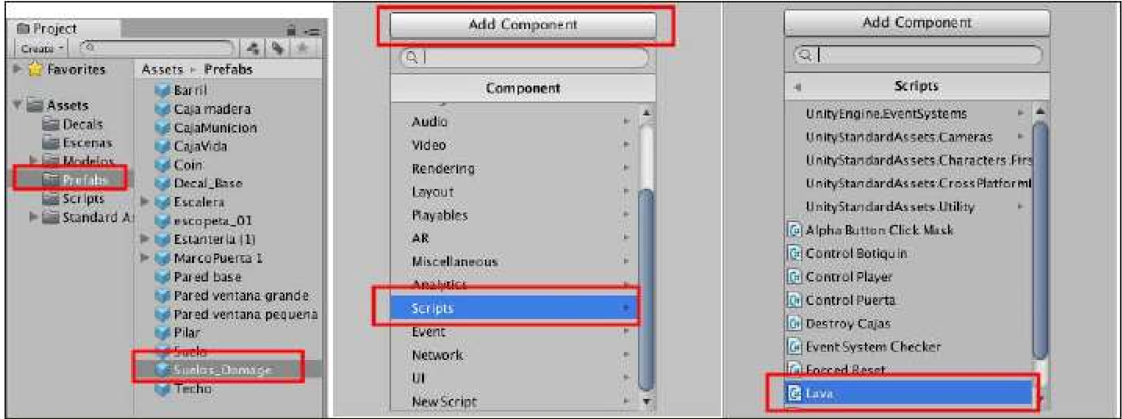


Fig. 9.78

Ahora vamos a hacer doble clic a este script **Lava** desde la carpeta Scripts por ejemplo para empezar a editarlo.

Script:Lava.cs

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

public class Lava : MonoBehaviour
{
    public float tiempoAtake;
    public int dolorAtake;
    ControlPlayer salud;

    private GameObject player;
    private bool playerDentro;
    private float tiempo;

    void Awake ()
    {
        this.tiempoAtake = 0.5f;
        this.dolorAtake = 10;
        this.player = GameObject.FindGameObjectWithTag ("Player");
        this.salud = this.player.GetComponent<ControlPlayer> ();
    }

    void OnTriggerEnter(Collider otro)
    {
        if (otro.gameObject == this.player)
        {
                this.playerDentro = true;
        }
    }

    void OnTriggerExit(Collider otro)
    {
        if (otro.gameObject == this.player)
        {
                this.playerDentro = false;
        }
    }

    void Atake()
    {
        this.tiempo = 0f;
        if (this.salud.vidaActual > 0)
        {
                this.salud.SentirDolor (this.dolorAtake);
```

```
                }
        }

        void Update ()
        {
           this.tiempo += Time.deltaTime;
           if (this.tiempo >= this.tiempoAtake && this.playerDentro && this.
salud.vidaActual > 0)
           {
                this.Atake ();
           }
        }
}
```

Primero vamos a crear las siguientes variables: una de tipo float con nombre tiempoAtake para determinar el tiempo que va ha estar atacando, otra de tipo int con nombre dolorAtake que se va a encargar de informar al FPS de la cantidad de dolor, luego he creado una variable Salud de la clase ControlPlayer para tener acceso al Script en concreto. Las otras tres variables son una de tipo GameObject con nombre player para acceder al objeto FPS otra de tipo booleana con nombre playerDentro que utilizaremos para saber cuando esta dentro nuestro FPS y para finalizar una variable de tipo float con nombre tiempo para determinar como su nombre indica el tiempo.
En la función Awake() determinamos en la variable tiempoAtake el valor que queremos que tenga; en este caso le he dado un valor de 0,5f. Para la variable dolorAtake le he dado el valor de 10. Para localizar el FPS utilizo la variable player con el método FindGameObjectWithTag,que localiza el objeto que en este caso lleve una etiqueta con el nombre "Player". (Debemos asegurarnos que la lleva antes de ejecutar la escena). Para finalizar la variable salud la igualo a la variable player y accedemos a los componentes pero en este caso del Script ControlPlayer.
Creamos dos funciones OnTrigger una para entrada y la otra para la salida. Para la de la entrada le damos la condición de que si el objeto que entra es el objeto this.player que hemos declarado en el Awake que es nuestro FPS, entonces sabemos que esta dentro así que utilizamos la variable playerDentro que es de tipo booleana y la igualamos a true (verdadero). Para la salida del Trigger es exactamente igual que a la de entrada pero en este caso la variable playerDentro es falsa por lo tanto la igualamos a false.
Ahora vamos a crear una función que dañe a nuestro FPS, le he puesto el nombre de Atake() y dentro primero igualaremos el tiempo a 0 y crearemos una condición relacionada con la variable salud accediendo al atributo vidaActual y que dice que si este atributo es mayor de 0 (Eso es que todavía está vivo) se cumplirá que salud accediendo al método SentirDolor le daremos un argumento que es la variable dolorAtaque. Dicho de otra forma esto le dirá al FPS que le están haciendo daño. Pero esto no va a funcionar si no llamamos a esta función.
Dentro de la función Update() es donde vamos a llamar al método Atake para ello primero voy a declarar que la variable tiempo es igual a si misma más Time.deltaTime y luego creo if con tres condiciones la primera es que el tiempo sea mayor o igual que el tiempoAtake la segunda que playerDentro y la última que salud.vidaActual se mayor que 0. Una vez se cumple todo esto el objeto lava va a realizar un ataque es decir llamará a la función Atake();
Guarda el script para que tenga efecto en Unity.

Una vez hemos vuelto a Unity vamos a asegurarnos de que nuestro **FPSController** tiene aplicada la etiqueta Player. Para ello selecciona desde la ventana Hierarchy el

nombre **FPSController** y en la ventana Inspector miramos en la opción **Tag** si tiene el nombre de Player. Si no lo tiene haz clic encima de **Untagged** y seleccionamos la opción Player.

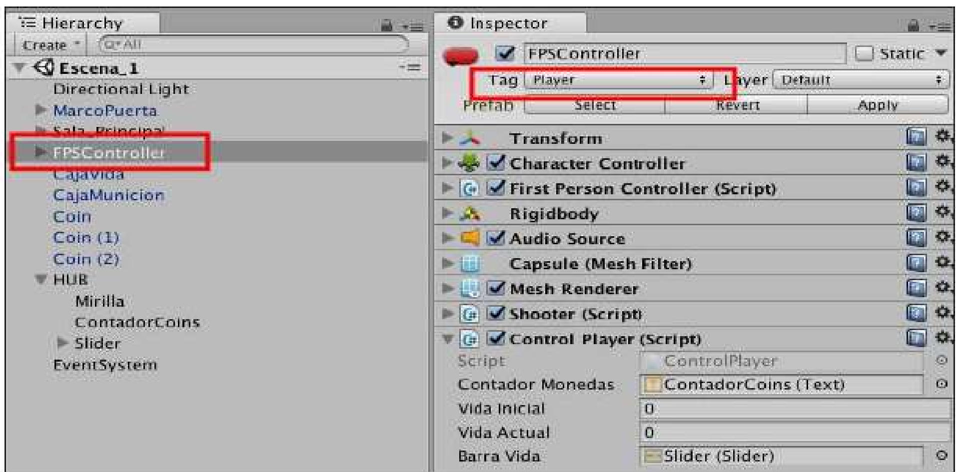


Fig. 9.79

Ahora ya puedes ejecutar la escena y comprobar que cuando pisas la lava y te mantienes en ella el slider cambia de color.

Fig. 9.80

13. Cómo curar a nuestro FPSController

Ahora que disponemos de un objeto en escena que nos provoca daño vamos a configurar el script con nombre **ControlPlayer** para que cuando toquemos el **Trigger** de la caja de vida recuperemos toda la vida. También podemos incorporar un temporizador o algún

tipo de sistema para que la caja de vida vuelva a aparecer al cavo de uno instante y así disponer de curación infinita.

Para realizar la curación abrimos el script ControlPlayer y vamos a añadir el siguiente código.

Script: ControlPlayer

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

public class ControlPlayer : MonoBehaviour
{
    private int contador;
    public Text contadorMonedas;
    public int vidaInicial;
    public int vidaActual;
    public Slider barraVida;

    public float tiempoBotiquin;
    private float tiempoActual;
    private GameObject miBotiquin;

    void Awake ()
    {
        this.contador = 0;
        this.vidaInicial = 100;
        this.vidaActual = this.vidaInicial;
        this.miBotiquin = GameObject.FindWithTag ("Botiquin");
        this.tiempoBotiquin = 5f;
        this.tiempoActual = Time.time;
    }
    public void SentirDolor(int cantidad)
    {
        this.vidaActual -= cantidad;
        barraVida.value = this.vidaActual;
    }
    public void Curarse()
    {
        this.vidaActual = 100;
        barraVida.value = this.vidaActual;
    }

    void OnTriggerEnter(Collider otro)
    {
```

```
        if (otro.gameObject.CompareTag (“Item”))
        {
         otro.gameObject.SetActive (false);
         this.contador += 1;
         this.ContadorTexto ();
        }

        if (otro.gameObject.CompareTag (“Botiquin”))
        {
         if (this.vidaActual < 100)
         {
               otro.gameObject.SetActive (false);
               this.Curarse ();
         }
        }
    }

    void Update ()
    {
        if ((Time.time-this.tiempoActual) > this.tiempoBotiquin)
        {
         this.miBotiquin.SetActive (true);
         this.tiempoActual = Time.time;
        }
    }

    public void ContadorTexto()
    {
        this.contadorMonedas.text = “COINS: “ + this.contador.ToString ();
    }
}
```

Empezamos por crear tres variables: la primera es pública de tipo float y con nombre tiempoBotiquin donde le daremos un valor que será el tiempo de espera. La segunda variable es de tipo float y le he puesto el nombre de TiempoActual y para finalizar la última variable es de tipo GameObject y la he llamado miBotiquin que servirá para localizar el objeto cajaVida.
En la función Awake() he utilizado la variable miBotiquin para localizar el objeto en escena con etiqueta “Botiquin”, si te fijas en Unity veras que la caja vida tiene esta etiqueta. La siguiente línea he dado el valor de 5 al tiempoBotiquin y al tiempoActual le doy el valor del Tiempo utilizando el método Time.time.
El siguiente paso que he realizado es crear una función para curar que la he llamado Curarse() dentro de esta función le digo que la vidaActual vuelva a ser su máximo es decir 100. También utilizo la variable barraVida con el atributo value y lo igualo a la vida Actual para que se rellene.
Dentro de la función OnTriggerEnter creo otra condición que cuando colisionemos con un objeto que tenga la etiqueta “Botiquin” sucederá otra condición. Esta nueva condición es para asegurarnos de que la vida es menor a 100, si se cumple el objeto se desactivará y llamaremos a la función curarse

Dentro de la función Update() creamos una condición en donde restamos el tiempo fijo con el tiempo actual si es mayor que el tiempoBotiquin entonces activaremos otra vez miBotiquin y volveremos a darle el valor del Tiempo a la variable tiempoActual.
Guardamos el script antes de volver a Unity para que tenga efecto.

Volviendo a Unity, ya puedes ejecutar la escena y comprobar que cuando has perdido vida al pisas la lava y te acercas a la caja de vida tu **Slider** vuelve a rellenarse totalmente. En este caso vamos a tener una caja de vida que va a reaparecer siempre que la necesites.

14. Cómo limitar el número de munición

Otro de los aspectos que quiero tocar es como limitar el numero de disparos que podemos realizar , tener un texto como el que hemos utilizado en el recuento de monedas para que nos informe del numero de balas que disponemos y para terminar recargar nuestra munición con el objeto caja munición.

Antes de afrontar el reto vamos a ver que queremos conseguir:

- Crear un texto que nos de información sobre el numero de balas que necesitamos
- Nuestro **FPSController** no puede disparar ningún **Decal** una vez el contador de balas llega a 0.
- Cuando nos acercamos a la caja munición tengamos un numero de balas y podamos volver a dispara.

Crear el contador de balas

Primero vamos a crear el objeto de tipo Texto accediendo a la barra de menús principal **GameObject > UI > Text**. Automáticamente veremos como se añade un elemento de tipo texto debajo del **Slider** dentro del **HUB** (Canvas). También veremos que en la ventana **Game** se muestra un texto con un color "negro". Igual que hicimos con el texto **ContradorCoins**, en el nuevo texto primero vamos a ponerle el nombre de **ContradorMunicion** y podemos arrastrarlo a una posición superior entre **ContadorCoins** y **Slider** en la ventana **Hierarchy** como te muestro en la siguiente imagen.

Fig. 9.81

Ahora vamos a posicionarlo y darle un aspecto igual que el que teníamos con el texto **contadorCoins**. Primero configuramos el anclaje desde el cuadro a la posición superior derecho. Dentro del **RectTransform** introducimos los siguientes valores de posición **X=-80, Y=-70, Z=0**, los valores de **Width= 150, Height=30**. En el componente de **Text(Script)** en la caja de texto introducimos la palabra en mayúsculas "BALAS:". En el parámetro **Character** so-

lamente tenemos la fuente Aria el tipo de fuente utilizamos **Bold** y el tamaño le damos un valor de 24. para finalizar le damos un color amarillo.

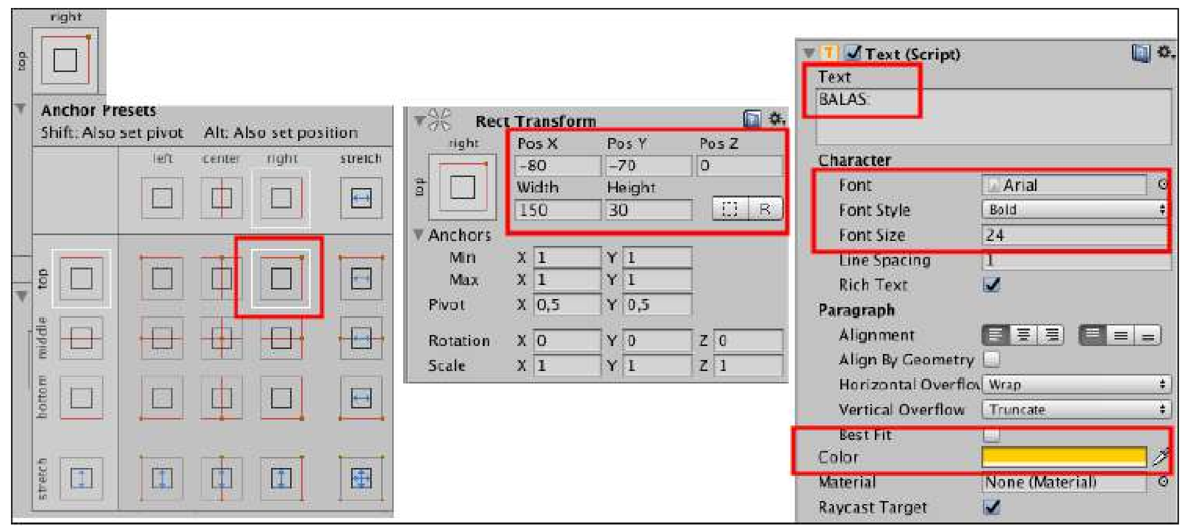


Fig. 9.82

Si todo es correcto deberías tener un nuevo texto con la palabra **BALAS:** debajo del texto **COINS** en la ventana **Game**, como te muestro a continuación.

Fig. 9.83

Sistema mediante Script para contar las balas

Para hacer un recuento de munición vamos a utilizar el script **Shooter** porque es el script que dispone del sistema de disparo de nuestro **FPSController**. Vamos a hacer doble clic encima del script **Shooter** para abrir el editor **Monodevelop** si es que no lo tienes abierto.

Script:Shooter.cs

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

public class Shooter : MonoBehaviour
{
        private Ray rayo;
        private RaycastHit hit;
        public float distanciaDisparo;
        private Camera camara;
        private Vector2 centroCamara;
        public GameObject[] decalsPrefabs;
        public GameObject[] createdDecals;
        public int decalIndex;
        public float tempoDisparo;
        private float tempoUltimoDisparo;
        private Quaternion rotDecal;
        private Vector3 posDecal;
        public LayerMask decalLayerMask;
        public float fuerzahit = 150f;
        public int escopetaDamage =1;
        //---------Recuento de Balas------------//
        public int balas;
        public Text contbalas;
        private int balasActual;

        void Awake()
        {
                this.camara = gameObject.transform.GetChild (0).
GetComponent<Camera>();
                this.centroCamara.x= Screen.width/2;
                this.centroCamara.y= Screen.height/2;
                this.tempoUltimoDisparo = Time.time;
                this.balas = 10;
                this.ContadorBalas ();

                for(int decalNum=0; decalNum<this.createdDecals.Length;
decalNum++)
                {
                        this.createdDecals[decalNum] = GameObject.Instantia-
te(this.decalsPrefabs[0], Vector3.zero,Quaternion.identity)as GameObject;
                        this.createdDecals[decalNum].GetComponent<Renderer>().
enabled=false;
                }
```

```
                this.decalIndex = 0;
        }
        public void ContadorBalas()
        {
                this.contbalas.text = "BALAS: " + this.balas.ToString ();
        }

        void Update()
        {
                this.ContadorBalas ();
                if(Input.GetButtonDown("Fire1"))
                {
                        if (this.balas < 1)
                        {
                                return;
                        }
                        if((Time.time-this.tempoUltimoDisparo)>this.
tempoDisparo)
                        {
                                this.rayo= this.camara.ScreenPointToRay(this.
centroCamara);
                                this.tempoUltimoDisparo = Time.time;

                                if(Physics.Raycast (this.rayo,out this.hit,-
this.distanciaDisparo,                                          this.
decalLayerMask))
                                {
                                        this.rotDecal=
                                                        Quaternion.From-
ToRotation(Vector3.forward,this.hit.normal);
                                        this.posDecal= this.hit.point+this.hit.
normal* 0.01f;
                                        this.createdDecals[this.decalIndex].
transform.position=this.posDecal;
                                        this.createdDecals[this.decalIndex].
transform.rotation=this.rotDecal;
                                        this.createdDecals[this.decalIndex].
transform.parent=null;
                                        this.createdDecals[this.decalIndex].
GetComponent<Renderer>().enabled=true;

                                        if(this.hit.collider.tag=="Puerta" ||
this.hit.collider.tag=="Caja")
                                        {
```

```
                                                  this.createdDecals[this.decalIn-
dex].transform.parent=
this.hit.collider.gameObject.transform;
                                          }
                                          this.decalIndex++;

                                          if(this.decalIndex>9)
                                          {
                                                  this.decalIndex=0;
                                          }
                                          DestroyCajas salud = hit.collider.
GetComponent<DestroyCajas>();

                                          if(salud != null)
                                          {
                                                  salud.Damage(escopetaDamage);
                                          }
                                          if(hit.rigidbody != null)
                                          {
                                                  hit.rigidbody.AddForce (-hit.
normal*fuerzahit);
                                          }
                                          this.balas -= 1;
                                  }
                          }
                  }
          }
}
```

Primero de todo para poder utilizar elementos de la UI primero cargaremos en la parte superior using UnityEngine.UI, para poder utilizar sin problemas el objeto Text que tenemos en escena.
Primero crearemos las siguientes variables un publica de tipo int con nombre balas para guardar el valor de las balas, otra también pública con de tipo Text con nombre contbalas que nos permitirá acceder al componente texto y una última variable privada de tipo int con nombre balasActual, en donde guardaremos el número de balas que tenemos en cada momento.
Dentro de la función Awake() vamos a declarar que this.balas va a empezar con 10. Luego verás que tenemos una función this.ContadorBalas()que te explico a continuación.
La función ContadorBalas() la he creado por comodidad, de este modo solo tengo que llamar a la función cuando la necesite y esta función contiene lo que se va a mostrar en el texto. Para ello accedo a la variable de tipo Text this.contbalas y accedo a su atributo text en donde le digo que me muestre el texto “BALAS: ” acompañado del valor de la variable balas. Como es un texto lo que quiero que se muestre utilizo el método ToString para que trate los valores numéricos como texto.
Dentro de la función Update() delante de la condición Input llamamos a la función this.ContadorBalas() una vez que pulsamos la tecla para disparar vamos a condicionar que si las balas son menores que uno crearemos un retorno y no avanzará en el código. Para que las balas se vallan restando cada vez que se dispara dentro de la condición Physics.Raycast al final de todo diremos que las balas menos 1.
Guarda el script para que tenga efecto al volver a Unity.

Con el nuevo script **Shooter** editado y guardado correctamente, debemos arrastrar el objeto **CajaMunición** en la casilla **Contbalas** del script **Shooter** dentro de la ventana **Inspecto** del objeto **FPSController,** para que el script tenga efecto, como te muestro en la siguiente imagen.

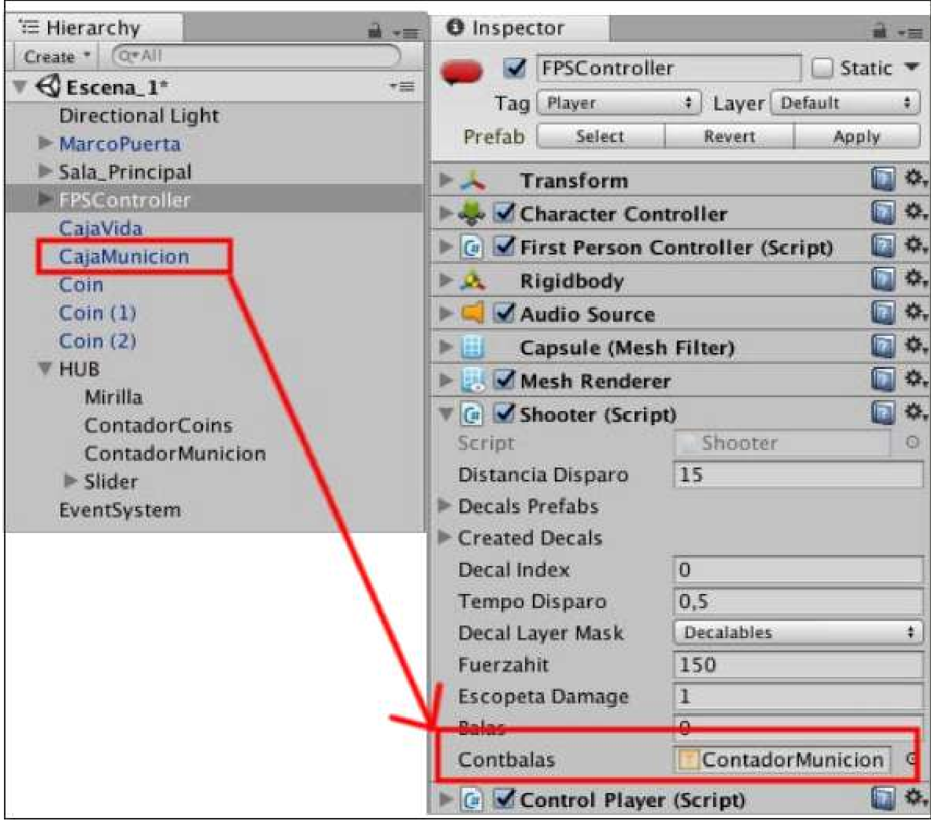


Fig. 9.84

Ahora si quieres puedes ejecutar la escena y comprobar como el texto Balas se llena con el valor 10 y cuando disparas con el botón izquierdo del ratón va disminuyendo el numero de balas hasta llegar al valor 0 el cual no te permite disparar.

Recargar la munición

En la escena tenemos un objeto con nombre **CajaMunicion** que vamos a utilizar para recargar nuestras balas. El objetivo es el mismo que hemos realizado con la caja de Vida, es decir cuando nuestro **FPSController** necesite munición pueda recargar 10 balas cada vez que recoja la **CajaMunición**, que también haremos que vuelva aparecer una vez la desactivemos.

Para realizar la recarga vamos a dirigirnos al Script Shooter y vamos a añadir las siguientes lineas de código.

Script:Shooter.cs

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

public class Shooter : MonoBehaviour
{
        private Ray rayo;
        private RaycastHit hit;
        public float distanciaDisparo;
        private Camera camara;
        private Vector2 centroCamara;
        public GameObject[] decalsPrefabs;
        public GameObject[] createdDecals;
        public int decalIndex;
        public float tempoDisparo;
        private float tempoUltimoDisparo;
        private Quaternion rotDecal;
        private Vector3 posDecal;
        public LayerMask decalLayerMask;
        public float fuerzahit = 150f;
        public int escopetaDamage =1;
        //---------Recuento de Balas-----------//
        public int balas;
        public Text contbalas;
        private int balasActual;
        public float tiempoMunicion = 10f;
        private GameObject miMunicion;
        private float tiempoActual;

        void Awake()
        {
                this.camara = gameObject.transform.GetChild (0).
GetComponent<Camera>();
                this.centroCamara.x= Screen.width/2;
                this.centroCamara.y= Screen.height/2;
                this.tempoUltimoDisparo = Time.time;

                this.miMunicion = GameObject.FindWithTag ("Municion");
                this.tiempoActual = Time.time;
                this.balas = 10;
                this.ContadorBalas ();
                for(int decalNum=0; decalNum<this.createdDecals.Length;
decalNum++)
```

```
			{
				this.createdDecals[decalNum] = GameObject.Instantia-
te(this.decalsPrefabs[0],				Vector3.zero,Quaternion.
identity)as GameObject;
				this.createdDecals[decalNum].GetComponent<Renderer>().
enabled=false;
			}
			this.decalIndex = 0;
		}

		void OnTriggerEnter(Collider otro)
		{
			if (otro.gameObject.CompareTag ("Municion"))
			{
				otro.gameObject.SetActive (false);
				this.Recargar ();
				this.ContadorBalas ();
			}
		}

		public void Recargar()
		{
			this.balas += 10;
		}


		public void ContadorBalas()
		{
			this.contbalas.text = "BALAS: " + this.balas.ToString ();
		}

		void Update()
		{
			this.ContadorBalas ();
			if ((Time.time-this.tiempoActual)>this.tiempoMunicion)
			{
				this.miMunicion.SetActive (true);
				this.tiempoActual = Time.time;
			}
			if(Input.GetButtonDown("Fire1"))
			{
				if (this.balas < 1)
				{
					return;
				}
				if((Time.time-this.tempoUltimoDisparo)>this.
tempoDisparo)
```

```
                        {
                                this.rayo= this.camara.ScreenPointToRay(this.
centroCamara);
                                this.tempoUltimoDisparo = Time.time;

                                if(Physics.Raycast (this.rayo,out this.hit,-
this.distanciaDisparo,                                              this.
decalLayerMask))
                                {
                                        this.rotDecal=
                                                                Quaternion.From-
ToRotation(Vector3.forward,this.hit.normal);
                                        this.posDecal= this.hit.point+this.hit.
normal* 0.01f;
                                        this.createdDecals[this.decalIndex].
transform.position=this.posDecal;
                                        this.createdDecals[this.decalIndex].
transform.rotation=this.rotDecal;
                                        this.createdDecals[this.decalIndex].
transform.parent=null;
                                        this.createdDecals[this.decalIndex].
GetComponent                                                        <Rende-
rer>().enabled=true;

                                        if(this.hit.collider.tag=="Puerta" ||
this.hit.collider.tag=="Caja")
                                        {
                                                this.createdDecals[this.decalIn-
dex].transform.parent=
this.hit.collider.gameObject.transform;
                                        }
                                        this.decalIndex++;

                                        if(this.decalIndex>9)
                                        {
                                                this.decalIndex=0;
                                        }
                                        DestroyCajas salud = hit.collider.
GetComponent<DestroyCajas>();

                                        if(salud != null)
                                        {
                                                salud.Damage(escopetaDamage);
                                        }
                                        if(hit.rigidbody != null)
                                        {
```

```
                                        hit.rigidbody.AddForce (-hit.
normal*fuerzahit);
                                }
                                this.balas -= 1;
                        }
                }
        }

    }
}
```

Para empezar crearemos 3 nuevas variables; la primera de tipo float con nombre tiempoMunicion al que le daremos un valor de 10, otra variable de tipo GameObject con el nombre miMunicion que servirá para referenciar el objeto CajaMunicion y la otra variable es de tipo float con nombre tiempoActual.
Dentro del Awake() utilizamos la variable miMunicion para guardar la localización del Objeto al que queremos hacer referencia con el método FindWithTag y entre paréntesis el nombre de la etiqueta ("Municion"). Para finalizar tiempoActual lo igualaremos la tiempo.
El siguiente código que agregamos es una función OntriggerEnter en donde creamos una condición que cuando el otro collider sea el que tiene la etiqueta ("Municion") entonces desactivaremos el objeto llamaremos a una función con nombre Recargar() que se ha creado después y llamamos a la otra función ContradorBalas().
La función Recargar() contiene la variable balas que será sumada por el valor de 10 cada vez que se llame a la función.
Para finalizar en el Update crearemos una condición que cuando el tiempo supere al tiempoMunicion, el objeto se volverá activar y para que esto funciones diremos que el tiempoActual sea igual al tiempo.
Guarda el script antes de volver a Unity.

Una vez que hemos añadido el código anterior nos dirigimos a Unity y ejecutamos la escena para comprobar como se nos gasta las balas y podemos recargar al acercarnos a la caja de munición.

Fig. 9.85

15. Pantalla de fallecimiento

Para terminar le capitulo vas a crear una imagen con un sprite de color rojo semi-transparente que por defecto va ha estar desactivado y se va a activar cuando la vida del FPS se termine. Esto va a ser un pequeño ejercicio en el que te recomiendo que intentes solucionar, tu mismo. Voy a darte algunas pistas para resolver este problema y también podrás ver la solución importando el paquete de la escena terminado con todos los scripts terminados.

La primera pista es que primero debes crear una Imagen para el canvas y ponerle el sprite muerte.

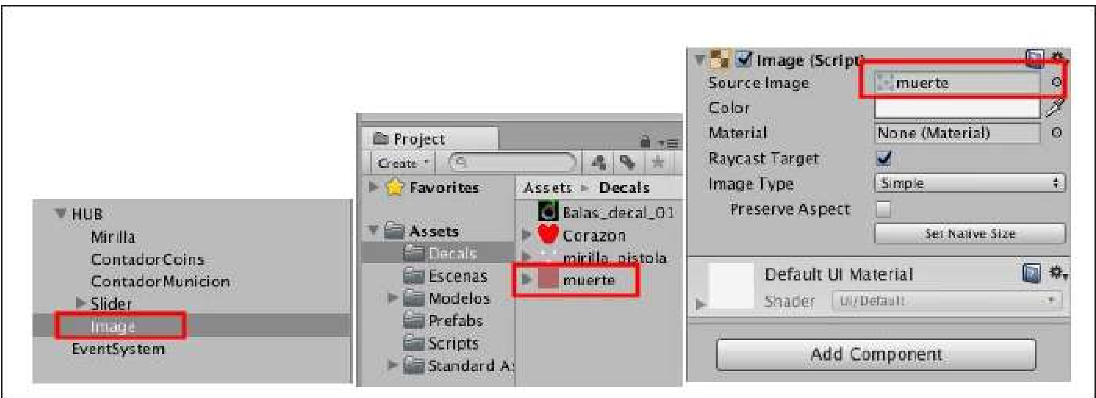


Fig. 9.86

Esta imagen debe ocupar todo el canvas y debe de estar desactivado por defecto en la ventana Inspector.

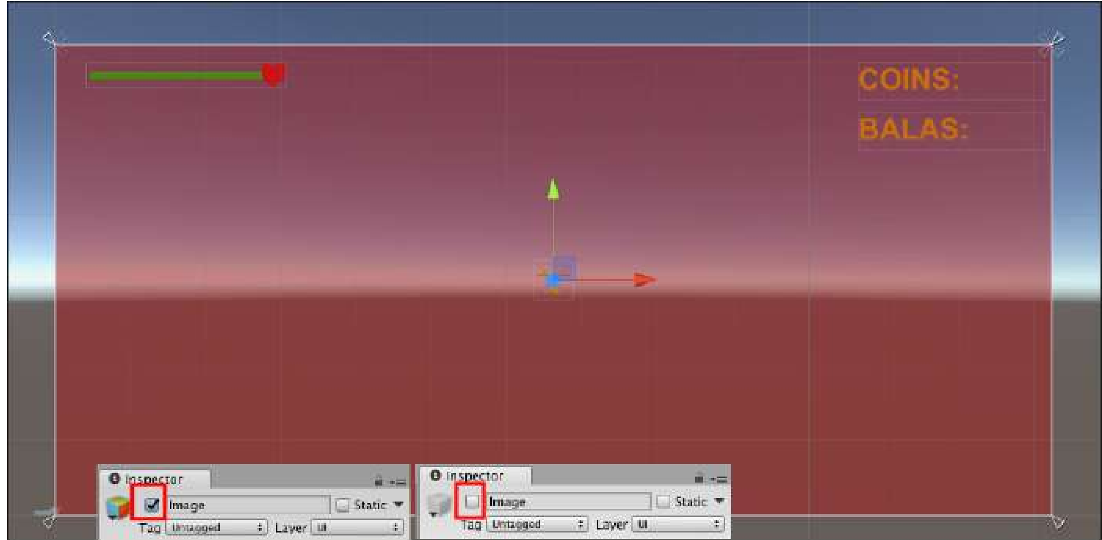


Fig. 9.87

Otro aspecto es que el código lo tienes que añadir al script **Control Player** dentro del **FPSController.** Para este problema tienes que localizar la imagen y para ello necesitaras crear una variable de tipo **Image,** para poder activar la imagen cuando la vida llegue a 0.

Intenta resolverlo por ti mismo, pero en el caso de que no consigas encontrar una solución en este o en alguno de los ejemplos que hemos ido realizando a lo largo del capitulo puedes importar en un proyecto nuevo el paquete **Solucion_Capitulo_9.unitypackage,** que contiene todas las actividades y scripts de este capitulo .

Capítulo 10

Animación

- Animación en Unity
- El flujo de trabajo
- Animation Clips
- Ventana Animation
- Animation Contoller
- Máquina de estados
- Proyecto de un soldado

1. Animación en Unity

Las características de Animación de Unity incluyen animaciones reorientables, control total de pesos de animación en tiempo de ejecución, llamadas a eventos desde la reproducción de la animación, jerarquías y transiciones sofisticadas de máquinas de estados, formas de combinación para animaciones faciales y mucho más.

Vista general del sistema de Animación en Unity

Unity dispone de un sistema de animación bastante sofisticado y a veces es posible que intimide a simple vista. Este sistema tienes varias características que te enuncio a continuación:

- Un flujo de trabajo sencillo configurable de animación para todos los elementos de Unity, incluidos objetos, personajes y propiedades.
- Clips de animación que podemos realizar dentro de Unity o si prefieres soporta la importación de clips de animación creados en otros programas.
- Podemos reorientar las animaciones de tipo humanoide, es decir podemos transferir una animación de un modelo de personaje a otro.
- Tenemos una forma muy cómoda de visualizar los clips de animación, manipular la transición e interacción entre clips.
- También podemos animar diferentes partes del cuerpo de un modelo con diferente lógica. Tenemos capas y características de enmascaramiento.

2. El flujo de trabajo

Las animaciones de Unity se basan en clips de animación. Estos clips contienen información sobre, cómo ciertos objetos se mueven y se rotan durante un periodo de tiempo. Cada clip se considera una grabación lineal. Estos clips normalmente se crean externamente con programaras como Max, Maya o Blender, pero también puede venir a partir de estudios que hacen capturas de movimiento.

Los clips de animación se organizan en un sistema estructurado similar a un diagrama de flujo llamado **Animator Controller**. Este controlador actúa como una máquina de estados que gestiona el clip que debería reproducirse a cada momento y como combinar estas animaciones.

Un ejemplo simple del **Animator Controller** podría ser la animación de una puerta abriéndose o cerrándose en donde solo haría falta dos clips de animación. Por el contrario también podríamos tener controles más avanzados en donde el **Animator Controller** podría contener docenas de animaciones para un personaje con todas las acciones necesarias para el desarrollo de un player y no solo las animaciones, también proporcionar la mezcla entre los diferentes clips para crear movimientos fluidos.

Otro aspecto que hay que nombrar en el sistema de animación de Unity es la característica de crear Avatares de tipo **humanoid**. Esta característica permite que podamos crear animaciones de otros artistas con nuestros propios personajes.

En resumen en Unity disponemos de los sistemas:

- Animation Clips.
- Animation Controller
- Avatar

Estos sistemas los unimos dentro de un **GameObject** a través de un componente Animador que tiene referencia a un **Animator Controller** y en ocasiones a un Avatar para el modelo. El **Animator Controller** a su vez contiene las referencias a los clips de animación que se utilizan. A continuación vamos paso a paso para entender como funciona todo.

3. Animation Clips

Si no tenemos clips de animación no vamos a poder animar nada, por eso los clips de animación son uno de los elementos principales del sistema de animaciones en Unity. Debes tener claro que los clips de animación se pueden importar de fuentes externas y no necesariamente crearlas en Unity.

Para crear animaciones en Unity desde cero utilizamos el editor **Animation** que podemos encontrar accediendo al menú principal **Window > Animation**. Cuidado no confundir con **Animator.**

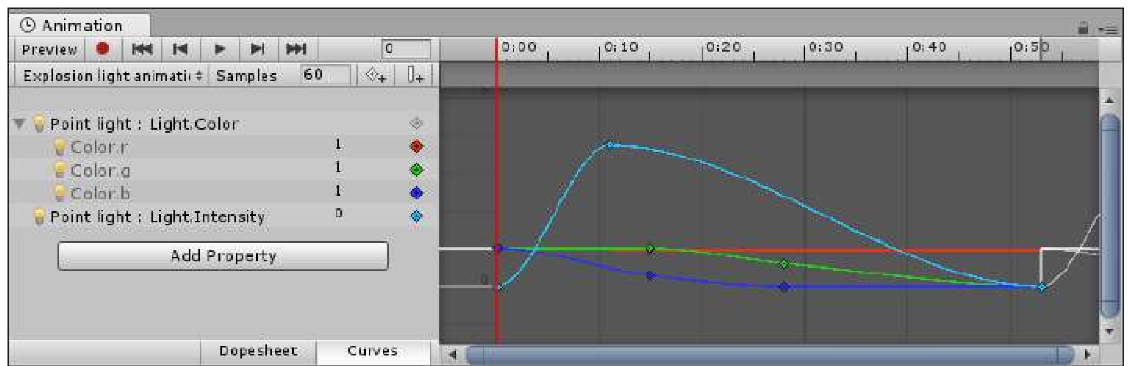


Fig. 10.1

En esta nueva ventana podemos crear y editar clips de animación. Estos clips pueden ser animaciones de;

- Posición, rotación y escala de un GameObject.
- Propiedades de los componentes, como el color del material, la intensidad de la luz, el volumen de un sonido.
- Propiedades dentro de los propios scripts que incluyen variables de tipo float, integer, enum, vector y booleanas.
- El tiempo de las llamadas a métodos dentro de los propios scripts.

Hemos nombrado antes que los clips de animación los podemos crear desde cero en Unity y también los podemos importar de una fuente externa, a continuación vamos a ver como podemos crear clips de animación desde cero. Para el siguiente apartado crea un proyecto nuevo 3D con el nombre que quieras. Este proyecto es a modo explicativo al que vas a utilizar si quieres para realizar varios ejemplos.

4. Ventana Animation

Podemos abrir este editor accediendo al menú principal **Window > Animation**. Este editor nos permite crear modificar clips de animación dentro de Unity. Ademas de animar

el movimiento también podemos animar variables materiales y componentes. Cuando abrimos por primera vez esta ventana veremos que nos aparece una ventana flotante, te aconsejo que arrastres la ventana en la parte inferior de manera que te quede las ventanas **Scene** y **Game** en la parte superior y la ventana **Animation** debajo al lado de la Consola.

Uno de los aspectos que veremos de esta ventana es que en un principio esta vacía con un mensaje en el centro que nos invita a crear nuestro primer clip de animación.

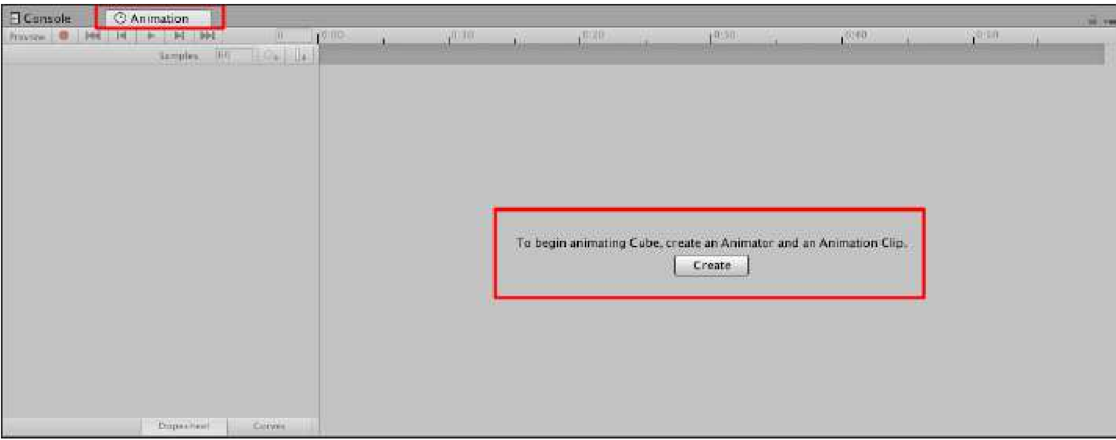


Fig. 10.2

Es tentador pero antes debes saber que para animar un simple GameObject en Unity, el objeto necesita un componente Animador adjunto y este debe hacer referencia a un **Animation Controller** que a su vez contiene referencias a uno o mas clips de Animación, pero tranquilo todos estos elementos se crearán y configurarán automáticamente al crear nuestro primer clip con el editor Animation.

Primero vamos a crear un GameObject Cube y nos aseguraremos de tenerlo seleccionado antes de pulsar el botón "Create" seguidamente nos pedirá que guardemos con un nombre, en este caso le he puesto el nombre de clip_1. Este clip de animación se guardará en algún lugar dentro de la carpeta Assets. Si lo deseas, guardalo allí por defecto y luego puedes crear una carpeta ClipsAnimation dentro de la ventana Project.

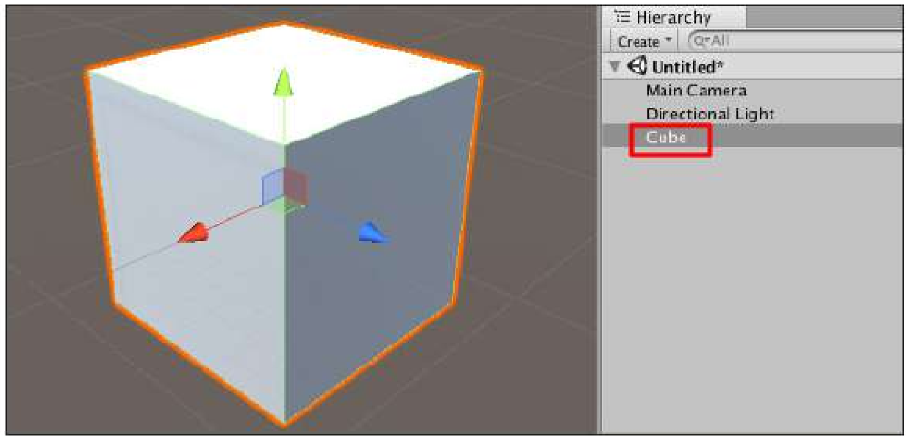


Fig. 10.3

Una vez hemos creado el clip y le hemos dado nombre si miramos en la ventana **Project**, veremos que se crea el clip de animación **Clip_1** y otro elemento con el nombre del Objeto que es la referencia a un **Animation Controller.** En mi caso he creado una carpeta con el nombre **ClipAnimation** para tenerlo todo ordenando.

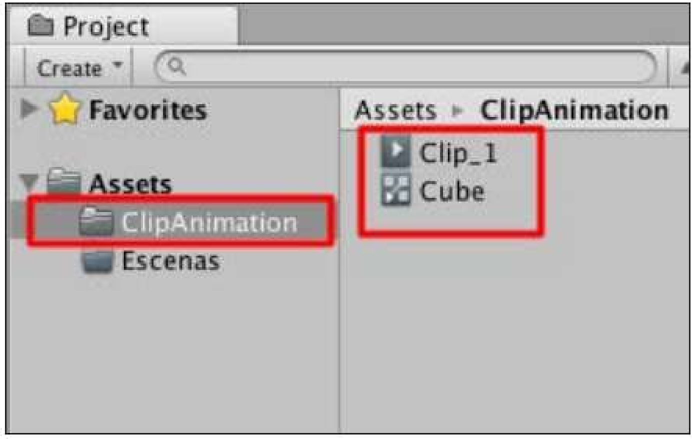


Fig. 10.4

Ahora vamos a inspeccionar el editor **Animation** que a tomado un aspecto nuevo y ahora si esta disponible para animar.

En el lado Izquierdo del editor hay una zona en donde se agrupa en forma de lista las propiedades que se animan, en estos momentos esta vacía. En la parte derecha tenemos una linea de tiempo y es donde podremos añadir puntos clave para la animación.

Fig. 10.5

Si hacemos clic encima de la parte derecha por encima del botón Add Property se nos aparecerá un menú con las propiedades que podemos seleccionar del GameObject, en este caso son los componentes del GameObject. Estos componentes son propiedades del objeto que a su vez contienen sub-propiedades. Despliega las propiedades de Transform y selecciona la sub-propiedad Rotation, puedes hacer clic encima del símbolo (+) que tiene a su lado.

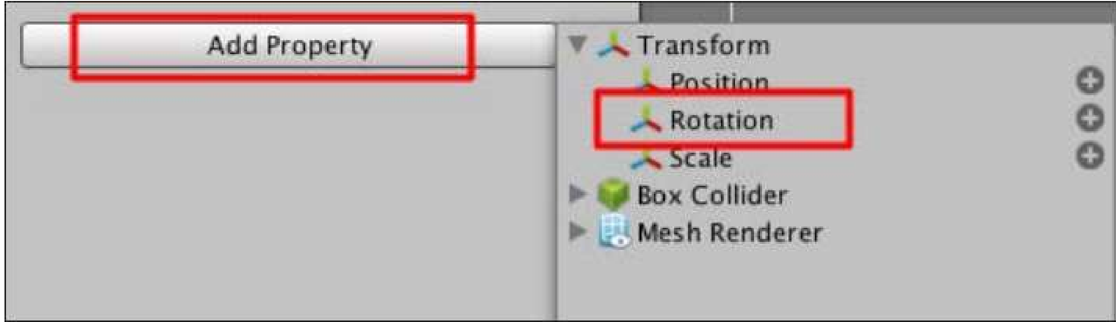


Fig. 10.6

Ahora el editor tiene en la parte izquierda la propiedad **Rotation** y en la derecha nos aparece unos puntos de color azul. Estos puntos son los puntos clave o **keyframes** que nos permiten guardar una posición, rotación o escalado en un punto de la barra de tiempo. Para saber en que momento de la animación nos encontramos, disponemos de una linea muy fina de color blanco, es la linea de tiempo. En esta parte del editor poniendo el cursor encima de el y utilizando la rueda del ratón, podemos escalar la barra de tiempo. Para movernos de derecha a izquierda debemos pulsar la ruedecilla del ratón.

Fig. 10.7

En la parte izquierda superior encontramos el control de reproducción de animaciones. Son los botones que podemos encontrar en cualquier reproductor de audio y video. El único elemento que voy a destacar es el punto rojo que nos permite grabar de una forma automática cada cambio que producimos en la parte derecha, es decir en la parte de edición de **keyframes**.

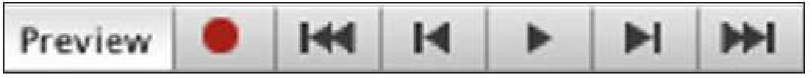


Fig. 10.8

Ahora vamos a crear una pequeña animación con el cubo para ver como funciona. En la parte de la izquierda, desplegamos las propiedades de Rotación y seleccionamos el eje Y, debes fijarte que cada eje esta representado por un **Key frame**. Eso nos permite poder animar la rotación de un objeto en los distintos eje independientemente. También

quiero destacar dos secciones que encontrarás en la parte superior debajo del control de reproducción; son el valor de **samples** que tiene la animación y el símbolo de **Keyframe** que utilizaremos para crear nuevos **keyframes**.

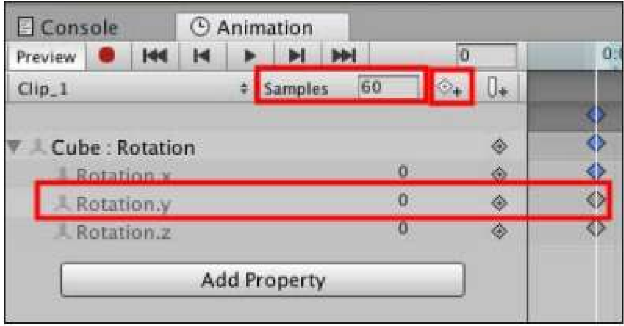


Fig. 10.9

Ahora vamos a introducir el valor 60 en la caja de textos que te muestro a continuación o puedes utilizar la linea de tiempo y arrastrarla mediante el botón izquierdo del ratón por la barra de tiempo hasta la posición de 1 minuto.

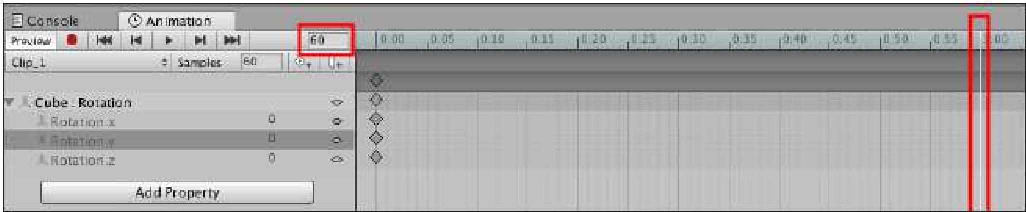


Fig. 10.10

El paso anterior es el primero de todos marcar en barra del tiempo el momento y ahora vamos a cambiar la rotación en el eje y desde la ventana Inspector que por defecto está en 0. Ahora le pondremos el valor de 360 y después de introducir el valor debemos añadir un **Keyframe.** Para ello pulsamos en el icono que se encuentra en al lado del parámetro **Samples.** Al añadir un nuevo keyframe veremos que se nos aparece nuevos puntos en la linea de tiempo pero solo en el eje de rotación y en la parte superior. Esto indica que el objeto tiene una rotación en ese momento del tiempo.

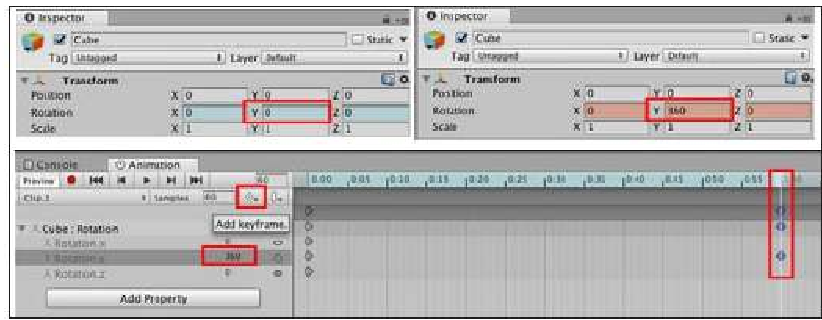


Fig. 10.11

Ahora si todo esta correctamente podemos ejecutar la animación pulsando en **play** del control de reproducción y comprobar como rota nuestro cubo en el eje y.

El editor de animación (**Animation**) tiene dos apartados o dos secciones que se utilizan en todos los programas de animación. La sección **Dopesheet** que es la que hemos utilizado ahora y el editor de curvas **Curves** que es el que vamos a ver a continuación ya que tenemos una pequeña animación.

Para acceder a los distintos apartados o secciones disponemos en la parte inferior del editor, dos botones con los respectivos nombres de las distintas secciones. Simplemente debemos pulsar en la sección que queremos.

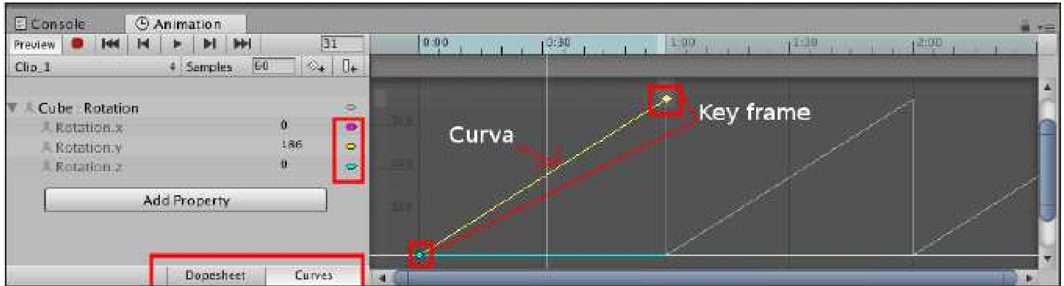


Fig. 10.12

Al acceder al apartado de edición de curvas la parte izquierda no varia nada exceptuando que los ejes de rotación de nuestro cubo toman un color distinto. Eso se produce para poder ayudarnos a identificar que curva estamos editando. En el caso de nuestro ejemplo tenemos la curva de color amarilla y las terminaciones o puntos de inicio y final son los **Keyframes** del apartado **Dopesheet.**

De momento no vamos a profundizar en esta sección y vamos a seguir con la visión general del editor Animation.

En resumen hemos creado un GameObject Cube, este objeto tiene un clip de animación en donde hemos animado su componente de transformación rotación. Ahora vamos a crear otro clip con el nombre Clip2 en el que vamos a animar su escalado en el eje (y). No te preocupes por el nombre porque luego lo cambiaremos.

Para crear un nuevo Clip de animación para nuestro Cube debemos seleccionar siempre el objeto que queremos animar y accedemos dentro de la ventana Animation en el menú donde pone Clip__1 y seleccionamos la opción Create New Clip.

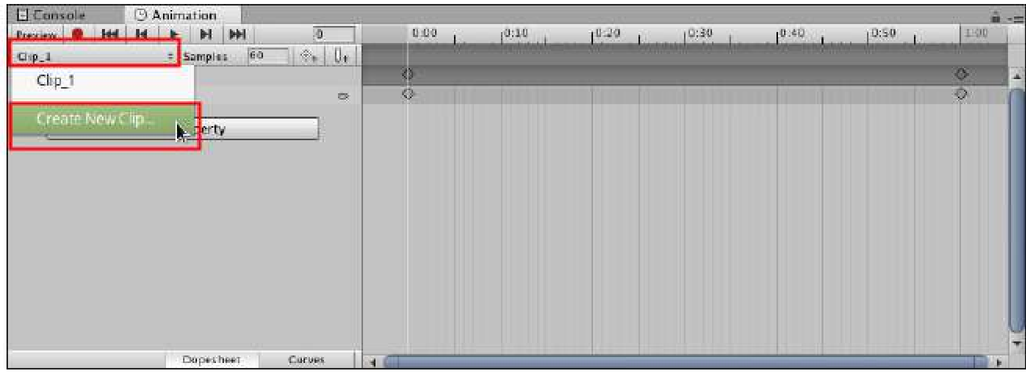


Fig.10.13

Unity nos pedirá que le demos un nombre y lo guardemos en algún lugar del sistema. En este caso el Clip de animación se llama Clip_2 y lo he guardado en la carpeta Assets > ClipAnimation que he creado anteriormente para tener un orden en el proyecto.

La ventana Animation vuelve a estar limpia de Keyframes. Ahora debemos añadir una nueva propiedad desde Add Property. En este caso vamos a escalar el cubo, entonces seleccionamos la propiedad Transform> Scale.

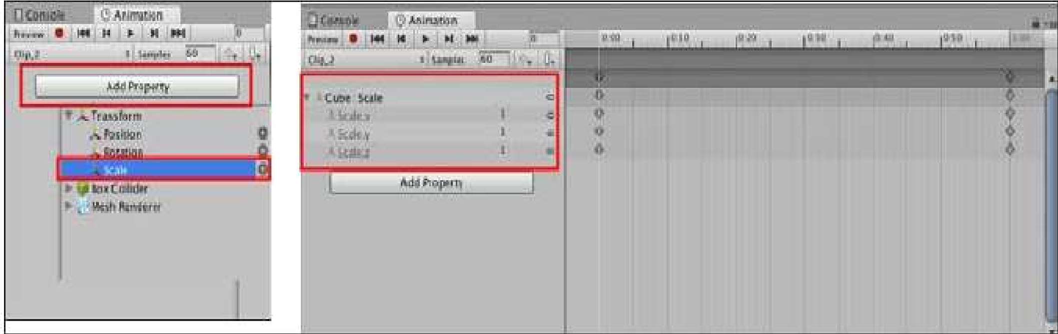


Fig. 10.14

El editor nos mostrará la nueva propiedad en la parte Izquierda y en la parte derecha ya disponemos de los Keyframes iniciales y finales.

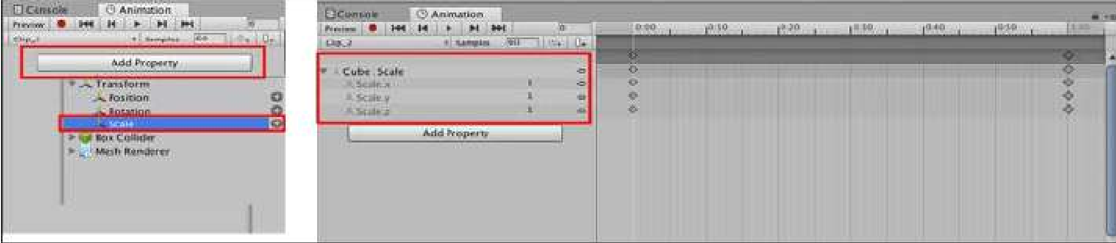


Fig. 10.15

Ahora vamos a posicionar primero la linea de tiempo a la mitad (0:30), y luego seleccionamos el eje Scale.y para determinar que eje queremos editar.

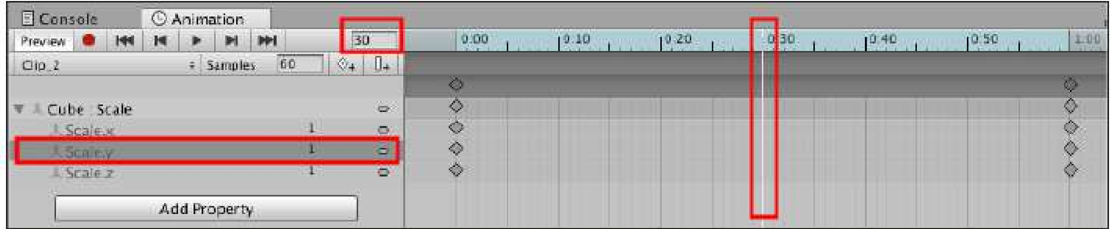


Fig. 10.16

El siguiente paso es dirigirnos a la ventana Inspector y dentro del componente Transform introducir el valor de 4 en el eje (y) del parámetro Scale. Una vez tenemos cambiado el valor pulsamos el botón de añadir Keyframe de la ventana Animation para que se guarde la información, como te muestro en la siguiente imagen.

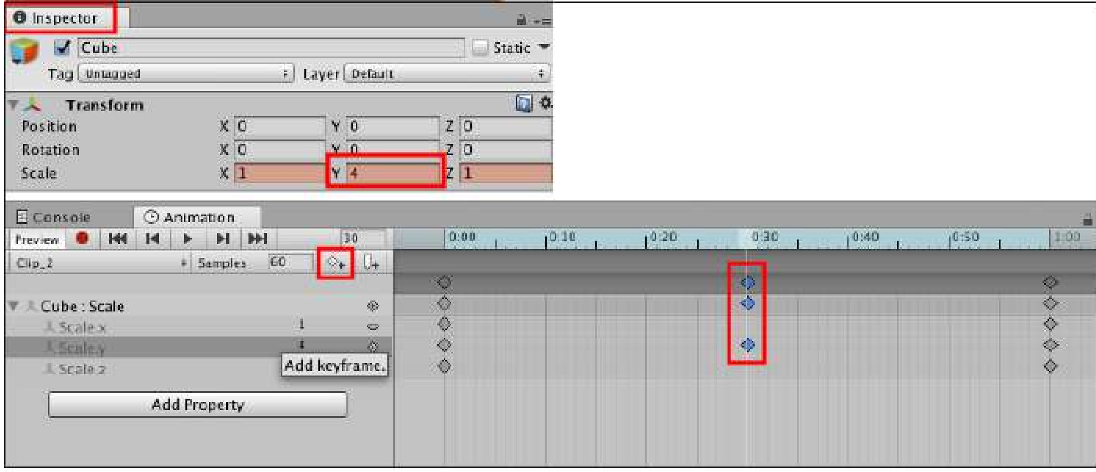


Fig. 10.17

Puedes darle al botón Play del controlador de reproducción y comprobar como la animación del cubo se estira y vuelve a su posición original.

Si en algún momento queremos eliminar un Keyframe que no nos gusta o hemos creado por error podemos eliminarlos haciendo clic encima del Keyframe que queremos eliminar, con el botón derecho del ratón y seleccionando la opción Delete Key. Esta forma de eliminar Keys elimina todos los keys creados en todos los ejes.

Bien, disponemos de dos Clips de animación dentro de un GameObject. Pero ahora:

- ¿Como puedo hacer que los clips interactúen en una escena?
- ¿Como puedo hacer que se active uno u otro según necesite?

La respuesta la encontramos en el Animation Controller (El controlador de animaciones), que nos permite gestionar diversos clips de animación dentro de un mismo GameObject.

5. Animation Controller

Antes de entrar de lleno en como funciona el **Animation Controller** vamos a ver que elementos se han creado al crear los clips de animación. En la ventana **Project** en el ejemplo anterior dentro de la carpeta **ClipAnimation** que he creado anteriormente, dispongo de dos clips de animación y un **elemento** con nombre **Cube**.

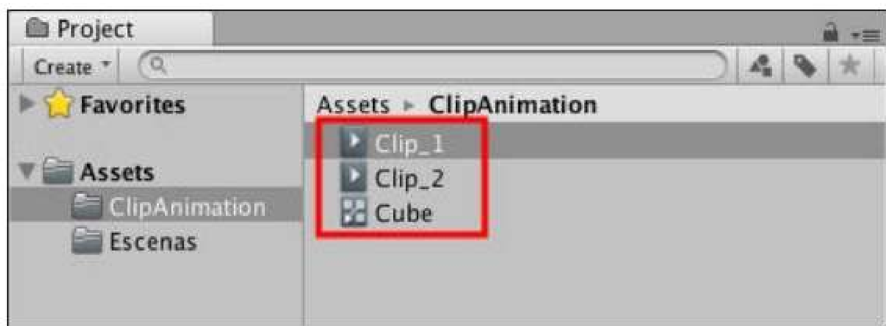


Fig. 10.18

Este **elemento** es el controlador de animaciones, aparece automáticamente cuando creamos clips de animación en Unity por el contrario si importamos modelos con animaciones externas deberemos crearlo manualmente. Este elemento permite unir estas animaciones dentro del objeto. Para ello debemos hacer doble clic encima de el para acceder a la ventana **Animator**.

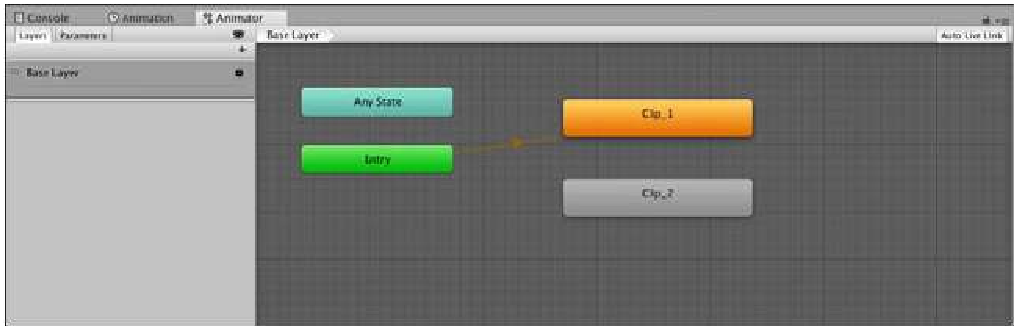


Fig. 10.19

Ventana Animator

La ventana **Animator** tiene dos secciones principales: el área principal en donde se ven representados los clips de animación por rectángulos, que son en realidad nodos que representan estados, con un fondo de cuadricula gris oscuro y el panel izquierdo con dos apartado; **Layers y Parameters**.

En el área principal podemos crear nuevos nodos de estado, haciendo clic con el botón derecho encima de la cuadricula y seleccionando la opción **Create States> Empty**.

Para desplazarnos por la vista debemos pulsar el botón central del ratón o la opción **Alt / Opción** y arrastrar el cursor para desplazarse por la vista.

Si hacemos clic encima de los nodos como por ejemplo el **Clip_1** ,podemos editarlos en la ventana Inspector. Como por ejemplo cambiar el nombre, en la siguiente imagen he cambiado el nombre del **Clip_1** por **Rotacion** y te muestro como seleccionando el nodo nos aparece las propiedades del clip en la ventana Inspector. Recuerda estoy cambiando el nombre que se muestra en el nodo, no el nombre del clip. También voy a cambiar el nombre del nodo **Clip_2** por **Escalado**.

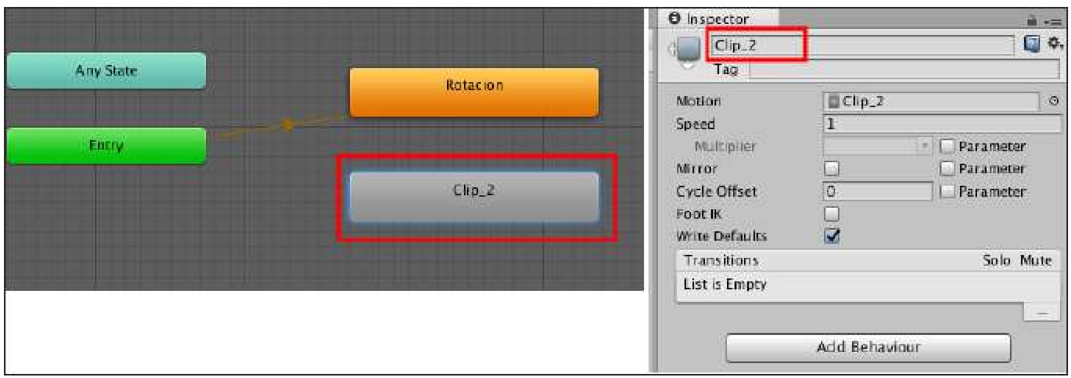


Fig. 10.20

6. Máquina de estados

Seguimos en la misma ventana **Animator** pero comúnmente cuando nos referimos a un personaje u objeto del juego animado, que contiene varias animaciones diferentes, correspondientes a varias acciones del juego, hablaremos de la maquina de estados. En este apartado vamos a ver de un modo general las funciones más básicas.

Parámetros

Son variables que se definen dentro de un controlador de animación al que podemos acceder y asignar valores desde scripts.

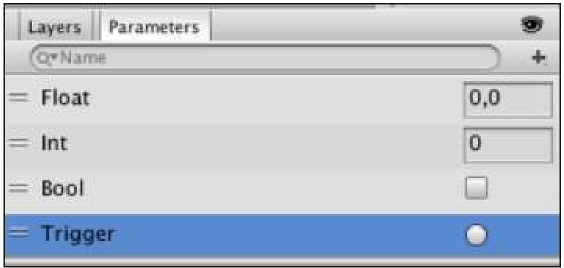


Fig. 10.21

- **Int** – numero entero
- **Float** – numero decimal
- **Bool** – valor verdadero o falso se representa con una casilla de verificación.
- **Trigger** – un disparador de tipo booleano y que se representa con un botón circular.

El panel de la izquierda se puede cambiar entre la vista Parámetros y la vista de Capas. La vista de parámetros te permite crear, ver y editar los parámetros del controlador **Animator**. Estas son variables que se definen como entradas en la **máquina de estado**. La maquina de estados es todo el conjunto que caracteriza la conexión de estos nodos, es decir cada nodo supone un estado del objeto y la gestión de estos constituye la maquina de estados.

Para agregar un parámetro, hacemos clic en el ícono (+) y seleccionamos el tipo de parámetro en el menú, por ejemplo **Bool.**

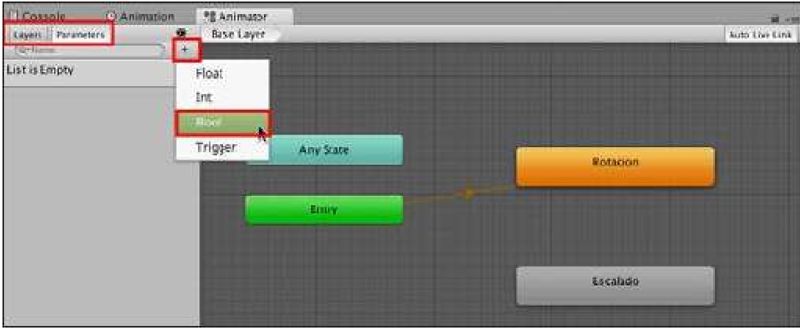


Fig. 10.22

Para eliminar un parámetro, seleccionamos el parámetro hacemos clic con el botón derecho del ratón y seleccionamos **Delete**. Estos parámetros nos permiten determinar un valor para identificar en que momento se utiliza un estado u otro. Esto lo veremos mas adelante por ahora debes saber que existe.

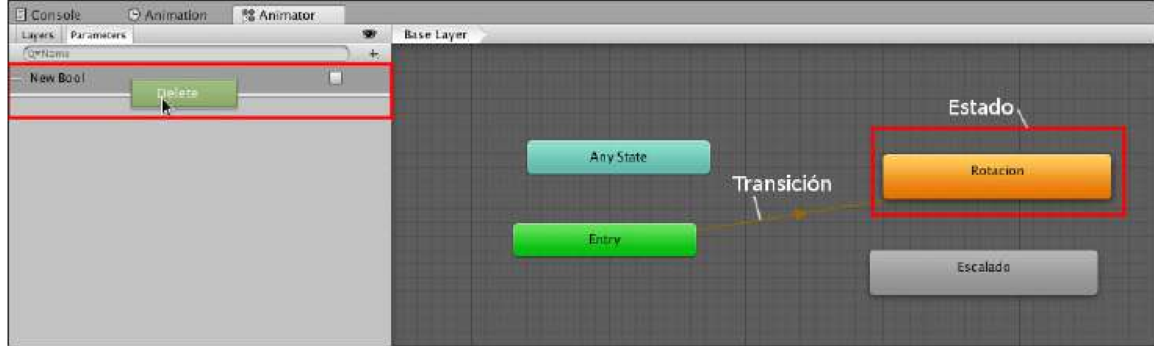


Fig. 10.23

Transiciones

Las transiciones nos ayudan a simplificar la gestión de los estados de una forma lógica y ordenada. En el área principal tenemos los nodos y siempre tendremos un nodo de Entrada y otro de salida. El nodo de entrada se utiliza para hacer la transición inicial al estado principal. Antes de pasar la información del nodo de entrada al de destino se comprobará que cumpla las condiciones establecidas. Las condiciones vienen dadas por los parámetros que hemos visto anteriormente.

Para empezar el nodo de entrada siempre tiene un estado predeterminado por el que va a empezar siempre. Normalmente el estado predeterminado de un player en todos lo juegos es el estado IDLE que es cuando el personaje esta en reposo, respirando o en ocasiones mirando a los lados.

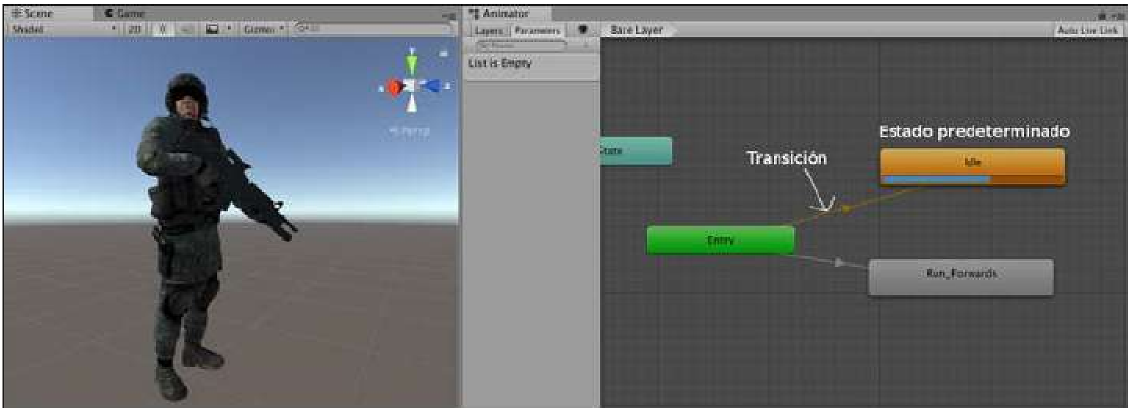


Fig. 10.24

Luego, se puede agregar transiciones adicionales desde el nodo Entrada a otros estados, para controlar si la máquina de estado debe comenzar en un estado diferente.

También existe un nodo de salida que se usa para indicar que la máquina de estados debe salir.

Podemos crear una configuración tan extensa como estados tengamos y creando transiciones para ambos lados, es decir podemos pasar de un estado en reposo a un estado de caminar y volver otra vez al estado de reposo.

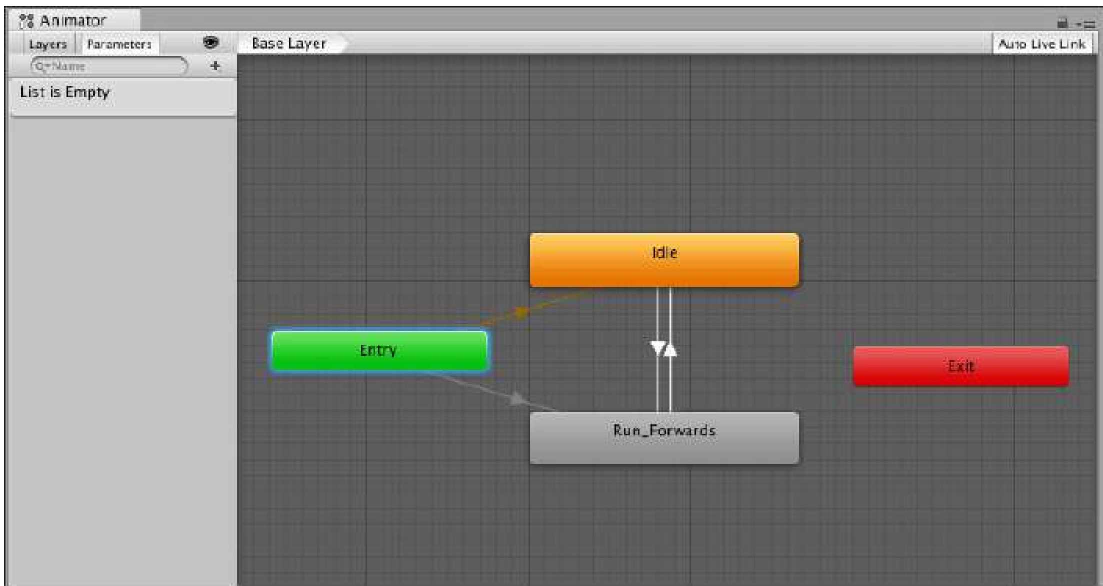


Fig. 10.25

Para crear una transición de un nodo a otro, hacemos clic encima con el botón derecho y seleccionamos la opción **Make Transition** después debemos hacer clic encima en el nodo de destino.

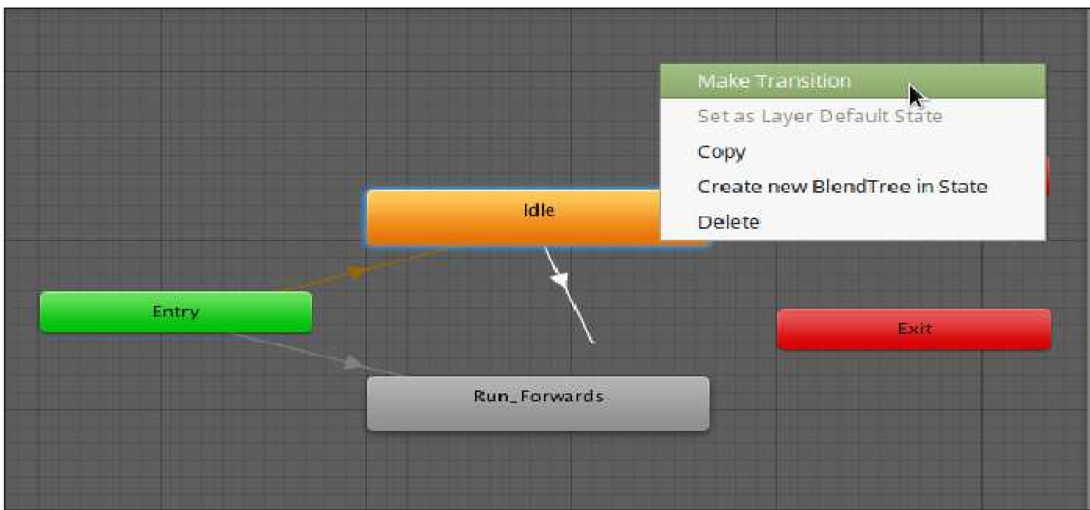


Fig. 10.26

En el próximo apartado veremos como desarrollar los estados básicos de un personaje para poder controlar sus clips dentro de una escena.

7. Proyecto de un soldado

En este proyecto vamos a trabajar con un personaje que contiene animaciones y vamos a ver como podemos crear con el sistema de animación que tiene Unity un conjunto de estados relacionadas entre si para que el personaje pueda ser controlado mediante un script.

Para facilitar la explicación y que puedas entrar en detalle dispones dentro del material que acompaña la obra un paquete de assets preparados para trabajar con el proyecto. El paquete pertenece al capitulo 10 y se llama **Proyecto_Soldado.unitypackage.**

Crea un proyecto nuevo 3D, ponle el nombre que desees en este caso le he llamado ProyectoSoldado. Una vez cargado el proyecto accedemos al menú principal y seleccionamos Assets > Import Package > Custom Package y buscamos el paquete y seleccionamos el Proyecto_Soldado.unitypackage. Cargamos todo el paquete, que contiene varias carpetas que se te muestran en la ventana Project entre ellas la carpeta escenas a la que accederemos y haremos doble clic en Escena1.

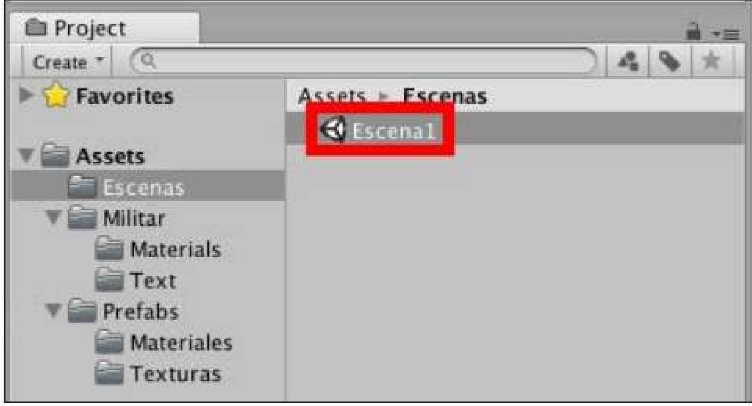


Fig. 10.27

Una vez cargada la escena veremos como en la ventana Scene nos aparece un escenario compuesto por un suelo, varios elementos que son en realidad dos cubos uno grande y otro pequeño que están duplicados y un soldado en el centro de la escena que es con el que vamos a trabajar.

Fig. 10.28

El modelo

Si seleccionamos al objeto **ArmyPilot** desde la ventana **Hierarchy** y desplegamos la Jerarquía de este objeto veremos los siguientes **subobjetos**:

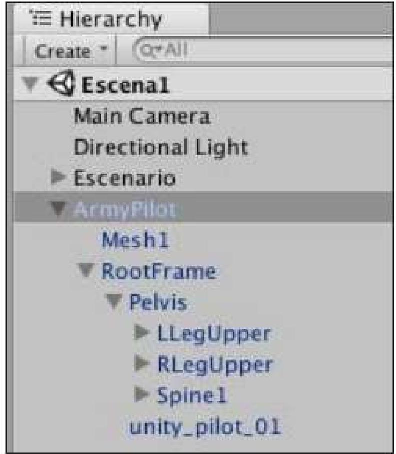


Fig. 10.29

El objeto padre es el llamado **ArmyPilot** y si miramos sus componentes en la ventana inspector veremos que contiene un **Transform** que es el componente que tienen todos los objetos por defecto y un componente **Animator** que veremos en profundidad más adelante.

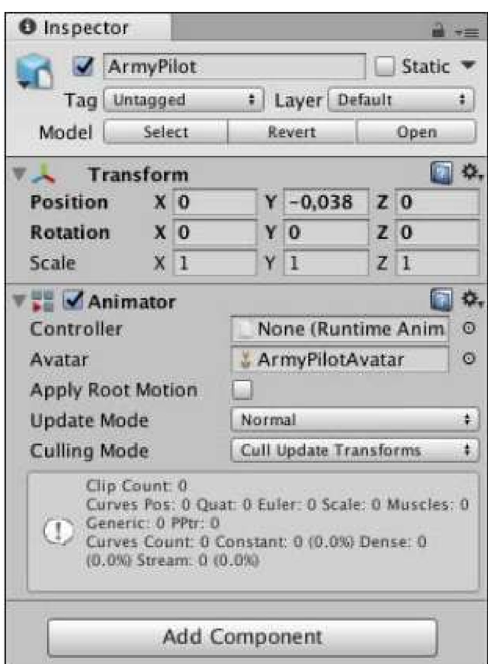


Fig. 10.30

El siguiente **subobjeto** se llama Mesh1 y contiene la malla y los materiales del objeto. En este caso lleva 3 materiales distintos uno para el cuerpo otro para la cabeza del personaje y otro material para el arma.

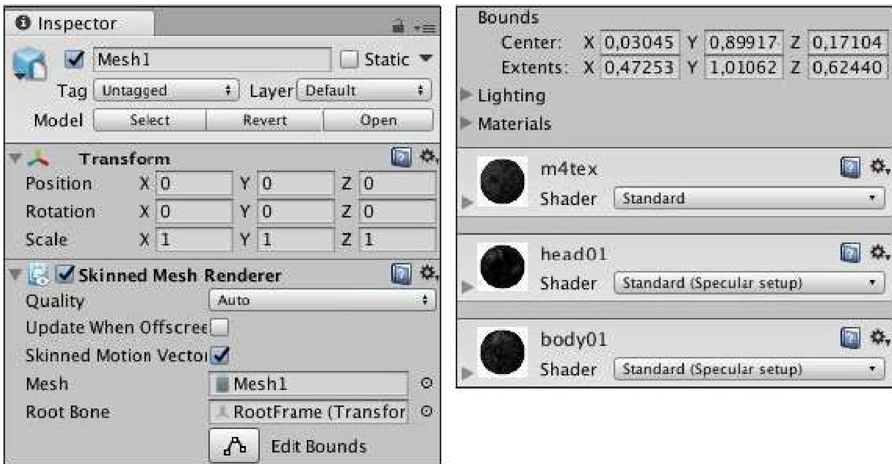


Fig. 10.31

El otro subobjeto se llama **RootFrame** y es padre de otros objetos que componente la jerarquía o estructura de huesos del personaje. Sin esta estructura no podríamos mover al personaje.

Configuración de la importación del Modelo

Volvemos a seleccionar el objeto padre con nombre **ArmyPilot** desde la ventana **Hierarchy** y accedemos a los componentes para hacer clic en la opción Select como te muestro en la siguiente imagen.

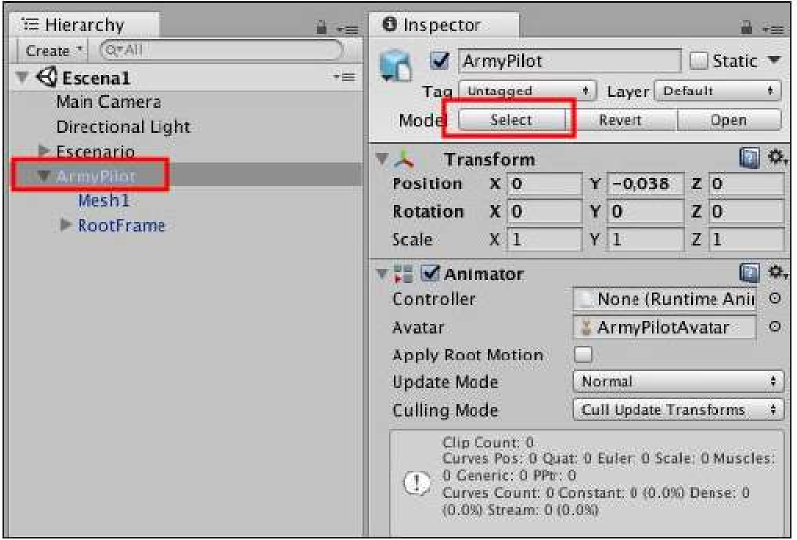


Fig. 10.32

De esta manera podemos ver dentro de la ventana Inspector la configuración de la importación del modelo. **En Unity existen cuatro opciones de importación de modelos :**

- La pestaña Model que tiene opciones para configurar el modelo 3D, como son los materiales, la malla, la escala del modelo, las normales etc.

- La pestaña Rig que tiene opciones para configurar que la estructura de huesos del modelo 3D sea compatible con la animación.
- La pestaña Animation para poder importar los clips de animación, en el caso de que el modelo las tenga. En el caso del soldado si que las tiene.
- La pestaña Materiales que tiene opciones para configurar y ajustar los materiales exportado del modelo 3D.

De estas cuatro opciones vamos a entrar en detalle con las de Rig y Animación pues es un tema muy extenso y existen distintas formas de importar modelos con animación. En este capitulo se va a intentar enfocar en el aprendizaje de como crear nuestro Player en 3ª persona aprovechando un modelo con animaciones ya creadas.

Fig. 10.33

Opción Rig

Esta opción nos permite asignar o crear que tipo de avatar va a ser el modelo importado. Esto quiere decir que depende de como se ha creado el modelo definiremos el tipo de animación que le vamos a asignar.

Existen tres tipos de formas de animar en Unity;

- **Generic:** Cuando tengas un personaje que no es humano, un cuadrúpedo por ejemplo o cualquier tipo modelo que no se asemeje a un humano, esta es la opción recomendada. Una vez seleccionada esta opción necesitas identificar un hueso en el menú desplegable para usarlo como nodo raíz. En estos casos te recomiendo que utilices modelos exportados en formato **fbx**. Este formato suele funcionar muy bien con esta opción en Unity. También puedes seleccionar esta opción cuando importes un modelo que disponga de animaciones creadas desde otros programas de edición 3D.
- **Humanoid:** Si tu modelo tiene dos piernas, dos brazos y una cabeza, entonces tienes un Humanoid. Esta opción es la que te ofrece Unity para crear un avatar humano, pero tengo que decir que el modelo debe coincidir en una serie de requisitos para que la jerarquía osea coincida.
- **Legacy:** Escoge esta opción si quieres usar el sistema de animación para la version 3.x. Unity desaconseja está opción para proyectos nuevos.

Opción Animations

En esta opción podemos ver la configuración de los clips de animación del modelo. En este apartado puedes ver en una lista todos los clips de animación que se han importado en el modelo. Puedes seleccionar uno de los clips y ver la animación en la parte inferior donde se muestra dentro de una ventana el modelo realizando la acción.

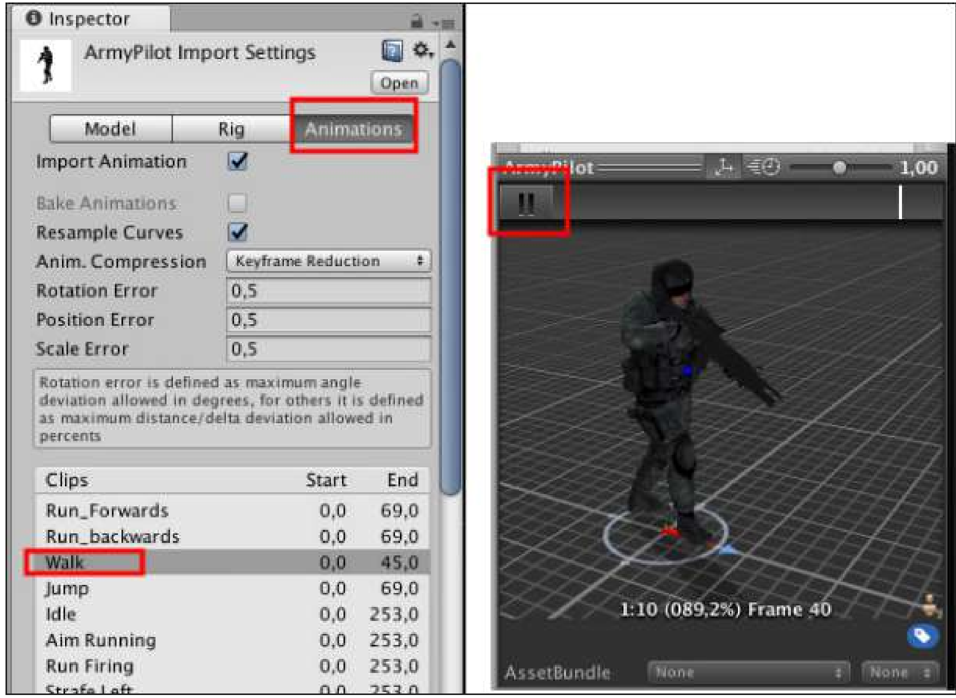


Fig. 10.34

Este apartado dispone también de un sistema para manipular los clips de animación, es como un editor de animaciones, por el momento y para no complicar la explicación vamos a ver dos parámetros que son importantes en la importación de modelos, que son el **Loop Time** y el **Loop Pose.**

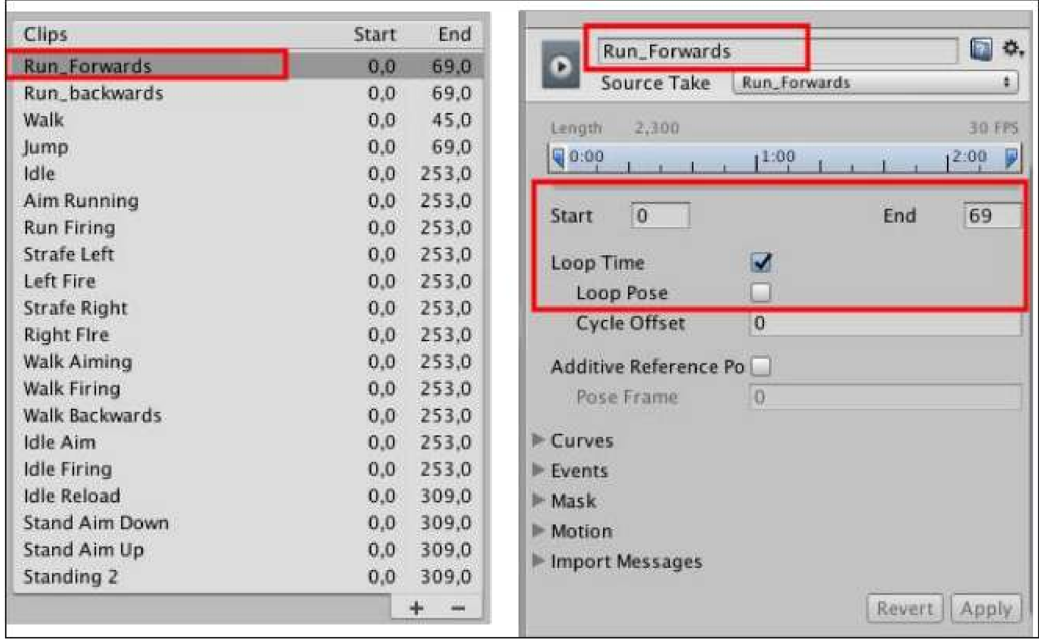


Fig. 10.35

Si seleccionamos el clip **Run_Forwards** y bajamos hasta las propiedades del clip de animación encontramos los siguientes parámetros, para configurar este clip. El objetivo es entender que tipo de animación realiza, en este caso correr hacia delante y esta es una animación que en principio se va a repetir constantemente, por lo tanto será un loop. En la imagen de arriba suelo fijarme en el tiempo que dura la animación que viene marcado en los parámetros **Start** y **End**, si el parámetro **Loop Time** debe estar activado o no según si la animación debe repetirse en el juego y el parámetro **Loop Pose** se utiliza cuando el loop no es del todo correcto y Unity crea una pose especifica para que el loop sea más fluido.

Siempre que cambiemos parte de la configuración de importación de un modelo Unity nos pedira que guardemos los cambios.

Character Controller en el Soldado

Una vez hemos visto de una forma general el modelo y la configuración que lleva al ser importado, vamos a añadirle un componente **CharacterController** para poder controlar el objeto.

Si no recuerdas muy bien que es esto del **CharacterController** deberías volver al capitulo 7 Creación de un Player.

Asi pues seleccionamos el Objeto ArmyPilot y en la ventana Inspector añadimos el componente Character Controller desde el botón Add Component > Physics > Character Controller. Este componente se representa como una capsula de alambre de color verde a la cual debemos re- situar para que coincida con nuestro modelo. En la siguiente imagen se ha movido el Character Controller con un valor de 1,05 en el eje Y y un valor en el radio de 0,8. Este componente lo ponemos de esta forma provisionalmente y veremos cuando estén las animaciones puestas como colocarlo correctamente.

Fig. 10.36

Ahora para que podamos ver el player en 3 persona vamos a posicionar la cámara de nuestra escena de la siguiente manera. Selecciona la cámara **MainCamera** de la ventana **Hierarchy** y le damos los siguientes valores en posición y rotación:

Position (x=0 y=1,689 z=-2,186)

Rotation (x=11,943 y= 0 z=0)

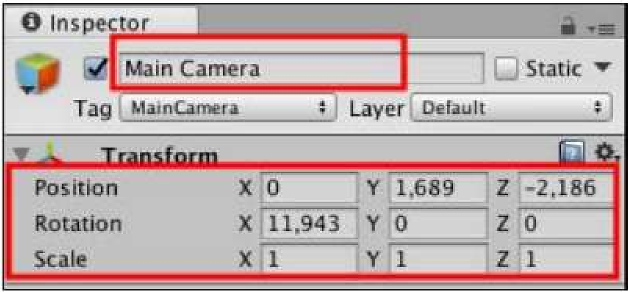


FIg. 10.37

Luego arrastramos **MainCamera** encima del objeto **ArmyPilot** para que pase a ser hijo, como te muestro en la **siguitente** imagen.

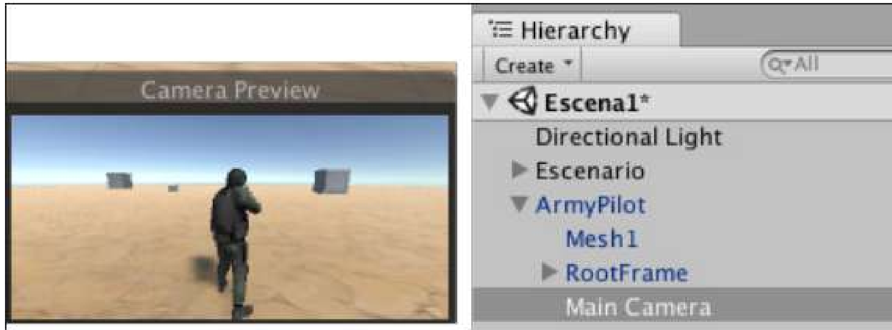


Fig. 10.38

Ahora vamos a crear una carpeta para guardar los scripts y creamos un script con nombre **ControlPlayer** y lo añadiremos al **gameObject ArmyPilot**.

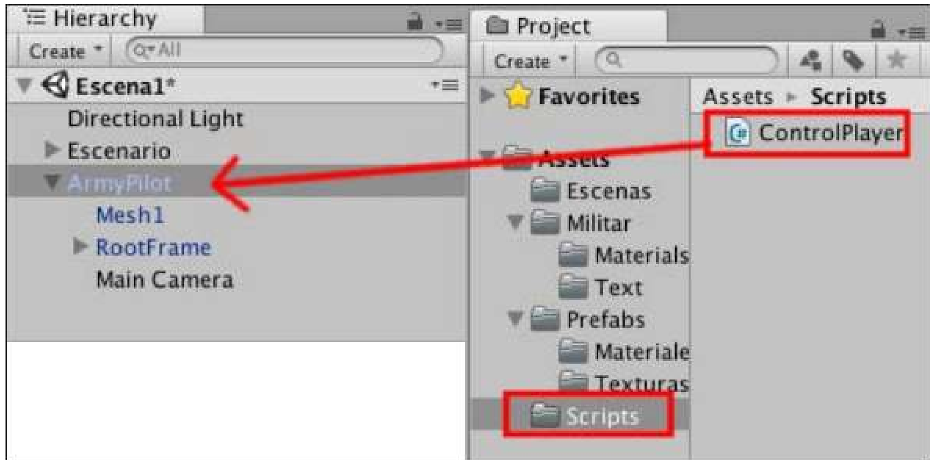


Fig. 10.39

A continuación voy a explicarte que queremos conseguir con este script y como vamos a combinarlo con las animaciones.

En primer lugar el **CharacterController** nos permite controlar el movimiento de nuestro personaje utilizando físicas y no hay que confundir con las animaciones, es decir nuestro personaje será controlado por medio del **CharacterController** como se hizo con una capsula en el capitulo 7 y luego utilizaremos el componente **Animator** para controlar las animaciones de manera que estos dos componentes trabajen de forma sincronizada.

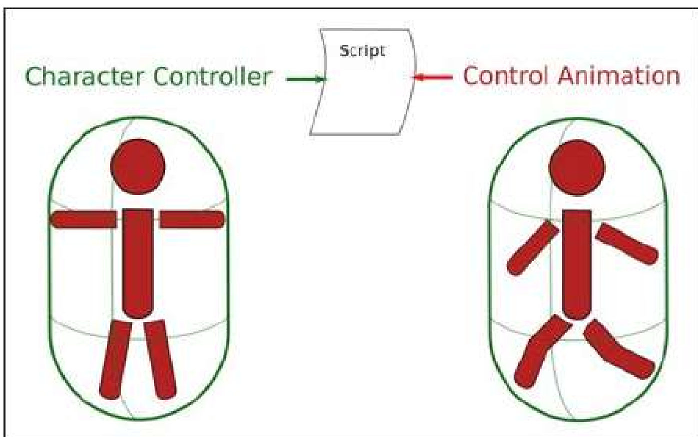


Fig. 10.40

Como te muestro en la imagen anterior tenemos que conseguir que se active la animación correspondiente según el movimiento que le demos al **character controller**.

Si ya hemos creado el script **ControlPlayer** y se lo hemos añadido a nuestro player **ArmyPilot** vamos a editarlo con **Monodevelop** haciendo doble clic encima del script.

Script: ControlPlayer.cs

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

public class ControlPlayer : MonoBehaviour
{
   private Vector3 direccion;
   private CharacterController controlador;
   public float walkingSpeed;
   public float runningSpeed;
   public bool isWalking;
   private Vector3 momentoSpeed;

   void Start (){
      this.controlador = gameObject.GetComponent<CharacterController> ();
   }
   void Update ()     {

      this.direccion = new Vector3 (Input.GetAxis ("Horizontal"), 0, Input.
GetAxis                        ("Vertical"));
      this.isWalking = !Input.GetKey (KeyCode.LeftShift);

      if (this.isWalking)
      {
        this.momentoSpeed = this.walkingSpeed * this.direccion;
      }

      else
      {
        this.momentoSpeed = this.runningSpeed * this.direccion;
      }
      this.controlador.SimpleMove (this.momentoSpeed);
   }
}
```

Primero las variables: tenemos un variable privada de tipo Vector3 que llamamos **direccion** en donde vamos a guardar los **inputs** para el vector. Otra variable privada de tipo **CharacterControler** con nombre controlador donde guardaremos la referencia del componente de este tipo.

Las siguientes variables son de tipo **float** y la dos son de tipo pública para poderlas ver representadas en la ventana inspector estas variables nos servirán para darle una velocidad a nuestro **CharacterController** para andar y para correr al que les he puesto el nombre de **walkingSpeed** (Andar) y **runningSpeed** (correr).

También he creado una variable pública de tipo booleana con nombre isWalking para saber cuándo camina y cuándo no. Para terminar esta primera parte he creado otra variable privada de tipo Vector3 para almacenar en que momento se camina y en que momento se corre y la he llamado momentoSpeed.

En la función Start() desde la variable controlador vamos a acceder al componente CharacterController.

En la función Update() vamos a crear el control para la direccion para ello diremos que this.direccion es igual a un nuevo Vector3 y en el eje x utilizaremos el método GetAxis (“Horizontal”) para poder mover con las teclas de izquierda a derecha, en el eje y lo dejamos en 0 y en el eje z utilizamos el método GetAxis (“Vertical”) para controlar el movimiento con las teclas arriba y abajo.
En la variable anterior hemos guardado el vector dirección para que el personaje pueda moverse pero tenemos dos tipos de velocidad una para correr y la otra para andar para saber cuando ando y cuando corro he utilizado la variable booleana isWalking para decirle que cuando pulse la tecla Shift de mi teclado sea falsa es decir no camina, para ello igualamos this.isWalking y negamos ! su valor cuando pulsamos la tecla shift . En este caso para encontrar la tecla he utilizado el método GetKey y entre paréntesis accedemos a KeyCode y enlazamos con el valor que necesitamos en este caso la tecla shift izquierda.
Ahora que sabemos cuando está caminando y cuando no vamos a una condición si isWalking es verdadero se cumplirá que this.direccion tendrá una velocidad de caminar por eso mismo multiplicamos la variable this.direccion, con la variable this.walkingSpeed. Todo esta operación la guardamos en la variable this.momentoSpeed. Si la condición no se cumple entonces diremos que this.direccion tiene una velocidad de this.runningSpeed, que también guardamos en la variable this.momentoSpeed.
Ahora que sabemos cuando camina y cuando no y que velocidad tiene uno y otro solo nos falta crear el movimiento para ello seleccionamos la variable que contiene el componente CharacterController y utilizamos un método sencillo, el SimpleMove al que le tenemos que pasar un Vector3 que en este caso va a ser el momentoSpeed que contiene la dirección y la velocidad según se camine o se corra.
Guarda el script antes de volver a Unity para que tenga efecto.

Una vez volvemos a Unity selecciona a el objeto **ArmyPilot** y en la ventana **Inspector** en el componente **Script** verás que aparecen los atributos **Walking Speed y Running Speed.** Antes de ejecutar la escena debemos darle un valor a cada atributo yo te recomiendo para **WalkingSpeed = 2 y para RunningSpeed = 4.** Si todo está correcto ejecuta la escena y comprueba que las teclas coinciden con el movimiento del personaje, también debes comprobar que cuando mantienes pulsada la tecla Shift mientras pulsas en una de las direcciones la velocidad se acelera.

Fig. 10.41

Animator Controller

La primera parte que se encarga del movimiento ya la tenemos creada. Te habrás dado cuenta que el modelo esta estático y no tiene animaciones, de esto nos vamos a encargar a continuación. Como te he explicado antes debemos combinar las animaciones con el movimiento y que todo encaje perfectamente, para ello primero vamos abrir la ventana **Animator** si no la has abierto. Para abrir esta ventana vamos al menú principal y accedemos a **Window > Animator.** Por comodidad yo he anclado esta ventana al lado de la ventana consola como te muestro en la siguiente imagen.

Fig. 10.42

Como veras esta ventana está vacía, eso es debido a que esta ventana esta relacionada con el componente **Animator** que tenemos en el modelo **ArmyPilot** en donde tenemos un parámetro que hace referencia al control de los clips de animación. Este parámetro es **Controller** y actualmente debes de tenerlo vacío porque no disponemos de ningún controlador de animación.

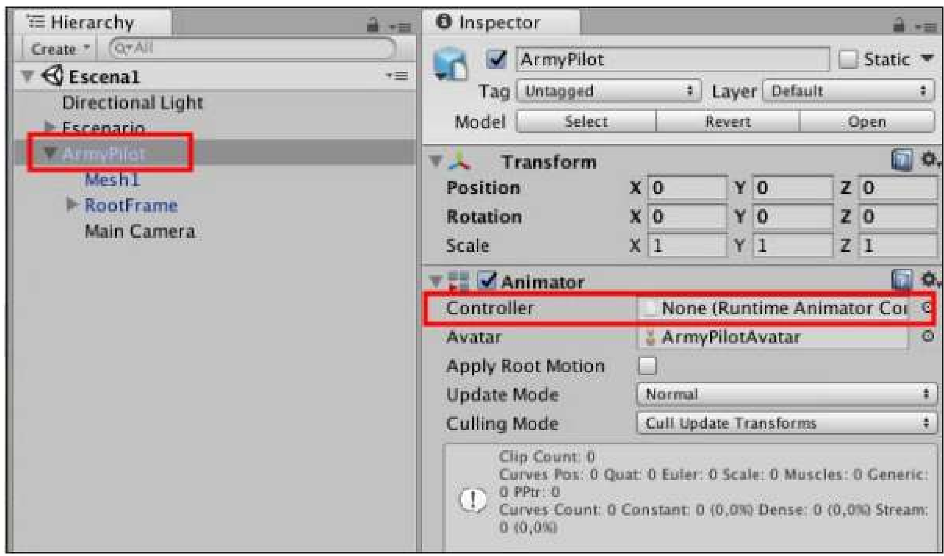


Fig. 10.43

Ahora vamos a crear un controlador de animación para ello en la ventana **Project** creamos una carpeta nueva que llamaremos **Animators.** Seleccionamos esta nueva carpeta y desde el botón Create seleccionamos la opción Animator Controller. Se nos creara un controlador de animación al que le ponemos el nombre de ArmyPilot. Puedes ponerle el nombre que quieras, pero intenta que sea descriptivo.

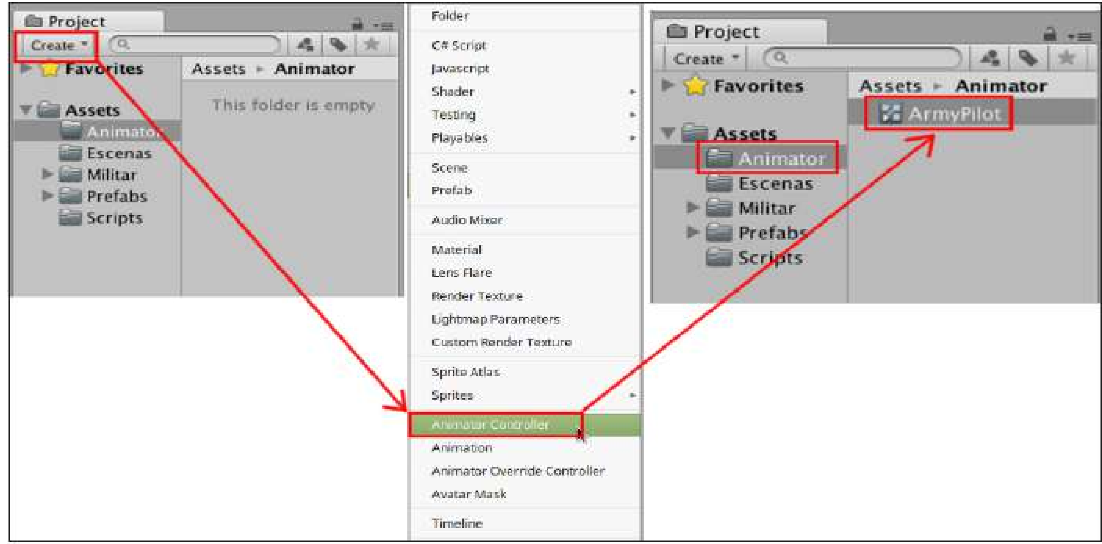


Fig. 10.44

Ya tenemos un **AnimatorController** que vamos a utilizar para gestionar las animaciones de nuestro player **ArmyPilot**. Para que los cambios que tengan efecto debemos arrastrar el **AnimatorController** hasta la ventana Inspector dentro del componente **Animator** en el parámetro **Controller.**

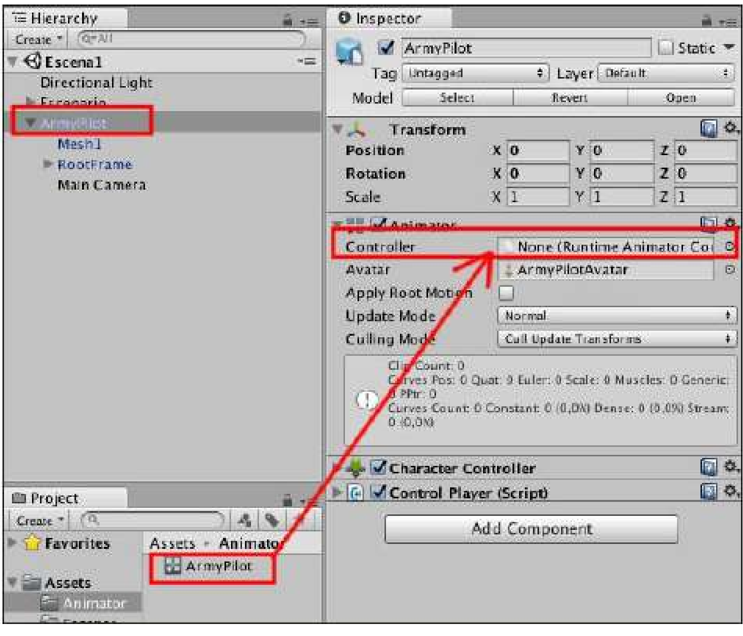


Fig. 10.45

Si te resulta engorroso tener que arrastrar elementos también puedes seleccionar el objeto ArmyPilot desde la ventana Hierarchy acceder a sus componentes desde la ventana Inspector y dentro del Componente Animator en el parámetro Controller pinchar encima del símbolo con forma de diana y en el nuevo menú que aparece seleccionar el Animator Controller con nombre ArmyPilot.

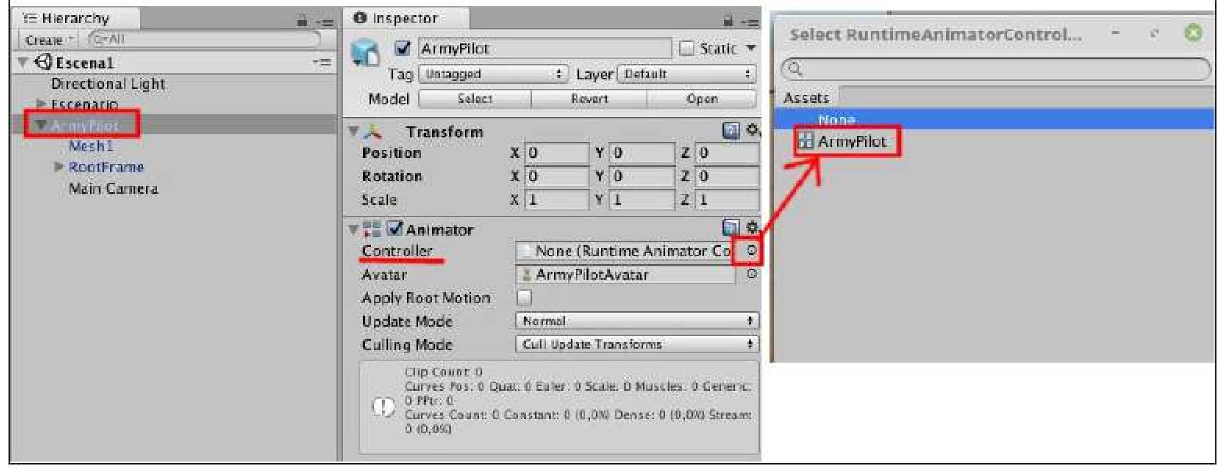


Fig. 10.46

Máquina de Estados

Ahora ya tenemos un controlador para nuestras animaciones ahora debemos introducir las distintas animaciones con ayuda de la ventana **Animator**. En el capitulo en el apartado de ventana Animator se explica o se introduce a la maquina de estados. Esto es lo que vamos a realizar a continuación cada clip de animación se va a considerar como un estado nuevo.

Por defecto en la ventana **Animator** tenemos tres estados uno de color azul que no vamos a utilizar, el estado **Entry** que es el estado de inicio y el estado **Exit** para finalizar.

Fig. 10.47

Para crear el primer estado o un nuevo estado hacemos clic con el botón derecho del ratón encima de la cuadricula gris y nos aparece un menú en donde seleccionamos la opción **Create State>Empty.**

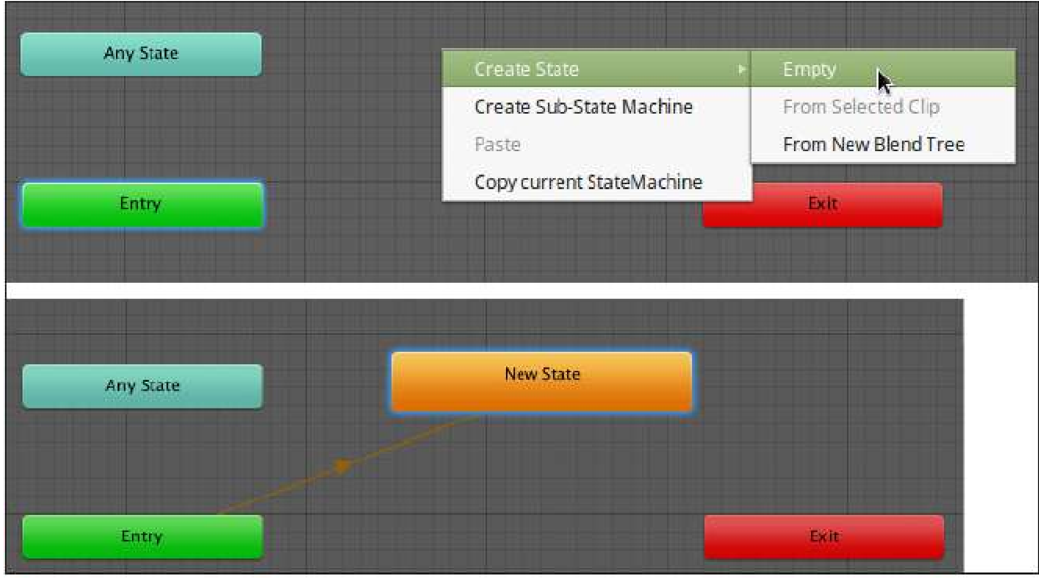


Fig. 10.48

El nuevo estado está vació y al ser el primero vemos que aparece una linea que la une con el estado de entrada. Entonces si está vacío: **¿Cómo podemos incluir un clip de animación?** Primero seleccionamos el nuevo estado y luego nos dirigimos a la ventana inspector en donde podemos ver las opciones de configuración de este estado.

Fig. 10.49

En el inspector podemos cambiar el nombre accediendo a la caja en donde por defecto se llama **New State** , en este caso le llamaremos **Idle** que es como se suele llamar al primer estado de un personaje o estado en reposo.

La siguiente opción es donde añadiremos el clip de animación que queremos para este estado, si hacemos clic encima del símbolo con forma de diana podemos seleccio-

nar en la ventana que aparece la animación que nos interesa, en este caso el que tiene el nombre **Idle**.

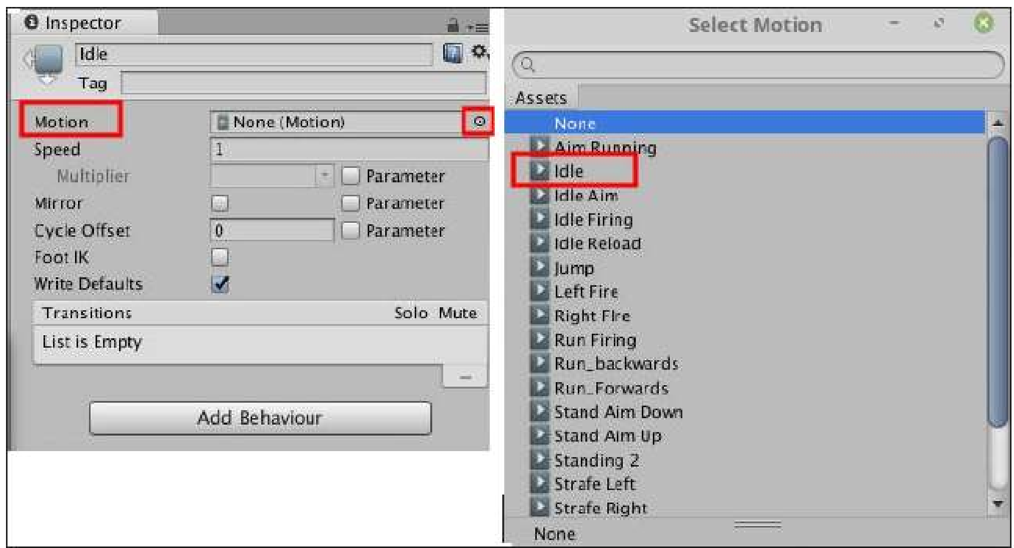


Fig. 10.50

Ahora nuestro estado que se llama **Idle** tiene una animación **Idle**. En el caso de que esta animación no estuviera bien hecha la siguiente propiedad es **Speed** y se encarga de acelerar la animación siempre que el valor sea mayor de 1, siendo 1 el valor por defecto.

¿Pero como sabemos que la animación es correcta? Podemos ejecutar la escena ahora mismo y ver que nuestro personaje ya tiene puesta la animación **Idle.** También puedes ver como la maquina de estados carga la animación de este estado representado con una barra de cargado que nos da una idea del tiempo que dura ese estado.

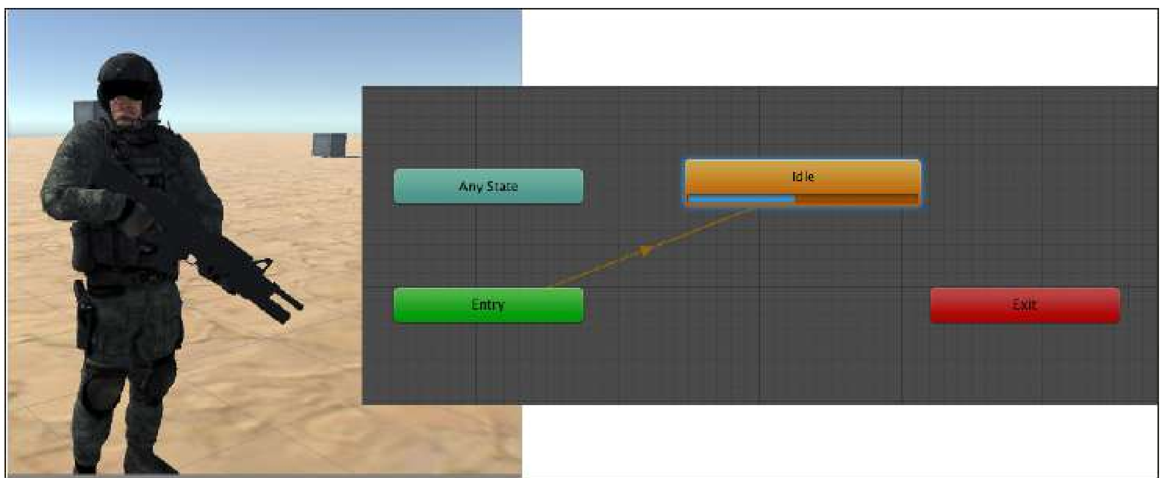


Fig. 10.51

Esto va tomando forma, ahora vamos a crear los estados de **Walk** (Caminar) y de **Run** (Correr), como hemos hecho con el estado **Idle**. Para ello hacemos clic con el botón derecho del ratón encima de la cuadricula gris y nos aparece un menú en donde se-

leccionamos la opción **Create State>Empty.** **Seleccionamos este estado, le cambiamos el nombre por el de Walk en la ventana Inspector y le añadimos el clip de animación Walk.**

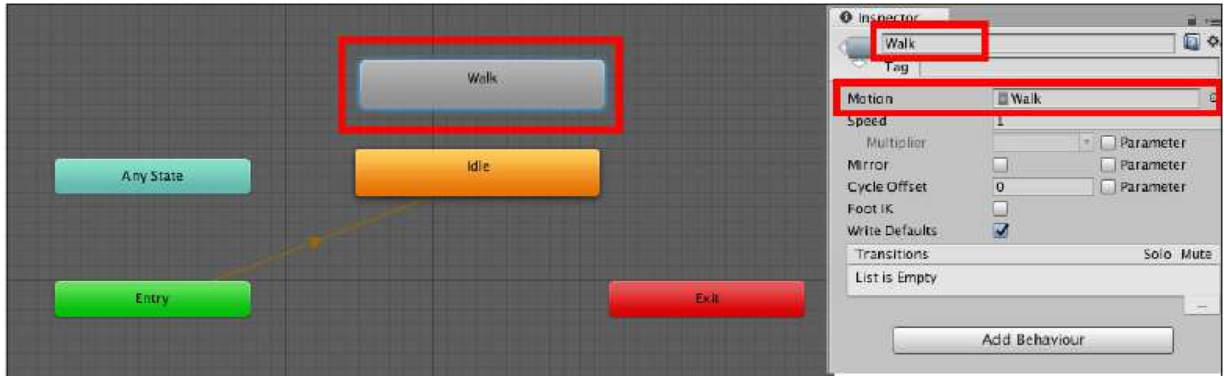


Fig. 10.52

En este caso al crear otro estado este no se conecta con el estado de entrada porque solo ocurre cuando creamos el primero. Ahora deberás crear el **Run** tal y como se a explicado al crear el estado **Idle**. En el caso de **Run** en la opción **Motion** es decir el clip de animación que corresponde a este estado es el que tiene el nombre de **Run_Forwards**.

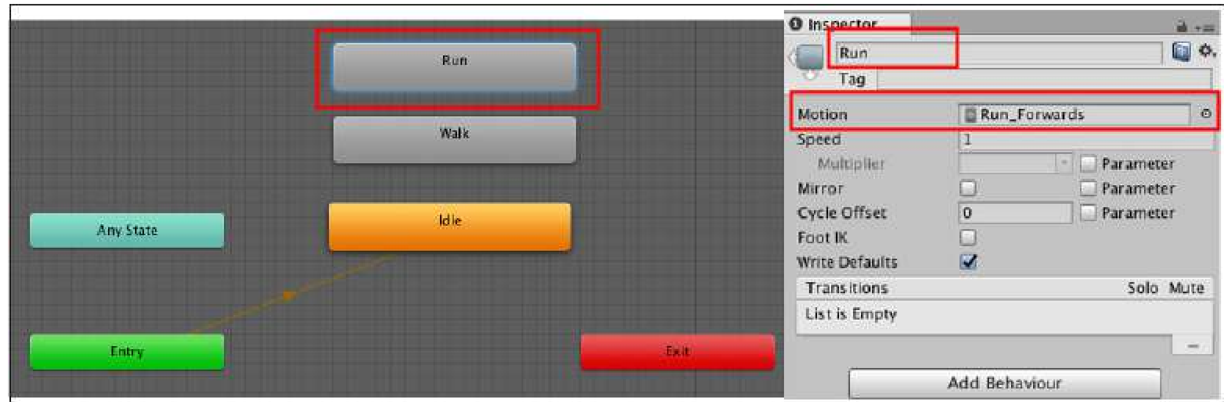


Fig. 10.53

Una vez hemos creado los tres estado **Idle, Walk y Run** debemos crear transiciones entre es decir conexiones que nos permitan acceder de Idle a Walk, de Walk a Run, de Idle a Run, etc.

Para crear una transición de **Idle** a **Walk** primero seleccionamos el estado **Idle** pulsamos el botón derecho del ratón y en el menú que aparece seleccionamos la opción **Make Transition.** Seguidamente aparecerá una flecha que sigue la dirección de nuestro cursor, debemos indicarle el estado de destino y seleccionamos el estado Walk.

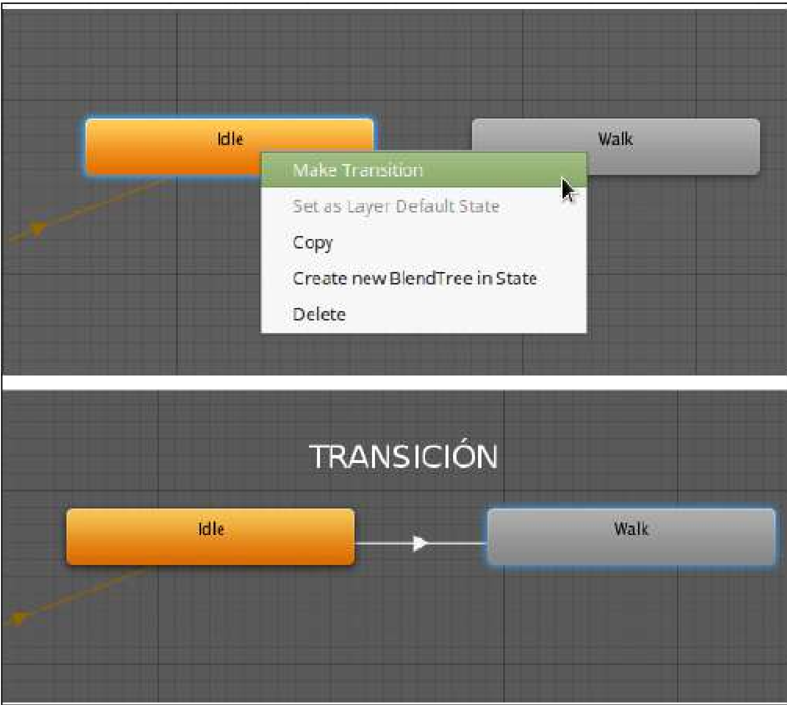


Fig. 10.54

Si ahora seleccionamos el estado **Idle** veremos que en la ventana Inspector aparece la transición en la parte inferior. Siempre que creemos una transición hacia otro estado se veran representadas en la ventana inspector. Si hacemos clic encima de esta transición en la ventana Inspector nos aparecerán las propiedades de esa transición.

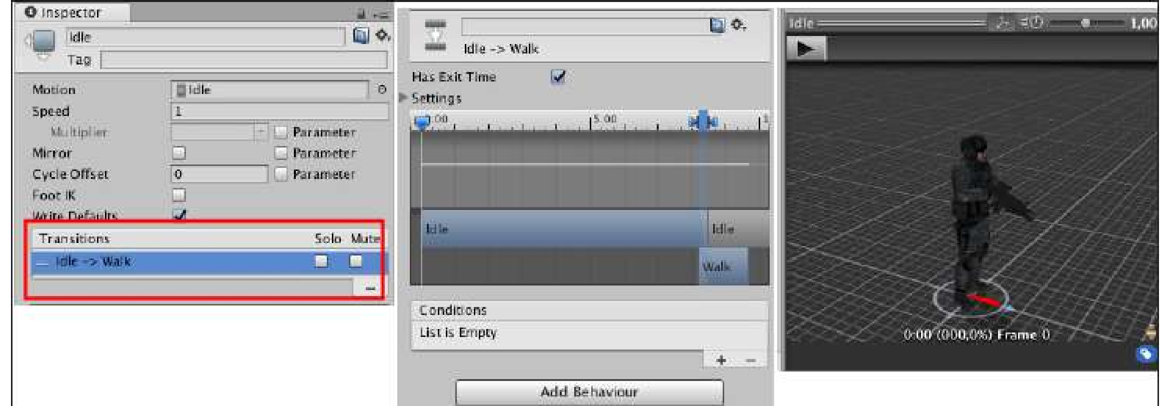


Fig. 10.55

En la imagen anterior se ha distribuido horizontalmente la ventana inspector con los distintos apartados que ahora se explicaran. En la imagen de la izquierda tenemos marcado la transición que indica que sale de Idle a Walk, si hacemos clic encima de esta transición aparecerán las propiedades de esta transición que se muestran en la imagen central y la imagen de la derecha es la representación de nuestro personaje en donde podremos visualizar la transición como se ejecuta.

En las propiedades de la transición disponemos de varias secciones:

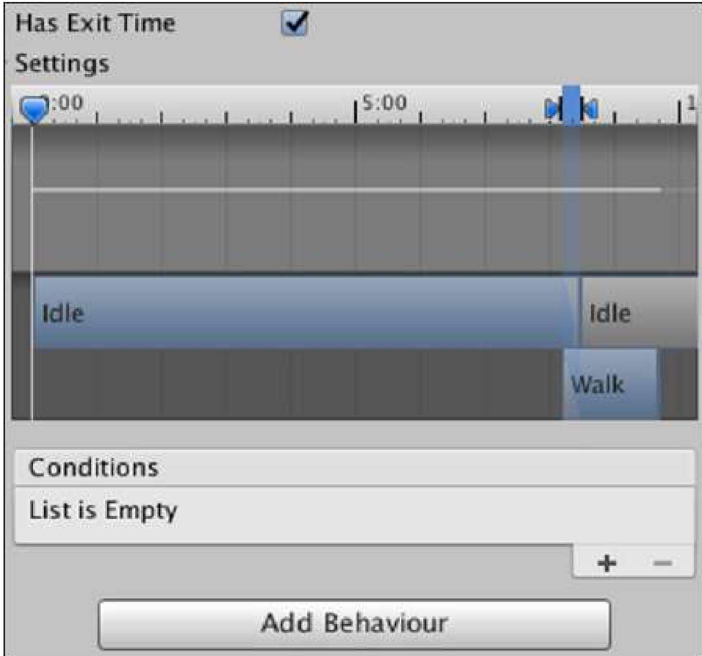


Fig. 10.56

Settings: En rasgos generales es la representación de la evolución de la transición es decir como se enlazan el clip de animación de **Idle** y el clip de animación de **walk.** Si hacemos clic en el play de la representación del personaje podrás ver como se ejecuta la transición.

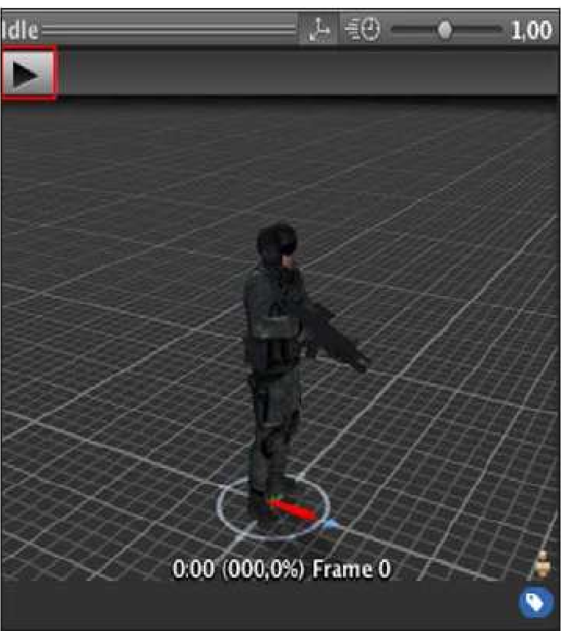


Fig. 10.57

El principal problema que podemos apreciar es que tarda demasiado tiempo des de el estado Idle hasta que empieza a caminar. En estos casos es muy recomendable que se escale el tiempo de la transición. Para que no reproduzca tanta animación **Idle** debemos

seleccionar la flecha de la izquierda que encontramos en la barra de tiempo y la desplazamos al inicio.

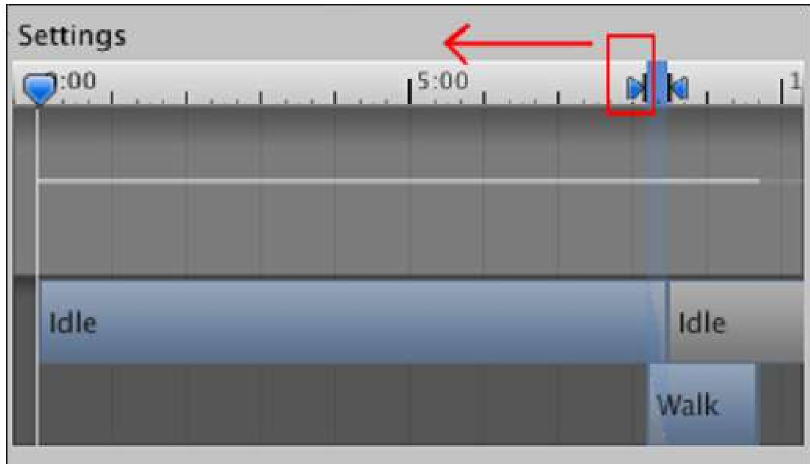


Fig. 10.58

La idea es que cuando se realice la transición del estado **Idle** a **Walk** sea de una forma **rapida**, si no es así, el jugador cuando pulse la tecla de caminar o avanzar, verá que su personaje avanza pero todavía está en la posición **Idle**. Para mejorar la transición a continuación te muestro la configuración que he utilizado para escalar la transición.

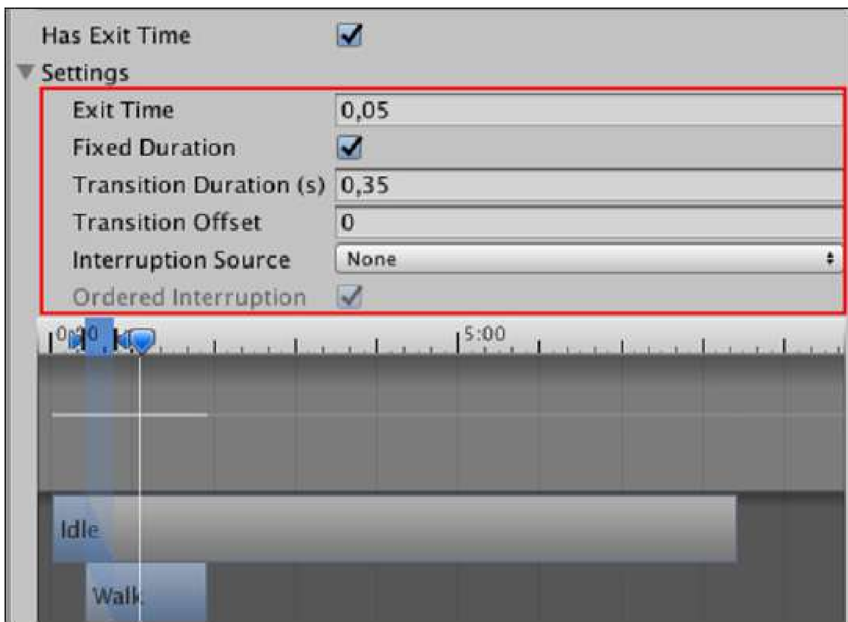


Fig. 10.59

En el apartado **Settings** al hacer clic encima de la flecha podemos acceder a las propiedades de la transición. Crear transiciones correctamente creo que es trabajo de un animador, pero en el caso de este ejemplo puedes utilizar estos o puedes probar los tuyos propios. Los parametros son los siguientes:

- **Exit Time:** este valor es el tiempo directamente antes de entrar en la transición, es decir el tiempo de salida antes de que empiece la transición. En este caso en concreto lo he desactivado a pesar de que veas en la imagen que está activo.
- **Fixet Duration:** cuando está activado se determina una duración fijada, en este caso es de 2 segundos si miras lo que dura la animación.
- **Transition Duration (s)**: la duración en segundos que dura la transición.
- **Transition Offset:** si queremos desfasar la transición. En este caso no nos interesa y por lo tanto el valor es 0.
- **Interruption Source:** en ocasiones se puede interrumpir una transición y esta opción nos permite controla bajo que circunstancias se puede interrumpir.
  - **None:** la transición no será interrumpida.
  - **Current State:** la transición puede ser interrumpida por otras transiciones dentro del estado actual, pero no dentro del estado destino.
  - **Next State:** la transición puede ser interrumpida por transiciones definidas y el estado destino (siguiente), pero no dentro del estado real.
  - **Current State then Next State:** la transición puede ser interrumpida por transiciones ya sea en la actual o la siguiente. Sin embargo, si una transición se vuelve true en ambos la actual y la siguiente a la misma vez, el estado actual tomará prioridad.
  - **Next State then Current State:** la transición puede ser interrumpida por transiciones ya sea en la actual o siguiente. Sin embargo, si una transición se vuelve true en ambas la actual y la siguiente a la misma vez, el estado siguiente tomará prioridad.
  - **Ordered Interruption:** determina si una transición actual puede ser interrumpida por otras transiciones independientemente de su orden.
- **Has Exit Time:** determina si la condición de la transición puede tomar efecto en cualquier momento, o solamente durante el exit time (tiempo de salida) del estado.

En este caso del ejemplo voy a desactivar esta ultima opción **Has Exit Time** para que se vea más exagerado y ya tendremos la transición revisada.

Interacción de un estado a otro

Ahora ya tenemos una transición pero ahora nos falta relacionar esta transición de alguna forma para que podamos controlar cuando se activa un estado u otro. Para realizar esto debemos acceder a la sección izquierda de la ventana **Animator** en el apartado **Paremeters**.

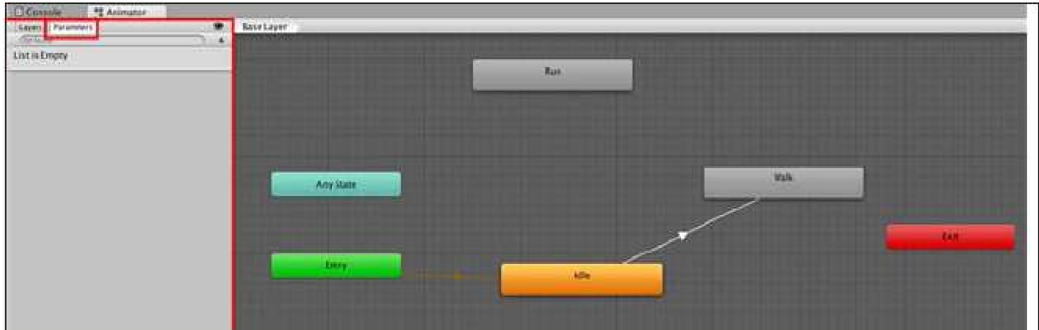


Fig. 10.60

Los parámetros son en lo que se va a basar la maquina de estados para poder pasar de uno a otro. Por el momento está vacía así pues vamos a añadir un parámetro pulsando en el símbolo más y seleccionamos la de tipo **Bool y una vez creada le ponemos el nombre Walking.**

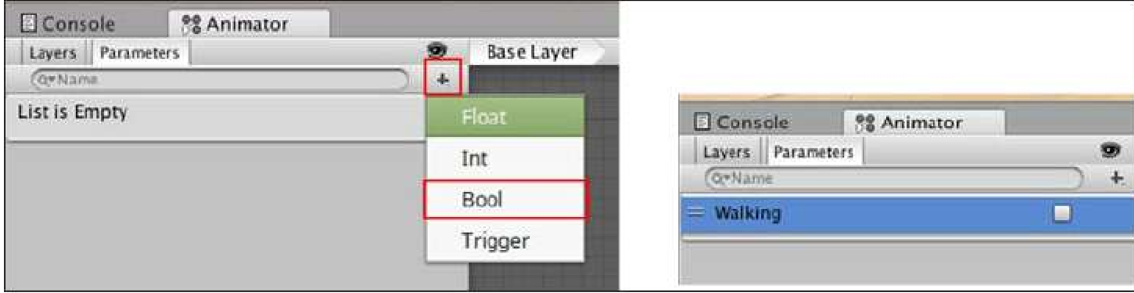


Fig. 10.61

Ahora seleccionamos es estado **Idle** para acceder a sus propiedades en la ventana Inspector y nos dirigimos a las propiedades de transición. En las propiedades tenemos la opción de crear condiciones. Pulsamos en el símbolo + para añadir una condición.

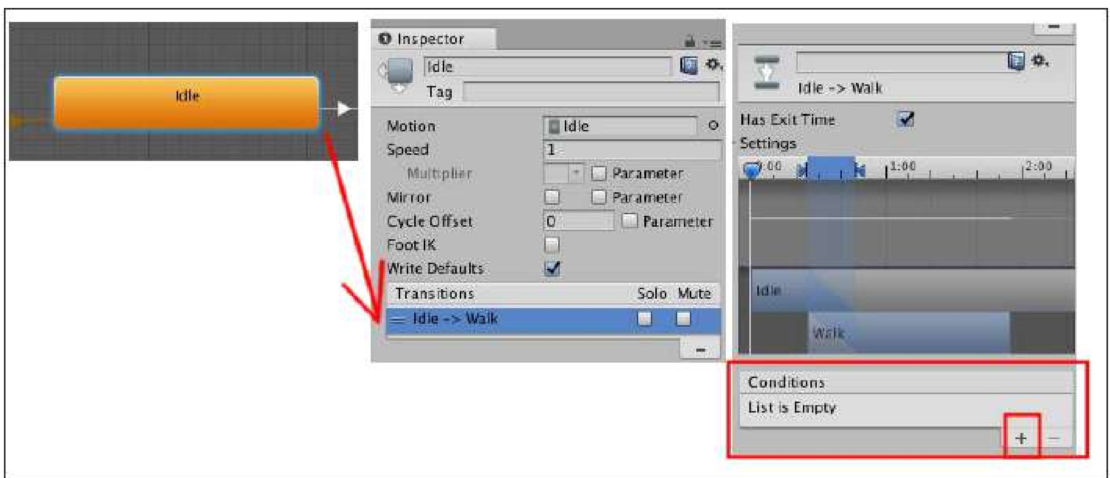


Fig. 10.62

Al añadir una condición en este ejemplo como solo hemos creado un parámetro nos aparecerá el parámetro que hemos creado por defecto. Para esta condición lo dejaremos en **Walking** true como te muestro en la siguiente imagen.

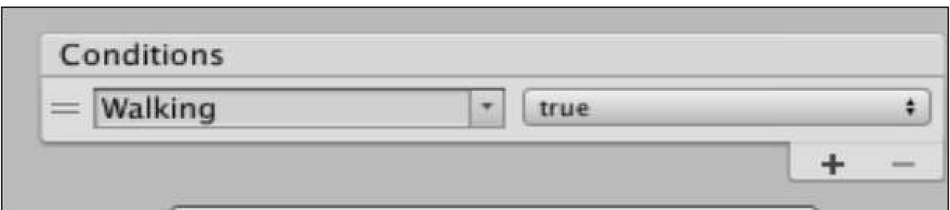


Fig. 10.63

Llegado hasta aquí debemos crear una transición de retorno, es decir del estado **Walk** al estado **Idle.** Para ello creamos una transición seleccionando Walk, pulsando botón derecho y seleccionado la opción Make Transition y seguidamente seleccionamos el estado Idle para que se cree la transición como te muestro en la siguiente imagen.

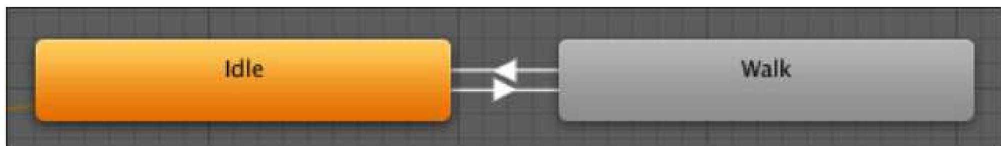


Fig. 10.64

Ahora vamos a crear otro parámetro para poder gestionar cuando esta caminando y cuando está parado. En este caso vamos a crear un parámetro de tipo **float** al que le daremos el nombre de **Speed** con un valor de **0,0**.

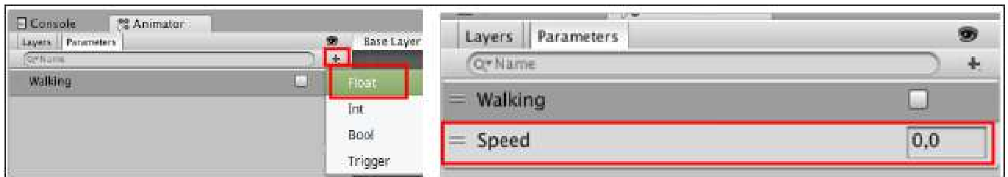


Fig. 10.65

Ahora vamos a ver que condiciones tienen las transiciones de cada uno de los estados. En el estado **Idle** teníamos **Walking** es verdadero, pero también podemos añadir otra condición que **Speed** sea mayor que 0. Para poner otra condición con el estado **Idle** seleccionado accedemos a la ventana **Inspecto** hacemos clic encima del apartado **Transitions** para que nos aparezca la propiedades de la transición. En el apartado de condiciones (**Conditions**) pinchamos en el símbolo (+) y nos aparecerá un duplicado de la condición anterior, debemos desplegar el menú y seleccionar el parámetro **Speed**. Para finalizar la condición tenemos una opción con **Greater** (Mayor que) y un **Less** (menor que) en donde hace la comparación al valor que introduzcamos en la caja de textos lateral. El valor lo dejamos en 0.

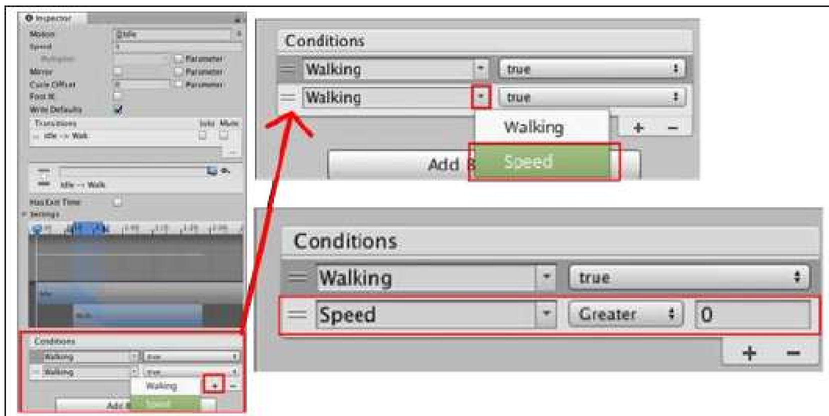


Fig. 10.66

Ahora seleccionamos el estado **Walk** y accedemos a sus propiedades de transición como hemos realizado con **Idle**. En sus propiedades accedemos a crear una nueva condición en este caso será el parámetro **Speed** con la opción **Less** y un valor de **0,1**.

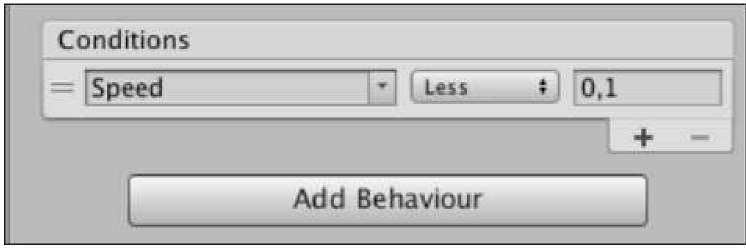


Fig. 10.67

En resumen el estado **Idle** pasara a **Walk** cuando **Walking** sea verdadero y la velocidad **Speed** sea mayor de cero. Por el contrario el estado **Walk** pasará a **Idle** cuando la velocidad **Speed** sea menor de **0,1**. Esta sería la lógica que vamos a utilizar.

Ahora es el momento de relacionar el parámetro con la acción y lo vamos a realizar mediante el Script **ControlPlayer** que hemos creado anteriormente. Antes de editar nuestro script me gustaría comentarte que en la **API** de Unity si ponemos en el buscador **Animator** podremos ver todos los métodos y atributos que disponemos para trabajar con este componente. Para facilitarte el trabajo puedes acceder desde el siguiente link;

https://docs.unity3d.com/ScriptReference/Animator.html

Si miras en esta documentación en el apartado de métodos públicos puedes ver los métodos **SetBool, SetFloat, SetInteger** que vamos a necesitar para la edición de nuestro script.

Accedemos al script ControlPlayer y vamos a editar las siguientes lineas.

Script: ControlPlayer.cs

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

public class ControlPlayer : MonoBehaviour
{
        Vector3 direccion;
        CharacterController controlador;
        public float walkingSpeed;
        public float runningSpeed;
        public bool isWalking;
        private Vector3 momentoSpeed;
        Animator animador;

        void Start ()
```

```
        {
                this.controlador = gameObject.GetComponent<CharacterContro-
ller> ();
                this.animador = gameObject.GetComponent<Animator> ();
        }

        void Animate()
        {
                this.animador.SetBool ("Walking", this.isWalking);
                this.animador.SetFloat ("Speed", this.momentoSpeed.magnitude);
        }

        void Update ()
        {
                this.direccion = new Vector3 (Input.GetAxis ("Horizontal"), 0,
                Input.GetAxis ("Vertical"));
                this.isWalking = !Input.GetKey (KeyCode.LeftShift);

                if (this.isWalking)
                {
                        this.momentoSpeed = this.walkingSpeed * this.
direccion;
                }

                else
                {
                        this.momentoSpeed = this.runningSpeed * this.
direccion;
                }


                this.controlador.SimpleMove (this.momentoSpeed);

                this.Animate ();
        }
}
```

Primero de todo debemos crear una variable de tipo Animator que he llamado animador para poder guardar la información de este componente.

En la función Start() accedemos al componente Animator con el método GetComponent y lo guardamos en la variable animador.

He creado una función antes del Update para gestionar las animaciones de la maquina de estado. Esta función la he llamado Animate y la llamare después dentro de la función Update() para que se ejecute a cada frame.

En esta función Animate() vamos a utilizar la variable animador en donde hemos guardado el componente **Animator** de nuestro personaje y ahora podemos acceder a todos sus métodos y atributos. Así utilizo primero el método **SetBol** que me pide como argumentos un nombre que va a ser el nombre del parámetro "**Walking**" y un valor verdadero o falso porque es booleano. Para el valor voy a utilizar la variable **isWalking** que creamos anteriormente para saber cuando camina y cuando no.
El segundo método utilizo el **SetFloat** que me pide los mismos argumentos un nombre en este caso es **"Speed"** y un valor **float.** En este caso la variable que controla el valor de la velocidad es **momentoSpeed,** pero en este caso no es una variable de tipo float, es una variable de tipo Vector3, para solucionar este problema he recurrido a un método de Vector3 que se llama magnitude que nos convierte un Vector3 a float.
Una vez tenemos la función **Animate()** creada la llamamos dentro del **Update** al final de todo.
Recuerda que debes guardar los cambios del script para que tenga efecto en **Unity.**

Una vez editado el script volvemos a Unity y ejecutamos la escena para comprobar que el player empieza con el estado **Idle** y cuando pulsamos la tecla arriba del teclado se activa el estado **Walk**.

Estado Walk a Running y de Runing a Walk

Con nuestro player ArmyPilot seleccionado accedemos a la ventana Animator y en la maquina de estados pinchamos encima del estado Walk. Vamos a crear una transición que vaya de Walk a Run. Con el estado Walk seleccionado pulsamos botón derecho del ratón y seleccionamos la opción Make Transition y seleccionamos el destino final que es el estado Run. Realizamos la misma acción desde el estado Run al estado Walk. La siguiente imagen te muestra como debería quedar la máquina de estados.

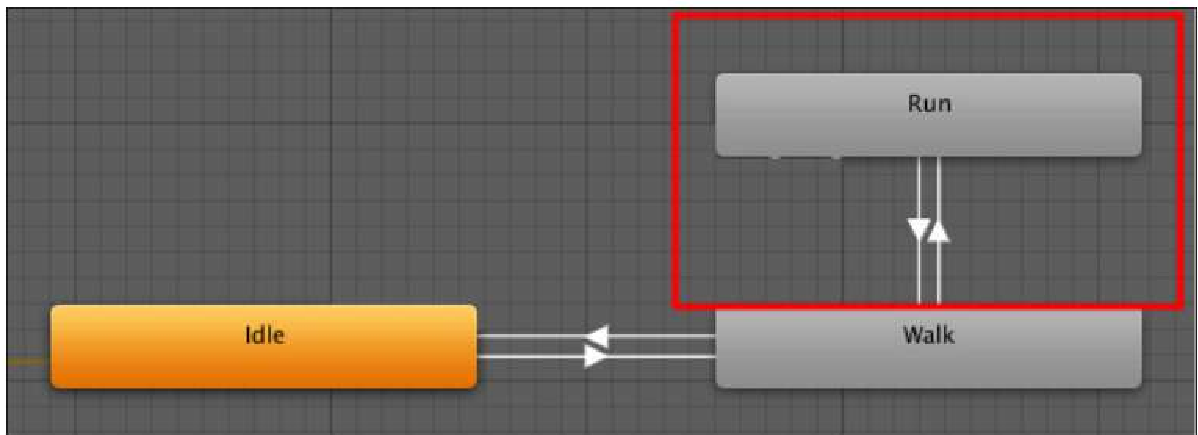


Fig. 10.68

Ahora vamos a crear un nuevo parámetro accediendo al símbolo **(+)** de la secciona izquierda de la ventana **Animator**. El parámetro será de tipo Booleano y le ponemos el nombre de **Running**.

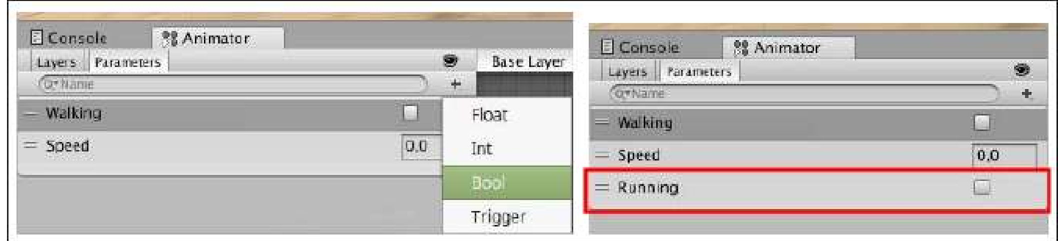


Fig. 10.69

Con el nuevo parámetro creado seleccionamos el estado **Walk** o si lo prefieres seleccionando la transición que va del estado **Walk** al estado **Run** podremos acceder a las características de la transición en concreto.

En primer lugar para este ejemplo en concreto la configuración de la transición de la animación de **Walk** a **Run** es la siguiente:

- **Has Exit Time:** desactivado.
- **Fixed Duration:** activado.
- **Transition Duration:** 0,26
- **Transition Offset:** 0,028

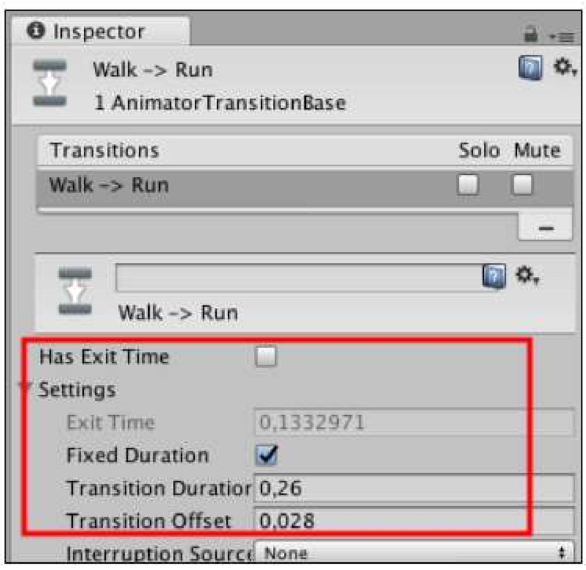


Fig. 10.70

En segundo lugar crearemos dos condiciones:

- **Speed** mayor de 2
- **Running** sea true.

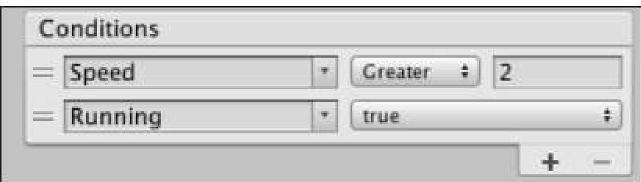


Fig. 10.71

Ahora tenemos que acceder a las propiedades de la transición, pero en sentido contrario, de **Run** a **Walk**. Como hemos hecho anteriormente vamos a pinchar en la transición que va del estado **Run** al estado **Walk** y en sus propiedades realizamos los siguientes cambios.

- **Has Exit Time:** desactivado.
- **Fixed Duration:** activado.
- **Transition Duration:** 0,149
- **Transition Offset:** 0

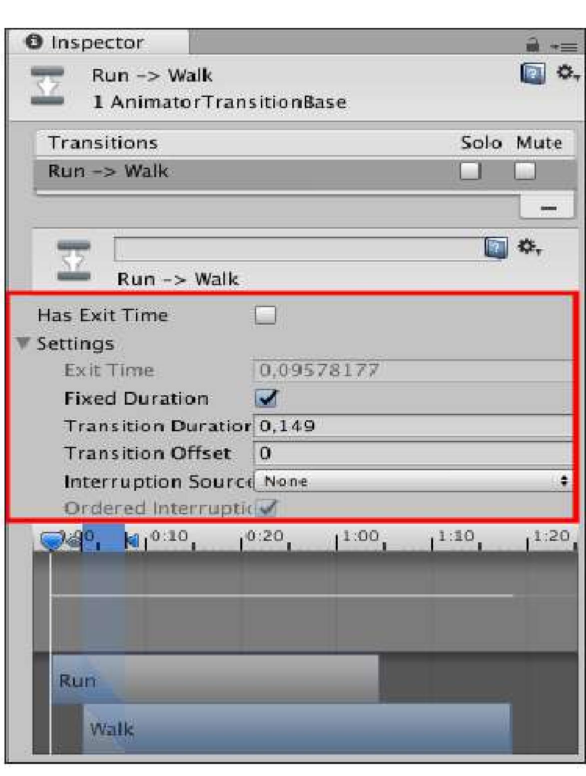


Fig. 10.72

Para las condiciones de esta transición diremos que **Running** es falso:

Fig. 10.73

Estado Idle a Running y de Runing a Idle

Antes de entrar en programación vamos a crear las transiciones para **Idle** y **Runing**, de este modo podemos empezar corriendo y no tener que caminar y luego correr. También podemos pasar de correr a estar parados.

El primer paso es crear las transiciones como hemos hecho hasta ahora uno que vaya de **Idle** a **Run** y otro de **Run** a **Idle**. Te muestro como debería quedarte en la siguiente imagen.

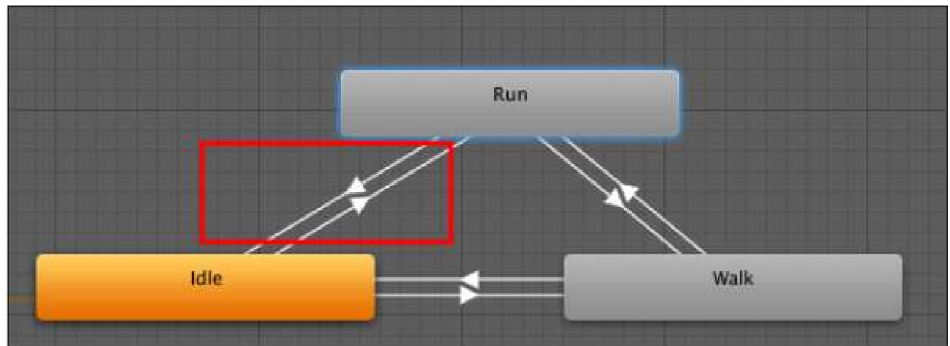


Fig. 10.74

Como ya tenemos un parámetro para relacionar el estado **Run**, vamos a pasar a configurar las propiedades de la transición.

De Idle a Run :

Para la transición de la animación

- **Has Exit Time:** desactivado.
- **Fixed Duration:** activado.
- **Transition Duration:** 0,17
- **Transition Offset:** 0

Para las condiciones:

- **Walking** con valor **False**
- **Running** con valor **True**
- **Speed** tenga un valor mayor de **2**

De Run a Idle :

Para la transición de la animación

- **Has Exit Time:** desactivado.
- **Fixed Duration:** activado.
- **Transition Duration:** 0,3
- **Transition Offset:** 0

Para las condiciones:

- **Running** con valor **False**
- **Speed** con valor menor a 0,1

Ahora que tenemos la maquina de estado con sus transiciones y con las condiciones vamos a mejorar nuestro script **ControlPlayer** para que nuestro player pueda reaccio-

nar a los distintos estados. Otro aspecto que voy a añadir a este script es un modo de rotación para que nuestro personaje rote hacia izquierda y derecha con las teclas flecha izquierda y flecha derecha del teclado.

Script: ControlPlayer.cs

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

public class ControlPlayer : MonoBehaviour
{
   Vector3 direccion;
   CharacterController controlador;
   public float walkingSpeed;
   public bool isWalking;
   private Vector3 momentoSpeed;
   public float runningSpeed;
   Vector3 rotacion;
   public float rotationSpeed;
   public bool isRunning;

   Animator animador;

   void Start ()
   {
      this.controlador = gameObject.GetComponent<CharacterController> ();
      this.animador = gameObject.GetComponent<Animator> ();
   }

   void Animate()
   {
      this.animador.SetBool(“Walking”, this.isWalking);
      this.animador.SetFloat (“Speed”, this.momentoSpeed.magnitude);
      this.animador.SetBool (“Running”, this.isRunning);
   }

   void Update ()
   {

      this.direccion = transform.TransformDirection (new Vector3 (0f, 0f,
Input.GetAxis                  (“Vertical”)));
      this.rotacion = new Vector3 (0f, Input.GetAxis (“Horizontal”), 0f) *
       this.rotationSpeed;
```

```
        this.isWalking = !Input.GetKey(KeyCode.LeftShift);

        if (this.isWalking)
        {
         this.momentoSpeed = this.walkingSpeed * this.direccion;
         this.isRunning = false;
        }
        else
        {
         this.momentoSpeed = this.runningSpeed * this.direccion;
         this.isRunning = true;
        }

        this.controlador.transform.Rotate (this.rotacion);
        this.controlador.SimpleMove (this.momentoSpeed);

        this.Animate ();
    }
}
```

Primero vamos a crear las variables referentes al estado run. Crearemos una variable pública de tipo float con nombre runningSpeed. He creado una variable de tipo Vector 3 con nombre rotacion, en este caso puedes hacerla privada puesto que no nos interesa variar el valor de esta. Otra variable que necesitamos es otra publica de tipo float con nombre rotationSpeed para poner una velocidad al rotar nuestro player. Para finalizar he creado una variable de tipo booleano con nombre isRunning para tener referencia al parámetro que hemos creado en la máquina de estados.
Dentro de la función Animate() que creamos anteriormente añadimos el parámetro Running para ello accedemos desde la variable animador al método SetBool y le facilitamos los siguiente argumentos, el nombre del parámetro al que hacemos referencia y a la variable booleana isRunning que es la que contiene el valor que modificara este parámetro.
En la función Update he realizado algunos cambios en la dirección en la variable dirección accedemos a la transformación del objeto en si y utilizamos su dirección local con TransformDirection y en el nuevo Vector 3 que le damos solamente controlaremos mediante el método de Input.GetAxis ("Vertical") el eje z que se encarga de mover nuestro player hacia delante y hacia atrás. Para rotar nuestro personaje utilizo la variable rotación con un nuevo Vector3 en donde utilizo el método de Input GetAxis en "Horizontal" en el eje y para que podamos rotar nuestro player a izquierda y derecha y lo multiplicamos por la variable rotationSpeed.
Dentro de la condición debemos decir que si esta Caminando nuestra variable isRunning debe ser falsa y por el contrario si isWalking no se cumple isRunning debe ser true. Una vez se revisan las condiciones debemos decir a this.controlador que es la variable que nos proporciona el movimiento, utilizaremos la transformación con el método Rotate y le daremos el argumento de la variable this.rotacion que contiene el vector con el Input.
Guardamos el script para que tenga efecto en Unity.

Una vez hecho todos los cambios en el script y volviendo a Unity debemos dar valores a las variables de la ventana inspector en el apartado Script en donde yo te propongo utilizar estos valores pero puedes experimentar probando con otros.

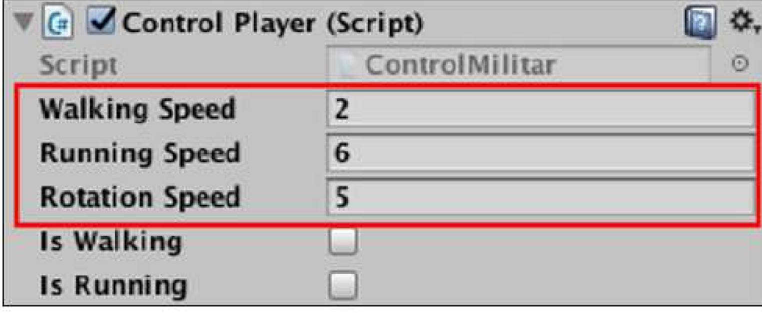


Fig. 10.75

Una vez añadidos estos valores ejecutamos la escena y podemos comprobar como podemos rotar nuestro personaje con las teclas izquierda y derecha y como las transiciones funcionan correctamente.

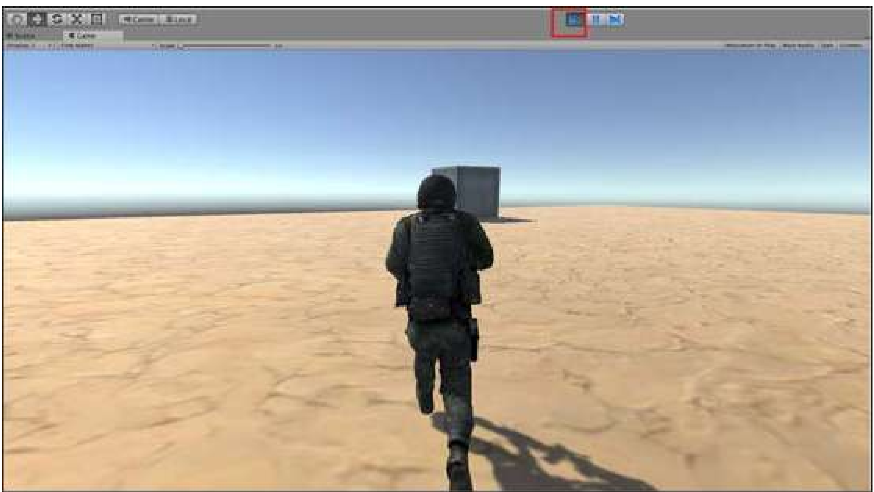


Fig. 10.76

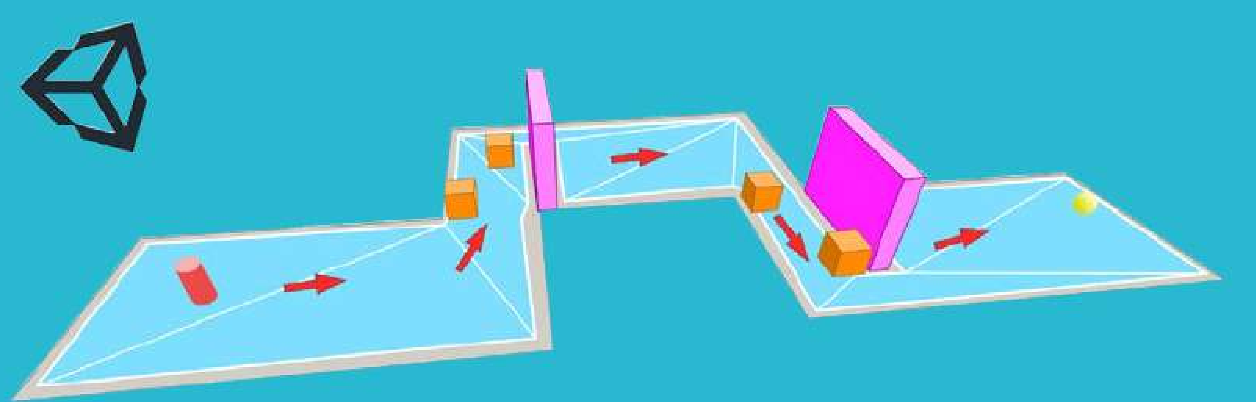
En este capítulo hemos visto una pequeña muestra de:

- Como funciona la ventana Animation y la ventana Animator.
- Como se puede crear un clip de animación.
- Como podemos configurar un personaje con animaciones creadas externamente
- Como integrar estas animaciones en un script, para se representen en nuestro personaje.

El objetivo es proporcionarte una visión general de como trabajar con Unity accediendo a las distintas partes que lo componen. Si has realizado toda la actividad valorarás mucho más cuando juegues a un videojuego y veas la cantidad de movimientos que puede llegar a tener un solo personaje. En el próximo capitulo veremos la parte de navegación.

Capítulo 11

Navigation y Pathfinding



- Vista general del sistema de navegación en Unity
- Cómo funciona el sistema de navegación
- Construir un NavMesh
- Crear un NavMesh Agent
- Crear un NavMesh Obstracle
- Crear un Off-mesh Link
- Proyectos de Navigation

Sistema de navegación

Este sistema nos permite crear personajes que pueden moverse por zonas con un tipo de inteligencia artificial por un escenario o un terreno utilizando mallas de navegación que se crean a partir de la geometría de la escena.

1. Vista general del sistema de navegación en Unity

Esta capacidad de moverse por todo un escenario se le proporciona a personajes del juego y a enemigos. Esta sistema permite crear una serie de normas para que el enemigo o personaje pueda tomar decisiones dentro del juego. Para ello el sistema de navegación se compone de las siguientes partes:

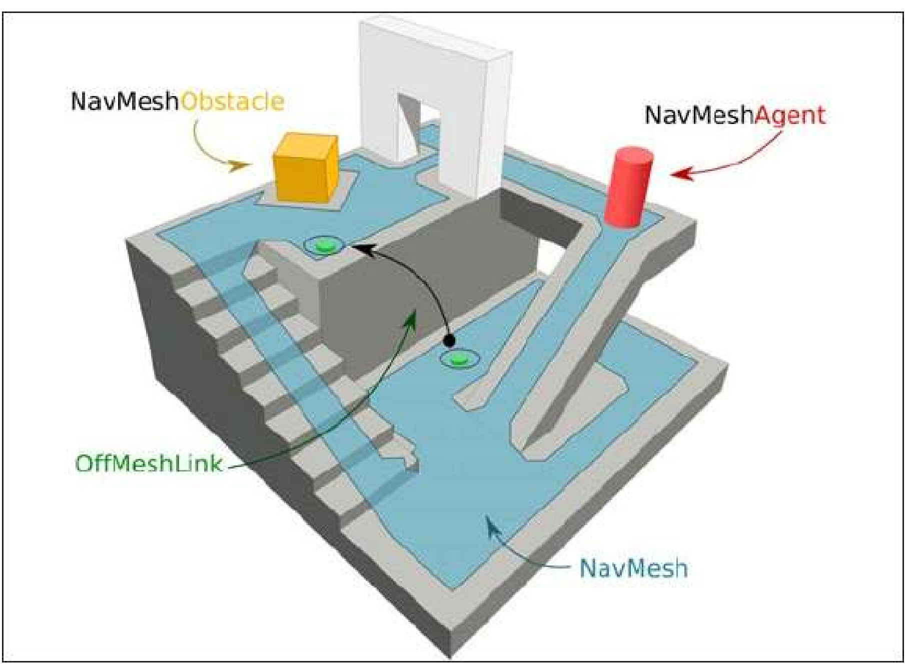


Fig. 11.1

- **NavMesh** (abreviatura de Navigation Mesh): Describe las superficies transitables del escenario o terreno del juego y permite que el objeto encuentre una la ruta alternativa en el caso de que el juego lo requiera. Esta estructura de datos la podemos crear o **backear (hornear)** automáticamente desde la geometría del escenario.
- **NavMesh Agentlo:** Es un componente que ayuda a crear personajes que se evitan mientras se mueven hacia su objetivo. Los agentes toman decisiones sobre la escena del juego utilizando el NavMesh, estos se detectan unos a otros y a obstáculos en movimiento.
- **Off-Mesh Link:** Es un componente que permite incorporar un atajos de navegación que no se pueden representar utilizando una superficie de la escena. Por ejemplo, saltar por encima de una barrera o tanca, abrir una puerta antes de atravesarla.
- **NavMesh Obstacle:** Es un componente que permite describir los obstáculos en movimiento que los agentes deben evitar mientras navegan por la escena. Un buen ejemplo de obstáculo sería un barril o una caja controlada por el sistema de física. Mientras el obstáculo se mueve, los agentes

hacen todo lo posible para evitarlo, pero una vez que el obstáculo se vuelve inmóvil, este creará una zona en la superficie de navegación que advierte a los agentes que tienen que cambiar su camino para rodearlo, de esta manera los agentes pueden encontrar otra ruta diferente.

2. Cómo funciona el sistema de navegación

Tenemos que entender que cuando movemos un objeto como agente, nos vamos a encontrar con dos situaciones a resolver la primera encontrar el destino y la otra como gestionar el trayecto hasta llegar allí. En el caso de encontrar el destino tiene un aspecto mucho más global y estático al tener en cuenta todo el escenario. El trayecto es de tipo local y más dinámico, ya que considera la dirección para desplazarse y como evitar colisiones.

Teniendo en consideración los dos aspectos anteriores vamos a ver los siguientes aspectos de la navegación.

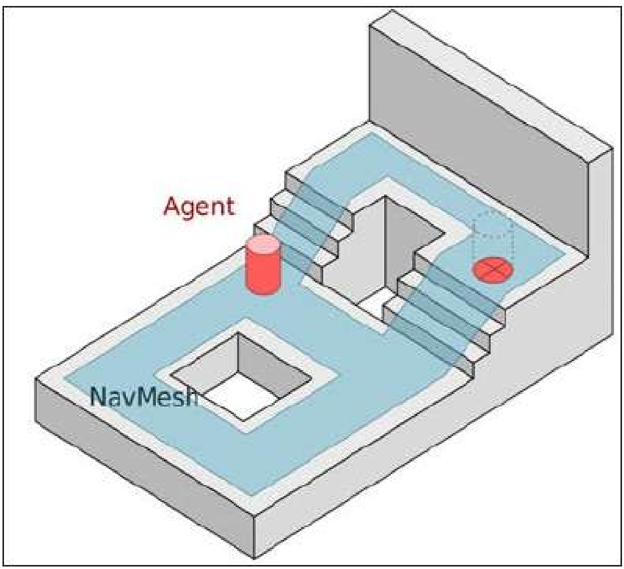


Fig. 11.2

Áreas Transitables

El sistema de navegación necesita sus propios datos para representar las áreas transitables en una escena de juego. Las áreas transitables definen los lugares en la escena donde el agente puede pararse y moverse. En Unity, los agentes son descritos como cilindros. El área para peatones se construye automáticamente a partir de la geometría de nuestro escenario. Luego, las ubicaciones se conectan a una superficie que se encuentra en la parte superior de la geometría de la escena. Esta superficie se llama malla de navegación (**NavMesh**).

El **NavMesh** almacena esta superficie como polígonos convexos. Los polígonos convexos son una representación útil, ya que sabemos que no hay obstrucciones entre dos puntos dentro de un polígono. Además de los límites del polígono, almacenamos infor-

mación sobre qué polígonos son vecinos entre sí. Esto nos permite razonar acerca de toda el área transitable.

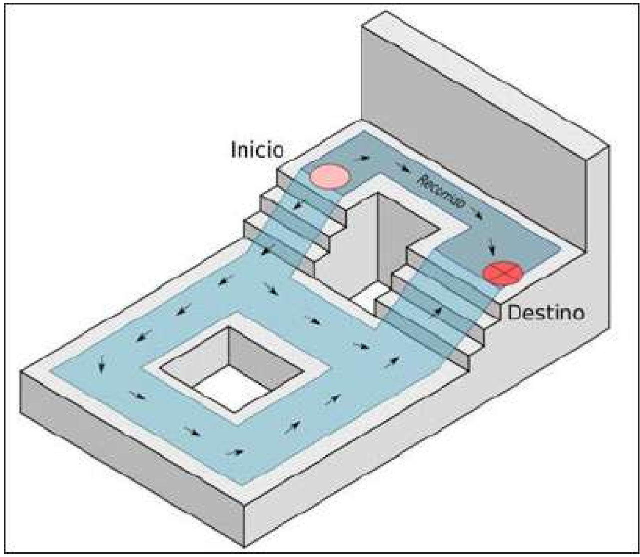


Fig. 11.3

Encontrar Caminos (Finding Paths)

Para encontrar la mejor ruta desde el inicio hasta el destino en una escena, primero debemos asignar las ubicaciones del inicio y destino en el escenario. Luego empieza una búsqueda desde la ubicación de inicio, explorando todos las posibles ubicaciones hasta llegar al lugar de destino. El recorrido de las zonas visitadas permite encontrar la ruta mas adecuada desde el inicio hasta el destino.

Seguimiento de recorrido

La secuencia de polígonos que describe la ruta desde el inicio hasta el destino se denomina corredor. El agente llegará al destino siempre dirigiéndose hacia la próxima esquina visible del corredor.

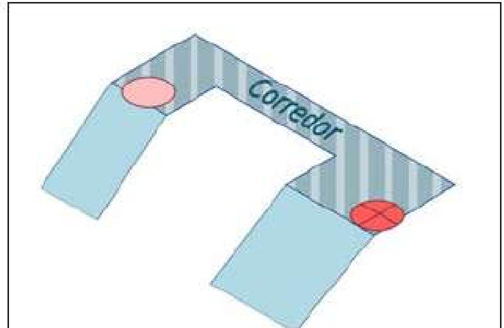


Fig. 11.4

Evitando Obstáculos

Cuando un agente se encuentra con un obstáculo la dirección lógica toma la posición de la siguiente esquina y, basándose en eso, calcula la dirección y la velocidad para alcanzar el destino. Usar la velocidad para mover el agente puede provocar una colisión con otros agentes.

Cuando un agente evita un obstáculo pueden ocurrir dos situaciones, una que busque una forma de bordear la zona y poder continuar con el recorrido. La otra situación es que no tenga ninguna opción de bordear el obstáculo y tenga que regresar por donde ha venido.

El movimiento de un Agente Global y Local

Un agente una vez tiene la dirección y evita obstáculos, calcula la velocidad final. En esta etapa es posible que la simulación del agente esté vinculado al sistema de animación de estados para mover el personaje, si no es así el sistema de navegación se encarga de ello.

Un agente utiliza la navegación de dos formas distintas global y local.

- **Global:** Se utiliza para encontrar el corredor en toda la escena. Es una operación requiere de bastantes recursos.
- **Local:** Este proceso intenta descubrir como moverse de forma eficaz hacia la siguiente esquina sin colisionar con otros agentes u objetos en movimiento.

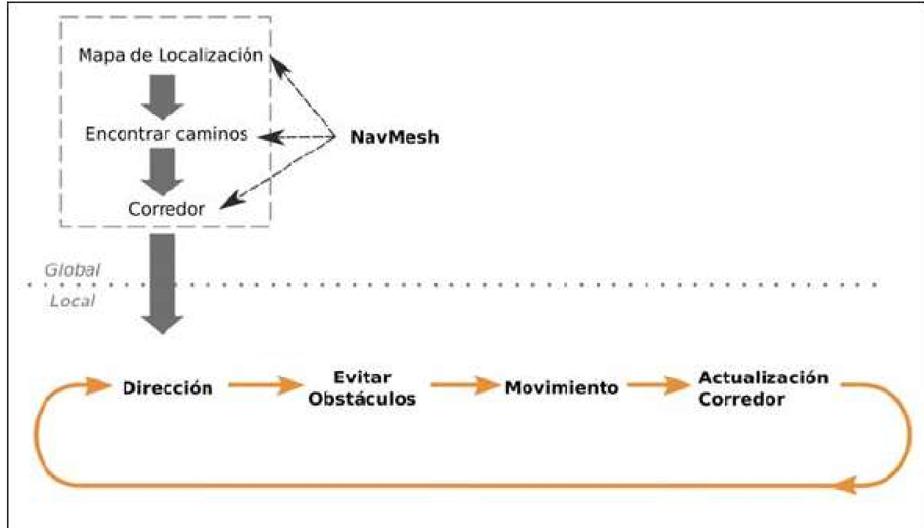


Fig. 11.5

3. Construir un NavMesh

Para continuar con el capitulo vas a tener que importar el paquete con las escenas preparadas para las siguientes explicaciones. El paquete lo encontraras como siempre en la carpeta de proyectos del capítulo correspondiente con el nombre **Assets_Escena_Capitulo_11.unitypackage.** Recuerda que para importar este paquete debes acceder al

menú principal **Assets > Import Package** y en la nueva ventana tener todos los archivos seleccionados y aceptar la importación. A continuación seguir las explicaciones siguientes.

Para este apartado accede a la carpeta escenas y selecciona la que tiene el nombre Escena_1. Esta escena es muy simple y servirá para mostrarte que hacer realmente un **NavMesh** y de que se caracteriza.

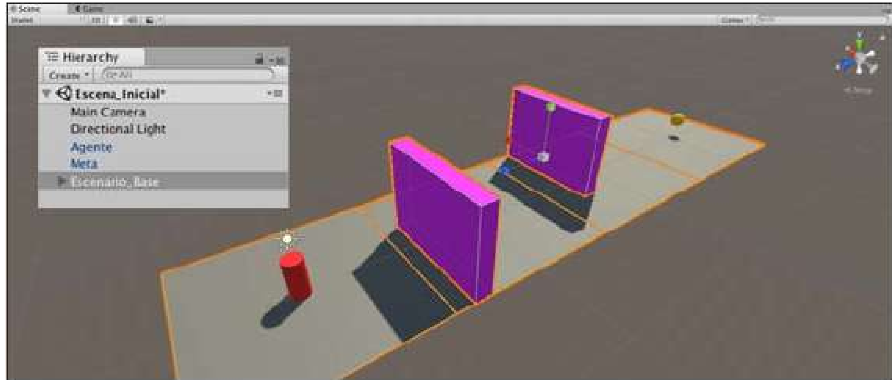


Fig. 11.6

Esta escena se compone de un escenario base que puedes desplegar en la ventana **Hierarchy,** que tiene cuatro planos como suelo y dos cubos que hacen de pared. Luego tenemos una esfera de color amarillo con el nombre Meta y un cilindro rojo con el nombre Agente.

Estos objetos tienen ciertas particularidades que vamos ha ir explicando pero por ahora solo quiero que sepas que tiene cada objeto de especial.

El objeto Agente dispone de un componente **Mesh Agent** y un script para que se mueva y busque un destino, en este caso sera el objeto Meta. Por otro lado las paredes también tienen un componente **NavMesh obstacle** que se explicara más adelante.

Si has ejecutado o ejecutas la escena tal y como esta actualmente no va a funcionar, porque no hemos creado el **NavMesh** para que nuestro agente pueda encontrar un camino para llegar hasta su objetivo o destino Meta.

Para crear un **NavMesh** primero debemos acceder a la barra de herramientas principal **Window > Navigation.** Por defecto aparecerá en forma de pestaña al lado de la ventana Inspector como te muestro a continuación.

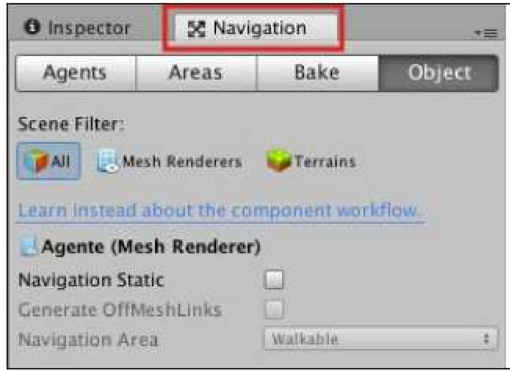


Fig. 11.7

Esta ventana tienes cuatro botones en la parte superior por el momento el que nos interesa es el botón **Object**. Tenemos en el apartado **Scene Filter** tres opciones que podemos seleccionar para filtrar que tipo de objeto queremos que contenga la superficie de **Navegaión**, por ahora dejamos seleccionada la primera opción **("All"), luego vamos a activar la opción de debajo Navigation Static, esto significa que los objetos que van a utilizarse para el recorrido van a ser estáticos.**

Ahora pinchamos en la pestaña Inspector y seleccionamos los objetos de la escena que forman parte del suelo, es decir todos los objetos suelo, tenemos que seleccionarlos.

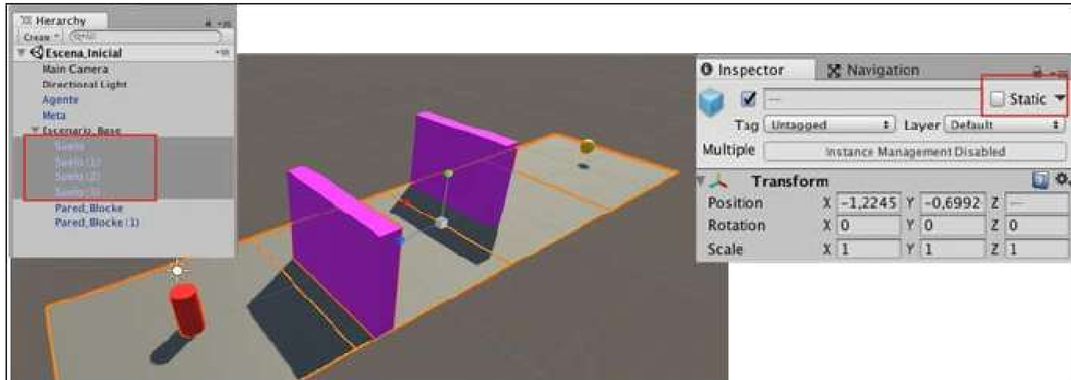


Fig. 11.8

Una vez seleccionados accedemos a la ventana inspector y activamos la opción **Static**. Unity permite hacer cambios a múltiples objetos a la vez. Todavía con los objetos suelo seleccionados volvemos a la ventana **Navigation** y pinchamos en el botón **Bake**.

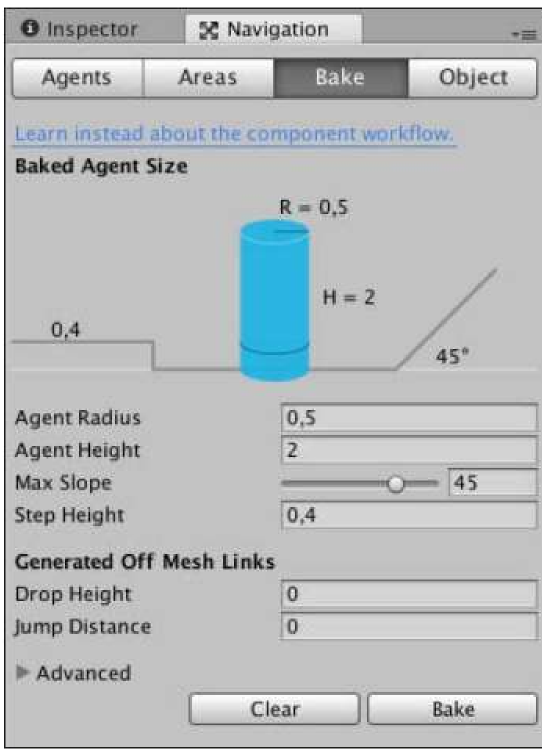


Fig. 11.9

Este apartado es el responsable de crear la superficie de navegación. Dentro de esta sección encontramos la configuración del Bake que contiene los siguientes parámetros básicos.

- **Agent Radius:** Define la distancia de cercanía que puede estar el centro del Agente de una pared o una repisa.
- **Agent Height**: Define la la altura a la que el agente puede alcanzar.
- **Max Slope**: Define la inclinación de una pendiente o rampa a la que el agente puede subir.
- **Step Height**: Define la altura a la que el agente puede pisar y acceder.

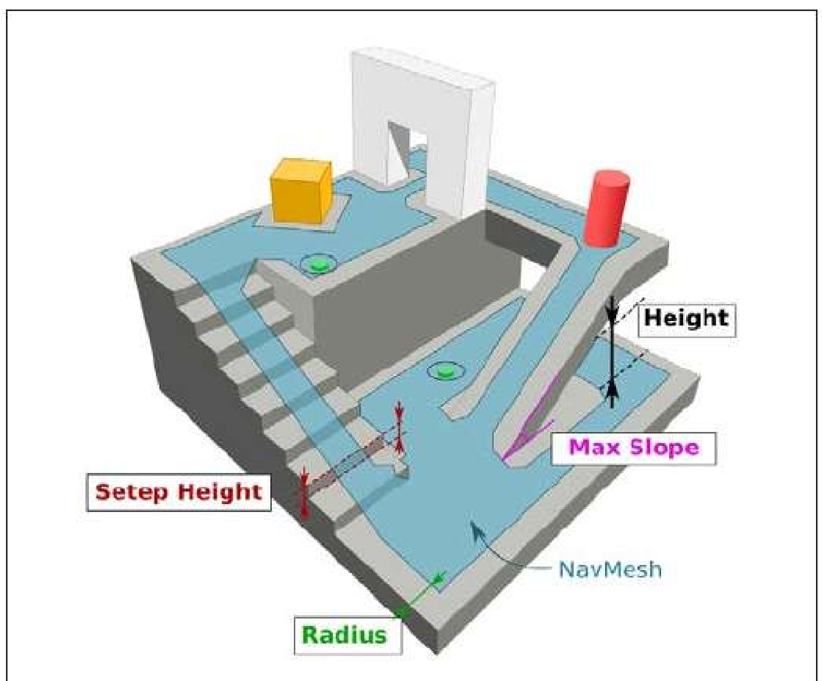


Fig. 11.10

Una vez que conoces los parámetros básicos del apartado **Bake**, y teniendo seleccionados todos los objetos suelo de la escena pinchamos en el botón **Bake**, para generar una superficie de navegación, que se representará en la ventana escena con un color azul. La escena debería de quedarte como te muestro en la siguiente imagen.

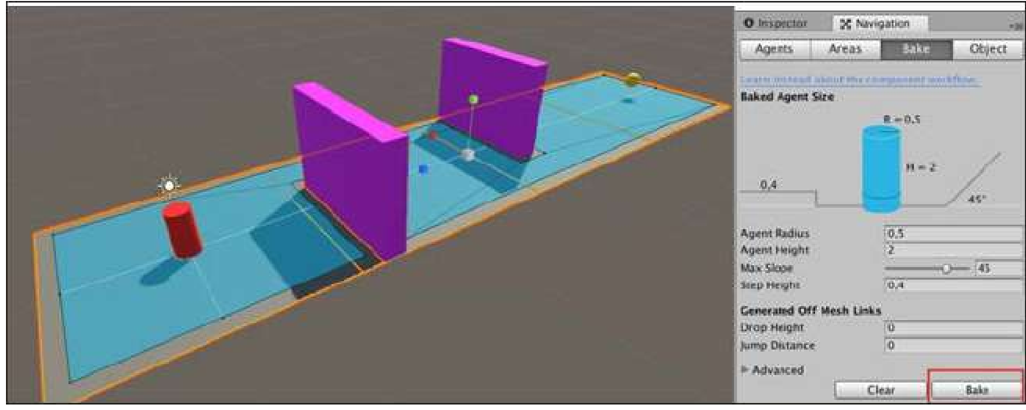


Fig. 11.11

El botón que hay al lado de **Bake** es decir **Clear**, sirve para eliminar la superficie de navegación y siempre que quieras modificar un parámetro te recomiendo eliminar el **bake** primero, cambiar los parámetros y volver a realizar el **bake**.

Si todo es correcto y te aparece la superficie azul, ahora si puedes ejecutar la escena y verás como el cilindro rojo va en búsqueda de la esfera amarilla.

4. Crear un NavMesh Agent

Ahora vamos a ver como se configura un **NavMesh Agent**. En el ejemplo anterior el objeto **Agent** disponía de un script y de los componentes ya preparados para su función en este próximo ejemplo vamos a ver como crearlo desde 0. Para el siguiente ejemplo vamos a la carpeta Escenas y pinchamos encima de la escena con el nombre Escena_2 dentro de la ventana **Projects**.

La escena que se cargará es como la que te muestro a continuación y no dispone de superficie de recorrido **NavMesh** ni el componente **Agent**.

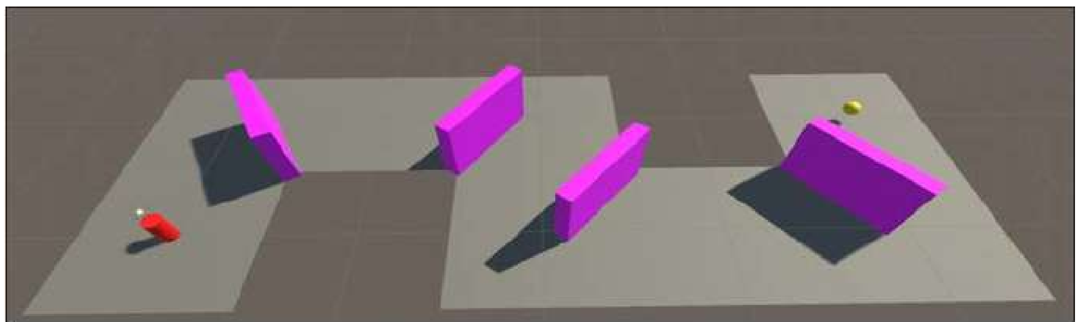


Fig. 11.12

Para empezar seleccionamos el objeto **Agent** desde la ventana **Hierarchy** que tiene el nombre de **Agent** y en la ventana inspector vamos a añadirle un nuevo componente. Pinchamos en **Add Component > Navigation > Nav Mesh Agent.**

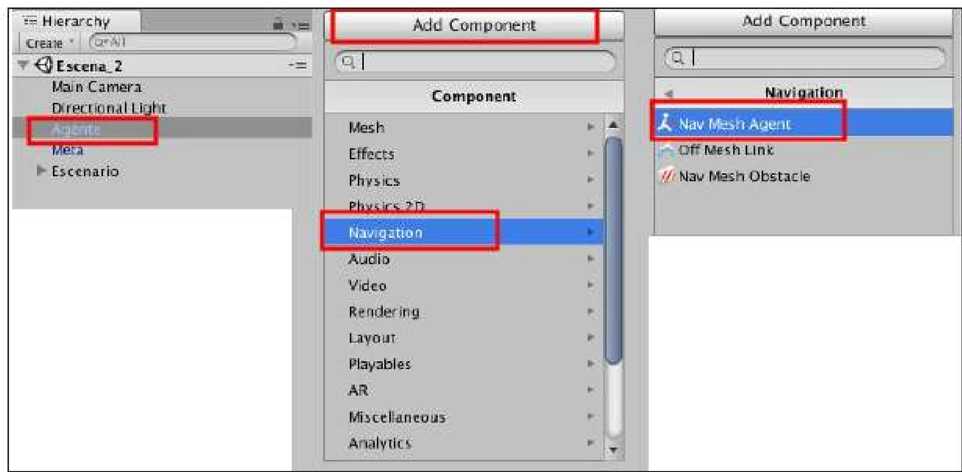


Fig. 11.13

Propiedades NavMesh Agent

El nuevo componente que se crea tiene una configuración que permite que este objeto evite otros Agentes y descubra nuevas rutas para recorrer el escenario.

A continuación te enuncio las distintas propiedades de este componente.

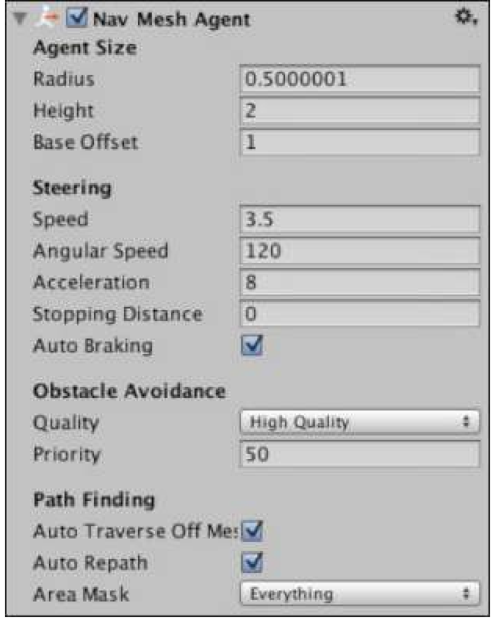


Fig. 11.14

- **Radius:** Este parámetro es utilizado para calcular colisiones entre obstáculos y otros agentes.
- **Height:** Este parámetro determina la altura necesaria para poder pasar por debajo o por encima de un obstáculo.
- **Base offset:** Es el desfase base del colisionador del cilindro en relación al punto pivote.
- **Speed:** Maxima velocidad de movimiento en unidades Unity por segundo.
- **Angular Speed:** Maxima velocidad de rotación en grados por segundo.
- **Acceleration:** Aumento de velocidad en unidades Unity por cuadro.
- **Stopping distance:** El agente se detendrá a cierta distancia cuando esté cerca de la ubicación del destino.
- **Auto Braking:** Cuando está activada esta opción, el agente se ralentizará cuando se acerque al destino. Se aconseja tener está opción cuando nuestro agente va a ser utilizado como un objeto que patrulla por la escena, de este modo permite que el objeto Agent se mueva sin problemas entre varios puntos.
- **Quality:** Determina la calidad para evitar obstáculos. Este parámetro permite configurar de modo eficiente la gestión de recursos del sistema en situaciones en las que tengamos una gran cantidad de Agentes en escena.
- **Priority:** Da prioridad a los Agentes, en el caso de un agente de menor prioridad este será ignorado al realizar la acción de evitar. Los valores deben ser valores comprendidos entre 0-99 tomando los valores más bajos como de mayor prioridad.
- **Auto Traverse OffMesh Link:** En el caso de que esta opción esté activada permite atravesar automáticamente los enlaces fuera de malla. Esta opción se recomienda desactivar cuando se quiera utilizar la animación o alguna otra forma específica de atravesar los enlaces fuera de malla.

- **Auto Repeath:** Al habilitar esta opción el agente intentará encontrar la ruta una vez más al llegar al final de una ruta parcial. Cuando no hay una ruta al destino, se genera una ruta parcial a la ubicación accesible más cercana al destino.
- **Area Mask:** Este parámetro determina qué tipos de área considerará el agente cuando encuentre una ruta. Cuando preparamos un **NavMesh** podemos determinar que tipo de malla será. Por ejemplo podemos crear Agentes que no puedan subir escaleras o no puedan pasar de ciertos limites.

Ahora que sabemos como funciona el componente vamos a crear una superficie **NavMesh** para nuestro agente. Para ello seleccionamos los objetos suelo y activamos la opción **Statics** de la ventana Inspector. Como hemos dicho en el ejemplo anterior Unity permite hacer una selección múltiple de varios objetos y ejecutar la misma acción para todos.

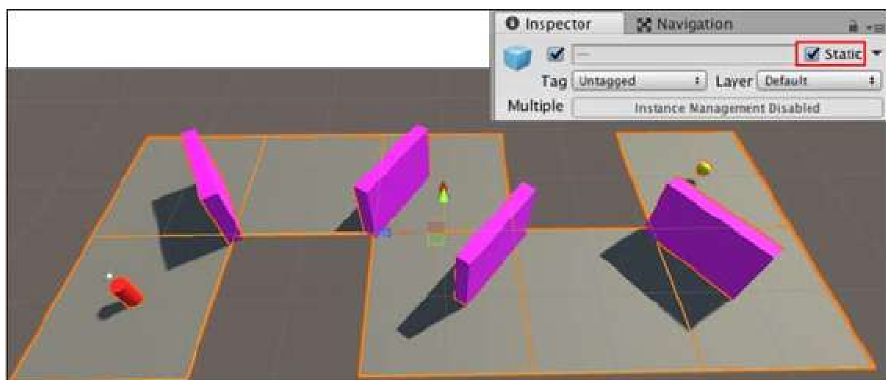


Fig. 11.15

Accedemos a la ventana **Navigation** para crear el **NavMesh**. Si no tienes abierta esta ventana puedes abrirla desde la barra de herramientas principal accediendo a **Window > Navigation.** Para empezar nos dirigimos al apartado **Object** y activamos la opción **Navigation Static.** En el apartado **Bake** pinchamos en el botón **Bake** para crear la superficie **NavMesh.**

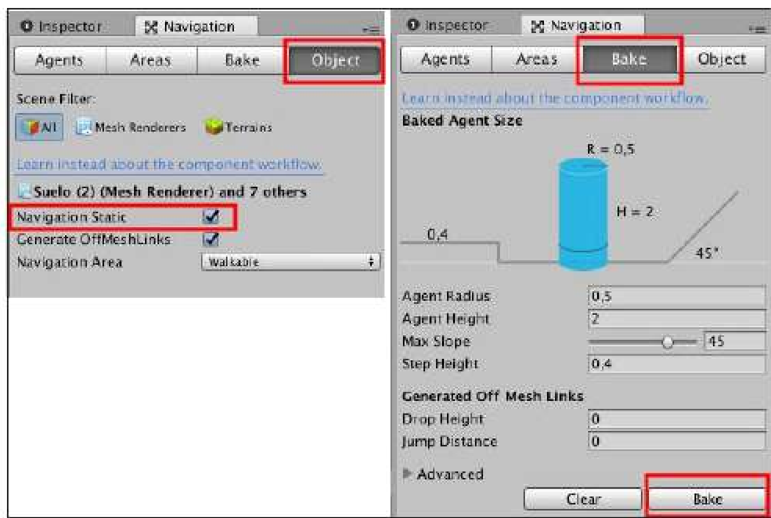


Fig. 11.16

Si todo es correcto la escena se debe mostrar de la siguiente imagen.

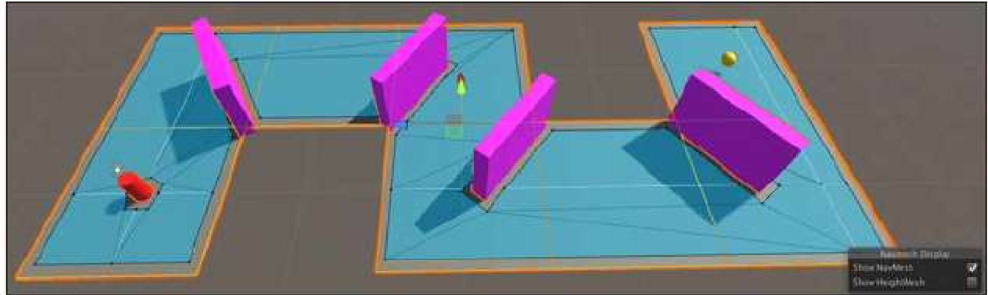


Fig. 11.17

Ahora debemos crear un script para crear la interactividad de nuestro Agente para que se mueva por la escena en la búsqueda de un destino. Antes de crear el script te recomiendo que mires en la documentación de Unity este tipo de clase que permite realizar consultas sobre el espacio, como pruebas para encontrar distintos caminos. Si lo deseas te facilito dos enlaces; el primero es para la clase **NavMesh** y el segundo se centra en la clase **NavMeshAgent.**

https://docs.unity3d.com/2018.1/Documentation/ScriptReference/AI.NavMesh.html

https://docs.unity3d.com/2018.1/Documentation/ScriptReference/AI.NavMeshAgent.html

Volviendo a Unity en la ventana **Project** accedemos a la carpeta Scripts y hacemos doble clic encima del script con el nombre **MoveTo**. A continuación se abrirá el editor **MonoDevelop** y vamos a comentar que contiene este script.

Script: MoveTo.cs

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

public class MoveTo : MonoBehaviour
{
    public Transform meta;
    private NavMeshAgent agente;

    void Start ()
    {
       this.agente = gameObject.GetComponent<NavMeshAgent> ();
       this.agente.destination = meta.position;

    }
 }
```

Primero de todo debemos decirle al script que utilice las clases que pertenecen a la librería de **UnityEngine.AI;**
Ahora vamos a crear dos variables la primera la hacemos pública para que se muestre en la ventana Inspector, que es de tipo Transform con nombre meta que utilizaremos para almacenar la posición del objeto Meta. La otra variable la hacemos privada y es de tipo NavMeshAgent con nombre agente en donde guardaremos el componente NavMeshAgent.
Dentro de la función Start() guardamos dentro de la variable agent el componente NavMeshAgent. Al tener acceso a este componente después podemos acceder dentro de la variable agent a la propiedad destination y le diremos que sea igual a la posición de meta.

Volviendo a Unity primero vamos a arrastrar este script encima del Objetos Agente como te muestro en la siguiente imagen.

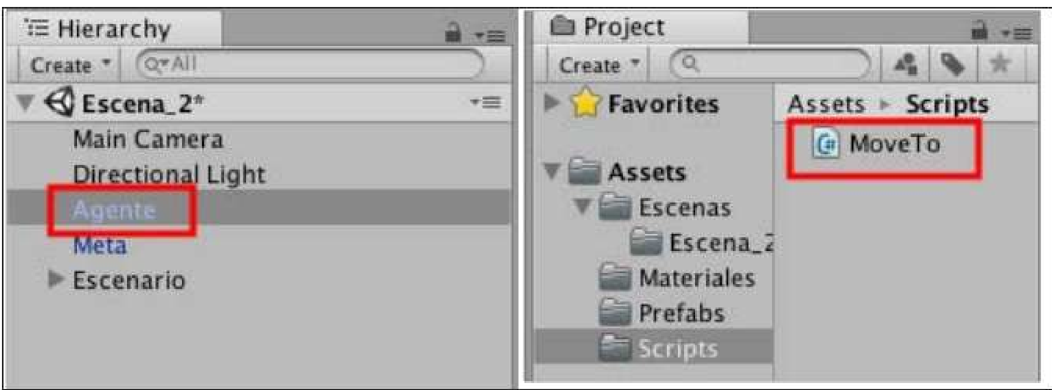


Fig. 11.18

Una vez añadimos el script a nuestro objeto Agente si miramos en su componente Script dentro de la ventana inspector veremos que aparece una propiedad con nombre Meta que está vacía por defecto porque en el script no se lo hemos indicado. Para rellenar esta propiedad seleccionamos desde la ventana **Hierarchy** el objeto Meta y lo arrastramos encima de la propiedad Meta. Otra forma es pinchando en el símbolo en forma de diana que encontramos en el lado derecho de la propiedad y en la ventana que aparece seleccionar el objeto Meta.

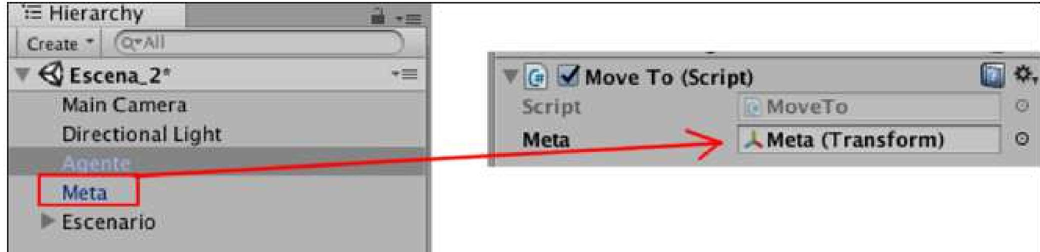


Fig. 11.19

Teniendo todos lo anterior preparado ya podemos ejecutar la escena y comprobar como el cilindro Agente se mueve por el escenario hasta la posición del objeto Meta.

5. Crear un NavMesh Obstacle

Los componentes **NavMesh Obstacle** los usaremos para determinar que objetos de la escena van a comportarse como obstáculos a los cuales los agentes deben evitar mientras navegan. Por ejemplo, los agentes deben evitar objetos controlados por físicas, como cajas y barriles mientras se mueven.

Para crear un **NavMesh Obstacle** accede a la carpeta escenas dentro de la ventana **Project** y realiza un doble clic encima de la **Escena_3** para que se cargue una escena preparada para facilitar la explicación. A continuación te muestro la imagen de como debería verse la escena.

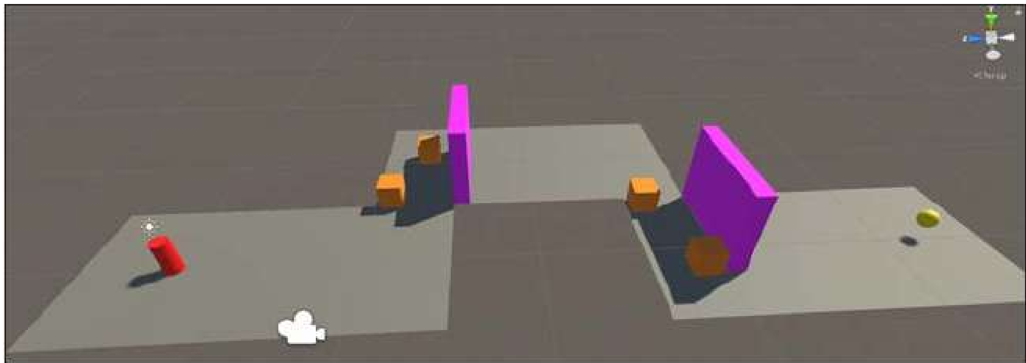


Fig. 11.20

Esta escena ya dispone de superficie **NavMesh** y también dispone de un Agente con el script aplicado como hemos realizado anteriormente. En este ejemplo si ejecutamos la escena veremos que el cilindro Agente se dirige hacia la esfera pero veremos que este atraviesa los cubos naranjas que tienen en la ventana **Hierarchy** el nombre de cajas. Para que el Agente no las atraviese deberemos añadir a cada caja el componente **NavMesh Obstacle**.

Para ello seleccionamos una de las cajas desde la ventana **Hierarchy** y accediendo a la ventana **Inspector** crearemos un nuevo componentes desde la opción **Add Component** > **Navigation**> **Nav Mesh Obstacle.**

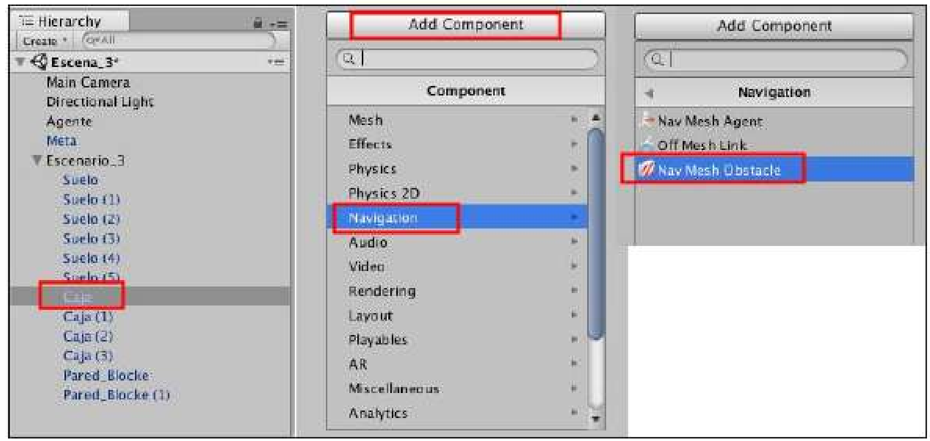


Fig. 11.21

Propiedades NavMesh Obstacle

Mientras el obstáculo se mueve, el **Agente** hace todo lo posible para evitarlo. Cuando el obstáculo es estacionario, hace un agujero en la superficie **NavMesh**. El Agente de malla de navegación cambia de ruta para rodearlo, o encuentra una ruta diferente si el obstáculo hace que la ruta se bloquee por completo.

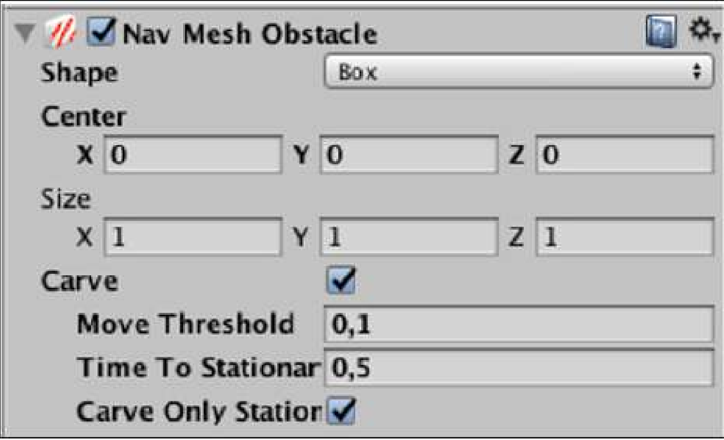


Fig. 11.22

- **Shape**: Esta propiedad nos permite determinar que forma geométrica va a adquirir el obstaculo.
- **Box**: La forma geométrica es la caja. Tenemos las propiedades **Center** que determina la posición de transformación del centro relativo de la propia caja. La otra propiedad **Size** se encarga de determinar el tamaño.
- **Capsule**: La forma geométrica es una capsula. Tenemos las propiedades de **Center** que determina la posición de transformación del centro relativo de la propia cápsula. Las otras propiedades son el radio (**Radius**) y la altura (**Height**) de la capsula.
- **Cave:** Cuando la casilla de verificación **Carve** está marcada, el Obstáculo de la malla de navegación crea un agujero en el **NavMesh**. Esto permite determinar que un trazado mas seguro.
    - **Move Threshold:** Los obstáculos cuando disponen de movimiento tienen una distancia establecida como umbral de movimiento, esta propiedad determina la distancia de este umbral.
    - **Time To Stationary:** Esta propiedad es el tiempo (en segundos) de espera hasta que el obstáculo se convierte en estacionario.
    - **Carve Only Stationary:** Cuando activamos esta opción el obstáculo crea un agujero en el NavMesh cuando está estacionario.

Una vez vistas la propiedades si hemos seleccionado una de las cajas y les hemos añadido el componente **NavMesh Obstacle** si ejecutas la escena podrás comprobar como el cilindro Agente evita la primera caja o en tu caso la caja a la que hayas aplicado el componente.

Esta actividad tienes varias cajas, selecciona una a una y añádeles el componente **NavMesh Obstacle** y varia los valores de los parámetros para ir probando como reacciona el Agente a las distintas opciones. En la siguiente imagen te muestro como utilizo dos de las cajas con la propiedad Cave activada y en las otras dos cajas con distintas formas geométricas.

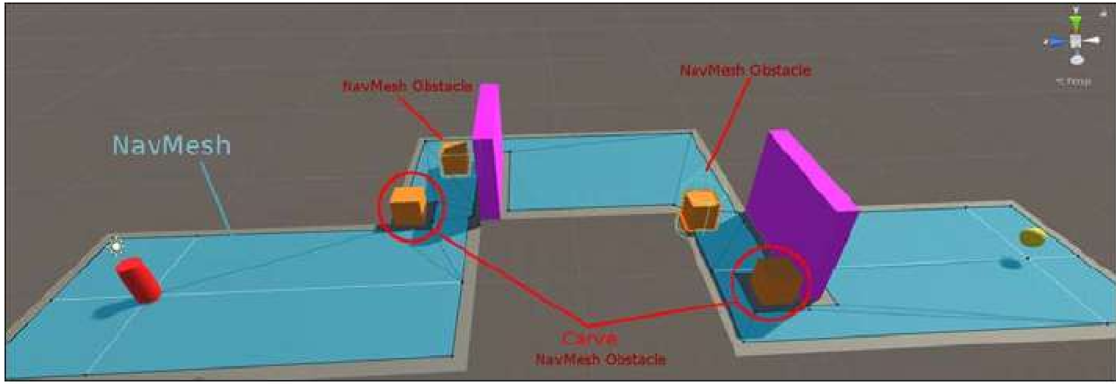


Fig. 11.23

6. Crear un Off-mesh Link

Los enlaces **Off-Mesh** se utilizan para crear rutas que acceden fuera de la superficie de la malla de navegación. Por ejemplo, saltar por encima de obstáculos o zonas , o abrir una puerta antes de atravesarla.

Para explicar los enlaces accede a la ventana **Project** y selecciona la carpeta escenas y la escena con el nombre **Escena_4**. La siguiente escena se te mostrará de la siguiente manera.

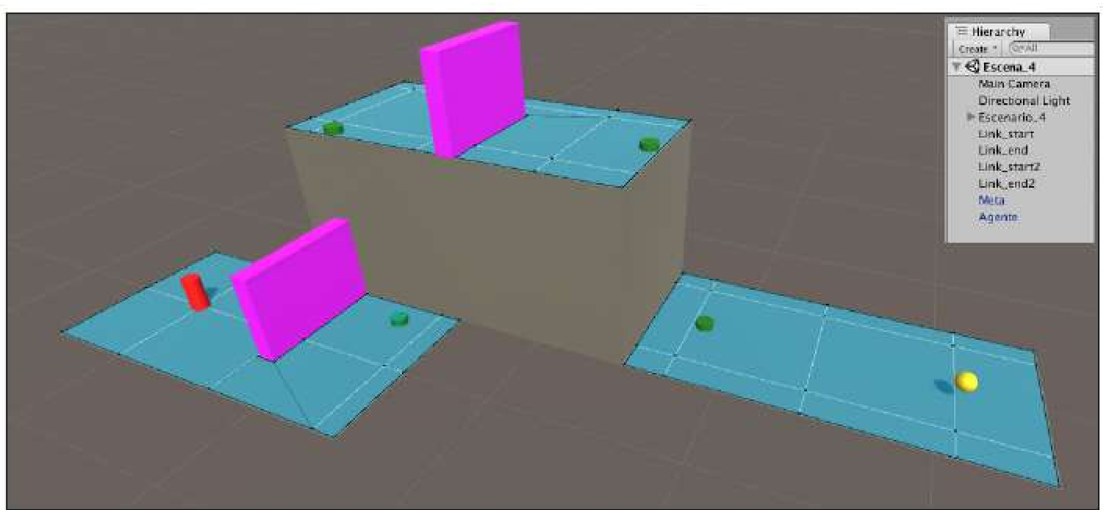


Fig. 11.24

Esta escena ya contiene un agente con un script para que este cuando ejecutemos la escena se desplace para buscar el objeto Meta que es la esfera amarilla. En este caso si ejecutamos la escena veremos como el Agente se desplaza en dirección a la esfera amarilla pero se queda bloqueado en al final cuando llega al muro.

Para resolver esta situación utilizamos estos objetos verdes que son simplemente cilindros escalados. Solamente a los que van a ser los enlaces de inicio les vamos a añadir un componente **Off-Mesh Link**

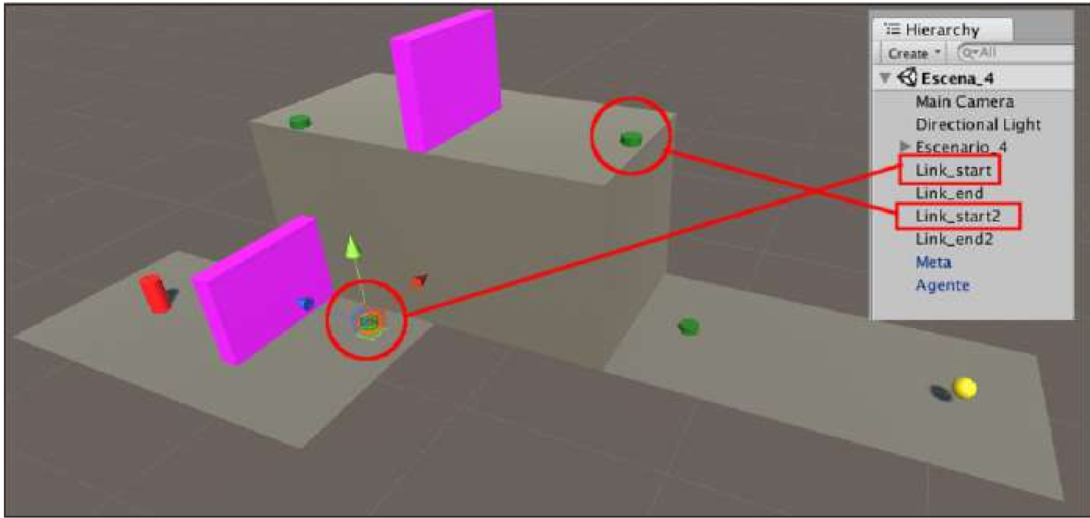


Fig. 11.25

Primero seleccionamos uno de los objetos que te muestro en la imagen anterior por ejemplo el **Link_start** y accede a la ventana Inspector y añadimos el siguiente componente desde el botón **Add Components > Navigation > Off Mesh Link.** Ahora realiza la misma acción con el objeto **Link_start2.**

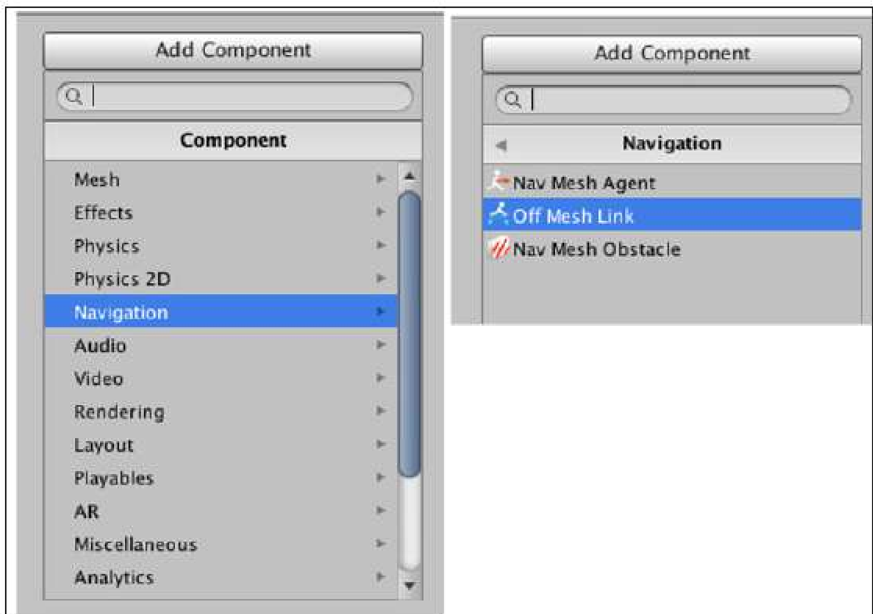


Fig. 11.26

Propiedades Off-Mesh Link

A continuación vamos a ver de que propiedades disponemos en el componente **Off-Mesh Link** y como podemos utilizarlo.

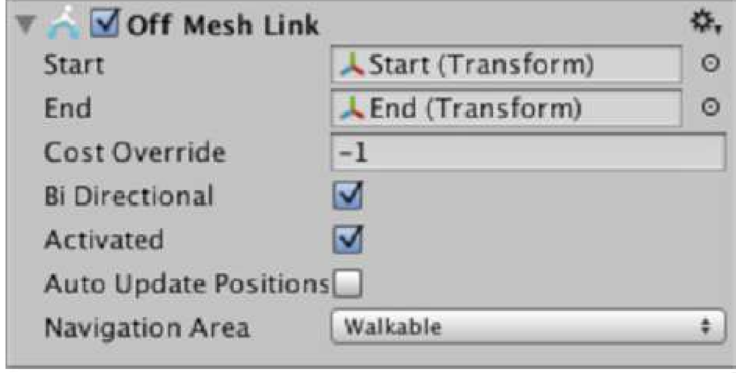


Fig. 11.27

- **Start:** Aquí es donde ponemos la referencia de la localización del objecto que determina el inicio del **Off-Mesh Link**.
- **End:** Aquí es donde ponemos la referencia de la localización del objecto que determina el final del **Off-Mesh Link.**
- **Cost Override:** Si el valor es positivo, se utiliza cuando se calcula el coste de procesamiento al encontrar la ruta. SI no es el caso, se utiliza el valor predeterminado (el costo del área a la que pertenece este objeto del juego). Si el **Cost Override** se establece en el valor 3.0, moverse sobre el enlace de malla oculta será tres veces más costoso que mover la misma distancia en un área **NavMesh** predeterminada. Al utilizar un valor negativo el **Cost Override** es mucho más favorable para que los agentes caminen, pero se usa el enlace sin malla cuando la distancia de caminata es claramente más larga.
- **Bi-Directional:** Si está habilitado, el enlace puede atravesarse en cualquier dirección. De lo contrario, solo se puede atravesar de principio a fin.
- **Activated:** Especifica si este enlace lo usará como ruta alternativa o simplemente se ignorará si se establece en false.
- **Auto Update Positions:** Cuando está habilitado, el enlace **Off-Mesh** se volverá a conectar al **NavMesh** cuando se muevan los puntos finales. Si se deshabilita, el enlace permanecerá en su ubicación de inicio incluso si se mueven los puntos finales.
- **Navigation Area**: Describe el tipo de área de navegación del enlace. El tipo de área nos permite aplicar un **Cost Override** común a tipos de área similares y evitar que ciertos objetos o caracteres accedan al **Off-Mesh Link** en función de la **Máscara de Área** del agente.

Una vez hemos visto las propiedades del componente vamos a indicarle el objeto que necesita el parámetro **start** y el objeto del parámetro **End** del objeto con el nombre **Link_start**. Es decir seleccionamos el objeto **Link_start** y accedemos a sus componentes en la ventana Inspector. Dentro de sus componentes accedemos al componente **Off Mesh Link** y en el parámetro **Start** podemos pinchar en el símbolo con forma de diana y seleccionar el objeto **Link_start** que en este caso es el propio objeto el que va a ser el enlace de inicio. En el parámetro **End** pincharemos en el símbolo con forma de diana que tenemos al lado y seleccionaremos el objeto **Link_end** que será el objeto que finalice el enlace.

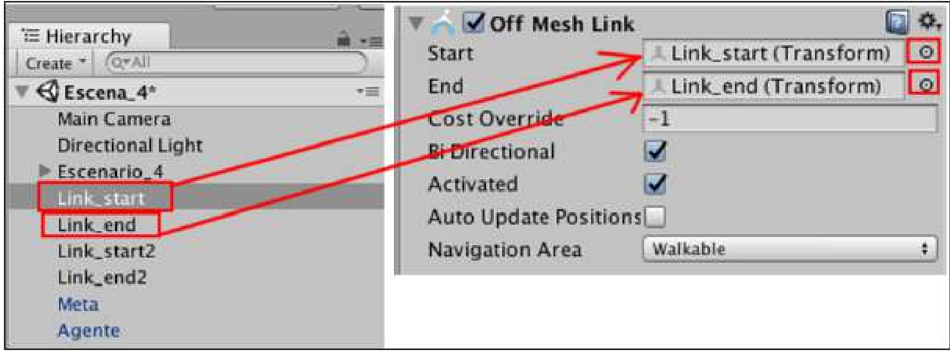


Fig. 11.28

Ahora debemos realizar la misma acción con el objeto **Link_start2**. Es decir seleccionamos el objeto **Link_start2** accedemos a la ventana Inspector y dentro de su componente **Off Mesh Link** añadimos para el objeto **Start** el propio objeto que se llama **Link_start2** y para el final del enlace **End** añadimos el objeto **Link_end2**.

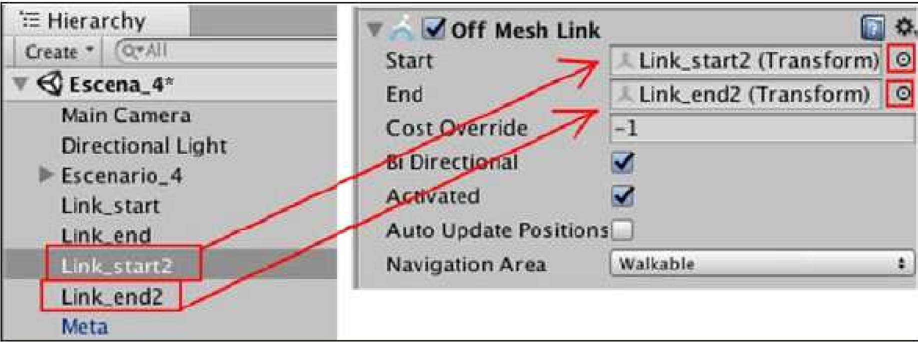


Fig. 11.29

Una vez hemos determinado los objetos de inicio y final de esta escena ya puedes ejecutar la escena y ver como el objeto Agente se desplaza utilizando los objetos enlace para llegar al objeto Meta.

7. Proyectos de Navigation

Ahora que hemos visto los elementos básicos te propongo una serie de actividades para ponerte a prueba con distintas situaciones en las que puedes utilizar la navegación. En un principio enuncio cual es el proyecto y luego te muestro como lo resuelvo. Te aconsejo que intentes realizar estos **mini-proyectos** sin mirar la solución he intentar llegar a la solución tu mismo. No existe una sola solución para resolver estos problemas así que es posible que encuentres soluciones mucho más satisfactorias que las que te muestro.

Recorrido de un Agente hasta un destino

Este primer proyecto es muy simple y es el mas básico de todos, debes crear un objeto Agente y un objeto Destino, crear una superficie de recorrido y si lo deseas crear objetos

que hagan de obstáculos. En este caso deberás crear un script que permita al agente encontrar su destino.

Para este proyecto dispones de una escena en el material de este capitulo con el nombre Proyecto_1 en donde tienes resuelto esta actividad, te recomiendo que intentes realizar la actividad tu mismo y que crees tu propio escenario y tu propio script.

A continuación te muestro el ejemplo que encontrarás en el proyecto.

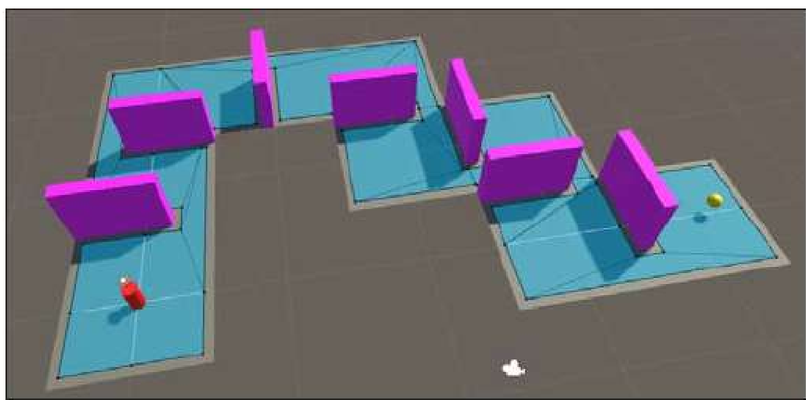


Fig. 11.30

Realizar este ejemplo no debería su ponerte ningún problema, si has seguido el capitulo desde el principio, porque exactamente lo mismo que hemos estado explicando desde el principio.

Recorrido de un Agente mediante el cursor.

En este segundo proyecto ya tienes conocimientos de como crear los distintos elementos para crear un ejemplo de Navegación. Ahora el objetivo es cambiar el destino del agente por el que marque el cursor del ratón. Para realizar este ejemplo recurro al temario que vimos en el capitulo 8 el **Raycast**. La idea es substituir el objeto de destino, por un **Raycast hit** realizado por un clic del ratón. La escena la encontraras resuelta en el material de este capitulo, sigo recomendándote que intentes realizar el ejemplo sin mirar la solución.

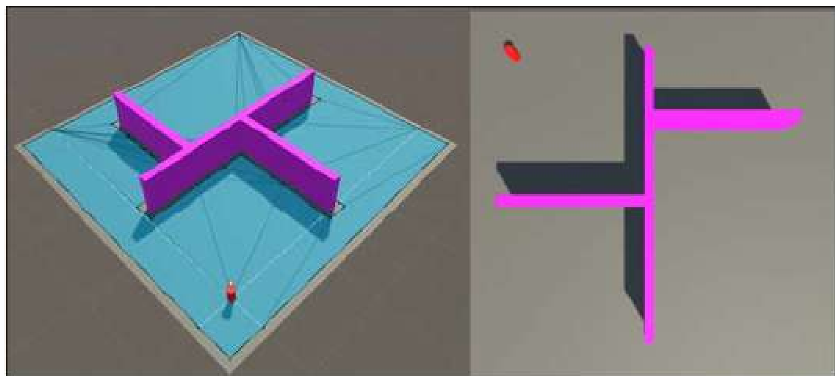


Fig. 11.31

El objetivo de estos ejercicios es mostrarte herramientas para que luego puedas utilizar para crear tus propios prototipos y proyectos.

Recorrido de un Agente siguiendo un patrón.

En el último proyecto vas a tener que crear un recorrido establecido para el agente para que una vez llegue a un destino vaya en busca de otro destino. En el caso del proyecto que tendrás en el material del capitulo se ha creado cuatro objetos meta que mediante un script con una array el agente recorrerá las distintas posiciones del escenario.

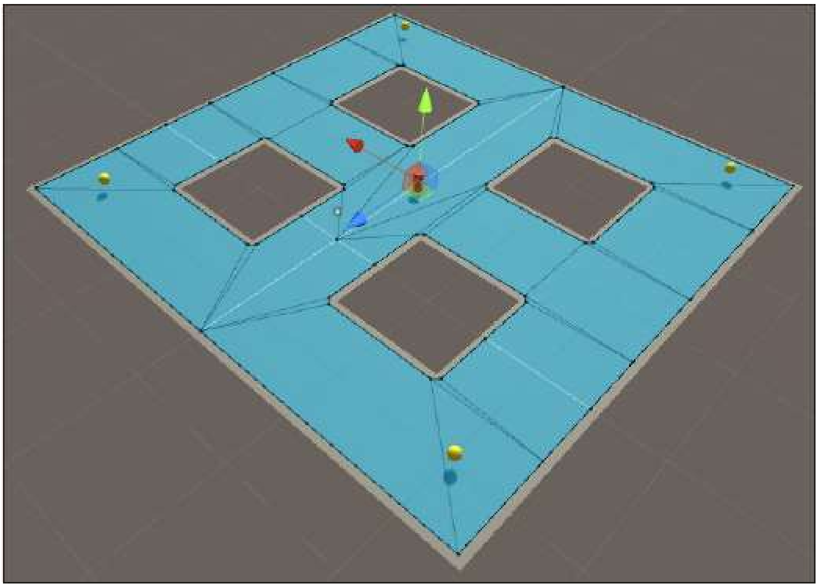


Fig. 11.32

Los componentes **NavMesh Obstacle** se pueden usar para describir los obstáculos que los agentes deben evitar mientras navegan. Por ejemplo, los agentes deben evitar objetos controlados por la física, como cajas y barriles mientras se mueven.

Capítulo 12
Iluminación

- Introducción
- Iluminación
- Apagar las luces
- Tipo de luces
- Propiedades de las luces
- Iluminación directa e indirecta
- Iluminación Baked
- Iluminación Mixed
- Práctica general
- Ponte a prueba

1. Introducción

La iluminación es un tema complejo no por las características técnicas sino por el conocimiento y la experiencia que se necesitan para poder adaptarse a los proyectos según la plataforma que se quiera utilizar. El objetivo es tener una buena iluminación sin abusar del proceso de calculo de nuestra estación de trabajo, es por eso que se empieza por explicar donde encontrar la configuración de la iluminación para después poder ver uno a uno los distintos puntos de luz.

Como siempre para seguir correctamente este capitulo debes importar un paquete de assets que te permitirán empezar con la materia directamente. El paquete lo encontraras como siempre en la carpeta de proyectos del capítulo correspondiente con el nombre **Assets_Escena_Capitulo_12.unitypackage.** Recuerda que para importar este paquete debes acceder al menú principal **Assets > Import Package** y en la nueva ventana tener todos los archivos seleccionados y aceptar la importación. A continuación seguir las explicaciones siguientes.

2. Iluminación

La iluminación es una parte muy importante dentro de cualquier videojuego, nos permiten crear entornos cargados de emociones y sensaciones si se utilizan correctamente. Para empezar a ver como funciona la iluminación vamos a ver donde podemos encontrar la configuración de esta pero antes de todos si has cargado el paquete de assets para el capitulo abre la escena **Iluminacion_base** que encontrarás en la carpeta Escenas dentro de la ventana **Project**.

Te encontrarás con una escena muy simple compuestas por varios objetos suelos agrupados en un objeto vacío con el nombre **Suelo_General** y dos **Prefabs**; un Barril y una Caja madera.

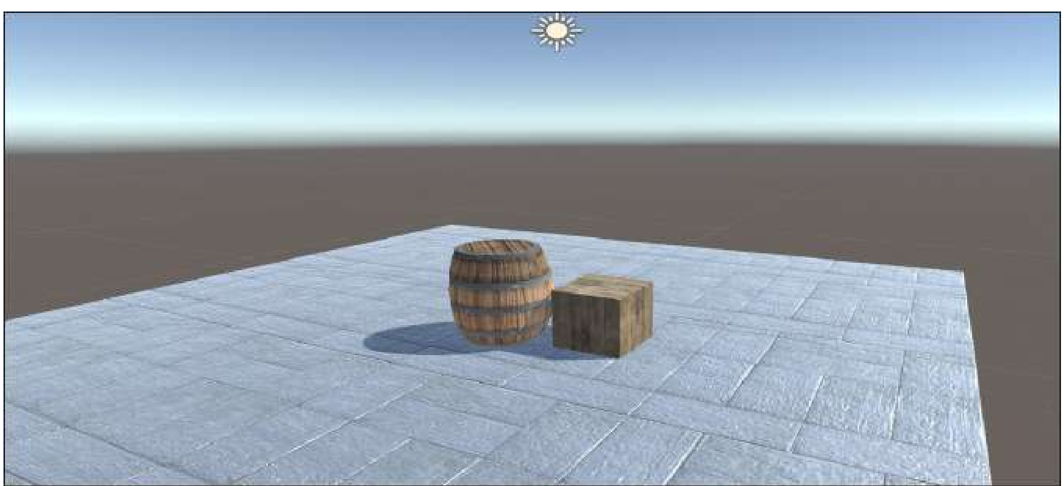


Fig. 12.1

En la barra de menús principal accedemos a **Window > Lighting > Settings** y se nos abrirá una ventana que contiene la configuración de la iluminación.

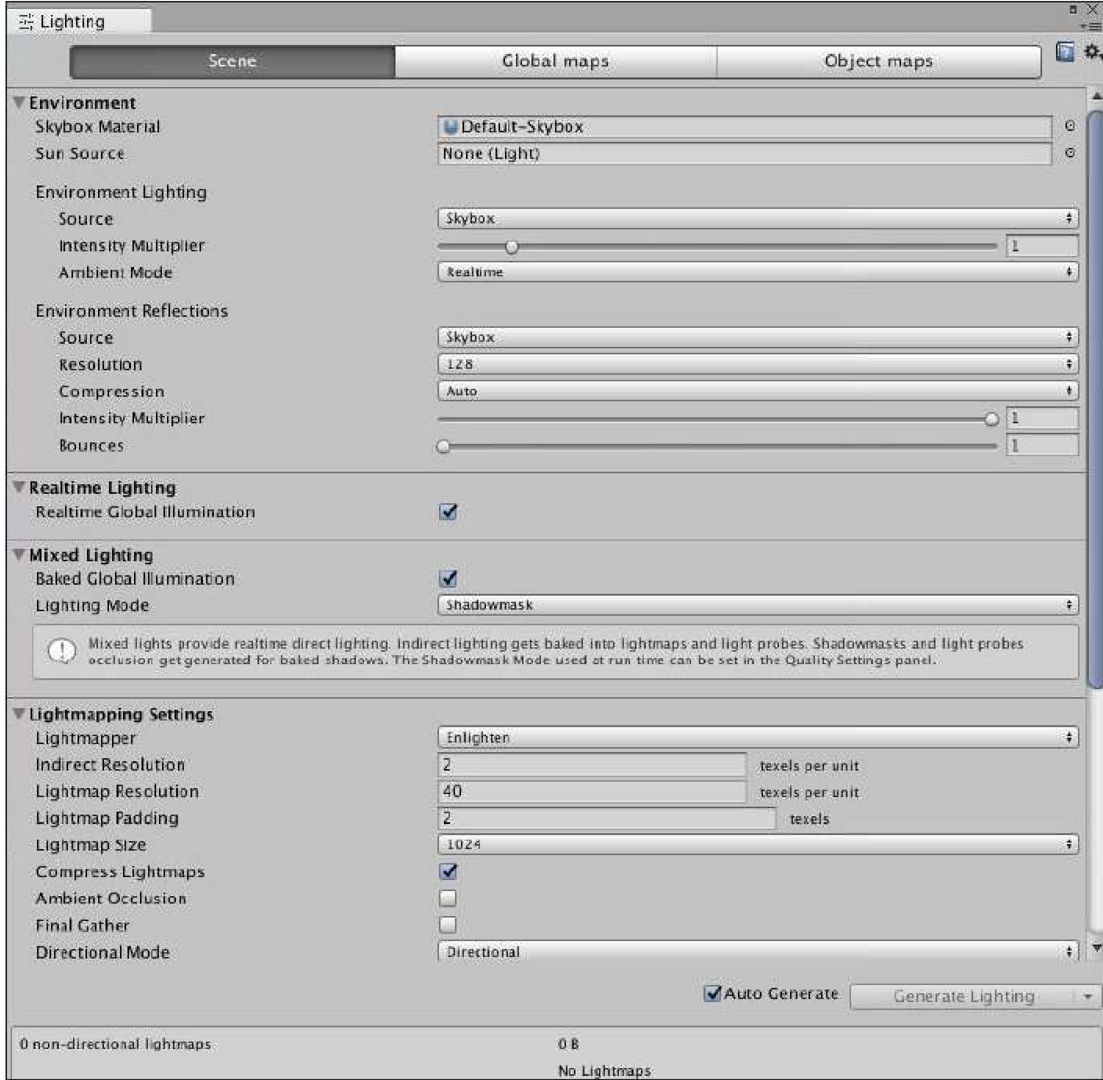


Fig. 12.2

Esta iluminación contiene 3 apartados que son Scene, Global maps y Object maps.

- **Scene:** Se aplica a la escena general en lugar de GameObjects individuales. Estas configuraciones controlan los efectos de iluminación y también las opciones de optimización.
- **Global maps:** Muestra todos los archivos activos que tienen que ver con los mapas de luz o lightmap que son generadas por un proceso llamado iluminación Global GI.
- **Object maps:** Muestra las vistas previas de lightmaps para los objetos seleccionados.

Auto Generate: Esta opción la encontramos abajo del todo en forma de casilla de verificación o checkbox. En este caso si tenemos activada esta opción Unity actualiza los datos de lightmap en casi en tiempo real mientras estamos editando nuestra escena, es decir la actualización no es instantánea pero se realiza segundos después de la variación. Por el contrario si la desactivamos se nos activa la opción que se encuentra a su lado Generate Lighting. Esta nueva opción nos permite actualizar la iluminación cuando lo pulsemos de este modo la escena no se actualizará a cada momento.

A continuación tienes un resumen general de que contienen estos parámetros en la opción Scena.

Propiedades del Lighting Scene.

Environment

Esta sección contiene la iluminación que envuelve el ambiente y para ello contiene parámetros para el skybox, la iluminación difusa y los reflejos.

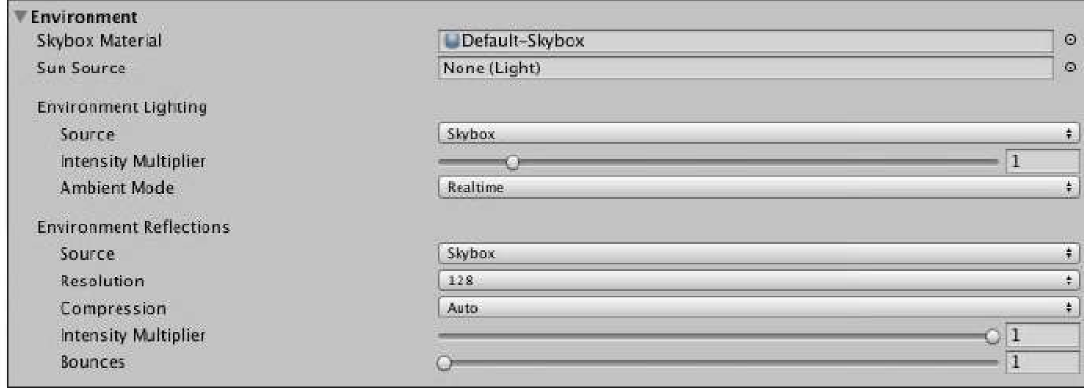


Fig. 12.3

- **Skybox Material:** Este material aparece detrás de todos los demás para dar la sensación o simular un cielo lejano. En este parámetro podemos seleccionar un material de este tipo. Por defecto Unity uno en todas la nuevas escenas. Si desactivamos esta opción accediendo al símbolo que tiene al lateral y seleccionando la opción None del menú que nos aparece la escena se nos muestra de este modo.

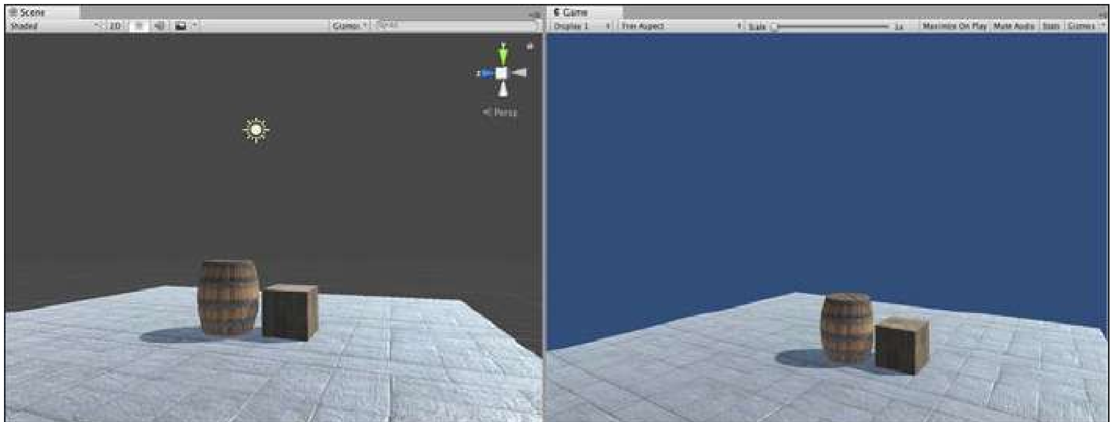


Fig. 12.4

- **Sun Source:** cuando disponemos de un Skybox este apartado podemos seleccionar un objeto con un componente de tipo Luz direccional para indicar la dirección de la luz o en el caso de que se determine como ninguno Unity toma por defecto la luz direccional más brillante de la escena como representante de la luz solar. En el caso del ejemplo que tenemos en escena Unity determina la luz direccional que hay como fuente de luz más brillante.

El siguiente apartado dentro de Lighting es el Environment Lighting que contiene parametros que afectan a la luz que proviene de la distancia del entorno o ambiente.

- **Source:** la luz ambiental es la luz que encontramos presente en toda la escena y no proviene de ninguna fuente de luz en concreto. Esto nos permite poder jugar con tres valores:
  - **Color:** Este valor lo podemos utilizar para tintar el ambiente y dar cierta atmósfera.
  - **Gradient:** Podemos separar los colores de la luz en un color para el cielo , otro para el horizonte y por ultimo uno para el suelo y luego crear un suavizado entre ellos.
  - **Skybox:** En el caso de que tengamos un material Skybox podemos utilizar esta opción para que el ambiente tome las especificaciones del propio material.

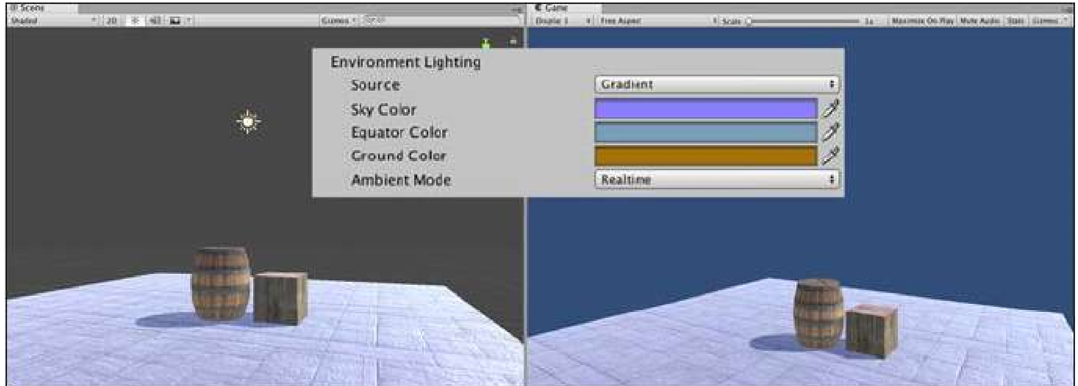


Fig. 12.5

Como veras en la imagen anterior he puesto un Degradado con estos colores para que se vea reflejado en el ambiente.

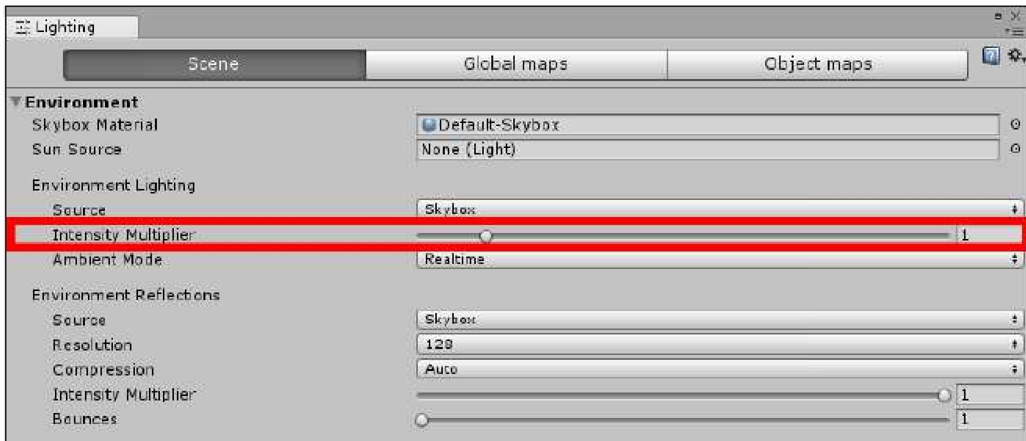


Fig. 12.6

- **Intensity Multiplier:** este parámetro nos permite establecer el brillo de la luz ambiental en la escena. Los valores van del 0 al 8 y por defecto encontraremos este valor en 1.
- **Ambient Mode:** este parámetro lo utilizaremos para especificar que modo de iluminación global va a utilizar la luz ambiental. Esta opción solo está disponible cuando están habilitadas tanto la iluminación RealTime como la Baked.

El siguiente apartado **Environment Reflections** se encarga de la configuración encargada de las reflexiones que se perciben en los objetos.

- **Source:** Aquí podemos determinar si utilizamos la **skybox** como efecto de reflexión o un mapa cubico (**cubemap**) que realiza la misma función que un **skybox**.
- **Compression:** Determina si las texturas de reflexión están comprimidas o no.
- **Intensity Multiplier:** El grado en que la fuente de reflexión es visible en los objetos que reflejan.
- **Bounces:** el rebote es un efecto que ocurre cuando un reflejo de un objeto es reflejado por otro objeto. Un ejemplo seria los dos espejos que se reflejan uno a otro infinitamente, pero en este caso si establecemos el valor en 1, Unity solo tomara la reflexión inicial.

Realtime Lighting

Si esta casilla de verificación (checkbox) está marcada, Unity calcula y actualiza la iluminación en tiempo real.

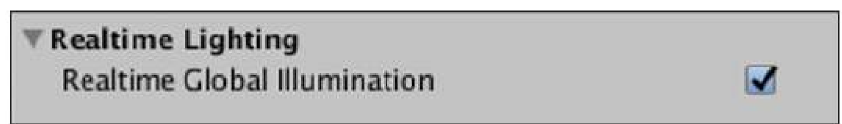


Fig. 12.7

Mixed Lighting

En este apartado tenemos los parámetros de un tipo de iluminación mixta.

- **Baked Global Illumination:** Si esta casilla de verificación está marcada, Unity precalcula la iluminación y la establece en la escena en tiempo de ejecución.
- **Lighting Mode:** El modo de iluminación demuestra la forma en que las luces y sombras mixtas funcionan con GameObjects en la escena.
- **Realtime Shadow Color:** Permite definir el color usado para renderizar sombras en tiempo real.

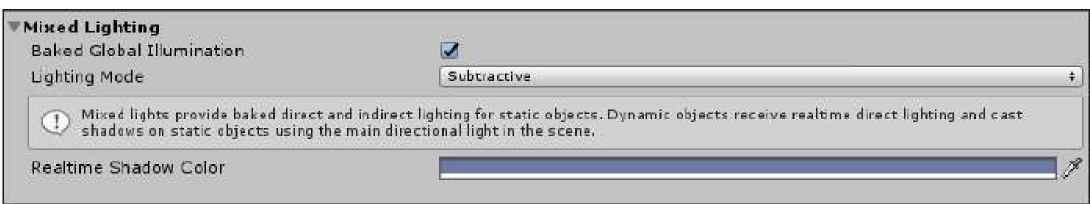


Fig. 12.8

Lightingmapping Settings

En este apartado tenemos una configuración que tiene parámetros propios pues estos no se comparten y son específicos del backend Lightmapper. No quiero entrar mucho en detalle porque podemos extender la explicación a varios capítulos pero dispone de dos opciones:

- **Progressive Lightmapper:** es un sistema de mapa de luz basado en rutas localizadas para proporcionar luces que están definidas por un mapa (Backed) y Light Probes que se actualizan en el editor.
- **Enlighten:** Unity proporciona dos técnicas distintas para el precalculo de la iluminación global (GI) y la iluminación rebotada. Estos son iluminación global (backed) y iluminación global precalculada en tiempo real. El sistema de iluminación Enlighten brinda soluciones para ambos.

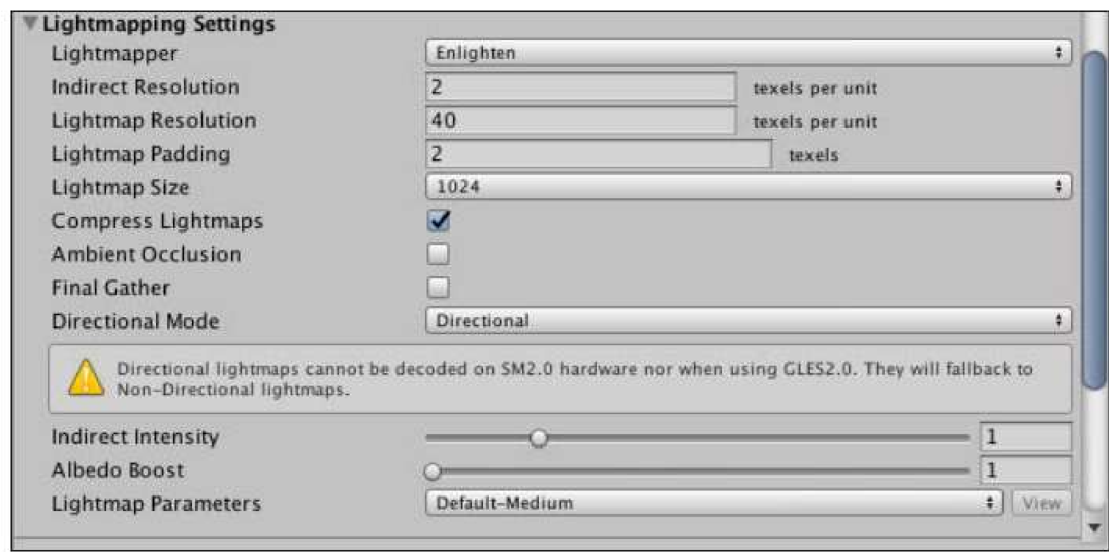


Fig. 12.9

Other Settings

En este apartado tenemos una configuración referentes a algunos efectos para el ambiente como son brillos y efectos de le luz.

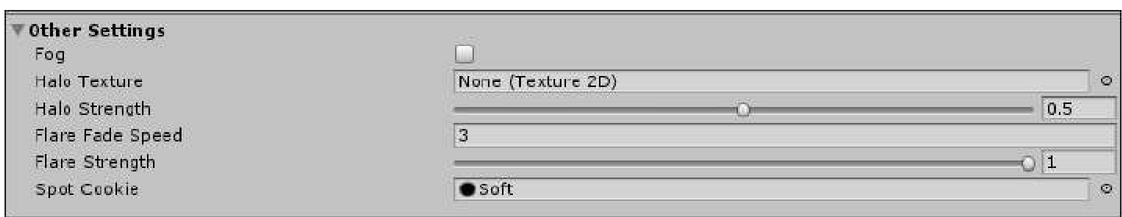


Fig. 12.10

- **Fog:** Activa o desactiva la niebla en la escena.
- **Halo texture:** Configura la textura que quieres para utilizar en el dibujo de un Halo al rededor de la luz.
- **Halo Strength:** Define la visibilidad de Halos alrededor de la luz, para ello utiliza un valor entre 0 y 1.
- **Flare Fade Speed:** Define el tiempo que tarda en segundos para que se desvanezca los reflejos de una lente después de ser reflejada.
- **Flare Strength:** Define la visibilidad de la lente utilizando valores que van desde el 0 hasta el 1.

- **Spot Cookie:** En las luces spot Light podemos configurar una textura para que esta determine la forma de la luz .

Debug Settings

En este apartado tenemos ciertas configuraciones para ayudar a depurar la escena. No quiero entrar en más detalle y prefiero centrarme a continuación en algo mucho más práctico.

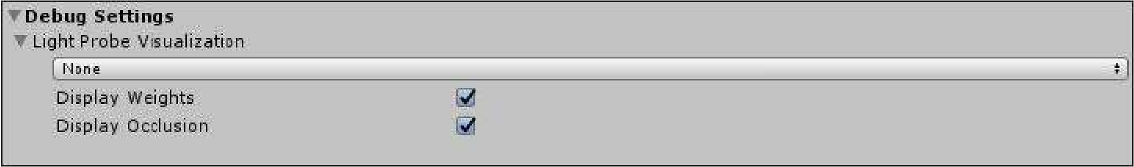


Fig. 12.11

3. Apagar las luces

Ahora vamos a empezar a ver la iluminación con un poco de práctica. Para ello si tienes abierta la escena Iluminacion_base que se encuentra en la ventana Project en la carpeta Escenas.

Para apagar toda la iluminación de una escena debemos acceder a la ventana Lighting que hemos explicado en el apartado anterior. Ahora accedemos a la pestaña Scene y en el apartado Skybox Material seleccionaremos la opción none. Realizamos la misma acción en Sun Source.

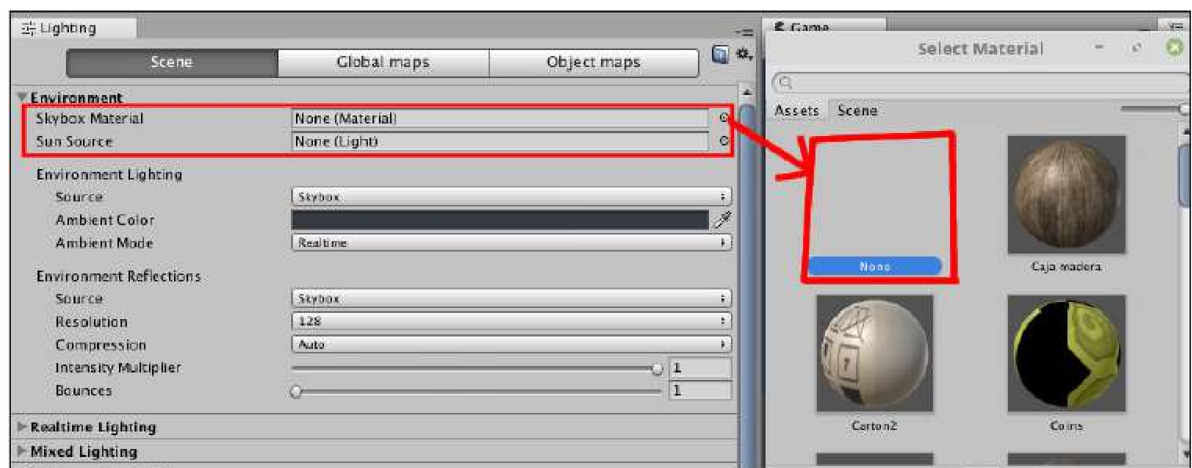


Fig. 12.12

Si realizas estos cambios la escena debería oscurecerse como te muestro a continuación.

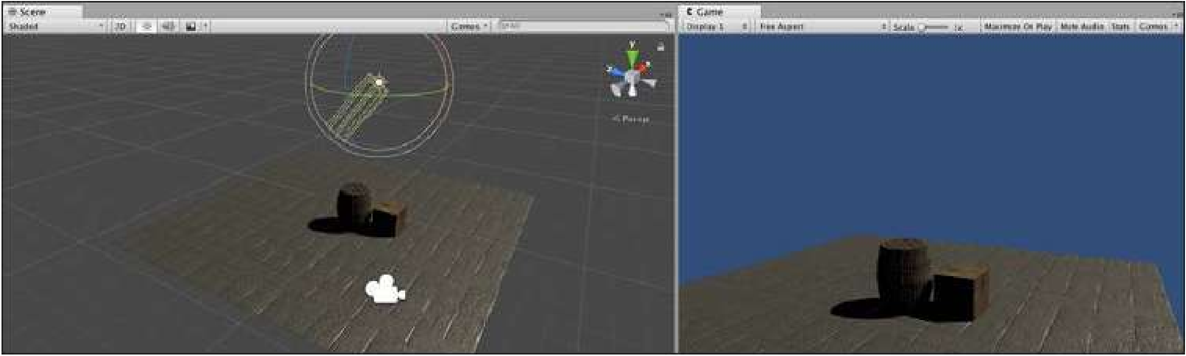


Fig. 12.13

Pero todavía no está oscura del todo y es porque tenemos una Luz direccional por defecto y unity toma el objeto mas brillante como punto de luz cuando no disponemos en el SunSource de la ventana Lighting, de ningún objeto seleccionado. A continuación selecciona la Luz direccional y desactivarla y veras como la escena queda a oscuras.

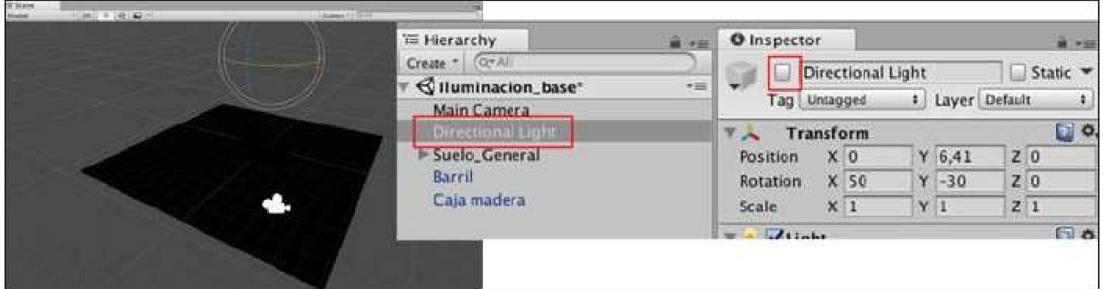


Fig. 12.14

Como puedes ver esta escena queda oscurecida, en el caso de que no te quede negra la escena deberás tocar el parámetro Ambient color.

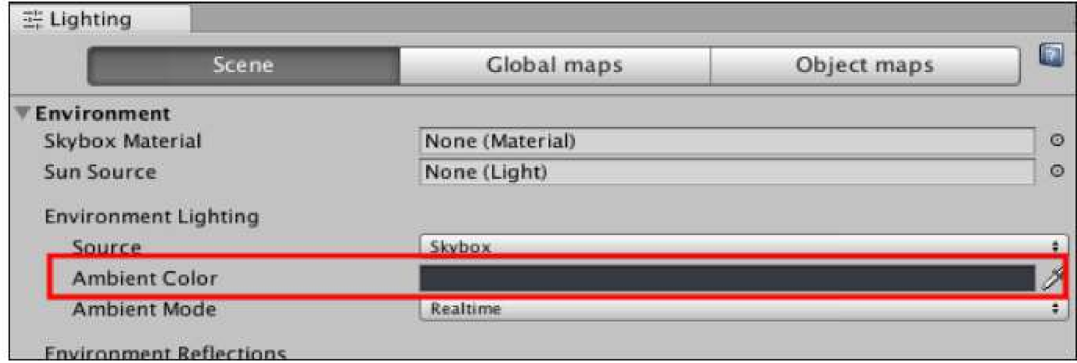


Fig. 12.15

Como consejo personal es que utilices negro en el caso de que quieras iluminar de una forma un poco más realista el entorno. De todos modos la iluminación tiene mucha relación con los shaders. En el caso de que el Ambient Color lo pusiéramos a blanco iluminaria los objetos a pesar de que no hubiera luces.

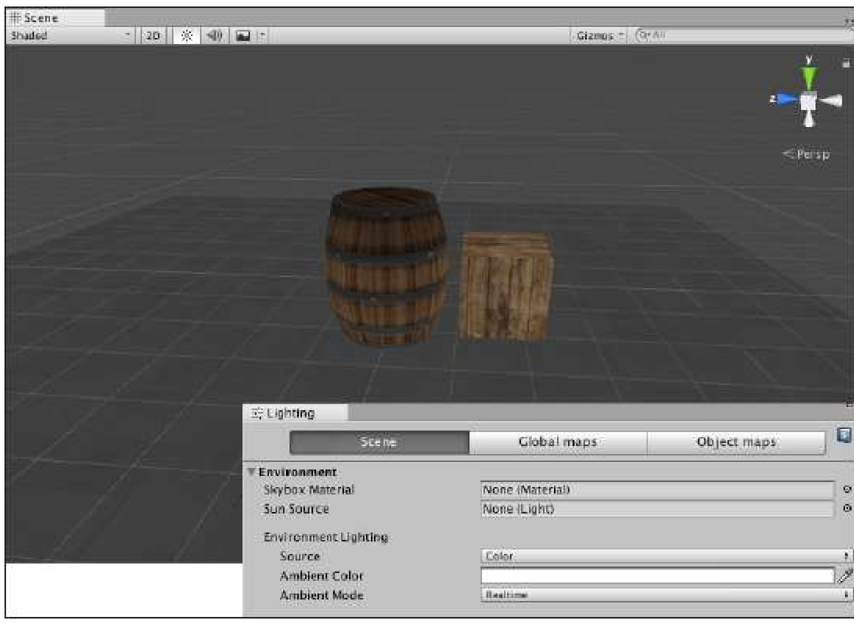


Fig. 12.16

4. Tipos de luces

Continuamos con la misma escena con la opción Ambient Color a Negro, también puedes abrir la escena que encontraras en la ventana Project > Escenas> Iluminación_Luces. En esta escena vamos a ver los tipos de iluminación básicos y ver como configurar-los. A continuación te muestro una imagen con el icono que representa cada luz en la ventana Escena.

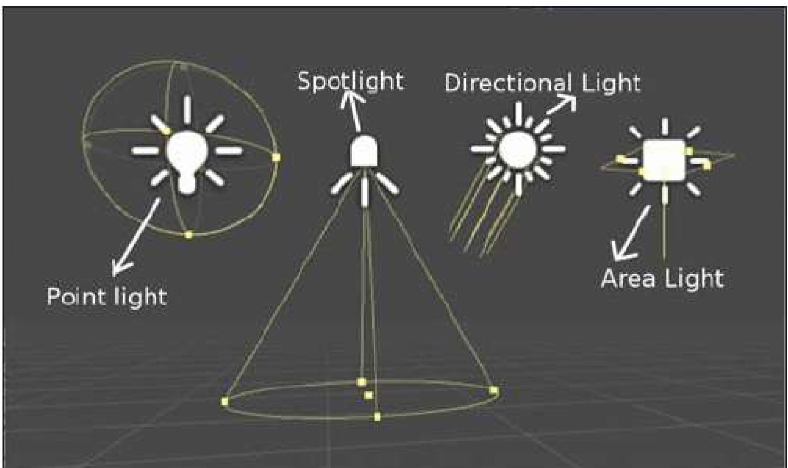


Fig. 12.17

Point lights: Se trata de una luz con forma de punto que proyecta luz en todas las direcciones por igual. Este tipo de luz pierde intensidad conforme el rayo proyectado se aleja del centro. La intensidad de la luz utiliza la “ley del cuadrado inverso” que nos dice que la intensidad de la luz es inversamente proporcional al cuadrado de la distancia

desde la fuente. Las luces de punto se utilizan mucho para simular lámparas y fuentes de luz localizadas. También se pueden utilizar para explosiones y otros efectos que necesiten de una iluminación puntual y localizada.

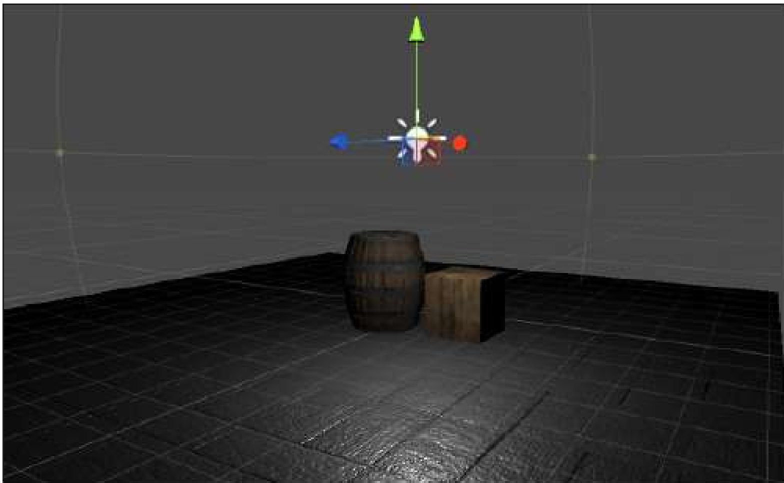


Fig. 12.18

- **Spot lights:** Este tipo de luz es parecido a la luz de punto puesto que podemos ubicarla en un punto concreto, pierde intensidad con la distancia, sin embargo, esta luz esta limitada por un angulo y esto nos proporciona una región iluminada en forma de cono. Esta luz proyecta un rayo de luz hacia delante (eje z), esta luz disminuye en los bordes del cono y al ensanchar el ángulo aumenta el ancho del cono y aumenta el tamaño de fundido de la luz o también conocido como penumbra. Los focos de luz se pueden utilizar para proyectar luces artificiales como flexos automóviles. También este tipo de luces nos permiten iluminar zonas concretas de una escena creando efectos mas emotivos.

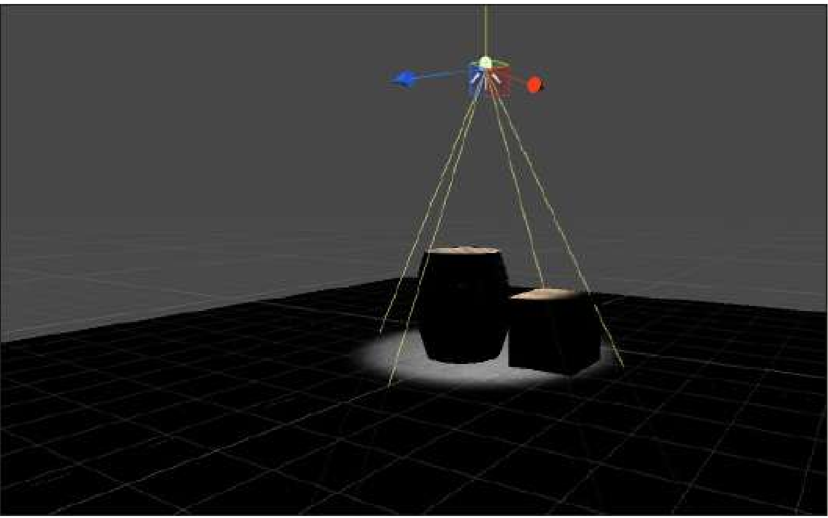


Fig. 12.19

- **Directional lights:** Este tipo de luz es direccional no es necesario posicionar lo en un punto concreto porque ilumina toda la escena, todos los objetos desde el mismo angulo. Otro aspecto

importante es que la intensidad no disminuye porque la distancia de la luz del objeto objetivo no está definida. Esta luz la utilizamos para simular el sol o la luna.

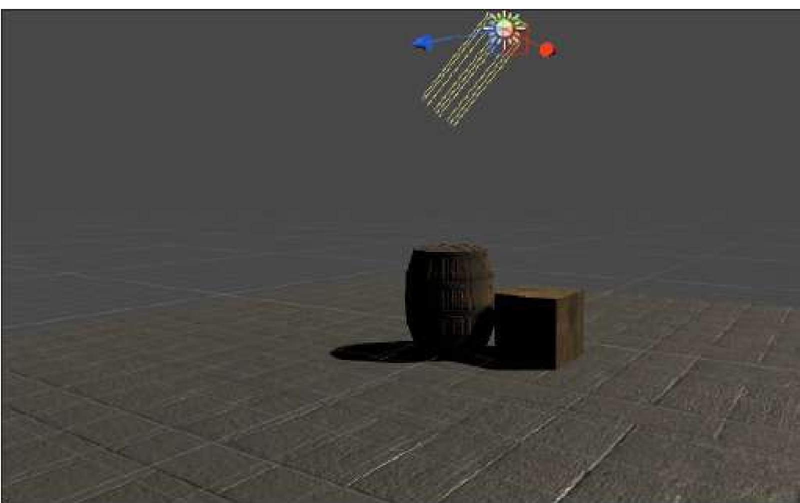


Fig. 12.20

- **Area lights:** Este tipo de luz esta definido por un rectángulo en el espacio. La luz se proyecta en todas las direcciones uniformemente a través de uno de los lados del rectángulo. El alcance de la luz no podemos controlarlo pero si que podemos escalar la luz, es decir al escalar el rectángulo que define la luz nos permite ampliar la superficie que emite la luz. Este tipo de luz también disminuye la intensidad al ampliar la distancia. El calculo de este tipo de luz repercute bastante en el gasto de recursos del procesador, es por ese motivo que se utilizan para crear mapas de luces.

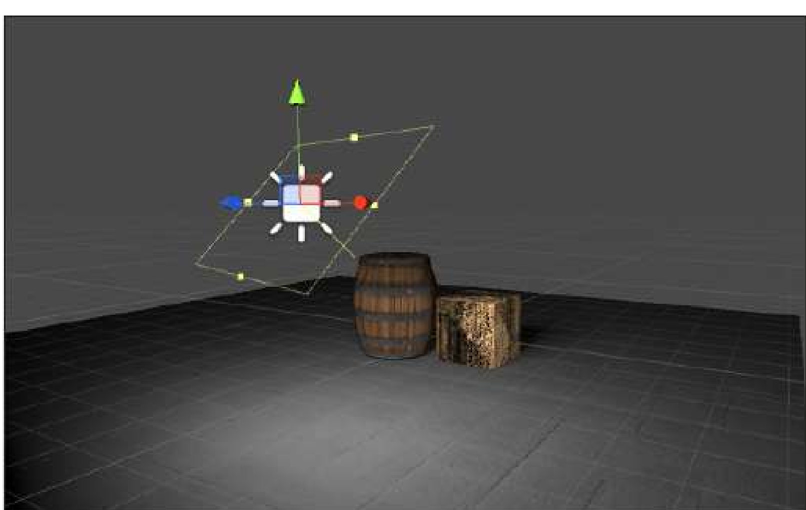


Fig. 12.21

También existe otra forma de iluminar y es utilizando objetos con un material que emiten luz. Para ello te pido que utilices el cubo que tenemos en esta escena y le añadas el material EmitirLuz. Te muestro como va a quedar la escena al final y luego pasaremos a explicar paso a paso que he utilizado.

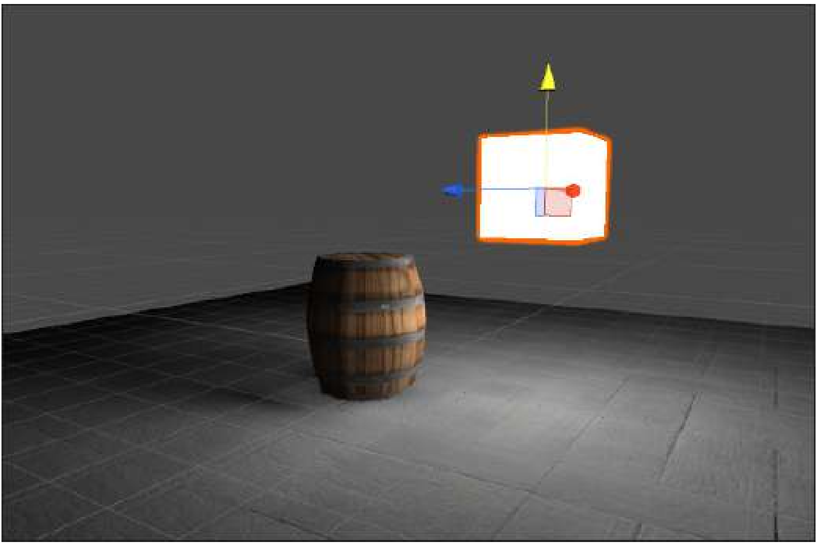


Fig. 12.22

Para acceder a este material debes ir a la ventana Project > Modelos > Materiales > EmitLuz. Si miramos los parámetros de este material veras que es un material de tipo Standard (Specular) y tenemos activada la opción Emission.

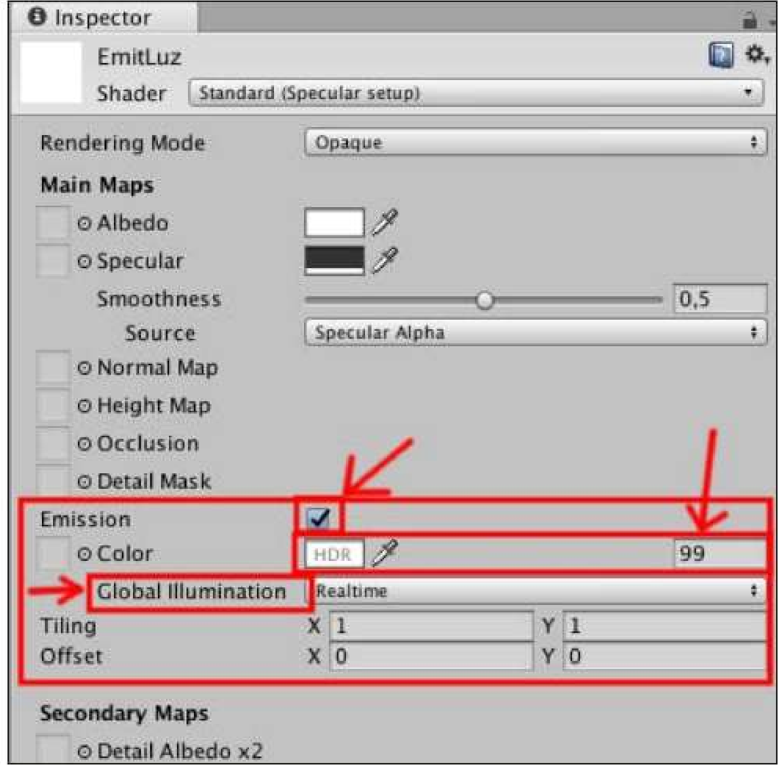


Fig. 12.23

En este apartado puedes variar el color y darle intensidad en este caso tiene un valor de 99 unidades. En el apartado Global Illumination se utiliza Realtime. Este ultimo parámetro es importante para configurar las propiedades de la ventana Lighting. Antes de

arrastrar el material al objeto Caja madera asegurate de seleccionar todos los objetos de escena que no son luces y activa la opción Static.

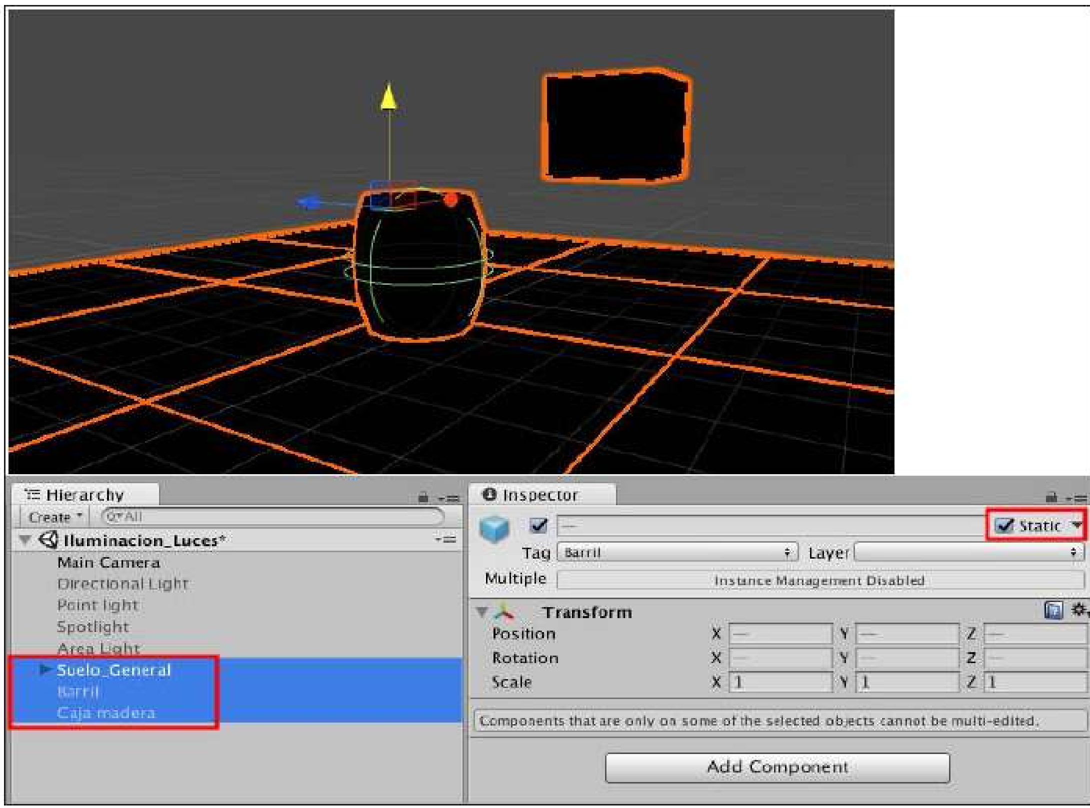


Fig. 12.24

El siguiente paso es asegurarnos de que en la configuración de la ventana Lighting tenemos activada la opción Realtime Lighting y Auto Generate. Por el contrario desactivamos si es que no lo está la opción Mixed Lighting.

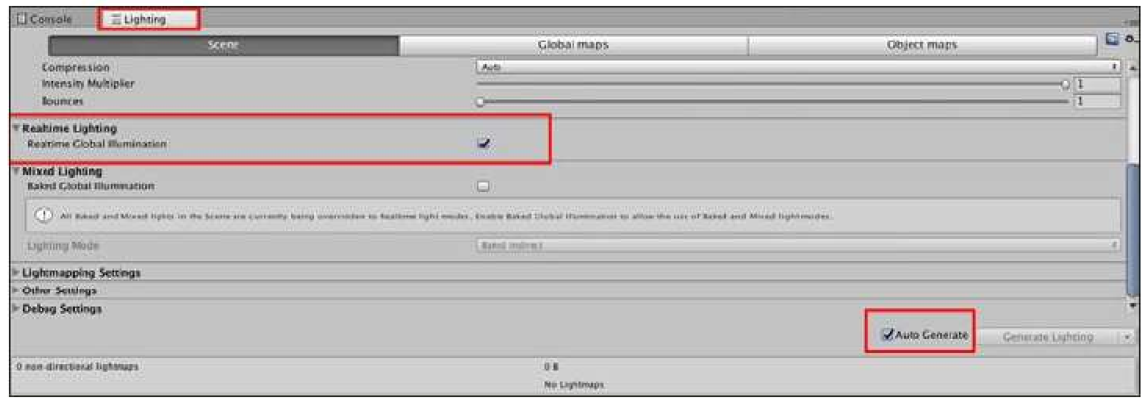


Fig. 12.25

Ahora puedes arrastrar el material EmitLuz encima de la caja y veras como esta se ilumina e ilumina la escena.

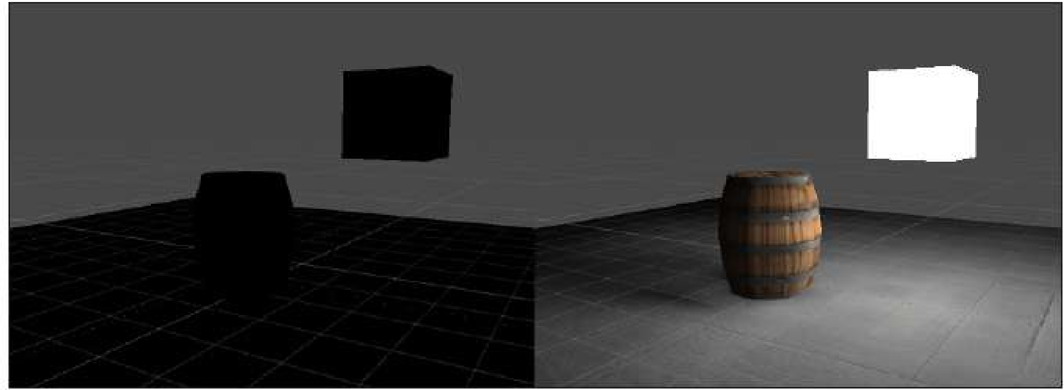


Fig. 12.26

5. Propiedades de las luces

Si seleccionamos una de las luces veras que en la ventana Inspector en el componente Light disponemos de una serie de propiedades para configurar la luz. Estas propiedades las divido en tres grupos principales: Tipo de luz, modo de iluminación, sombras y otros efectos.

En la escena selecciona una de las luces que tenemos desactivadas, por ejemplo la Point Light y miramos las propiedades que tiene dentro de la ventana Inspector en el componente Light.

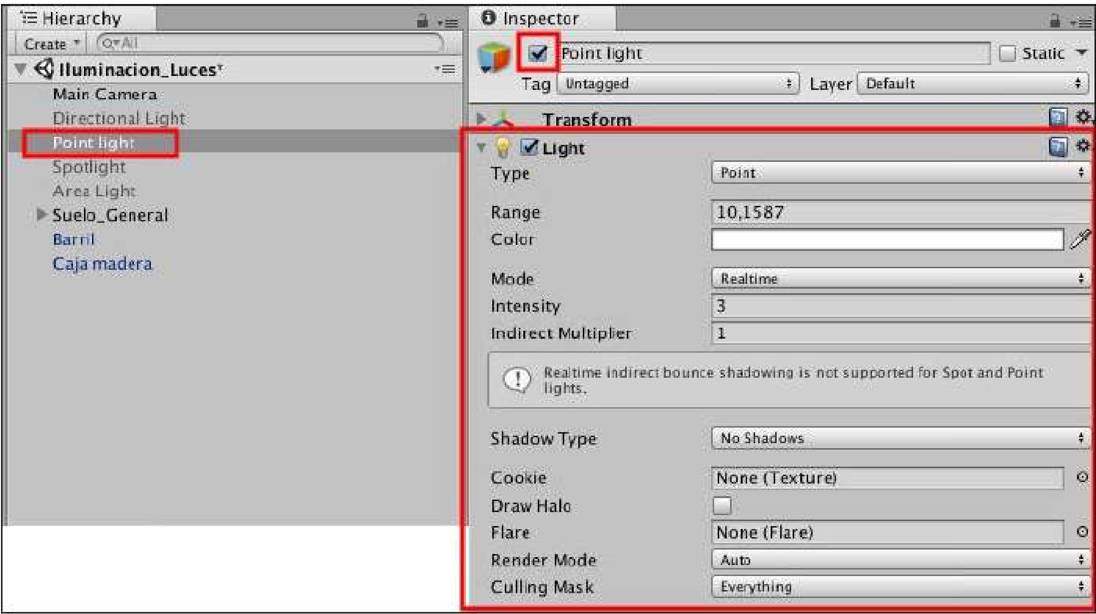


Fig. 12.27

Dentro del componente Light tenemos las siguientes opciones:

- **Type:** En este menu podemos escoger que tipo de luz vamos a utilizar, esto quiere decir que a pesar de que hayamos creado una luz de punto, esta misma se puede convertir en una luz direccional o cualquier otra.

- **Range:** Determina la distancia a la que la luz es emitida desde el centro del objeto. Este efecto solo se aplica a luces point y spot.
- **Spot Angle:** Esta opción nos aparecerá si seleccionamos el tipo de luz Spot Light y determina el angulo (en grados) en base del cono de la luz .
- **Color:** Utilizaremos esta opción para tintar de color la luz emitida.
- **Mode:** Esta opción es importante para especificar que método de iluminación vamos a utilizar para que la luz en concreto actúe en consecuencia. Los modos posibles son Realtime, Mixed y Baked.
- **Intensity:** Determina la cantidad de brillo de la luz que podemos modificar mediante valores numéricos.
- **Indirec Multiplier:** Este valor varia la intensidad de la luz indirecta. Esta luz indirecta es luz que rebota de un objeto a otro cuando el primer rayo impacta en el objeto. Esta opción nos permite determina el brillo de luz reflejada calculada por el sistema de iluminación global (GI). Cuando el valor es menor de 1 la luz rebotada se atenúa con cada rebote por el contrario un valor superior a 1 hace que la luz sea más brillante con cada rebote.
- **Shadow Type:** Esta opción permite que las luces arrojen sombras de distintas formas, es decir podemos obtener sombras duras (Hard Shadows), sombras suaves (Soft Shadows) o ninguna sombra.
- **Baked Shadow Angle:** Si el tipo de luz se establece en Direccional y las sombras son de tipo Soft, esta propiedad proporciona un suavizado artificial a los bordes de las sombras y les da un aspecto algo más natural.
- **Baked Shadow Radius:** Si el tipo de luz es Point o Spot y la sombra es de tipo Soft, esta propiedad también agrega un suavizado como la propiedad anterior al borde de las sombras.
- **Realtime Shadows:** Esta propiedad aparecerá siempre que la propiedad Shadow tenga configurada un tipo de sombra Hard Shadows o Soft Shadows. Esta propiedad sirve para controlar la configuración del render de las sombras en tiempo real.
    - **Strength:** Mediante el deslizador podemos controlar la oscuridad de las sombras.
    - **Resolution:** Este parámetro controla la resolución renderizada de los mapas de sombras.
    - **Bias:** Mediante el deslizador podemos controlar la distancia a la que las sombras se alejan de la luz. Este parámetro esta definido entre los valores 0 y 2.
    - **Normal Bias:** Mediante el deslizador podemos controlar la distancia a la que se reducen las superficies de fundido a lo largo de la superficie normal.
    - **Near Plane:** Mediante el deslizador podemos controlar el valor de cercanía del plano cuando se muestra las sombras.
    - **Cookie:** Especifique una máscara de textura a través de la cual se proyectan las sombras (por ejemplo, para crear siluetas o iluminación modelada para la luz).
    - **Draw Halo:** Cuando activamos esta casilla se dibujar un Halo esférico de luz con un diámetro igual al valor de Rango.
    - **Flare:** Si desea establecer un Flare para que se represente en la posición de la Luz debes arrastrar el asset dentro de este parámetro.
    - **Culling Mask:** Este parámetro se utiliza para excluir selectivamente grupos de objetos de ser afectados por la Luz.

6. Iluminación directa e indirecta

La iluminación directa diremos que es aquella que utilizaremos para iluminar a partir de las luces que proyectan directamente desde una fuente original como por ejemplo una bombilla o el sol. En este caso solo interactúa el punto de luz y el objeto iluminado.

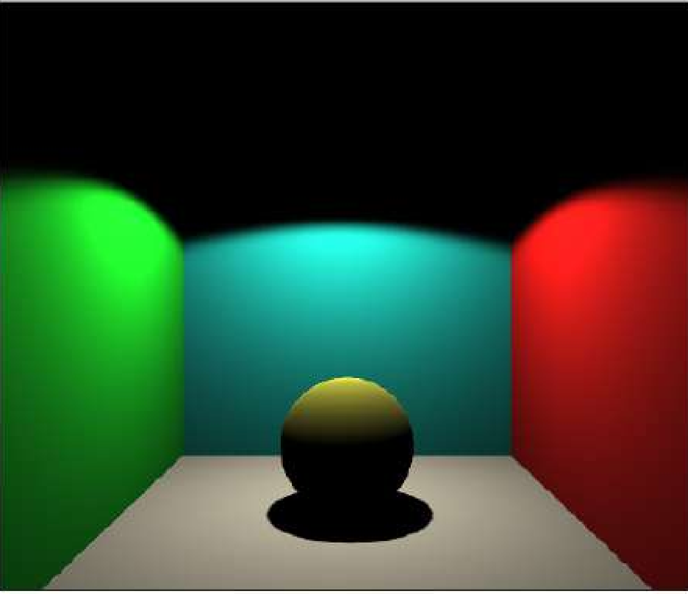


Fig. 12.28

Por el contrario una iluminación indirecta es la luz que se refleja o rebota de otros objetos proporcionando luz a su entorno.

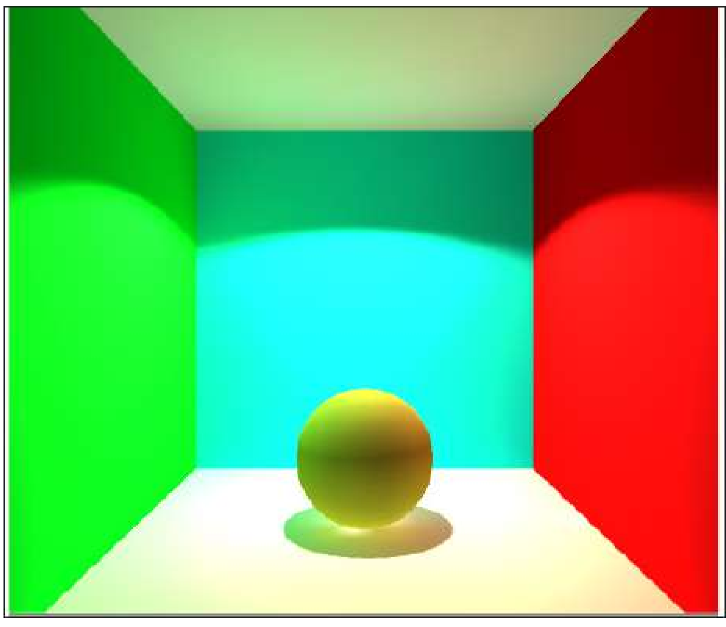


Fig. 12.29

Para trabajar estos aspectos y entender mejor como iluminar en Unity selecciona la escena I**luminacion_Directa_Indirecta** que encontrarás en la ventana **Project >Escenas** si has importado el paquete de ejercicios de este capitulo.

En esta escena encontrarás una escena con un habitación con varias paredes de colores que en realidad son **Quads**, una esfera en el centro y una Luz de tipo **Spot** para iluminar. Todos los objetos excepto la Luz y la cámara, se les ha activado la opción **Static** de la ventana Inspector.

Este escenario por defecto verás como está iluminado con luz directa, para crear una iluminación indirecta accedemos a la ventana **Lighting** que hemos explicado al principio del capitulo, y sin tocar ningún parámetro de la luz ni de la escena activaremos la opción **Realtime Global Ilumination.**

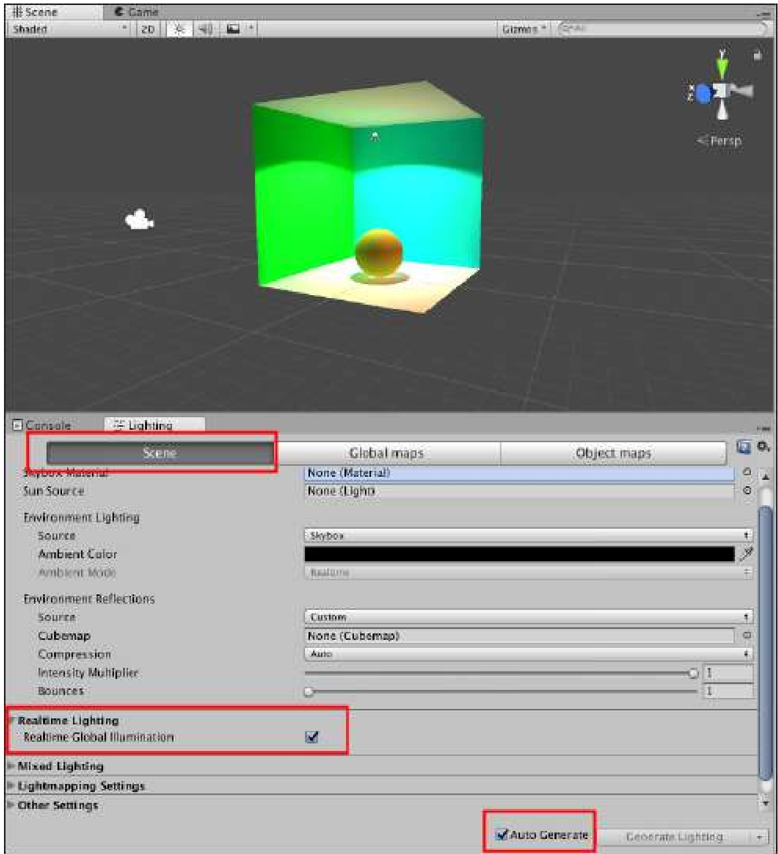


Fig. 12.30

Este tipo de iluminación consume bastantes recursos y no es la forma más efectiva de iluminar un videojuego. Si miras las otras opciones de la ventana Lighting verás que están desactivadas. Te recomiendo que antes de empezar a iluminar tengas claro como vas a iluminar la escena.

Con la opción Real Global Ilumination activada seleccionamos la Luz que tenemos en la escena y en la ventana Inspector vamos a fijarnos en el parámetro Mode.

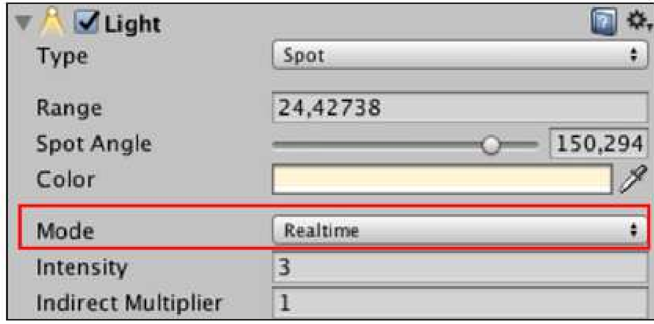


Fig. 12.31

El modo de la luz es el que está vinculado a la configuración de la ventana Light, porque a continuación vamos a ver algunas de las formas que podemos utilizar para iluminar nuestra escena. En este ejemplo tenemos Realtime que se calcula en tiempo real, es decir en cada frame tiene que calcular las luces e iluminar una escena de videojuegos de este modo no seria una buena elección. A continuación vamos a ver como trabajar con el modo Baked.

7. Iluminación Baked

En el mismo escenario que hemos abierto en el apartado anterior seleccionamos la luz de la escena y en la ventana Inspector cambiamos el Modo de la luz a Baked. Automáticamente la escena dejara de iluminarse y pasara a estar a oscuras.

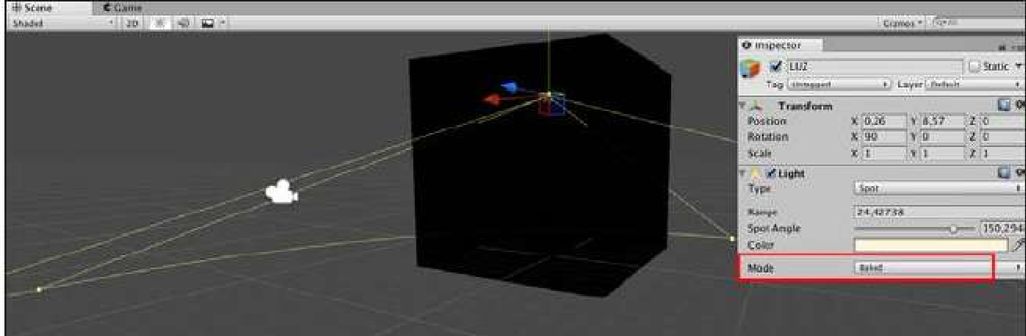


Fig. 12.32

Ahora volvemos a la ventana Lighting y desactivamos la opción Realtime Global Illumination y veremos como la luz vuelve a iluminar la escena pero en este caso de una forma directa como te muestro en la siguiente imagen.

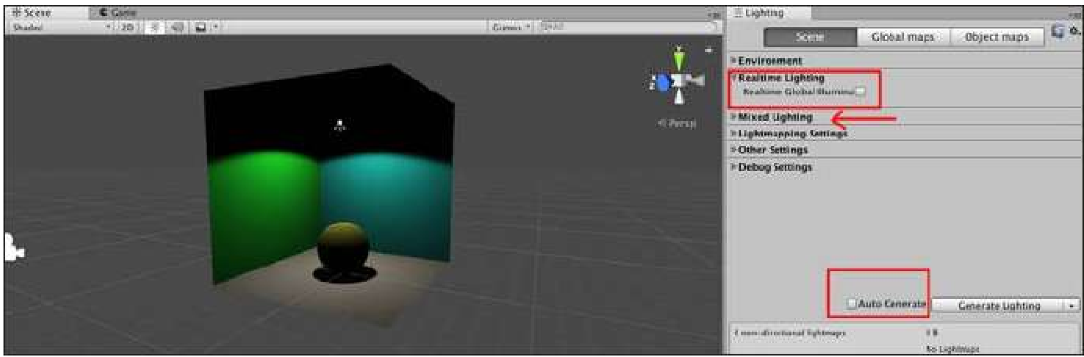


Fig. 12.33

Ahora vamos a crear un Baking de la escena. Para ello desactivamos el Auto Generate y abrimos la sección Mixed Ligting. Dentro de Mixed Ligting activamos la opción Baked Global Ilumination y en este caso yo he seleccionado el modo de iluminación Lighting Mode Baked Indirect. Luego hacemos clic encima de la opción Generate Lighting, se ejecutará un calculo que puede tardar unos segundos y al terminar nuestra escena quedará iluminada.

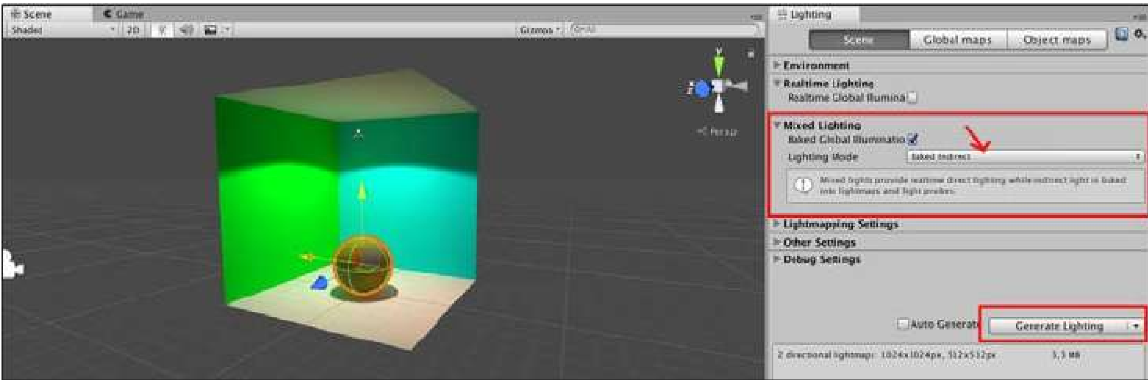


Fig. 12.34

Unity Bakea iluminación directa e indirecta desde luces hasta un mapas de luz (para iluminar GameObjects estáticos). El Baked de luces no puede emitir luz especular, incluso en GameObjects dinámicos.

Este tipo de iluminación no cambian en respuesta a las acciones tomadas por el jugador, o eventos que tienen lugar en la escena. Son principalmente útiles para aumentar el brillo en áreas oscuras sin necesidad de ajustar toda la iluminación dentro de una escena.

En resumen si ahora seleccionas la esfera y la desplazas a un lado, podrás comprobar que la sombra no se mueve y la esfera no reacciona a la luz que está emitiendo.

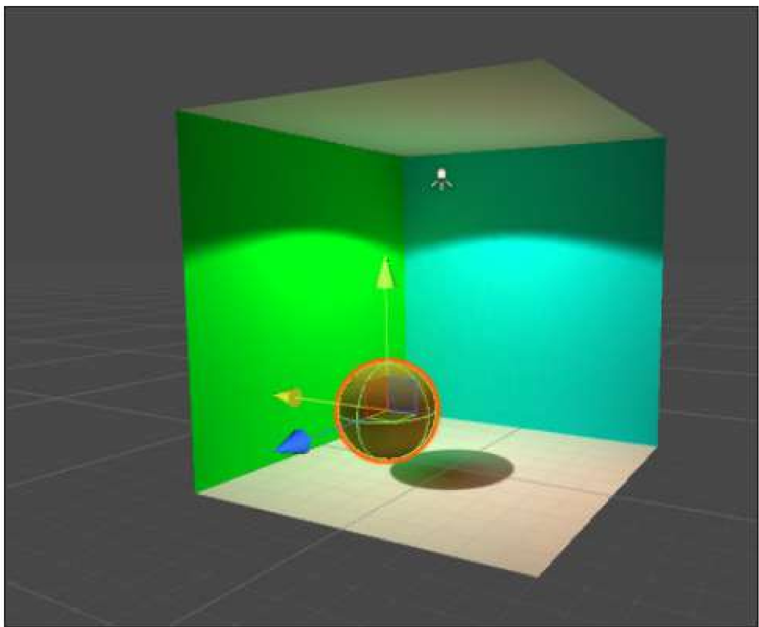


Fig. 12.35

Esto se debe a que Unity a creado un mapa de iluminación de los objetos estáticos de la escena, para poder ver estos mapas, desde la ventana Lighting en la opción Global maps.

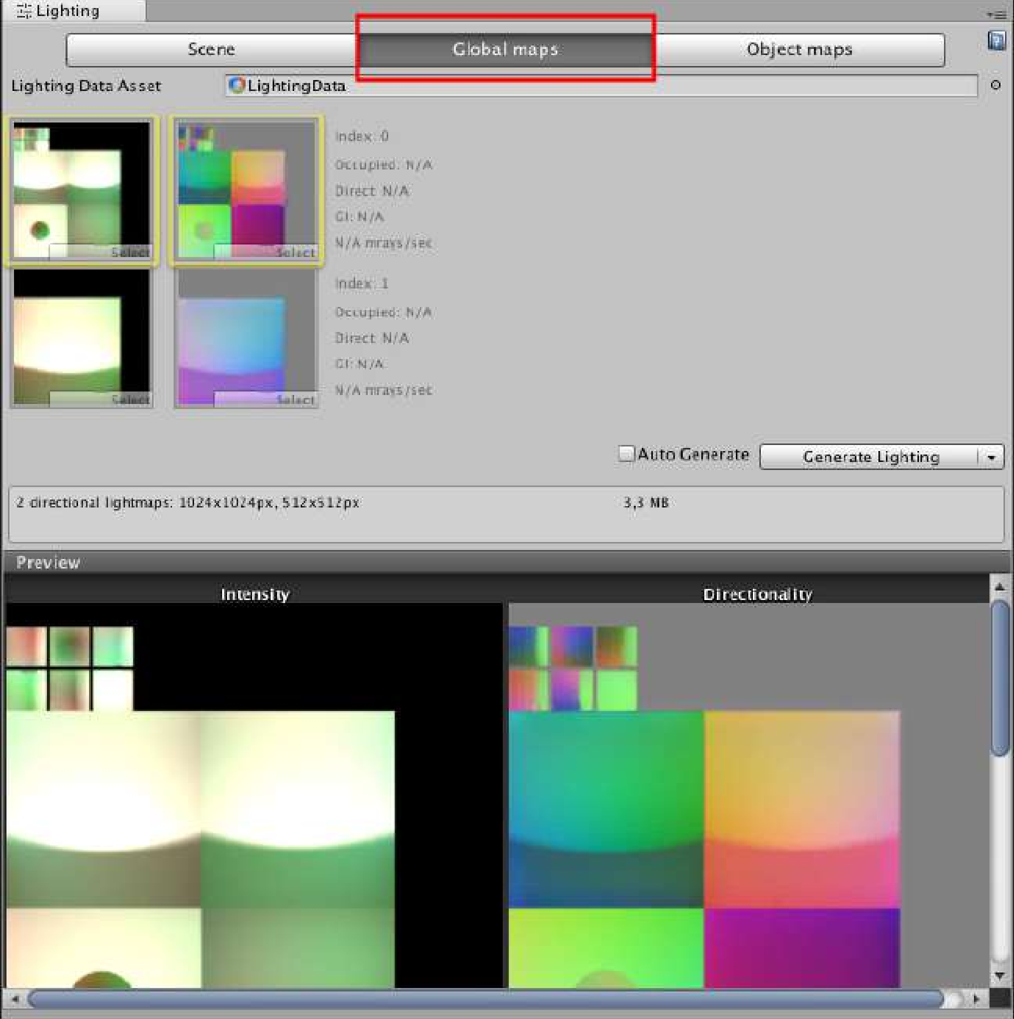


Fig. 12.36

En estos casos es útil si no tenemos objetos que interactúen con la luz es decir los objetos de la escena sean estáticos. En el caso de que tengamos un personaje o personajes que se muevan y tengan que interactuar con el escenario podemos utilizar otro modo de la luz que se llama Mixed, que utiliza una mezcla de ambas.

En realidad el ejemplo que hemos realizado anteriormente hemos utilizado un modo de iluminación Mixto, la diferencia reside en el modo que hemos utilizado en la luz.

8. Iluminación Mixed

Normalmente los objetos de un videojuego son estáticos exceptuando los objetos que interactúan de alguna forma. Para este ejemplo abrimos la escena Iluminación_Mixed que contiene la misma escena que tenemos en el apartado anterior pero con un cubo que no tiene activada la opción estática. La luz es de tipo Spot y tiene el modo Mixed activado en sus propiedades.

La escena que te debería de aparecer es como la que te muestro en la siguiente imagen.

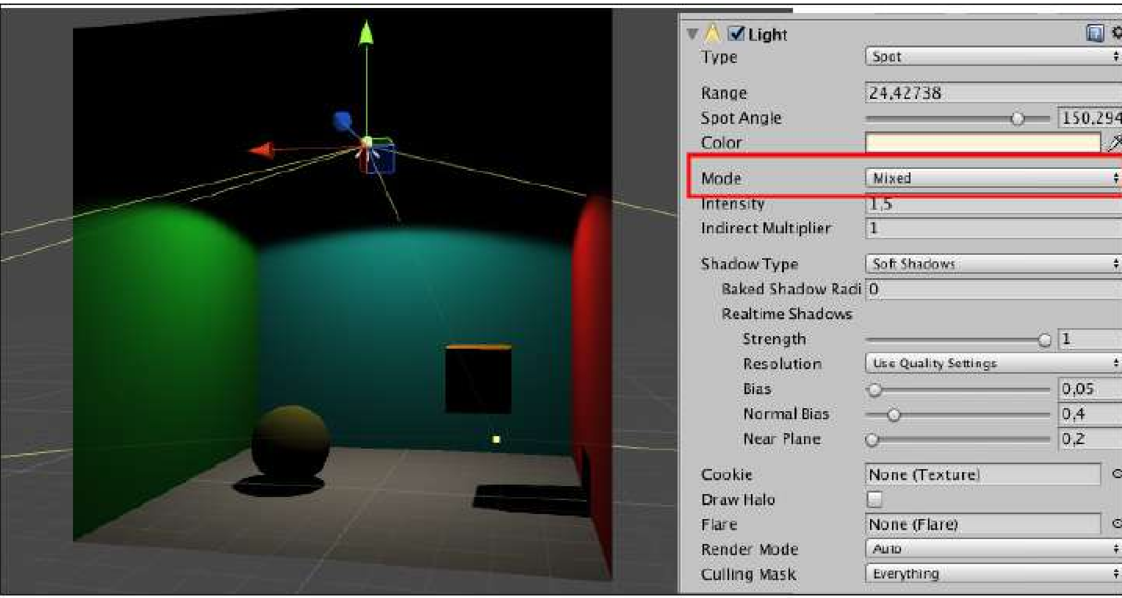


Fig. 12.37

Ahora vamos a dirigirnos a la ventana Lighting y vamos a crear un Baking de la escena para ver como funciona este tipo de iluminación. Para ello en la ventana Lighting asegurate de que las opciones de Real Time Light y Auto Generate están desactivados, y la opción Mixed Lighting debe estar la opción Baked Global Illumination activada. Una vez está todo correctamente hacemos clic encima de Generate Lighting.

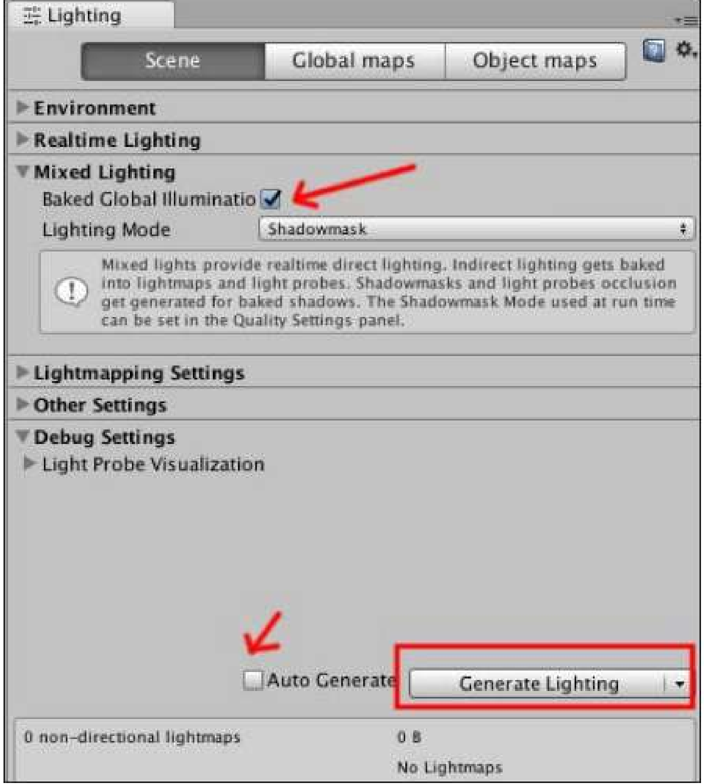


Fig. 12.38

Cuando generes el backing puede que tarde unos segundos dependiendo del tipo de estación de trabajo que utilices. Una vez finalicen los cálculos la escena quedará iluminada de forma global, con la diferencia de que esta vez si movemos algún objeto de la escena este interactúa con la iluminación.

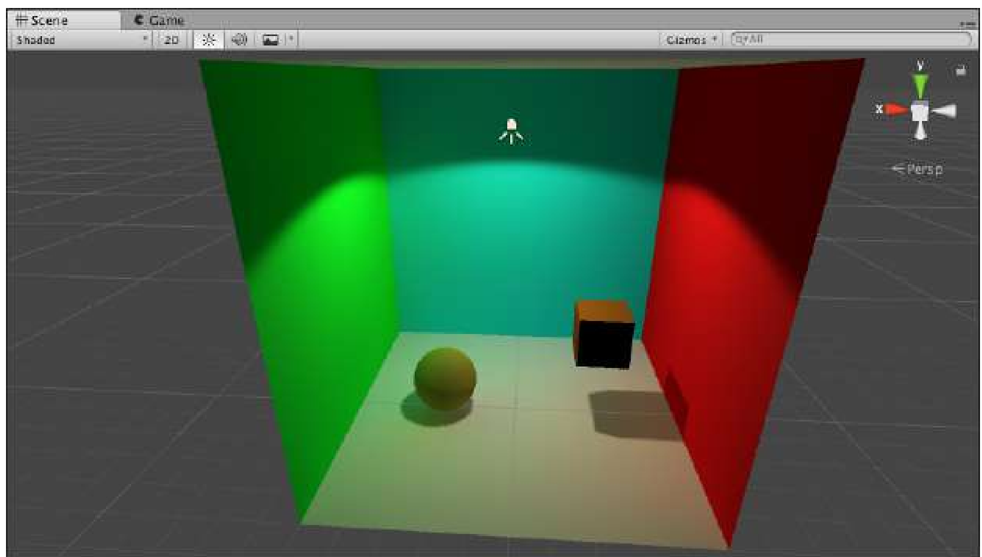


Fig. 12.39

Este ejemplo es muy generalizado y por eso mismo a continuación vamos a ver que modos podemos bakear con este tipo de iluminación Mixed.

Primero vamos a limpiar el bake anterior para dejarlo limpio y poder ver todos los modos uno por uno. Para limpiar el bake anterior en la ventana Lighting accedemos al menú que encontraremos al lado del botón Generate Lighting.

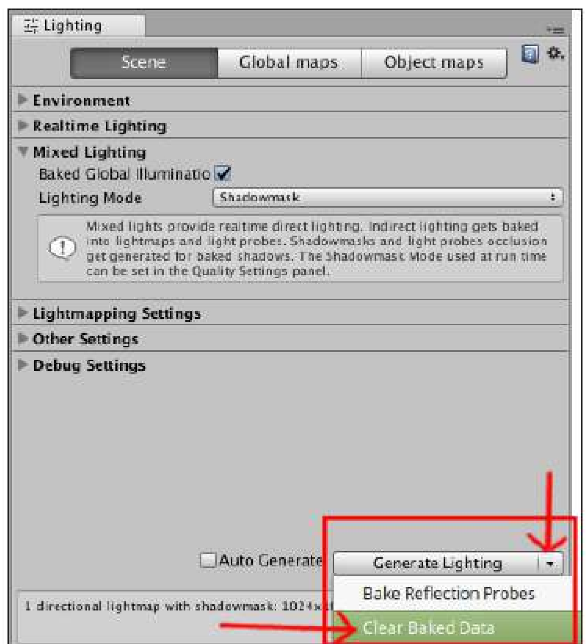


Fig. 12.40

La escena dejara de tener la iluminación global que tenia antes y pasará a estar iluminada directamente. A continuación vamos a ver como funcionan los siguientes modos:

Subtractive

Es el único modo de iluminación mixto que bakea la iluminación directa en un mapa de luz y descarta la información que Unity usa para crear sombras dinámicas y estáticas como en otros modos de iluminación mixta. Debido a que la Luz ya está bakeada en el mapa de luz, Unity no puede realizar ningún cálculo de iluminación directa en tiempo de ejecución.

Si realizamos la prueba y creamos un bakeado de la escena veremos que no tenemos

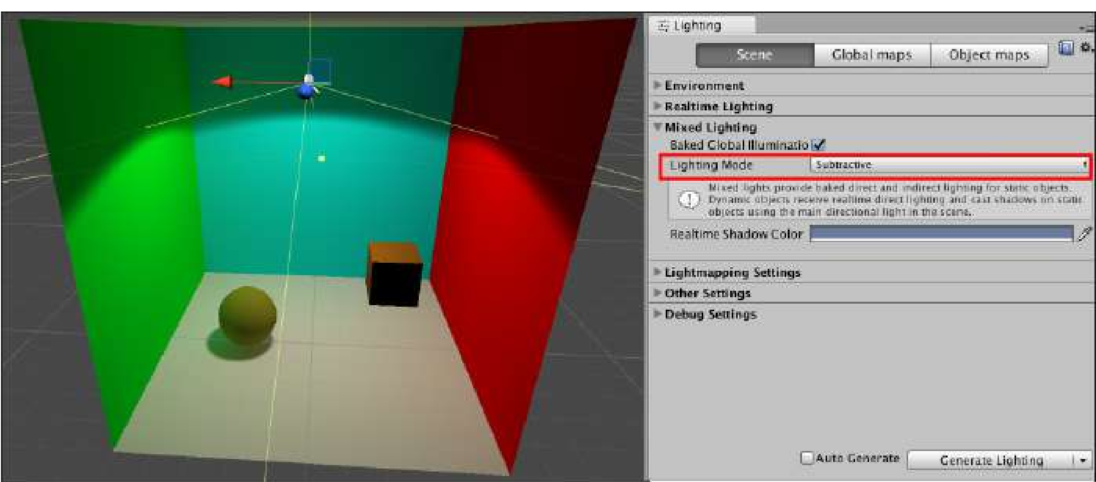


Fig. 12.41

Las principales características de este tipo de iluminación son que los GameObjects estáticos no muestran ningún brillo especular o brillante en Mixed Lights. Tampoco pueden recibir sombras de GameObjects dinámicos, a excepción de la Luz direccional principal. Otro aspecto es que los GameObjects dinámicos reciben iluminación en tiempo real y admiten reflejos brillantes. Sin embargo, solo pueden recibir sombras los GameObjects estáticos.

Baked Indirect

Este tipo de configuración Unity solo calcula previamente la iluminación indirecta, y no lleva a cabo cálculos previos ocultos. Las sombras son totalmente en tiempo real dentro de Shadow Distance. En otras palabras, las luces indirectas del baking se comportan como luces en tiempo real con iluminación indirecta adicional, pero sin sombras más allá de la distancia de la sombra.

Para acceder a la distancia de las sombras debemos acceder a Edit>Project Settings > Quality. En la ventana Inspector nos aparece las QualitySetting en el apartado Shadows tenemos el parámetro Shadow Distance que por defecto tiene el valor 150.

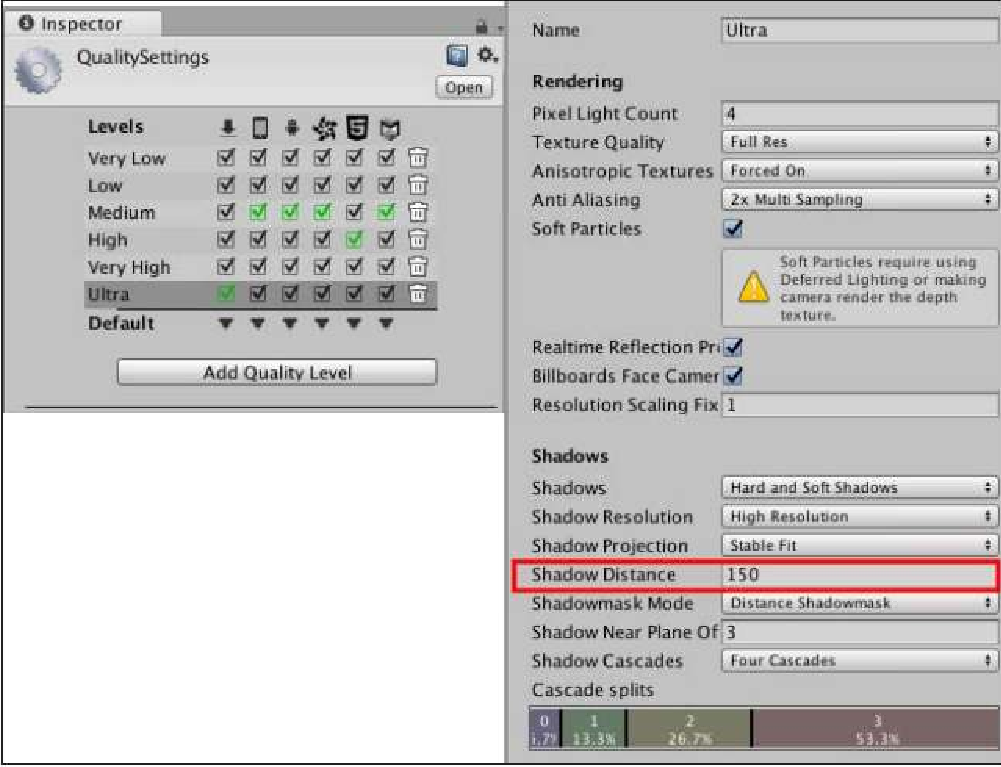


Fig. 12.42

Un buen ejemplo de cuándo podría ser útil el modo Baked Indirect es si está construyendo un juego en interiores en habitaciones conectadas con pasillos en donde las distancias de visualización son limitadas, por lo que todo lo que sea visible debería ajustarse dentro del parámetro Shadow Distance. Este modo también es útil para crear una escena al aire libre con niebla, ya que se puedes usar la niebla para ocultar las sombras que faltan en la distancia.

Vamos a realizar la prueba con nuestra escena pero antes vamos a limpiar el bake anterior como ya hemos explicado accediendo al menú que encontraremos al lado del botón Generate Lighting y seleccionando la opción Clear Baked Date. Luego seleccionamos el modo Baked Indirect y hacemos clic en Generate Lighting.

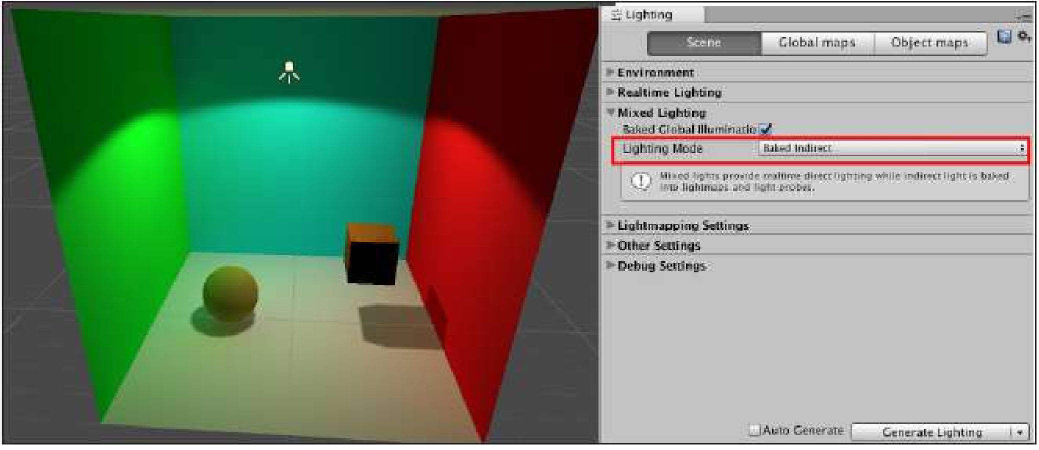


Fig. 12.43

Baked Shadowmask

Unity pre-calcula las sombras emitidas desde un GameObject estático a otro, y los almacena en una textura Shadowmask separada para hasta 4 luces superpuestas. Si se superponen más de 4 luces, cualquier luz adicional volverá a estar en Bake Lighting. El sistema de Baking determina cuál de las luces corresponde a Bake Lighting y se mantiene constante en todos los bakeos, a menos que se modifique una de las luces superpuestas.

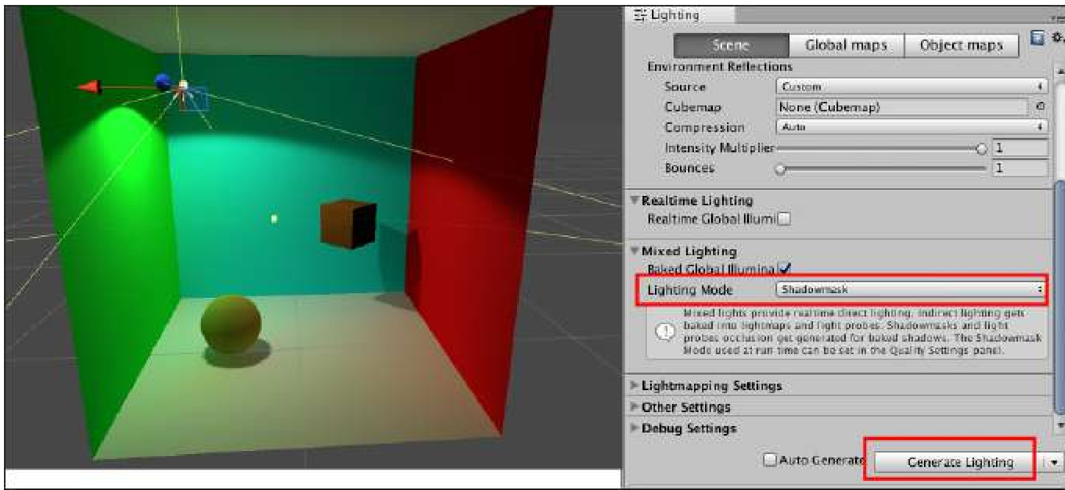


Fig. 12.44

La superposición de luz se calcula independientemente de los objetos receptores de sombra. Entonces, un objeto puede obtener la influencia de 10 luces diferentes mezcladas todas desde el mismo canal Shadowmask / Probe, siempre y cuando esos volúmenes de luz no se superpongan en ningún punto en el espacio. Si algunas luces se superponen, se utilizan más canales. Y, si una luz se superpone mientras los 4 canales ya han sido asignados, esa luz volverá a estar completamente bakeada.

9. Práctica general

Fig. 12.45

Para realizar esta práctica puedes crear tu mismo una escena con los prefabs que contiene el paquete de este capitulo. Esta pequeña escena se encuentra en la escena Iluminacion_textura y es una escena ya iluminada, la puedes utilizar para ver como la he iluminado.

A continuación voy a ir explicando los pasos que he realizado, te recomiendo que no te saltes esta parte, porque el siguiente apartado deberás iluminar un pequeño escenario tu mismo.

¿Que objetos son estáticos?

Primero de todo debes saber que objetos no van a tener ningún tipo de interacción, por ese mismo motivo debemos seleccionarlos y activar la opción Static en la ventana Inspector. En el caso del ejemplo son estáticos todos los prefabs suelo, paredes, pared con ventana y Barril. El objeto que vamos a tener con interactividad es la caja de munición que en este ejemplo he utilizado un script para que de vueltas. Otro aspecto importante que ya se ha explicado es apagar todas las luces para ver mejor el efecto de los puntos de luz.

Fuentes de luz

Antes de poner cualquier luz debes saber de donde viene la luz. En el ejemplo esta claro que va a ser de la ventana. En este caso en concreto no quiero que sea una ventana transparente, mas bien una ventana que deje pasar la luz y aprovechar para que emita luz.

Hay varios aspectos que debemos prestarle atención; el primero es una luz que entre por la ventana y una textura que emita esa luz brillante.

Para la luz de la ventana he utilizado una luz de tipo spot light. Debemos posicionar y rotar con cuidado la luz de forma que simule que entra por la ventana, es decir si el cono de la luz es muy grande es posible que se vean sombras extrañas provocadas por los objetos pared.

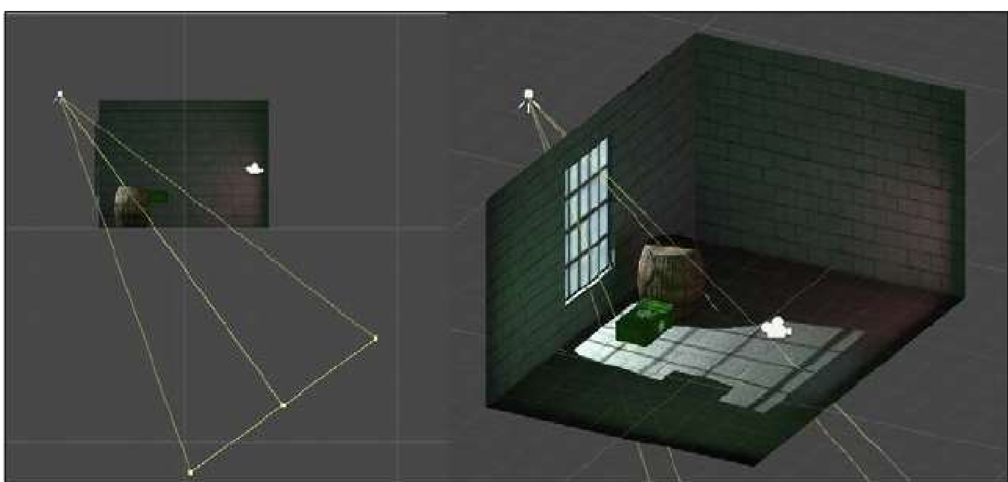


Fig. 12.46

Sobretodo no te preocupes si tu spot light no proyecta ninguna sombra de la ventana, eso es a causa de una textura y lo veremos más adelante, por ahora coloca bien el foco y vamos a ver la configuración del foco.

El Rango , el angulo y el color lo dejo a tu juicio puesto que las escenas pueden variar, pero si voy a prestarle especial atención Modo que vamos a utilizar (en este caso Mixed), la intensidad he puesto 6 unidades y indirect multiplier con un valor de 3 unidades. El tipo de sombra en el ejemplo he utilizado la sombra más dura (Hard Shadows) y por ultimo el causante de la sombra cuadriculada es el parámetro cookie al que se le ha añadido una textura que explico a continuación.

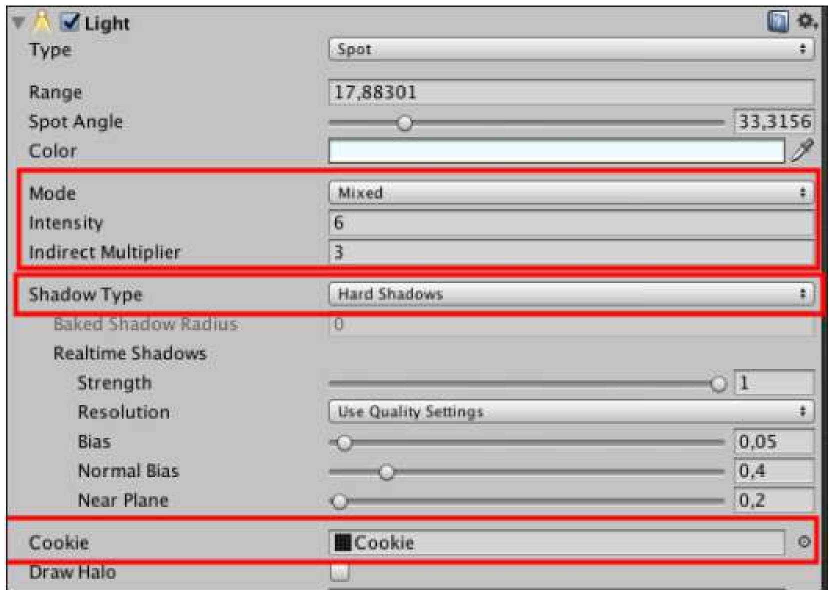


Fig. 12.47

Textura Cookie

Esta textura la encontrarás en la ventana Project >Modelos>Texturas> Cookie. Es una imagen cuadrada de 1024 x 1024 en blanco y negro.

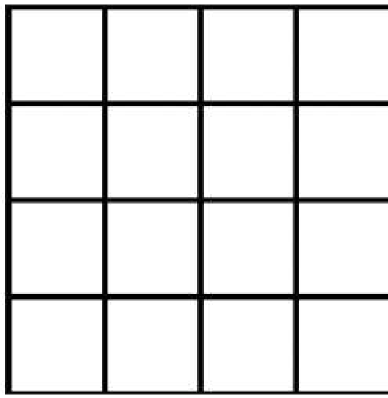


Fig. 12.28

Esta imagen tiene que configurarse siempre como Cookie en sus propiedades para poder utilizarla en una luz. Para ver sus propiedades selecciona la imagen y aparecerán en la ventana inspector.

Los parámetros más relevantes de este tipo de texturas son:

- **Texture Type:** en donde tienes que definir que es una cookie.
- **Light Type:** escoge el tipo de luz para el cual esta textura va ha ser utilizada.
- **Alpha Source:** Escoge la opción From Gray Scale, para que tome la imagen como escala de grises.

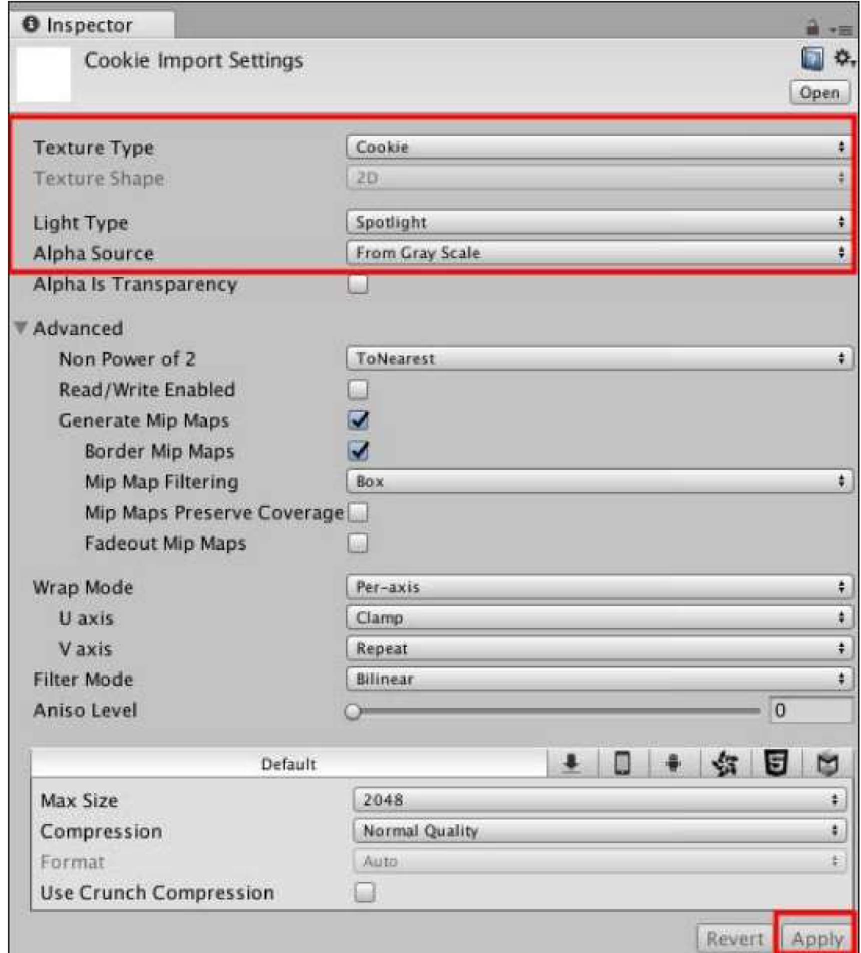


Fig. 12.49

Una vez se tenga la textura configurada, entonces se puede arrastrar directamente al parámetro de Cookie del SpotLight. Si todo es correcto debería proyectar las sombra de la textura. Si es así seguramente deberás mover la luz para que encaje mejor en la escena.

Iluminar zonas oscuras

En esta escena he querido utilizar las Area Light para iluminar zonas que han quedado muy oscuras y como modo de ejemplo les he dado un color distinto a cada una. En reali-

dad este tipo de luces solo funcionan cuando hacemos un bakeado de la iluminación. En la siguiente imagen te muestro como las he dispuesto. En este ejemplo a las tres luces les he dado una intensidad de 1,5 unidades.

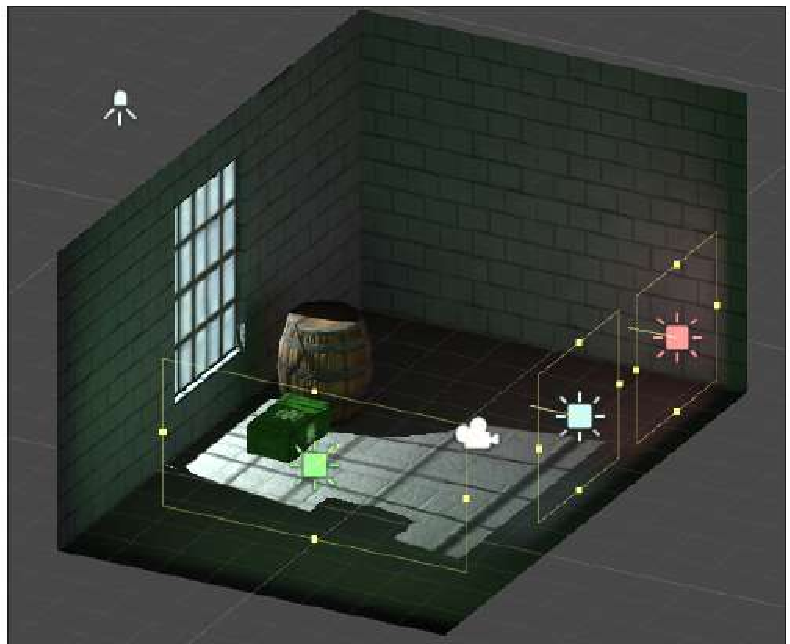


Fig. 12.50

Luz puntual

En una escena también podemos utilizar las luces para destacar una zona en concreto y de este modo dar pistas al jugador de que en esa zona puede haber algún objeto importante. En este caso simplemente he añadido una luz de punto a la caja de munición para que reflejara un color distinto.

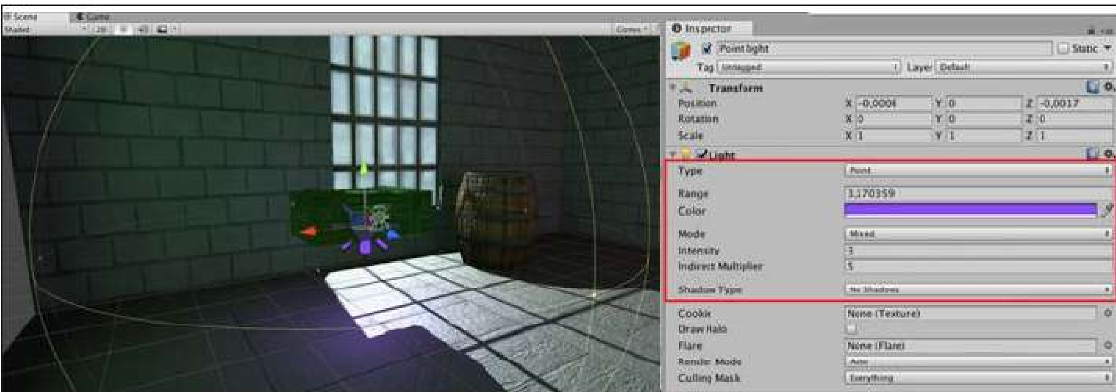


Fig. 12.51

Las imágenes que te muestro a continuación son la escena con todo pero en una no se le ha realizado el bake de iluminación y en el otro si.

Fig. 12.52

Si quieres ver las configuración que he utilizado solo tienes que abrir la escena como he dicho al principio de este apartado y hacer pruebas tu mismo para ver como podrías mejorarla. Si ejecutas esta escena veras como la caja de munición rota y hay interacción con las sombras.

10. Ponte a prueba

Ahora que hemos visto de forma general como podemos iluminar una escena, dispones en el material de este capitulo una escena para que la ilumines como quieras. La escena se llama **Escenario** y también dispones de otra escena con el nombre **Escenario_Iluminado** con la misma escena pero iluminada.

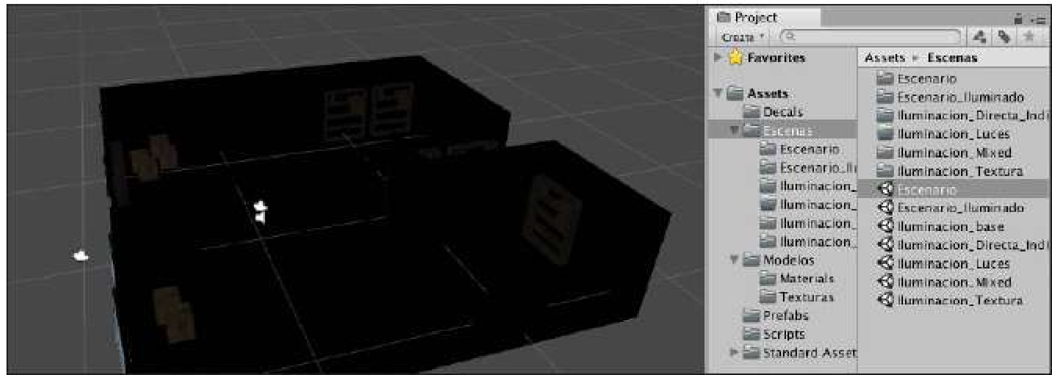


Fig. 12. 53

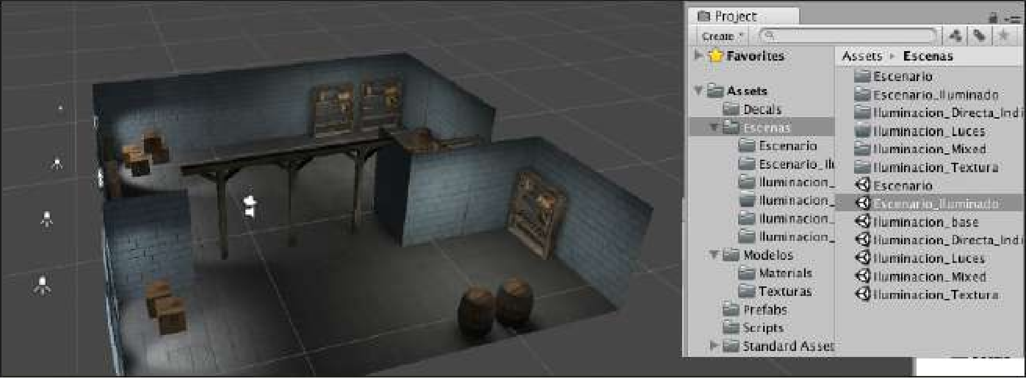


Fig. 12. 54

Estas **Escenas** las encontraras en la carpeta Escenas dentro de la ventana **Project**. Intenta utilizar todo lo aprendido en el capitulo. Esta escena dispone de un **FpsController** que te va a permitir explorar la iluminación de tu escena cuando la tengas iluminada.

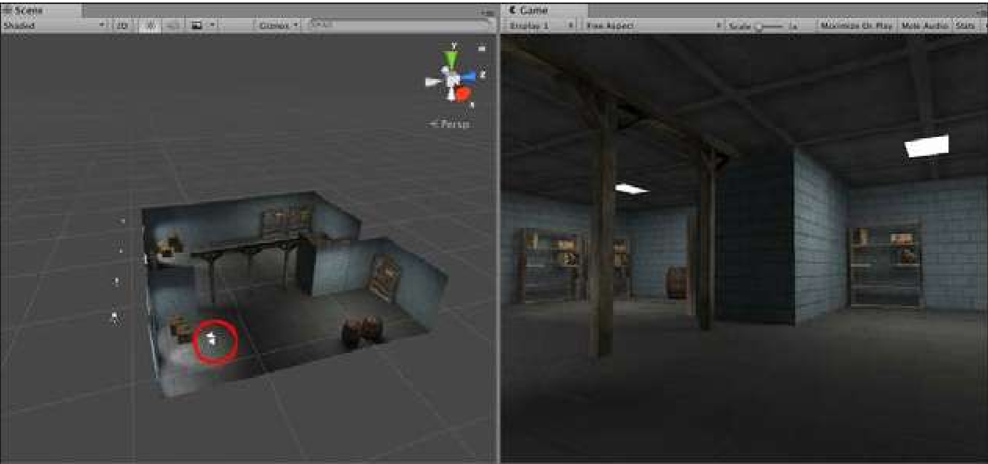
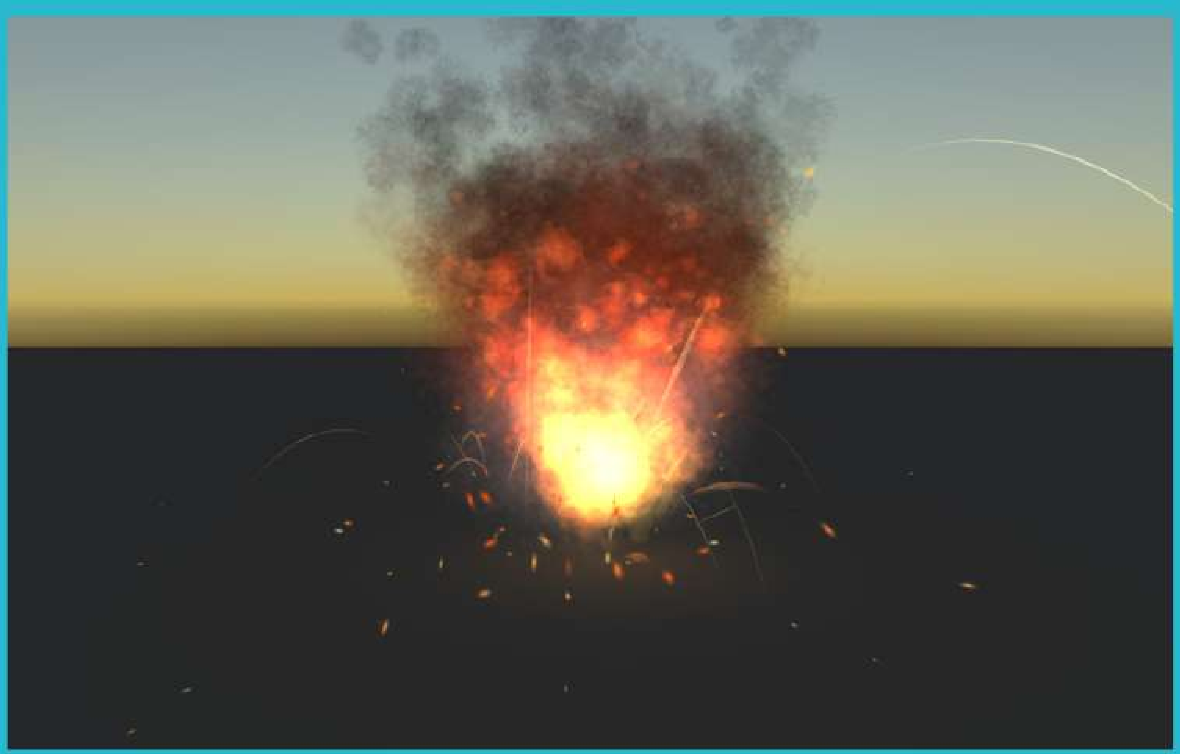


Fig. 12.55

Capítulo 13
Las partículas



- Introducción
- Sistema de partículas
- Creación de un sistema de partículas
- Editar las propiedades de las partículas

1. Introducción

El objetivo de este capítulo es acercarte un poco más a que son las partículas darte una visión general de todas sus propiedades así como un ejemplo completo para demostrarte el potencial de estas. Es un tema muy complejo y que requiere de muchas horas de pruebas.

Como siempre para seguir correctamente este capitulo debes importar un paquete de assets que te permitirá empezar con la materia directamente. El paquete lo encontraras como siempre en la carpeta de proyectos del capítulo correspondiente con el nombre **Assets_Escena_Capitulo_13.unitypackage.** Recuerda que para importar este paquete debes acceder al menú principal **Assets > Import Package** y en la nueva ventana tener todos los archivos seleccionados y aceptar la importación. A continuación dispondrás de una serie de carpetas con el material necesario. En este capitulo concreto este paquete se va a utilizar para realizar el proyecto final, pero dispones de una serie de texturas que podrás utilizar para hacer pruebas con las partículas.

2. Sistema de partículas

El sistema de partículas simula elementos fluidos como líquidos, nubes y llamas de fuego al generar y animar un gran número de pequeñas imágenes 2D en la escena. Cuando todas estas partículas se manifiestan juntas da la sensación de una entidad completa.

Antes de ver como crear , editar y ponerles materiales hay que entender que es la dinámica del Sistema y las Dinámicas de las partículas.

Cuando hablamos de **dinámicas del sistema** nos referimos a que tenemos un sistema que emite o genera partículas, cada partícula tiene un **lifetime** (tiempo de vida) predeterminado que dicho de otro manera es cuantos segundos dura la partícula antes de desaparecer. El sistema puede emitir partículas de varias formas: en posiciones aleatorias dentro de una región especifica que puede ser una esfera un cono, caja o cualquier superficie arbitraria que deseemos. Las partículas serán mostradas hasta que el tiempo se agote, luego son eliminadas. Otro parámetro a conocer es el **emission rate** que indica la cantidad de partículas que se emiten por segundo, aunque los tiempos exactos de la emisión son asignados al azar.

Por otro lado las **dinámicas de las partículas** son ajustes que afectan a las partículas individualmente. Cada una tiene un vector de velocidad llamado **velocity** que determina la dirección y la distancia con cada actualización de cuadro o **framerate**. Esta velocidad puede ser cambiada por la aplicación de fuerzas externas como la gravedad o un **wind zone** en un terreno. Otro aspecto que puede variar es el color el tamaño y la rotación de cada partícula. El color dispone de un componente **alpha** (transparencia) , para poder dar la sensación de que una partícula se desvanece gradualmente dentro y fuera de su vida es decir emisión y desaparición. Si usamos con conocimiento ambas dinámicas, podemos simular muchos tipos de efectos de forma aproximada a la realidad. Crear cascadas de agua o el fuego de las antorchas de una mina antigua etc. Las posibilidades son infinitas.

3. Creación de un sistema de partículas

Para crear un sistema de partículas podemos hacerlo de dos formas uno es agregar desde el menú **GameObject > Effects > Particle System** y tendremos un sistema prefabricado por Unity como se muestra en la siguiente imagen.

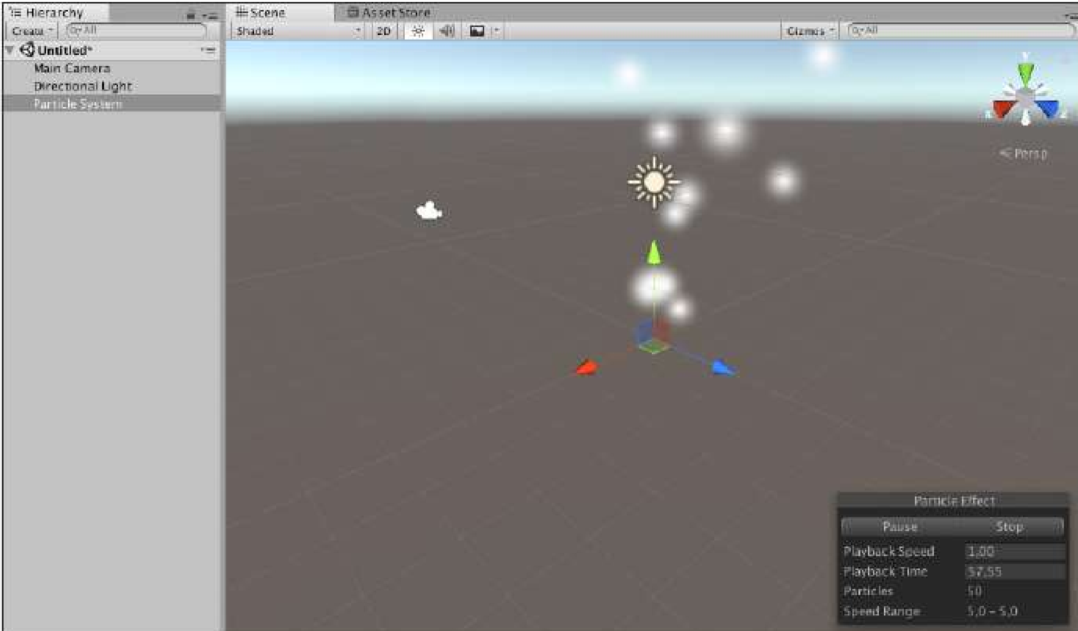


Fig. 13.1

Otra forma de crear sistemas de partículas es crear un objeto nuevo en la escena , por ejemplo un objeto vacío y agregarle un componente de tipo **Add Component > Effects > Particle System** y se añadirá un componente de sistema de partículas como se muestra en la siguiente imagen.

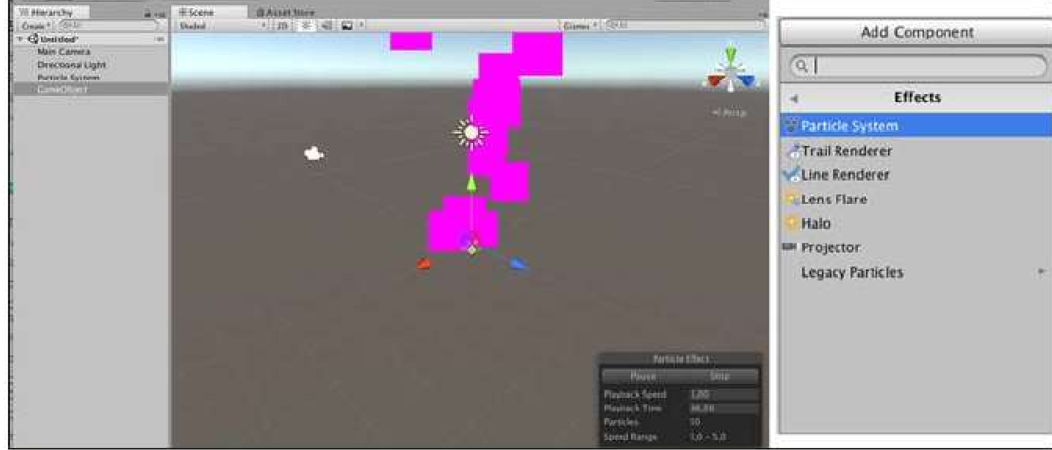


Fig. 13.2

Particle Effet en este panel encontramos un controlador sencillo que nos permite realizar cambios para visualizar las partículas. A continuación veremos los parámetros de este sistema de partículas.

4. Editar las propiedades de las partículas

En este apartado vamos a ver todas y cada una de los parámetros de Particle system e iremos viendo para que sirve cada opción. En el inspector los parámetros se organizan

en secciones o “módulos”. Cada módulo se puede expandir y contraer haciendo clic en la barra que muestra su nombre. En el lado izquierdo de la barra hay una casilla de verificación que se puede usar para habilitar o deshabilitar la funcionalidad de las propiedades en esa sección. Por ejemplo, si no queremos variar el tamaño de las partículas a lo largo de su vida útil, simplemente puedes desmarcar la sección.

Fig. 13.3

Aparte de las barras de módulo, el inspector contiene algunos otros controles. El botón Abrir Editor muestra las opciones en una ventana de editor separada que también te permite editar múltiples sistemas a la vez. La casilla de verificación **Resimular** determina si los cambios de propiedad se deben aplicar inmediatamente a las partículas ya generadas por el sistema. Otro aspecto es la casilla de **Bounds** que muestra los contornos de los objetos de malla utilizados para mostrar las partículas en la escena. A continuación vamos a ver los módulos de este componente.

Módulo principal

Es el primer modulo contiene las propiedades globales que afectan el sistema de partículas general. A continuación veremos en la imagen siguiente el modulo y como se ve representado en las partículas en la ventana **Scene**.

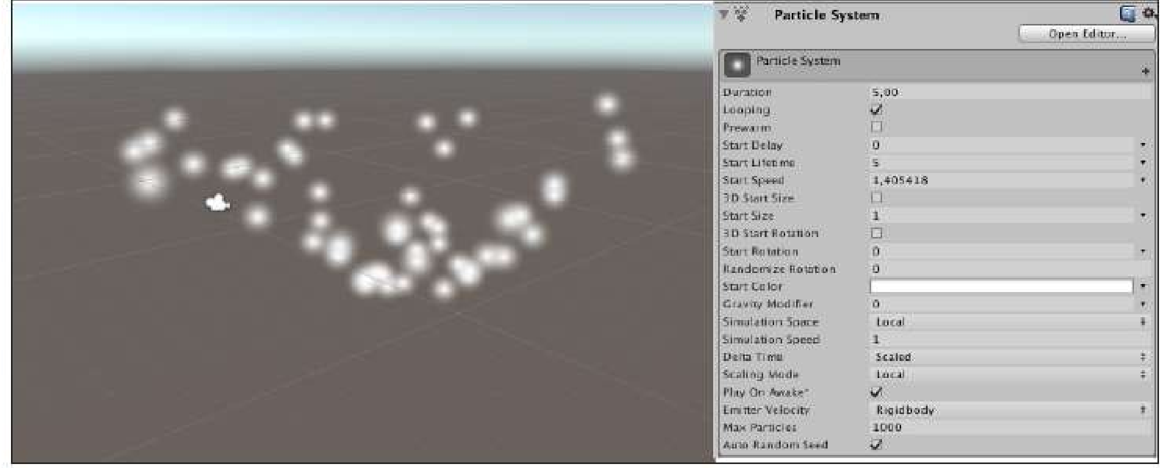


Fig. 13.4

Duration	La duración que el sistema va a ejecutarse.
Looped	Si está habilitado, el sistema va a iniciar nuevamente al final de su tiempo de duración y va a repetir continuamente el ciclo.
Prewarm	Si está habilitado, el sistema va a iniciarse tal como si hubiera completado un ciclo completo (solamente funciona si **Looping** también está habilitado).
Start Delay	Demora en segundos antes de que el sistema empiece a emitir una vez esté habilitado.
Start Lifetime	El tiempo de vida inicial para las partículas.
Start Speed	La velocidad inicial de cada partícula en la dirección apropiada.
Start Size	El tamaño inicial de cada partícula.
3D Start Rotation	La rotación viene dada por los ejes **X, Y, Z.**
Start Rotation	El ángulo re rotación inicial de cada partícula.
Start Color	El color inicial de cada partícula.
Gravity Modifier	Escala los valores de gravedad configurados en el **physics manager**. Un valor de cero va a apagar la gravedad.
Simulation Space	Determina si las partículas son animadas en el espacio local del objeto padre (y por lo tanto movido con el objeto) o en el espacio global (**world**)

Scaling Mode	Utiliiza la escala (**scale**) del **transform**. Configura la jerarquía Local o **Shape.** Local aplica solamente a la escala transform del sistema de partículas. El modo **Shape** aplica solamente la escala a la posición inicial de las partículas.
Play on Awake	El sistema inicia automáticamente cuando el objeto es creado
Max Particles	El máximo número de partículas en el sistema de una sola vez. Las partículas más viejas serán eliminadas cuando el limite sea alcanzado.

A continuación te propongo que alteres algunos de los parámetros de este modulo utilizando la tabla anterior para ir haciendo pruebas con el sistema de partículas.

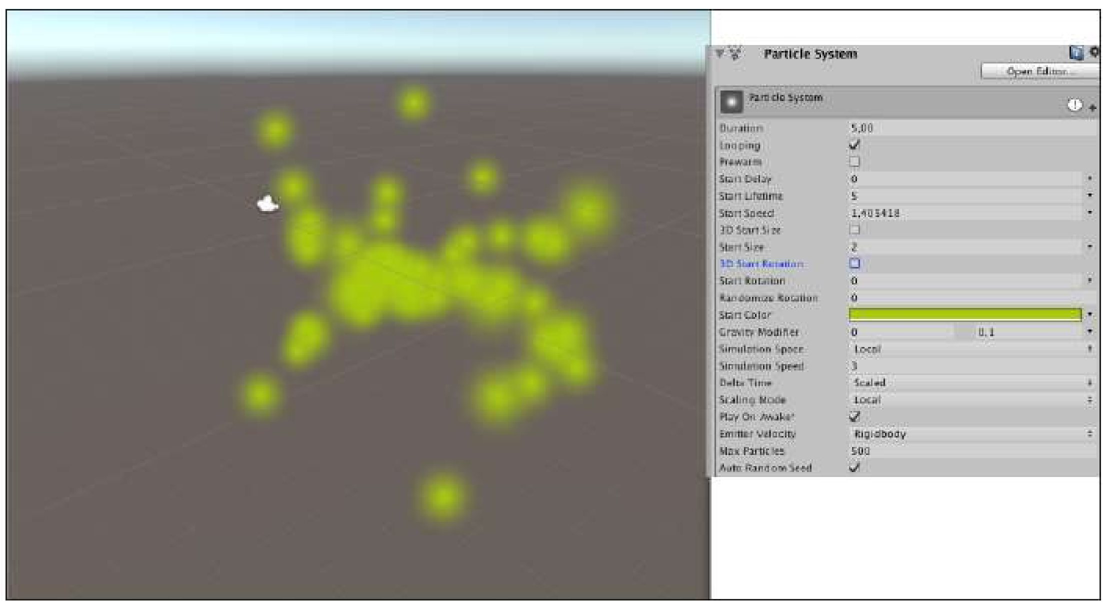


Fig. 13.5

Módulo Emission

Este modulo afecta a la cantidad y el tiempo de las emisiones de las partículas. La tasa (rate) de emisión puede ser constante o puede variar en el tiempo de vida del sistema de acuerdo a una curva. Si el modo *Distance* es seleccionado entonces un cierto número de partículas son soltadas por unidad de distancia, movida por el objeto padre. Con esta opción podemos simular partículas que en realidad están creadas por el movimiento del objeto. Algo importante a tener en cuenta es que el modo ***Distance*** solamente está disponible cuando ***Simulation Space e***stá configurado a ***World*** en la sección del **Particle System** (Sistema de Partículas).

Si la tasa (rate) es configurada en modo **Time** entonces el número que pongamos será el numero de partículas emitidas por segundo sin importar cómo el objeto padre se mueva. Adicionalmente, podemos agregar bursts (ráfagas) de partículas extras que aparecen en tiempos específicos. Solo tienes que pinchar encima de el símbolo (+) y agregar los valores correspondientes.

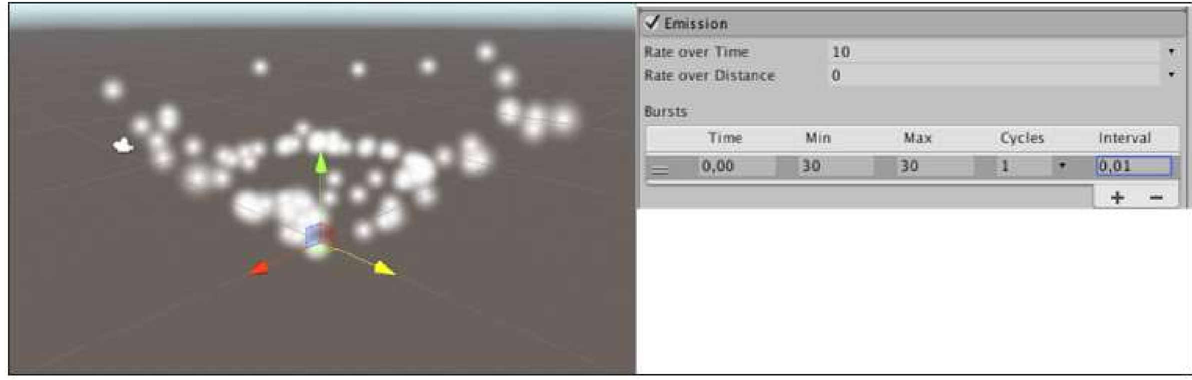


Fig. 13.6

Rate over Time	El número de partículas emitidas por unidad de tiempo
Rate over Distance	El número de partículas emitidas por distancia movida
Bursts	Permite que partículas extras sean emitidas en tiempos específicos.

Módulo Shape (Forma)

Como su nombre indica define la forma, volumen o superficie en donde se emiten las partículas y la dirección de la velocidad inicial.

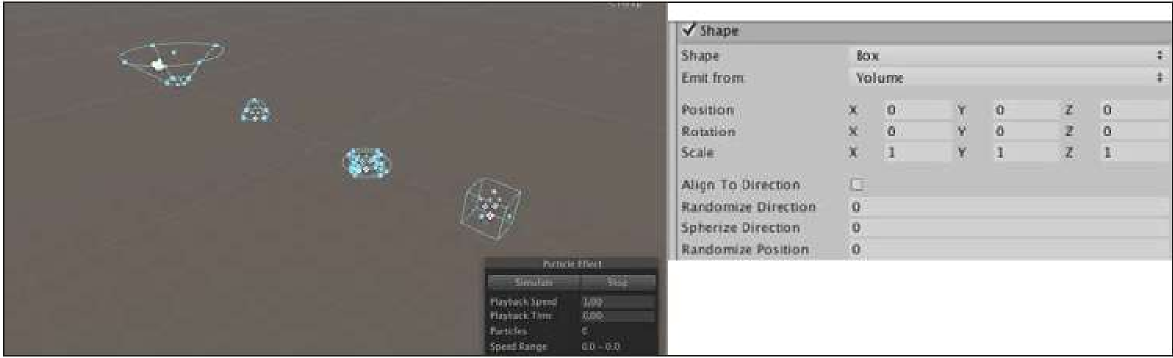


Fig. 13.7

La forma del emisor puede ser de cualquier tipo, en la opción Shape podemos seleccionar entre las siguientes que te muestro a continuación.

- **Sphere**: Emisión uniforme en todas las direcciones.
- **Hemisphere**: Emisión uniforme en todas las direcciones de un *solo lado* de un plano. Seria la mitad de una esfera.
- **Cone**: Emisión de la base de un cono. Las partículas divergen en proporción a sus distancia desde la linea del centro del cono.
- **Box**: Emisión del cuerpo de una forma **box** (caja). Las partículas se mueven en la dirección hacia adelante (Z) del objeto emisor.

- **Mesh**: Emisión desde cualquier forma arbitraría **mesh** proporcionada vía el inspector.
- **Mesh Renderer**: Emisión desde una referencia a un **Mesh** Renderer de un **Game Object**.
- **Skinned Mesh Renderer**: Emisión desde una referencia a un **Skinned Mesh** Renderer de un **Game Object**.
- **Circle**: Una emisión uniforme desde el centro de los bordes de un círculo. Las partículas se mueven solamente en un plano del círculo.
- **Edge:** Emisión desde un segmento de linea. Las partículas se mueven en la dirección hacia arriba (Y) del objeto emisor.

Shape	La forma del volumen de emisión. Para la forma Mesh, hay un menú extra para seleccionar si las partículas son emitidas desde los vértices, triángulos, o aristas del mesh.
Angle	El ángulo del cono en su punto (para Cone solamente). Un ángulo de 0 produce un cilindro mientras que un ángulo de 90 nos da un disco plano.
Radius	El radio del aspecto circular de la forma (para Sphere, Hemisphere, Cone, Circle y Edge solamente).
Length	La longitud del cono (para Cone solamente, cuando utilice uno de los modos de **volume emit-from**).
Box X, Y, Z	Anchura (**Width**), altura (**height**) y profunidad (**depth**) de la forma de la caja (para **Box** solamente).
Mesh	El mesh que proporciona la forma de la emisión (para **Mesh**, **MeshRenderer** y **Skinned Mesh** Renderer solamente).
Emit from Shell	Las partículas pueden ser emitidas desde la superficie externa en vez de con el volumen interno de la forma (Sphere y Hemisphere solamente).
Emit from	Selecciona la parte del cono desde dónde va a emitir: Base, Volume, Base Shell o Volume Shell (para Cono solamente).
Arc	La porción angular del círculo completo que forma la forma del emisor (para Circle solamente).
Emit From Edge	Las partículas pueden ser emitidas desde el borde del círculo en vez del centro (Para Circle solamente.)
Single Material	Las partículas pueden ser emitidas desde un sub-mesh en partículas (identificado por el número indice del material). Si está habilitado, se muestra un campo numérico que te permite especificar el número indice del material.
Use Mesh Colors	Utiliza, o ignora los colores mesh.

Normal Offset	Distancia de la superficie del mesh para emitir partículas (en la dirección de la normal de la superficie)
Random Direction	Cuando está habilitado, la dirección inicial de las partículas serán escogidas aleatoriamente.

Módulo Velocity over Lifetime

Este módulo se encarga de controlar la velocidad de las partículas durante el transcurso de su vida.

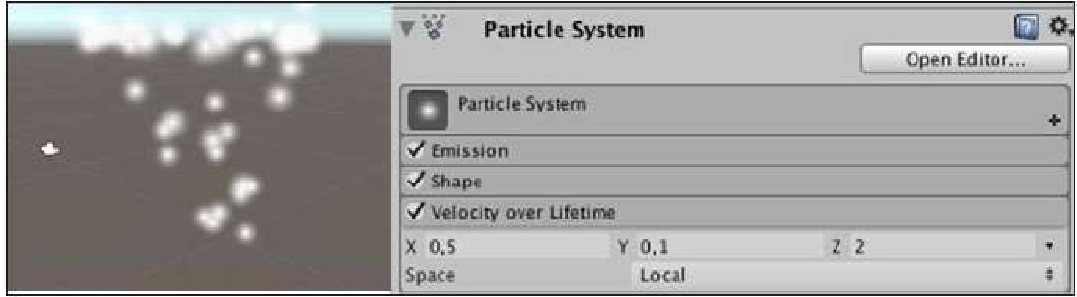


Fig. 13.8

X, Y, Z	Velocidad en los ejes X, Y, Z.
Space	Selecciona si los ejes X, Y y Z se refieren al espacio local o global (world).

Módulo Limit Velocity over Lifetime

Este modulo se encarga de reducir o limitar la velocidad de las partículas mientras perduren. En la imagen que se muestra a continuación son las formas o los métodos que podemos utilizar para manipular la velocidad . Para seleccionar una de las opciones debes pulsar encima de la flecha lateral y veras como se despliega un menú con las nuevas opciones.

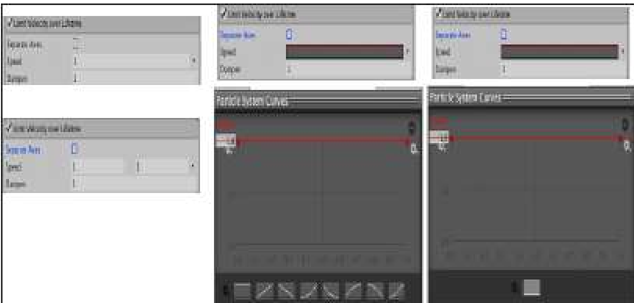


Fig. 13.9

La primera imagen es la opción **Constant** y la segunda es la **Random between two constans**.

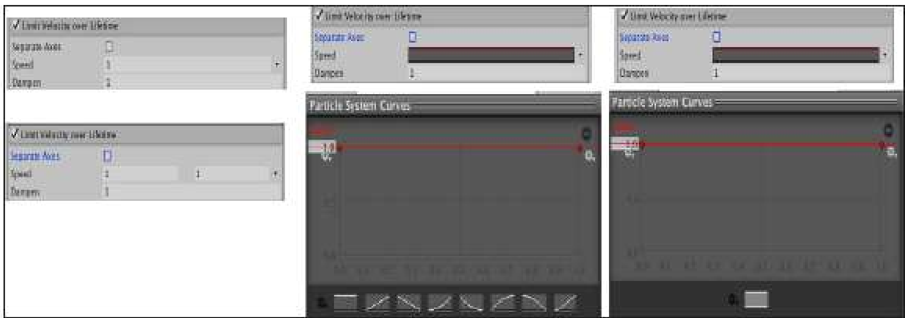


Fig. 13.10

De las imágenes anteriores la primera es la opción **curve** y la segunda es la **Random between two curves**.

Speed	Límite de velocidad (dividido en componentes separados X, Y y Z si Separate Axes está activado).
Space	Selecciona si el límite de velocidad se refiere al espacio local o global (solamente cuando Separate Axes está activado).
Dampen	La fracción en la cual la velocidad de una partícula será reducida cuando exceda el limite de velocidad (Imaginate que es un radar).

Esta opción es interesante para simular resistencia en las partículas, una explosión estalla en pedazos y los restos (partículas) van cayendo lentamente por el aire.

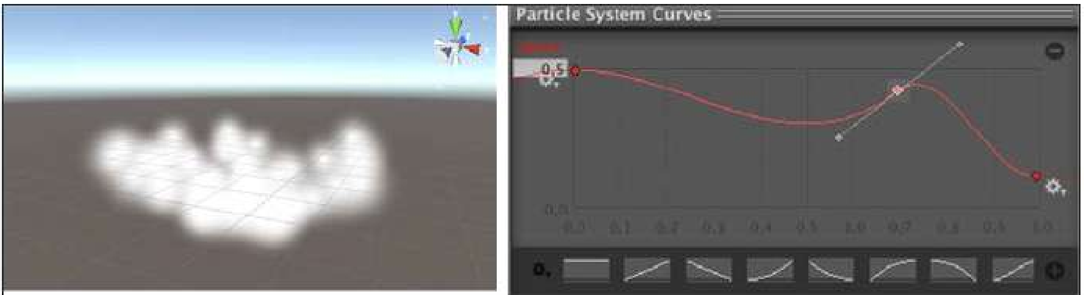


Fig. 13.11

Módulo Inherit Velocity

Este módulo nos permite controlar la velocidad de las partículas para que estás se adapten al tiempo.

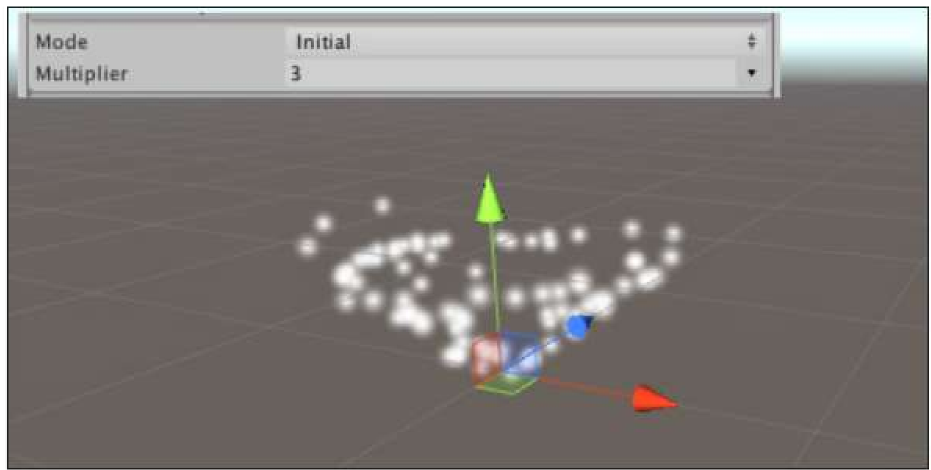


Fig. 13.12

En esta opción en el parámetro **Multiplier** también dispones de las opciones anteriores para controlar la velocidad **(Constant, Curve, Random between two constans, Random between two curves.)**

Mode	Especifica cómo la velocidad de emisión es heredada por las partículas. Current: velocidad es heredada constantemente, siguiendo la velocidad del emisor. Initial: la velocidad es heredada constantemente, utilizando un valor fijo de la velocidad del emisor cuando la partícula nació.
Multiplier	El porcentaje de la velocidad del emisor que la partícula debería heredar

Módulo Force over Lifetime

Este modulo nos permite manipular nuestras partículas mediante fuerzas que podemos variar utilizando distintos valores en los ejes (x,y,z) como te muestro en la siguiente imagen.

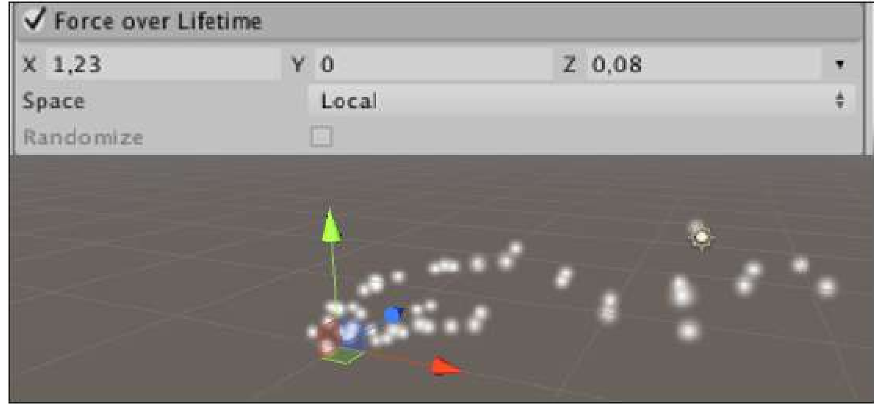


Fig. 13.13

Módulo Color over Lifetime

Este modulo nos permite especificar el color de una partícula y su transparencia en el tiempo. Podemos hacer que una partícula tenga un color claro en el inicio de su creación y hacer que se oscurezca conforme pase el tiempo como te muestro en la imagen siguiente.

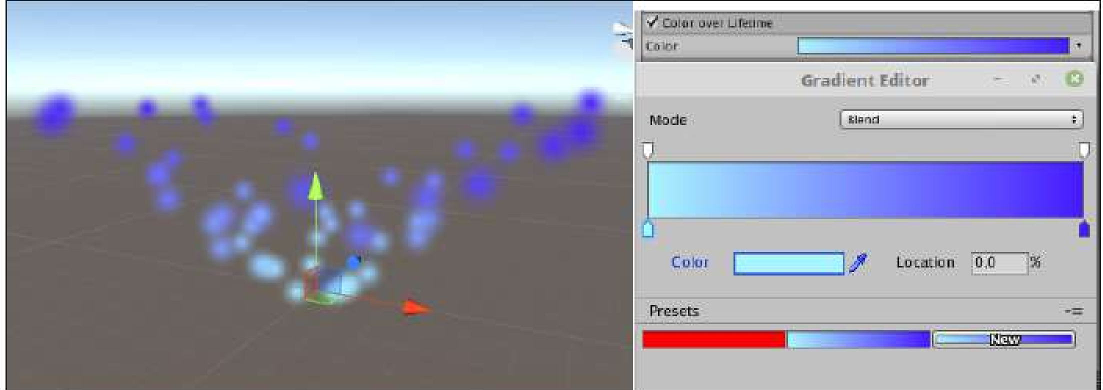


Fig. 13.14

Color	El degradado de color de una partícula en su tiempo de vida.

El parámetro color dispone de dos modos uno es el degradado simple **(Gradient)** y el otro te da la posibilidad de utilizar un doble degradado **(Random between two gradients).**

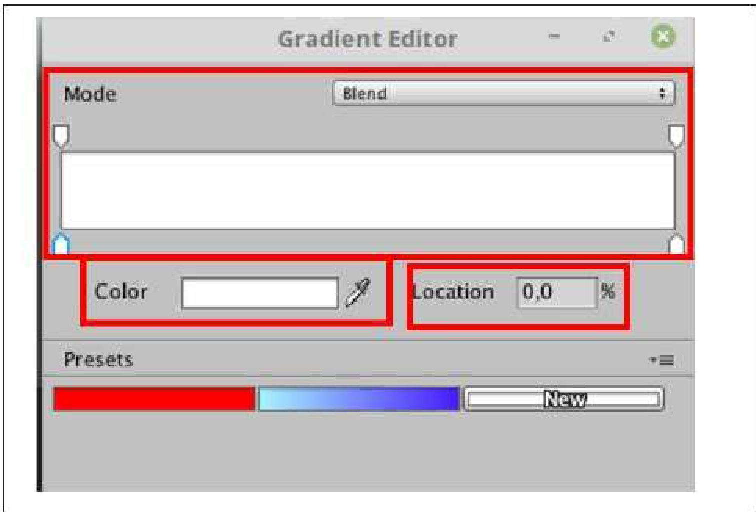


Fig. 13.15

El editor de degradado es muy sencillo en la opción **Mode** puedes seleccionar entre **Blend** y **Fixed**. **Blend** crea un suavizado entre los dos colores y el **Fixed** crea un corte fijo en donde empieza y termina los colores. Después tienes la barra de tonos que tiene una

especie de indicadores a los extremos de la barra. Cuando haces clic encima de uno de estos indicadores y seguidamente haces clic encima del selector color puedes seleccionar un color y su opacidad. Luego puedes mover este color a lo largo de la barra o bien moviendo el indicador o utilizando el parámetro **Location** en donde puedes introducir valores que van del 0 al 100.

Para finalizar debajo puedes guardar tus degradados. Cuando tengas un degradado que te gusta puedes pulsar en **New** y quedara tu degradado en la zona Presets, por si lo necesitas.

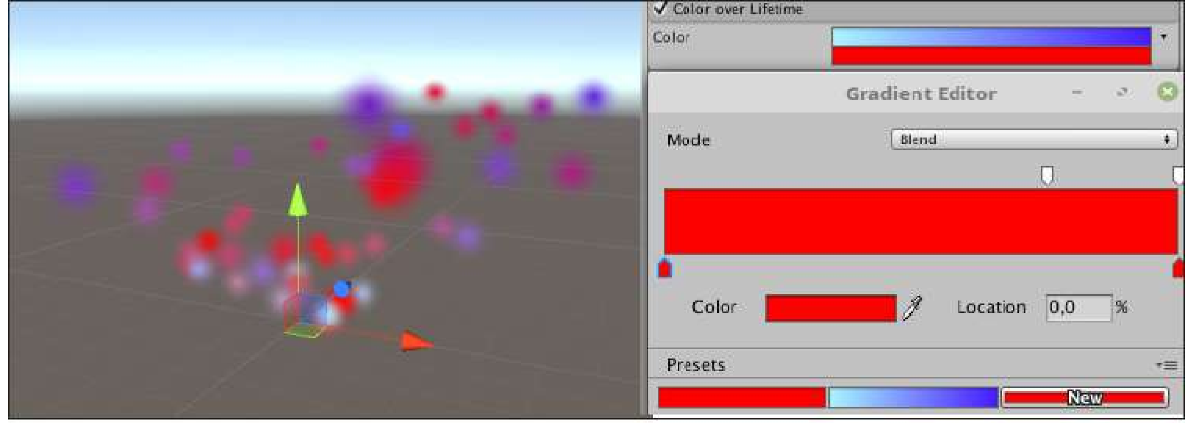


Fig. 13.16

Modulo Color by Speed

Con este modulo podemos hacer que nuestras partículas cambien de color de acuerdo a su velocidad en unidades de distancia por segundo.

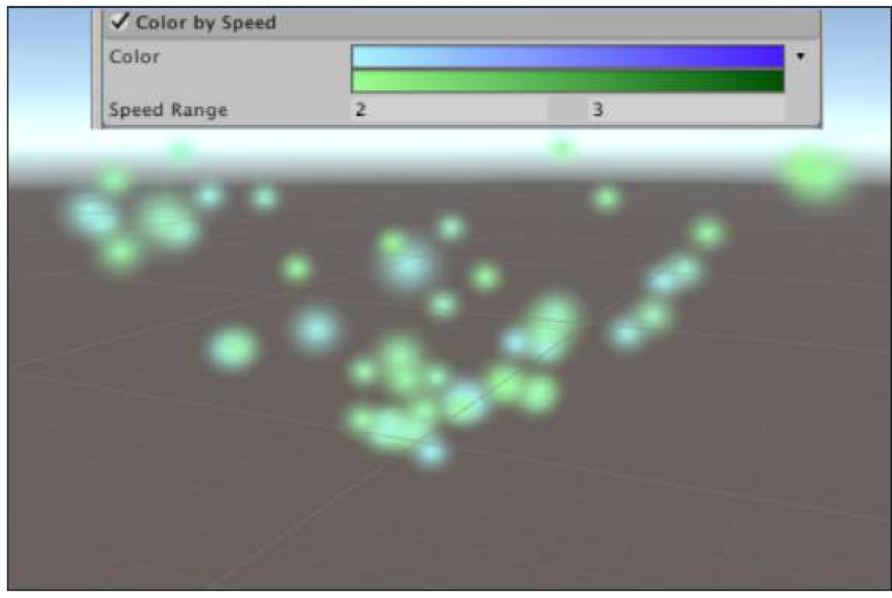


Fig. 13.17

Color	El degradado de color de una partícula definida sobre un rango de velocidad.
Speed Range	Los extremos inferiores y superiores del rango de velocidad el cual el degradado de color es mapeado (velocidades fuera de este rango serán mapeadas a los puntos extremos del degradado).

Módulo Size over Lifetime

En este modulo podemos cambiar el tamaño de las partículas mediante una curva.

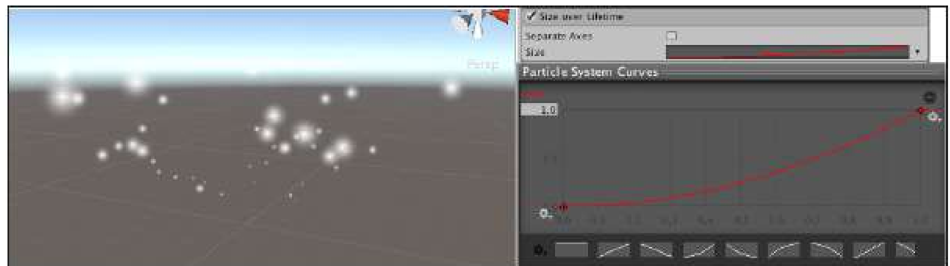


Fig. 13.18

Size	Una curva definiendo el tamaño de la partícula de acuerdo a su tiempo de vida.

En el parámetro **Size** (Tamaño) podemos utilizar las siguientes opciones (**Curve, Random between two constans, Random between two curves**).

Separate Axes	Si activamos esta opción podemos utilizar una curva para cada eje de coordenadas.

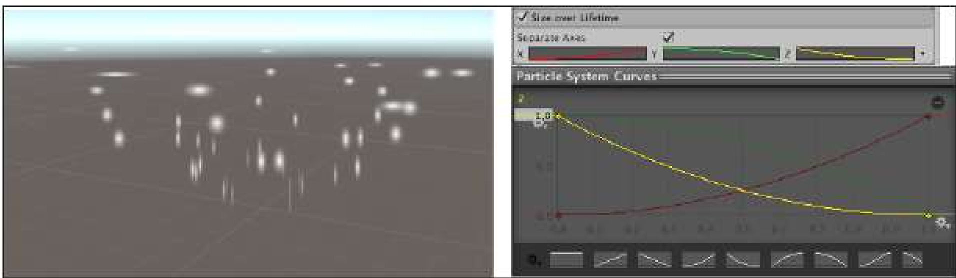


Fig. 13.19

Módulo Size by Speed

En este modulo podemos cambiar el tamaño de las partículas de acuerdo a su velocidad en unidades de distancia por segundo.

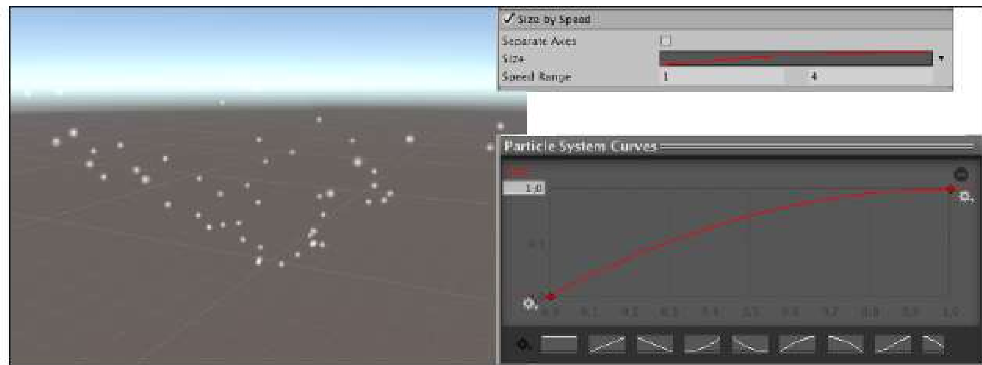


Fig. 13.20

Size	Una curva define el tamaño de la partícula sobre un rango de velocidad.
Speed Range	Las extremidades inferiores y superiores del rango de velocidad de la curva proporcionan el tamaño mapeado de esta (las velocidades fuera del rango, serán mapeadas a los puntos finales de la curva).

En el parámetro **Size** (Tamaño) podemos utilizar las siguientes opciones **(Curve, Random between two constans, Random between two curves).**

Módulo Rotation over Lifetime

En este modulo simplemente podemos configurar que las partículas giren a medida que se mueven. En este caso es muy útil si queremos que las partículas simulen las hojas de un árbol.

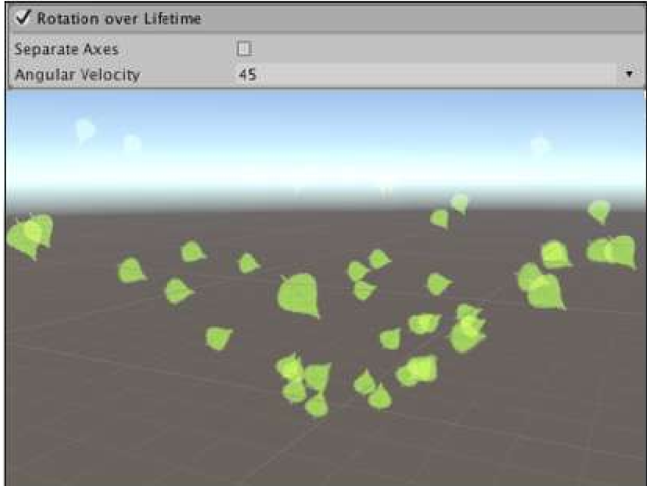


Fig. 13.21

En esta imagen he utilizado una textura para las partículas para facilitar la visualización de la rotación.

Separate Axes	Cuando activamos este parámetro nos permite especificar la rotación por eje.
Angular Velocity	Es la velocidad de rotación por segundo en grados.

Como hemos visto anteriormente podemos utilizar más opciones en Angular **Velocity** para configurar la rotación (**Curve, Random between two constans, Random between two curves**).

Módulo Rotation by Speed

En este modulo podemos configurar que las partículas cambien de acuerdo a su velocidad en unidades de distancia por segundo. En la documentación aseguran que se utiliza cuando las partículas representan objetos sólidos que se mueven sobre el suelo, es decir como rocas en un derrumbe. En el ejemplo he seguido utilizando hojas para jugar con la velocidad de rotación.

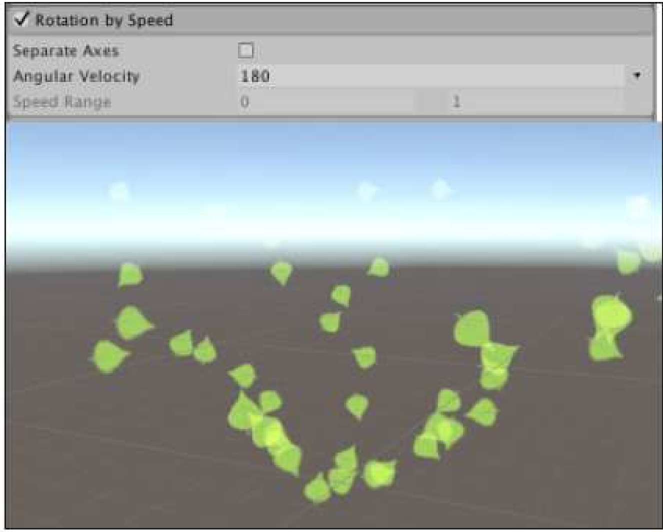


Fig. 13.22

Angular Velocity	Velocidad de rotación en grados por segundo.
Speed Range	Son los extremos inferiores y superiores del rango de velocidad a los cuales la curva de tamaño está mapeada (velocidades fuera del rango serán mapeadas a los puntos finales de la curva).

Módulo External Forces

Las partículas en ocasiones pueden estar afectadas por fuerzas externas como es el efecto **wind zone** que encontramos en un **Terrain**. Si activamos este modulo en el caso de que tengamos un **wind zone, este** podrá afectar al sistema.

Multiplier	Valor de escala aplicado a las fuerzas **wind zone** (zona de viento).

Módulo Noise

En este modulo podemos añadir una turbulencia al movimiento de partículas.

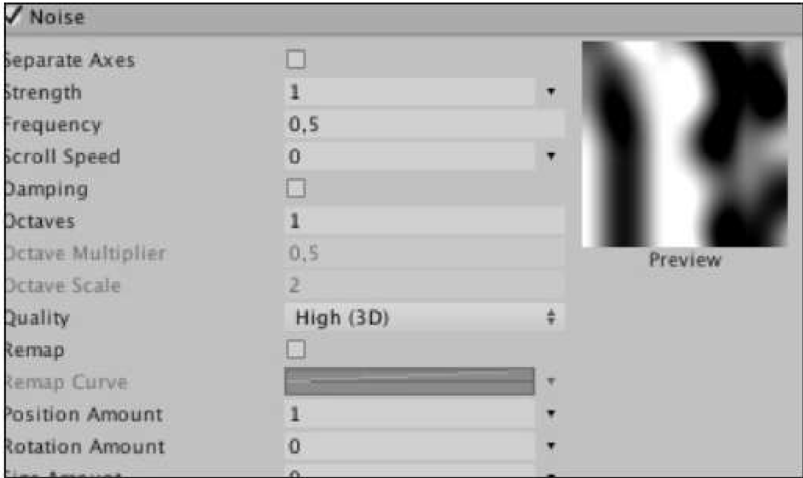


Fig. 13.23

Separate Axes	Cuando activamos esta opción podemos controlar la fuerza independientemente de cada eje.
Strength	Los valores más altos harán que las partículas se muevan más rápido. En el caso de que utilicemos una curva podremos definir la fuerza del efecto de ruido en una partícula a lo largo de su vida.
Frequency	Este parámetro controla la frecuencia con que las partículas cambian su dirección de desplazamiento y como de bruscos son esos cambios. Los valores bajos crean un ruido suave y los valores altos crean un ruido que cambia rápidamente.
Scroll Speed	El scroll mueve el campo de ruido a lo largo del tiempo para que el movimiento de la partícula sea más impredecible.
Damping	Cuando está habilitado, la fuerza es proporcional a la frecuencia. Al vincular estos valores, el campo de ruido se puede escalar manteniendo el mismo comportamiento, pero en un tamaño diferente.
Octaves	Especifica cuántas capas de ruido superpuesto se combinan para producir los valores de ruido finales. Cuanto más capas el ruido sera mas complejo, pero aumenta significativamente el consumo de recursos.
Octave Multiplier	Por cada capa de ruido adicional, ajusta la frecuencia con el multiplicador.

Octave Scale	Por cada capa de ruido adicional, reduce la fuerza por esta proporción.
Quality	Si ajustamos la calidad a menor reducimos significativamente el consumo de recursos, pero también afectan la apariencia del ruido. Si deseas un rendimiento máximo utiliza la calidad más baja que te ofrezca.
Remap	Reasigna los valores del ruido final a un rango distinto.
Remap Curve	La curva nos describe como se transforman los valores de ruido finales.
Position Amount	Es un multiplicador para controlar cuánto afecta el ruido a las posiciones de las partículas.
Rotation Amount	Es un multiplicador para controlar cuánto afecta el ruido a las rotaciones de partículas, en grados por segundo.
Size Amount	Es un multiplicador para controlar cuánto afecta el ruido a los tamaños de partículas.

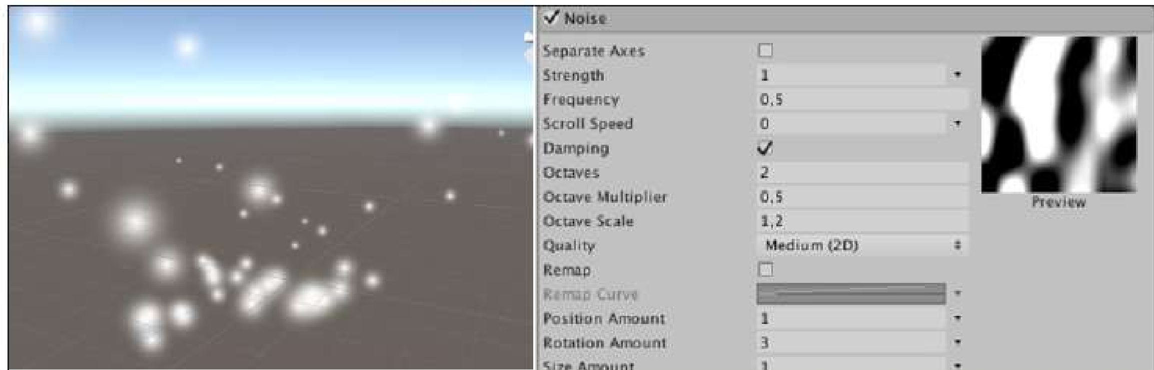


FIg. 13.24

Módulo Collision

Este modulo controla la forma en que las partículas colisionan con otros objetos sólidos en la escena. En este modulo debemos seleccionar primero la configuración de colisión que queremos trabajar (Planes o World). En el caso de que escojas World también debes definir que tipo de mundo es 2D o 3D.

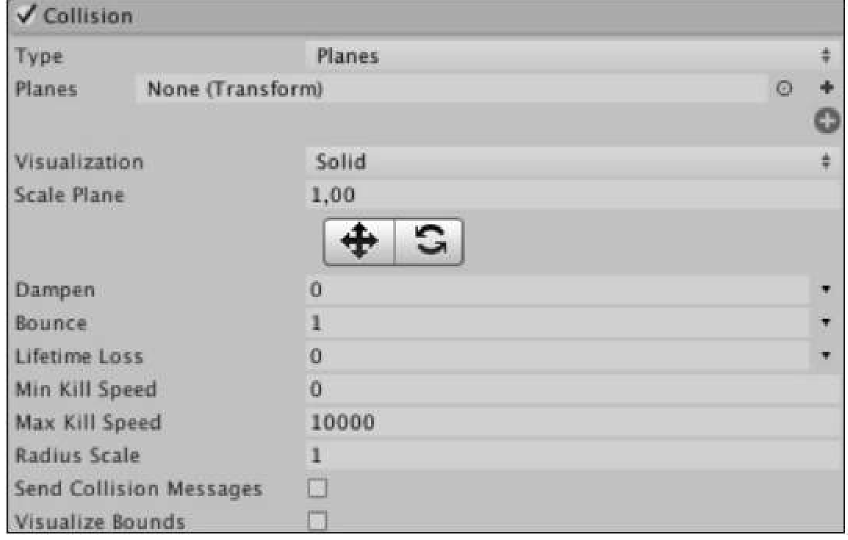


Fig. 13.25

Planes	Una lista ampliable de Transformaciones que definen planos de colisión.
Visualization	Selecciona si el plano de colisión Gizmos se mostrará en la vista de Escena como cuadrículas de alambre o planos sólidos.
Scale Plane	Tamaño de los planos utilizados para la visualización.
Dampen	La fracción de la velocidad de una partícula que pierde después de una colisión.
Bounce	La fracción de la velocidad de una partícula que rebota de una superficie después de una colisión.
Lifetime Loss	La fracción de la vida total de una partícula que pierde si colisiona.
Min Kill Speed	Las partículas que viajan por debajo de esta velocidad después de una colisión serán eliminadas del sistema.
Max Kill Speed	Las partículas que viajan por encima de esta velocidad después de una colisión serán eliminadas del sistema.
Radius Scale	Nos permite ajustar el radio de las esferas de colisión de partículas para que se ajuste mejor a los bordes visuales del gráfico de partículas.
Send Collision Messages	Si está habilitada la función OnParticleCollision puedes detectar colisiones de partículas a partir de secuencias de comandos.
Visualize Bounds	Representa los límites de colisión de cada partícula como una forma de estructura de alambre en la vista de Escena.

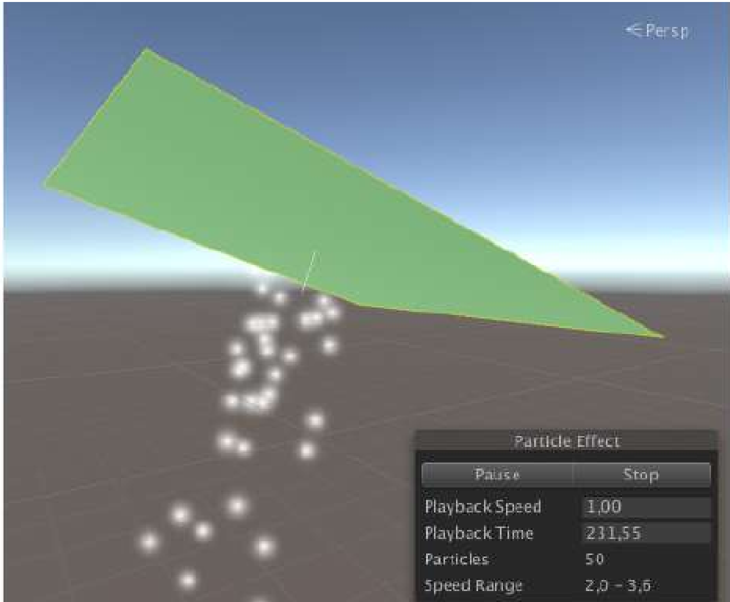


Fig. 13.26

World	Modo world (Mundo)
Collision Mode	3D o 2D.
Dampen	La fracción de la velocidad de una partícula que pierde después de una colisión.
Bounce	La fracción de la velocidad de una partícula que rebota de una superficie después de una colisión.
Lifetime Loss	La fracción de la vida total de una partícula que pierde si colisiona.
Min Kill Speed	Las partículas que viajan por debajo de esta velocidad después de una colisión serán eliminadas del sistema.
Max Kill Speed	Las partículas que viajan por encima de esta velocidad después de una colisión serán eliminadas del sistema.
Collision Quality	Usa el menú desplegable para establecer la calidad de las colisiones de partículas. Esto afecta la cantidad de partículas que pueden pasar a través de un colisionador. En niveles de calidad inferiores, las partículas a veces pueden pasar a través de colisionadores, pero requieren menos recursos para calcular.
Radius Scale	Configura la escala para 2D o 3D.
High	Cuando **Collision Quality** está configurada en **High** (Alto), las colisiones siempre usan el sistema de física para detectar los resultados de colisión. Esta es la opción más intensiva en recursos, pero también la más precisa.

Medium (Static Colliders)	Cuando **Collision Quality** se establece en **Medium** (Static Colliders), las colisiones usan una cuadrícula de **voxels** para almacenar en caché las colisiones anteriores, para una reutilización más rápida en los cuadros posteriores. La única diferencia entre **Medium** y **Low** es cuántas veces por cuadro el Sistema de Partículas consulta el sistema de física. **Medium** hace más consultas por cuadro que **Low**. ***ATENCIÓN: esta configuración solo es adecuada para colisionadores estáticos que nunca se mueven.***
Low (Static Colliders)	Cuando **Collision Quality** está configurada como **Low** (**Static Colliders**), las colisiones usan una cuadrícula de **voxels** para almacenar en caché las colisiones anteriores, para una reutilización más rápida en los cuadros posteriores.
Collides With	Las partículas solo colisionarán con objetos en las capas seleccionadas.
Max Collision Shapes	Especifica cuántas formas de colisión se pueden considerar para colisiones de partículas. El exceso de formas se ignora y los terrenos tienen prioridad.
Enable Dynamic Colliders	Permite que las partículas también colisionen con objetos dinámicos (de lo contrario, solo se utilizan objetos estáticos).
Enable Dynamic Colliders	Colisionadores dinámicos son cualquier colisionador no configurado como cinemático. Marcamos esta opción para incluir estos tipos de colisionador en el conjunto de objetos a los que responden las partículas en las colisiones. Desmarcamos esta opción y las partículas solo responden a colisiones contra colisionadores estáticos.
Voxel Size	Un **voxel** representa un valor en una cuadrícula regular en el espacio tridimensional. Al utilizar colisiones de calidad media o baja, Unity almacena colisiones en una estructura de cuadrícula. Esta configuración controla el tamaño de la cuadrícula. Los valores más pequeños dan más precisión, pero cuestan más memoria y son menos eficientes. ATENCIÓN: Solo puede acceder a esta propiedad cuando la Calidad de colisión está configurada en Medio o Bajo.
Collider Force	Aplica una fuerza a **Physics Colliders** después de una colisión de partículas. Esto es útil para empujar colisionadores con partículas.
Multiply by Collision Angle	Al aplicar fuerzas a colisionadores, escala la potencia de la fuerza en función del ángulo de colisión entre la partícula y el colisionador. Los ángulos esparcidos generarán menos fuerza que una colisión frontal.
Multiply by Particle Speed	Al aplicar fuerzas a colisionadores, escala la potencia de la fuerza en función de la velocidad de la partícula. Las partículas de movimiento rápido generarán más fuerza que las más lentas.

Multiply by Particle Size	Al aplicar fuerzas a colisionadores, escala la potencia de la fuerza en función del tamaño de la partícula. Las partículas más grandes generarán más fuerza que las más pequeñas.
Send Collision Messages	Verifique esto para poder detectar colisiones de partículas de las secuencias de comandos mediante la función **OnParticleCollision**.
Visualize Bounds	Obtenga una vista previa de las esferas de colisión para cada partícula en la vista de escena.

Módulo Triggers

Los sistemas de partículas tienen la capacidad de utilizar **Triggers** que nos permiten activar una devolución de llamada cuando las partículas interactúan con uno o más colisionadores en la escena. Esta devolución de llamada puede activarse cuando una partícula entra o sale de un Colisionador, o durante el tiempo que una partícula está dentro o fuera del Colisionador.

El módulo **Triggers** también ofrece la opción **Kill** para eliminar partículas automáticamente, y la opción Ignore para ignorar los eventos de colisión, que se muestran a continuación.

Inside	Selecciona **Callback** si desea que el evento se dispare cuando la partícula está dentro del Collider de un objeto. Selecciona **Ignore** para que el evento no se dispare cuando la partícula está dentro del Collider. Selecciona **Kill** para destruir las partículas de dentro del Collider.
Outside	Selecciona **Callback** si desea que el evento se dispare cuando la partícula está fuera del Collider de un objeto. Selecciona **Ignore** para que el evento no se dispare cuando la partícula está fuera del Collider. Selecciona **Kill** para destruir las partículas de fuera del Collider.
Enter	Selecciona **Callback** si desea que el evento se dispare cuando la partícula entra en el Collider de un objeto. Selecciona **Ignore** para que el evento no se dispare cuando la partícula entra en el Collider. Selecciona **Kill** para destruir las partículas que entran en el Collider.
Exit	Selecciona **Callback** si desea que el evento se dispare cuando la partícula salga del Collider de un objeto. Selecciona **Ignore** para que el evento no se dispare cuando la partícula salga del Collider. Selecciona **Kill** para destruir las partículas que salgan del Collider.

Radius Scale	Este parámetro establece los límites del Colisionador de la partícula, lo que nos permite que un evento parezca suceder antes o después de que la partícula toque el Colisionador. Un ejemplo sería que una partícula penetre en la superficie de un objeto Collider un poco antes de rebotar, en cuyo caso estableceríamos que Radius Scale sea un poco menor que 1. El valor 1 es para que el evento parezca suceder cuando una Partícula toca el Colisionador Los valores inferiores a 1 son para que el disparador parezca suceder después de que una partícula penetre en el colisionador Los valores superiores a 1 son para que el disparador parezca suceder antes de que una Partícula penetre en el Colisionador.
Visualize Bounds	Esto nos permite mostrar los limites del colisionador de partículas en la ventana del editor.

Fig. 13.27

En la imagen anterior se le ha añadido un script con nombre **colorParticulas** en **C#** al emisor de partículas para que cambien de color las partículas según en donde se encuentren. A continuación, te comparto el script por si quieres probarlo.

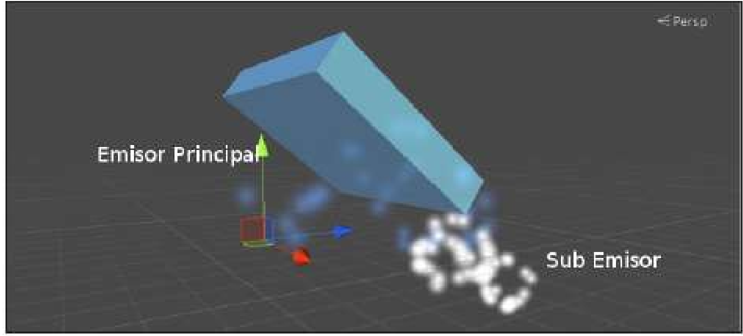
```
using UnityEngine;
using System.Collections;
using System.Collections.Generic;

[ExecuteInEditMode]
public class colorParticulas : MonoBehaviour
{
        ParticleSystem ps;
        // estas listas se usan para contener las partículas que coinciden
        // El trigger condiciona cada frame.
        List<ParticleSystem.Particle> enter = new List<ParticleSystem.
Particle>();
```

```
    List<ParticleSystem.Particle> exit = new List<ParticleSystem.Particle>();

    void OnEnable()
    {
        ps = GetComponent<ParticleSystem>();
    }

    void OnParticleTrigger()
    {
      // obtener las partículas que coinciden con las condiciones del trigger
de este frame
        int numEnter = ps.GetTriggerParticles(ParticleSystemTriggerEventType.
Enter, enter);
        int numExit = ps.GetTriggerParticles(ParticleSystemTriggerEventType.
Exit, exit);

      // Repite a través de las partículas que entran en el trigger y las hace
rojas
        for (int i = 0; i < numEnter; i++)
        {
               ParticleSystem.Particle p = enter[i];
               p.startColor = new Color32(255, 0, 0, 255);
               enter[i] = p;
        }

      // Repite a través de las partículas que salen del trigger y las hace
verdes
        for (int i = 0; i < numExit; i++)
        {
               ParticleSystem.Particle p = exit[i];
               p.startColor = new Color32(0, 255, 0, 255);
               exit[i] = p;
        }

      // reasignar las partículas modificadas de nuevo al sistema de partículas
        ps.SetTriggerParticles(ParticleSystemTriggerEventType.Enter, enter);
        ps.SetTriggerParticles(ParticleSystemTriggerEventType.Exit, exit);
    }
}
```

Este ejemplo no hace falta hacerlo simplemente lo he incluido como una curiosidad pero no es necesario que lo utilices.

Módulo Sub Emitters

Este módulo nos permite configurar **sub-emitters** (sub emisores de partículas). Estos son emisores de partículas adicionales que se crean en la posición de una partícula en ciertas etapas de su ciclo de vida.



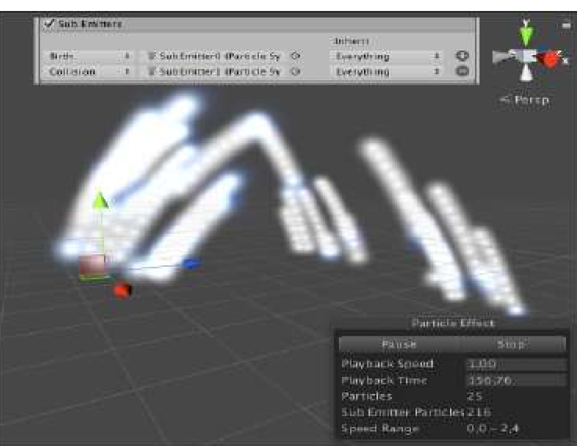


Fig. 13.28

Sub Emitters	Aquí podemos configurar una lista de sub-emisores y seleccionar su condición de activación, así como también las propiedades que heredan de sus partículas originales.

Hay tres condiciones que podemos utilizar para los trigger:

- **Birth** (Nacimiento); Cuando se crea la partícula.
- **Collision** (Colisión): Cuando la partícula colisiona con un objeto.
- ***Death** (Muerte): Cuando se destruyen las partículas.*

Módulo Texture Sheet Animation

El gráfico de una partícula no necesita ser una imagen fija. Este módulo le permite tratar la Textura como una cuadricula de subimágenes independientes que se pueden reproducir como cuadros de animación.

Propiedades del modo Grid

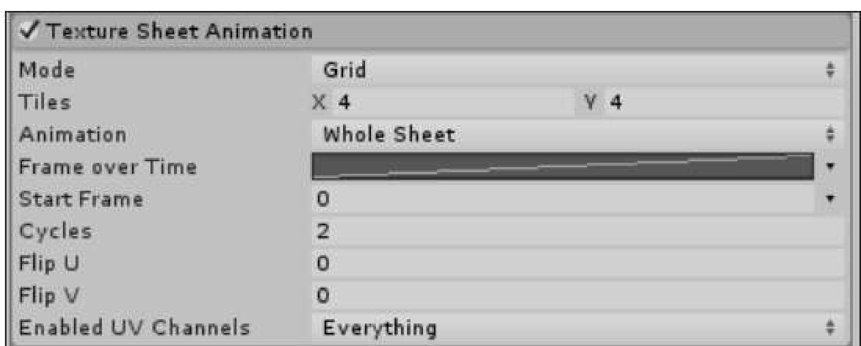


Fig. 13.29

Mode popup	Seleccionamos Grid
Tiles	El número de imágenes en que se divide la textura en las direcciones X (horizontal) e Y (vertical).
Animation	El modo Animación se puede establecer en **whole sheet** o **Single Row** (es decir, cada fila de la hoja representa una secuencia de Animación separada).
Random Row	Elige una fila de la hoja al azar para producir la animación. Esta opción solo está disponible cuando se selecciona **Single Row** como modo de animación.
Row	Selecciona una fila particular de la hoja para producir la animación. Esta opción solo está disponible cuando se selecciona el modo **Single Row** y **Random Row** está desactivada.
Frame over Time	Una curva que especifica cómo el marco de la animación aumenta a medida que avanza el tiempo.
Start Frame	Te permite especificar en qué marco debes comenzar la animación de la partícula (útil para dividir aleatoriamente la animación de cada partícula).
Cycles	La cantidad de veces que la secuencia de animación se repite durante la vida de la partícula.
Flip U	Horizontalmente refleja la textura en una proporción de las partículas. Un valor más alto arroja más partículas.
Flip V	Muestre verticalmente la textura en una proporción de las partículas. Un valor más alto arroja más partículas.
Enabled UV Channels	Te permite especificar exactamente qué corrientes de UV se ven afectadas por el sistema de partículas.

Propiedades del modo sprite

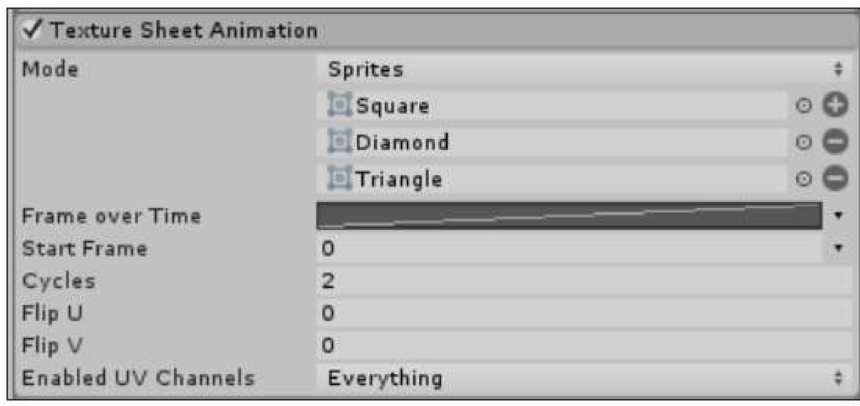


Fig. 13.30

Mode popup	Selecciona el modo sprite
Frame over Time	Una curva que especifica cómo el marco de la animación aumenta a medida que avanza el tiempo.
Start Frame	Te permite especificar en qué marco debe comenzar la animación de la partícula (útil para dividir aleatoriamente la animación de cada partícula).
Cycles	La cantidad de veces que la secuencia de animación se repite durante la vida de la partícula.
Flip U	Horizontalmente refleja la textura en una proporción de las partículas. Un valor más alto arroja más partículas.
Flip V	Muestra verticalmente la textura en una proporción de las partículas. Un valor más alto arroja más partículas.
Enabled UV Channels	Te permite especificar exactamente qué corrientes de UV se ven afectadas por el sistema de partículas.

Módulo Lights

Con este modulo podemos agregar una luz en tiempo real a un porcentaje de particular.

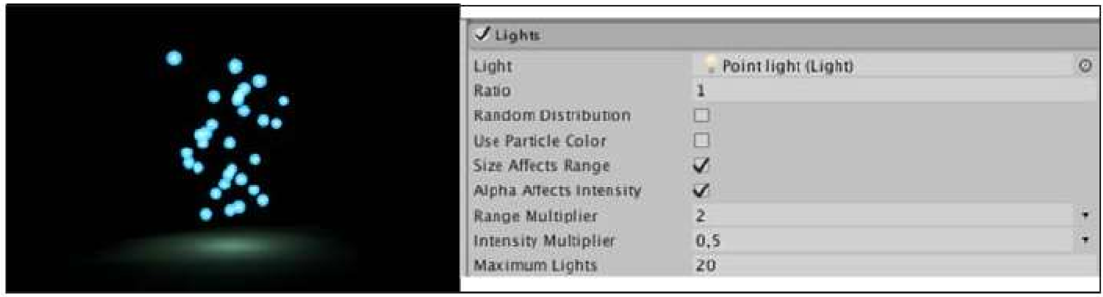


Fig. 13.31

Light	Asignamos que tipo de luz queremos para nuestras partículas.
Ratio	Un valor entre 0 y 1 describe la proporción de partículas que recibirán una luz.
Random Distribution	Eliges si las luces se asignan de forma aleatoria o periódica. Cuando se activa esta opción, cada partícula tiene una posibilidad aleatoria de recibir una luz basada en la relación. Los valores más altos aumentan la probabilidad de que una partícula tenga una luz. Cuando se establece en falso, la relación controla la frecuencia con la que una partícula recién creada recibe una luz (por ejemplo, cada enésima partícula recibirá una luz).
Use Particle Color	Cuando se activa esta opción, el color final de la Luz será modulado por el color de la partícula a la que está unido. Si se desactiva, el color de la luz se usa sin ninguna modificación.
Size Affects Range	Cuando está activada esta opción, el Rango especificado en la Luz se multiplicará por el tamaño de la partícula.
Alpha Affects Intensity	Cuando está habilitado, la Intensidad de la luz se multiplica por el valor alfa de la partícula.
Range Multiplier	Aplica un multiplicador personalizado al rango de la luz durante la vida útil de la partícula utilizando esta curva.
Intensity Multiplier	Aplica un multiplicador personalizado a la intensidad de la luz a lo largo de la vida de la partícula usando esta curva.
Maximum Lights	Utiliza esta configuración para evitar la creación accidental de una enorme cantidad de luces, lo que podría hacer que el Editor no responda o hacer que tu aplicación se ejecute muy lentamente.

Módulo Trails

Cuando hablamos de **Trails** estamos hablando de residuo o rastro que deja la partícula. En otras palabras podemos agregar a nuestras partículas un rastro o residuo. Este mo-

dulo es muy útil para crear una gran variedad de efectos, como humo, efectos de conjuros mágicos o proyectiles.

Ratio	Un valor entre 0 y 1, que describe la proporción de partículas que tienen un rastro asignado a ellos. Los rastros se asignan al azar, por lo que este valor representa una probabilidad.
Lifetime	La vida útil de cada vértice en el rastro, expresada como un multiplicador de la vida de la partícula a la que pertenece. Cuando se agrega cada nuevo vértice al rastro, desaparece después de que ha existido por más tiempo que su duración total.
Minimum Vertex Distance	Define la distancia que debe recorrer una partícula antes de que el recorrido reciba un nuevo vértice.
Texture Mode	Elije si la Textura aplicada al Camino se estira a lo largo de toda su longitud, o si repite cada N unidades de distancia. La velocidad de repetición se controla en función de los parámetros de Mosaico en el Material.
World Space	Cuando esta opción se habilita, los vértices del rastro no se mueven en relación con GameObject del Sistema de Partículas, incluso si se usa el Espacio de Simulación Local. En cambio, los vértices del rastro se dejan caer en el mundo e ignoran cualquier movimiento del Sistema de Partículas.
Die With Particles	Si está opción se habilita, los rastros desaparecen instantáneamente cuando mueren sus partículas. Si no está habilitado, el rastro restante expira de forma natural en función de su propia vida útil restante.
Size affects Width	Si la casilla está marcada, el ancho del rastro se multiplica por el tamaño de partícula.
Size affects Lifetime	Si la casilla está marcada, el tiempo de vida de rastro se multiplica por el tamaño de partícula.
Inherit Particle Color	Si la casilla está marcada, el color del rastro se modula por el color de la partícula.
Color over Lifetime	Una curva para controlar el color de todo el rastro a lo largo de la vida de la partícula a la que está unido.
Width over Trail	Una curva para controlar el ancho del rastro sobre su longitud.
Color over Trail	Una curva para controlar el color del rastro sobre su longitud.

Generate Lighting Data	Si la casilla está marcada, la geometría del rastro está construida con Normales y Tangentes incluidos. Esto nos permite usar materiales que usan iluminación de escena, por ejemplo, a través del sombreador estándar, o usando un sombreador personalizado.

Fig. 13.32

Módulo Custom Data

El módulo **Custom Data** (Datos personalizados) nos permite definir formatos de datos personalizados en el Editor para adjuntar los a las partículas. También puedes configurar esto en un script pero no lo vamos a ver en esta obra.

Las etiquetas predeterminadas para cada curva o degradado se pueden personalizar haciendo clic en ellas y escribiendo un nombre contextual. Al pasar datos personalizados a sombreadores, es útil saber cómo se usan esos datos dentro del sombreador. Un ejemplo seria que una curva se puede usar para pruebas alpha personalizadas, o se puede usar un degradado para agregar un color secundario a las partículas. Al editar las etiquetas, es simple mantener un registro en la interfaz de usuario del propósito de cada entrada de datos personalizada.

Módulo Renderer

Este modulo determina como la imagen de una partícula o malla es transformada, sombreada por otras partículas.

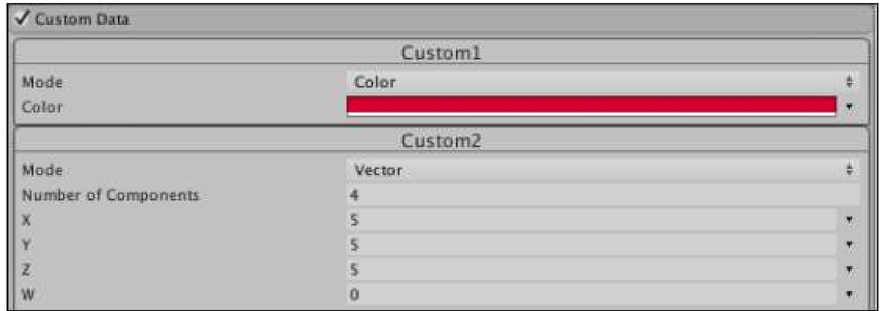


Fig. 13.33

Render Mode	Cómo se produce la imagen renderizada a partir de la imagen gráfica (o Mesh) • **Billboard:** la partícula siempre se ve de frente a la cámara. • **Stretched Billboard:** la partícula se ve de frente a la cámara pero con varias escalas aplicadas. • **Camera Scale:** estira las partículas según el movimiento de la cámara. Ajusta el valor a 0 para deshabilitar el movimiento de la cámara. • **Velocity Scale:** estira las partículas proporcionalmente a su velocidad. Ajusta el valor a 0 para desactivar el estiramiento según la velocidad. • **Lenght Scale:** estira las partículas proporcionalmente a su tamaño actual en la dirección de su velocidad. Ajusta el valor a 0 para que las partículas desaparezcan, dándole el valor 0 a longitud. • **Horizontal Billboard:** el plano de partículas es paralelo al plano "suelo" XZ. • **Vertical Billboard:** la partícula está en posición vertical en el eje Y en modo global, pero gira para mirar hacia la cámara. • **Mesh:** la partícula se representa desde una malla 3D en lugar de una textura. • **None:** esto puede ser útil cuando se utiliza el módulo Trails, si solo desea que se muestre los rastros y ocultar el renderizado predeterminado.
Normal Direction	Bias de las normales de iluminación utilizadas para los gráficos de partículas. Un valor de 1,0 las normales miran hacia la cámara, mientras que un valor de 0,0 las normales miran hacia el centro de la pantalla. (solo cuando utilizamos el modo Billboard)
Material	Material utilizado para mostrar las partículas.
Trail Material	Material utilizado para renderizar rastros de partículas. Esta opción solo está disponible cuando el módulo Trails está habilitado.
Sort Mode	El orden en que se dibujan las partículas (y por lo tanto superpuestas). Los valores posibles son By Distance(Por distancia desde la cámara), Oldest in Front (Más antiguo al frente) y Youngest in Front (Más joven al frente). Cada partícula dentro de un sistema se ordenará de acuerdo con esta configuración.
Sorting Fudge	Las bias del ordenamiento del sistema de partículas. Los valores más bajos aumentan la posibilidad relativa de que los sistemas de partículas se dibujen sobre otros GameObjects transparentes, incluidos otros sistemas de partículas. Esta configuración solo afecta dónde aparece un sistema completo en la escena; no realiza la clasificación en partículas individuales dentro de un sistema.
Min Particle Size	El tamaño de partícula más pequeño (independientemente de otras configuraciones), es expresado como una fracción del tamaño de la ventana gráfica. Hay que tener en cuenta que esta configuración solo se aplica cuando el modo de reproducción está configurado en Billboard.

Max Particle Size	El tamaño de partícula más grande (independientemente de otras configuraciones), es expresado como una fracción del tamaño de la ventana gráfica. Hay que tener en cuenta que esta configuración solo se aplica cuando el modo de reproducción está configurado en Billboard.
Billboard Alignment	Use el menú desplegable para elegir a qué dirección son alineadas las billboards de partículas. • **View:** las partículas son alineadas al plano de la cámara. • **World:** las partículas son alineadas con el eje global. • **Local:** las partículas son alineadas al componente de transformación de los GameObjects. • **Facing:** las partículas son alineadas a la posición directa de la cámara GameObject.
Pivot	Modifica el punto de pivote utilizado como el centro para partículas giratorias.
Visualize Pivot	Proporciona una vista previa de los puntos de pivote de la partícula en la vista de escena.
Custom Vertex Streams	Configura qué propiedades de partícula están disponibles en el Vertex Shader de tu Material.
Cast Shadows	Si está habilitado, el sistema de partículas crea sombras cuando una Luz shadow-casting brilla sobre ella. • **On:** para activar las sombras. • **Off:** para desactivar las sombras. • **Two Sided:** para permitir proyectar sombras desde cualquier lado de la Malla. • **Shadows Only:** para que las sombras sean visibles y la malla no.
Receive Shadows	Dicta si las sombras se pueden lanzar sobre las partículas. Solo los materiales opacos pueden recibir sombras.
Sorting Layer	Nombre de la capa de clasificación del Renderer.
Order in Layer	El orden de este Renderer dentro de una capa de clasificación.
Light Probes	Modo de interpolación de iluminación basada en sonda.
Reflection Probes	Si esta activado las pruebas de reflexión en la escena, se selecciona una textura de reflexión para este GameObject y se establece como una variable uniforme integrada de Shader.

Crear materiales para un sistema de partículas

Es muy sencillo crear materiales para nuestras partículas, para seguir un orden en la ventana **Project** crea una carpeta dentro de Assets con el nombre Materiales y luego dentro de esta carpeta crearemos 3 materiales que contendrán 3 imágenes una estrella,

un sol y una hoja. Para crear un material solamente tienes que ir a la barra principal de Unity acceder al menú ***Assets > Create > Material***. El nuevo material le pones un nombre y en el caso de que lo hayas creado fuera de la carpeta ***Assets > Materiales*** , lo arrastras a dentro de su carpeta desde la ventana **Project**.

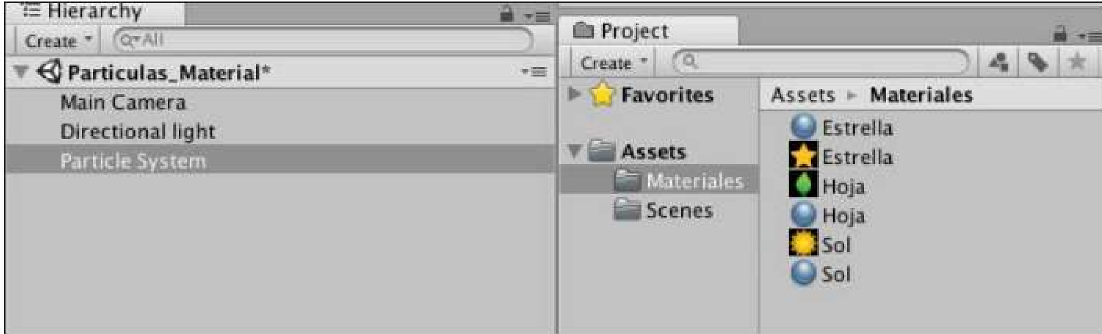


Fig. 13.34

Seleccionamos uno de los materiales por ejemplo en el caso que te muestro a continuación es el material Hoja. Por defecto al seleccionar este material nos aparece el shader Standard debemos hacer clic encima del menú y seleccionar la opción Particles > Additive o cualquiera de las opciones que existen dentro de Particles.

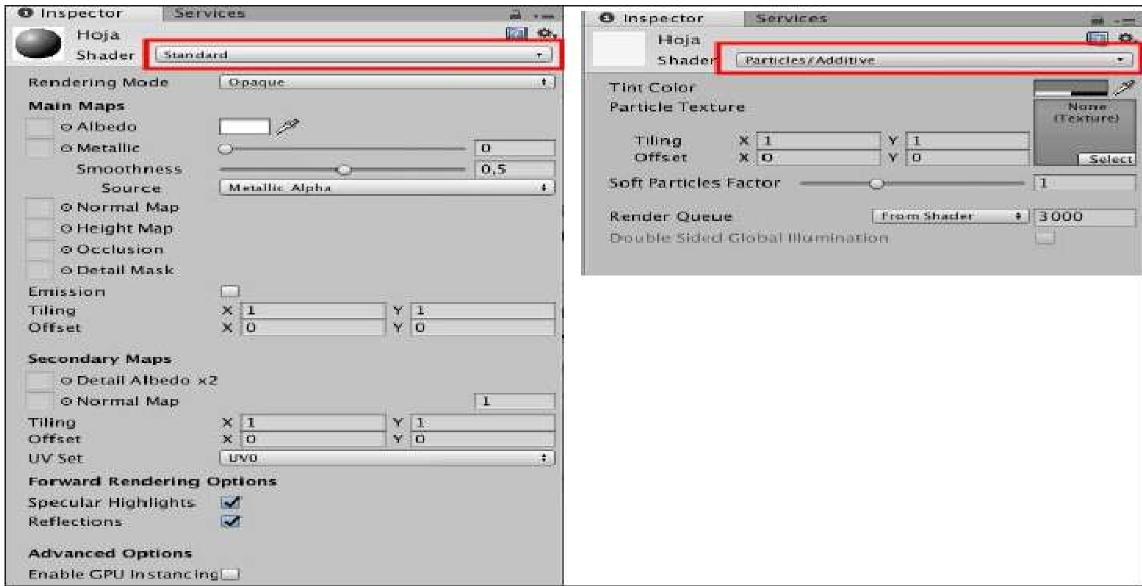


Fig. 13.35

El siguiente paso es el de seleccionar una imagen utilizando el botón select dentro de la sección Particle Texture y escogiendo la imagen de la hoja. Esta imagen ha sido guardada en el formato png porque guarda el canal alfa.

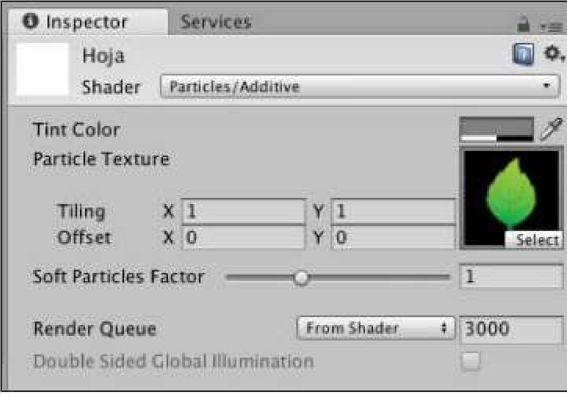


Fig. 13.36

Ahora si creas un sistema de partículas por defecto en Unity y accedemos a sus propiedades deberemos acceder a sus opciones de Renderer y en el apartado Material añadir el material con el nombre Hoja.

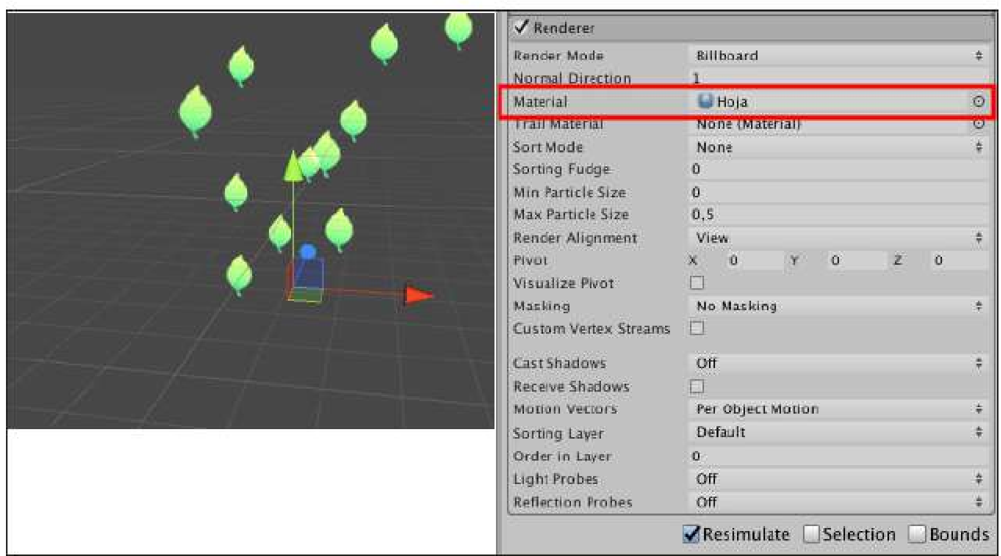


Fig. 13.37

Como puedes ver las hojas aparecen en el sistema de partículas pero el tipo de material que le hemos dado no es del todo correcto para el tipo de imagen que estamos utilizando. A continuación se explican que tipos de materiales disponemos para las partículas y sus propiedades generales.

Material Additive y Additive (Soft)

Este tipo de material crea la ilusión de un material que emite luz. La diferencia que tenemos entre estos dos tipos de material es que Additive tiene un parámetro Tint color que permite tintar la imagen o potenciar la emisión. Para que la imagen parezca más iluminada deberás aproximarte al color blanco en el caso contrario el color negro provocara que la imagen desaparezca o se apague.

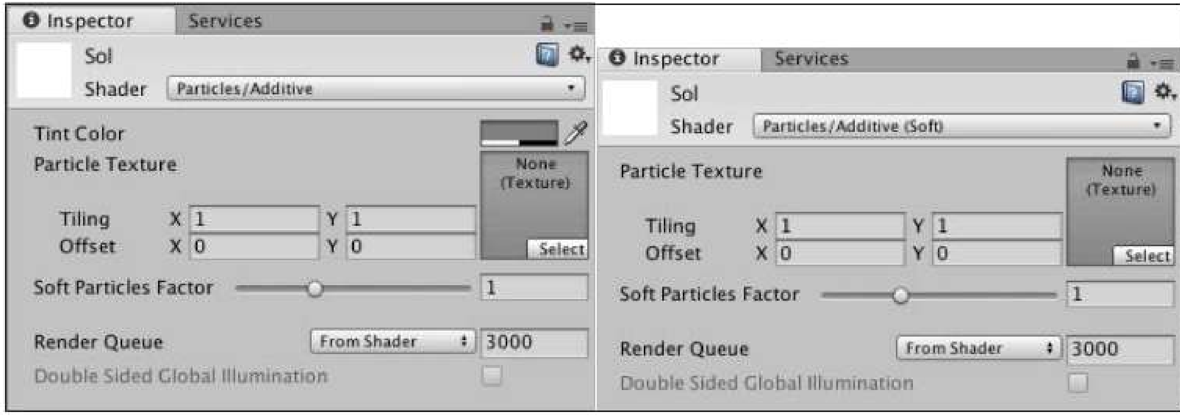


Fig. 13.38

Propiedades que encontraremos en todos los demás materiales son:

Particle Texture	Nos permite añadir una textura **Tiling** : las texturas son en 2 dimensiones y por ese motivo si la textura esta teselada es decir que podemos repetirla en los dos ejes y obtenemos un patrón. **Offset:** nos permite corregir la posición de la textura.
Soft Particles Factor	Nos permite suavizar la muestra de nuestra textura entre los valores 0,01 a 3.

Material Multiply , Multiply (Double) y Additive-Multiply

Este tipo de material crea el efecto contrario al anterior, es decir oscurece la textura. Es el mismo efecto que encontraras en los modos de fusión de cualquier programa de edición gráfico. Crea una transparencia en las partículas oscureciendo la textura. La diferencia principal es que el Additive -Multiply te permite tintar la textura o jugar con su canal alpha.

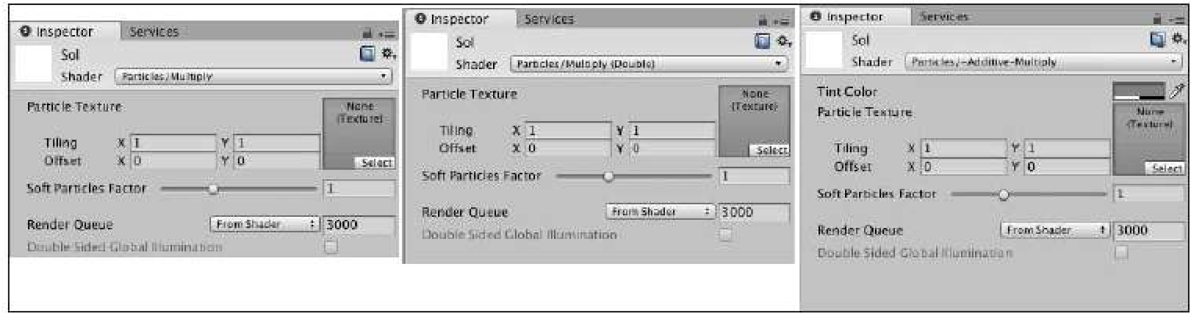


Fig. 13.39

Material Alpha Blended y Alpha Blended Premultiply

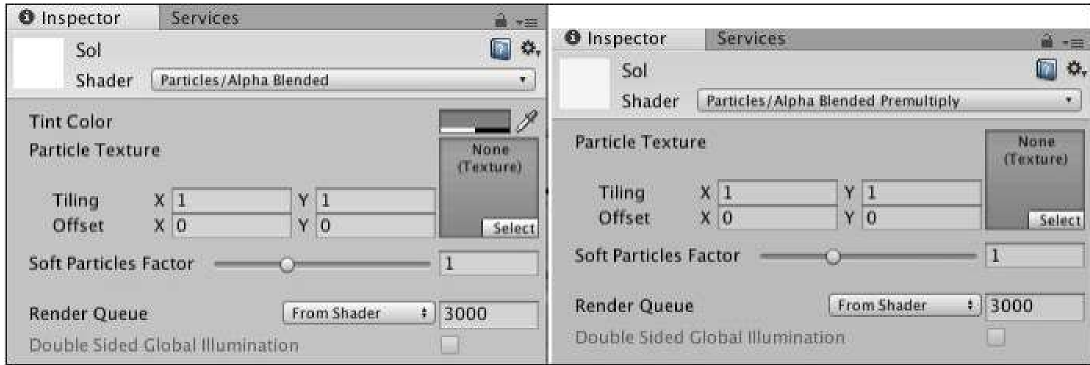


Fig. 13.40

Este material es ideal para texturas con canal **alpha** como el que te he mostrado anteriormente con la hoja. La principal diferencia es que en **Alpha Blended** te permite tintar la textura.

Material VertexLit Blended y Anim Alpha Blended

Estos dos materiales son distintos el de **VertexLit Blended** puedes hacer que la textura sea mas clara cuando el valor del color es mas próximo a blanco y cuando el valor es más próximo al color negro la textura puede llegar a ser negra.

En el caso del material **Anim Alpha Blended** podemos jugar con la posición del canal alfa de la textura.

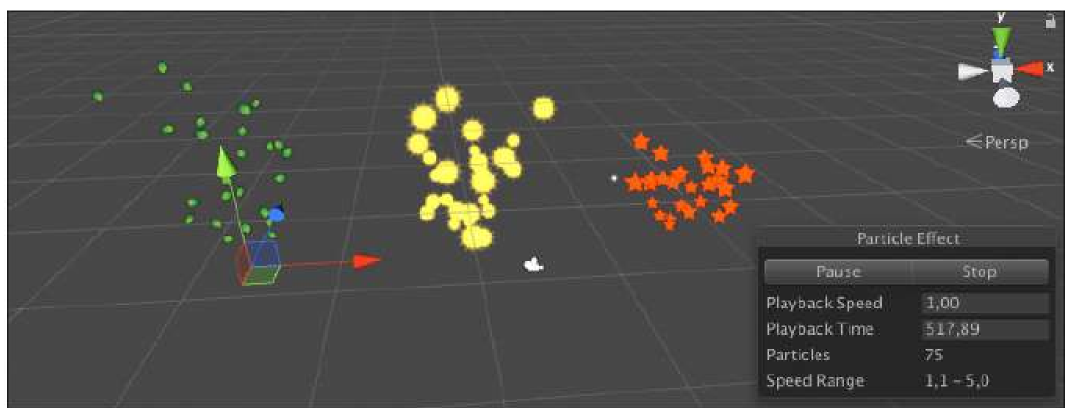


Fig. 13.41

El tema de partículas puede llegar a ser muy abrumador por la cantidad de parámetros de las cuales disponemos. Todas las propiedades que hemos enunciado anteriormente las veremos mucho mejor con el proyecto que viene a continuación.

Te propongo que utilices este capitulo a modo de consulta cuando no sepas para que sirve alguno de los parámetros, en el caso de que te interese especialmente este sector

lo ideal seria que intentaras emular los elementos como vamos a realizar a continuación con la creación de un fuego.

Creación de un fuego con partículas

Existen muchas formas de crear elementos con partículas en este proyecto vamos a crear un fuego con humo utilizando una textura que podemos crear con cualquier programa de pintura digital. En este caso en concreto te facilito la textura con el material que acompaña este capítulo.

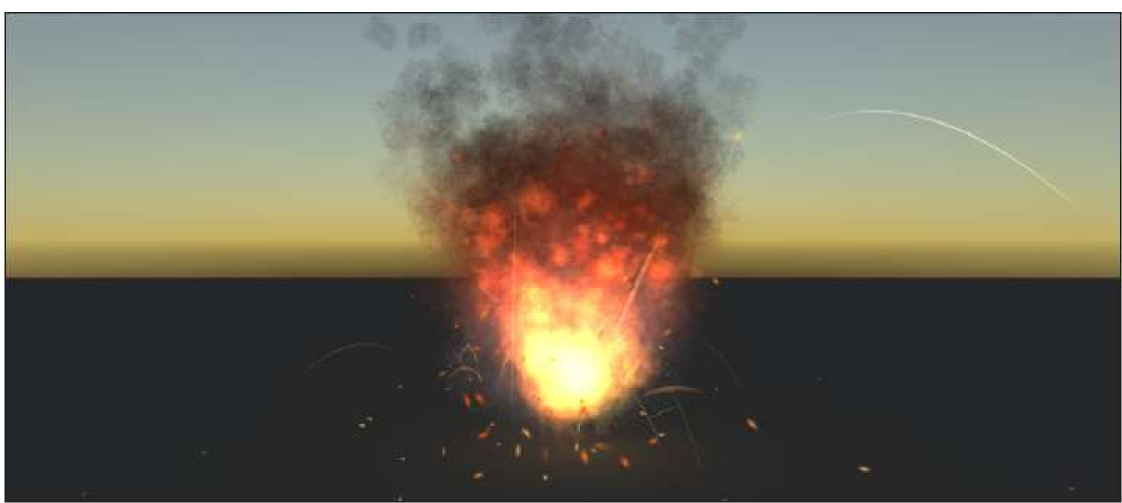


Fig. 13.42

Vamos a crear el fuego desde cero así que debes importar el paquete con el nombre Assets_Escena_Capitulo_13.unitypackage de la carpeta del capitulo, si no lo has hecho todavía. Primero de todo crea una escena nueva y guárdala en la carpeta escenas, en este caso yo la he llamado Proyecto.

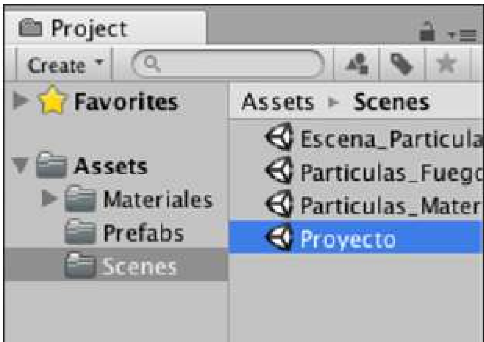


Fig. 13.43

Para el escenario vamos a crear un plano; accede al menú principal **GameObject > 3D Objects > Plane.** Se creará un objeto plano al que le pondremos el nombre de suelo y lo posicionaremos en el centro de la escena. En el ejemplo yo he cambiado la escala en 10 y en la posición en el eje y le he dado un valor de 0,5.

Fig. 13.44

Para finalizar el suelo, le pondremos un material. Accede a la carpeta Materiales de la ventana Project, seleccionamos el material Suelo2 y lo arrastramos encima del plano.

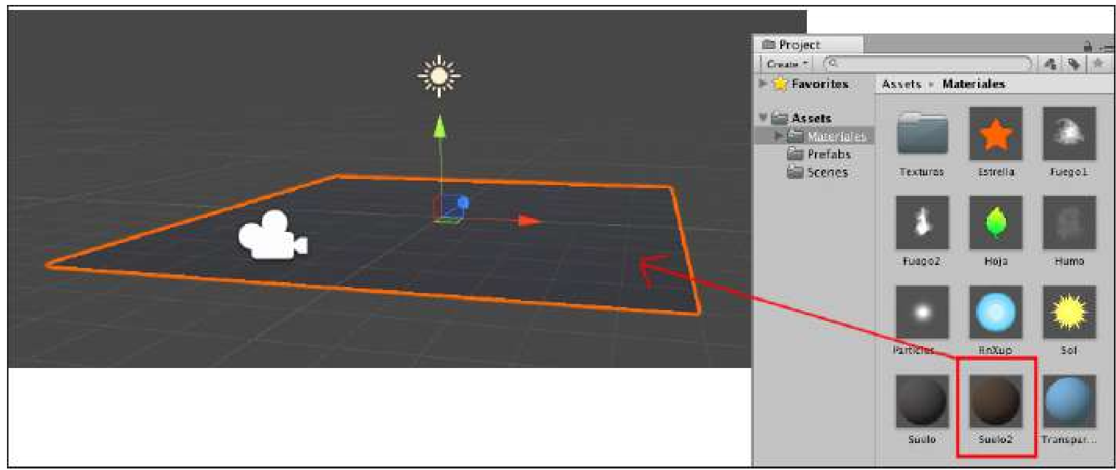


Fig. 13.45

Creación del objeto base

Ahora vamos a crear un objeto vacío que va a contener nuestro primer sistema de partículas. Dentro de la ventana Hierarchy, nos aseguramos de no tener ningún objeto seleccionado y pinchando encima del menú Create > Create Empty. Este objeto vacío va ha ser el contenedor de nuestros sistemas de partículas, en este caso lo he renombrado por PS_Fuego y lo he posicionado en el centro de la escena.

Fig. 13.46

Ahora dentro de la ventana Hierarchy y con el objeto PS_Fuego seleccionado pulsamos en el menú Create > Effecs > ParticleSystem. Este sistema de partículas debe de ser hijo del objeto PS_Fuego. Este sistema de partículas le pondremos el nombre de LlamaFuego01 y le daremos un valor de 0 en todos sus ejes de posición y en el de rotación en este caso tiene el valor -90 para que proyecte las partículas hacia arriba.

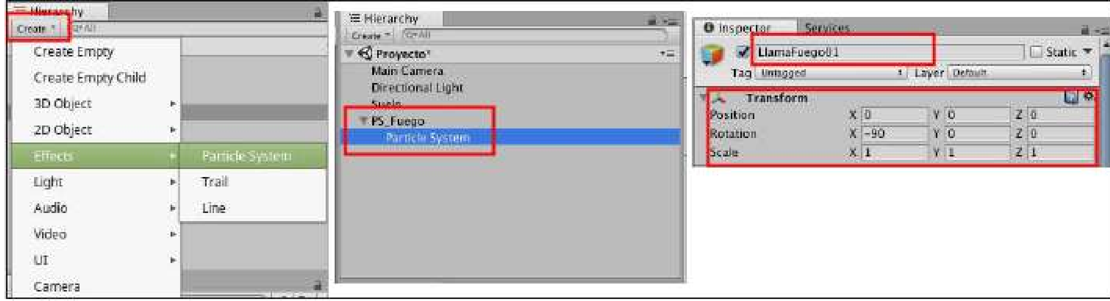


Fig. 13.47

Creación del material para LlamaFuego01

Ahora vamos a crear el material para este primer sistema de partículas que hemos llamado LlamaFuego01 y que es hijo de un objeto vacío que hemos llamado PS_Fuego. Nos dirigimos a la carpeta materiales dentro de la ventana Project y pulsamos en Create > Material. Le ponemos un nuevo nombre al New Material que se a creado por el de LlamasFuego01.

Ahora con el material renombrado y seleccionado accedemos a la ventana Inspector y en el menú Standard seleccionamos el tipo de material Particle > Additive.

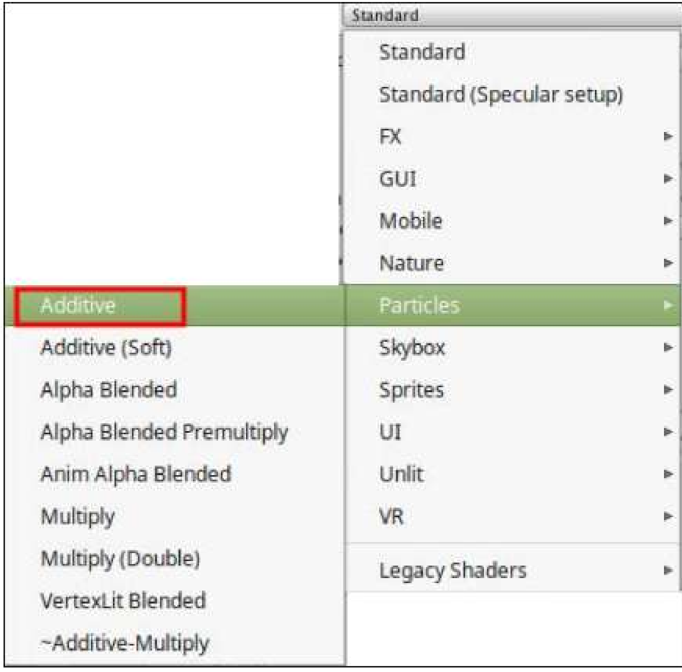


Fig. 13.48

Dentro de estas opciones vamos a agregar una textura para el material. Pulsa en la opción Select y en la ventana que te aparecerá selecciona la textura Fuego01. Estas texturas suelen ser archivos de tipo png que guarda el canal alfa y con fondo negro que Unity en este caso toma como transparencia.

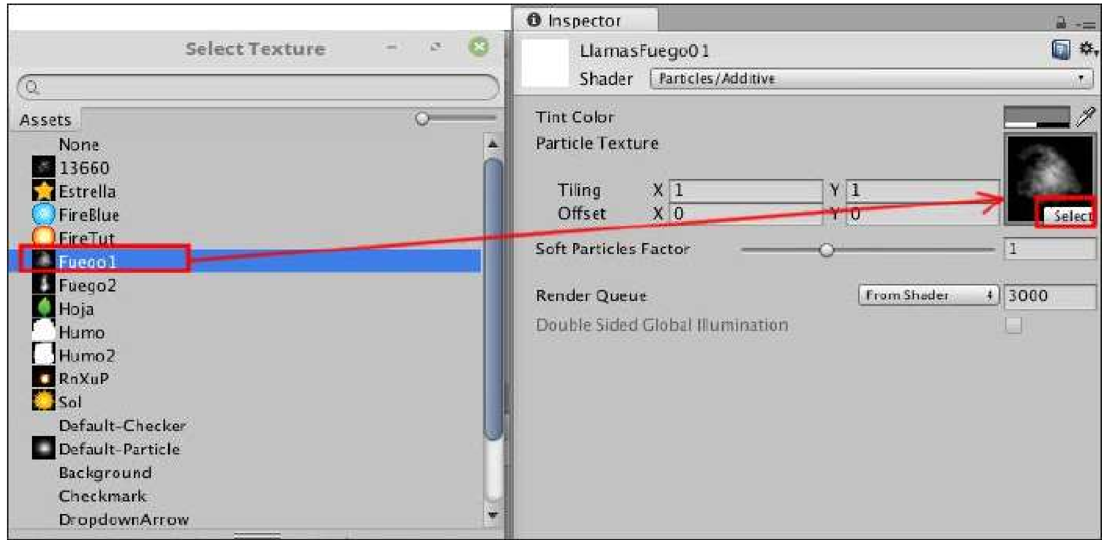


Fig. 13.49

Ahora selecciona el sistema de particular LlamaFuego01 y añádele este material. El icono del material cuando es de tipo ParticlesAdditive suele tomar la forma de la textura que le hemos puesto.

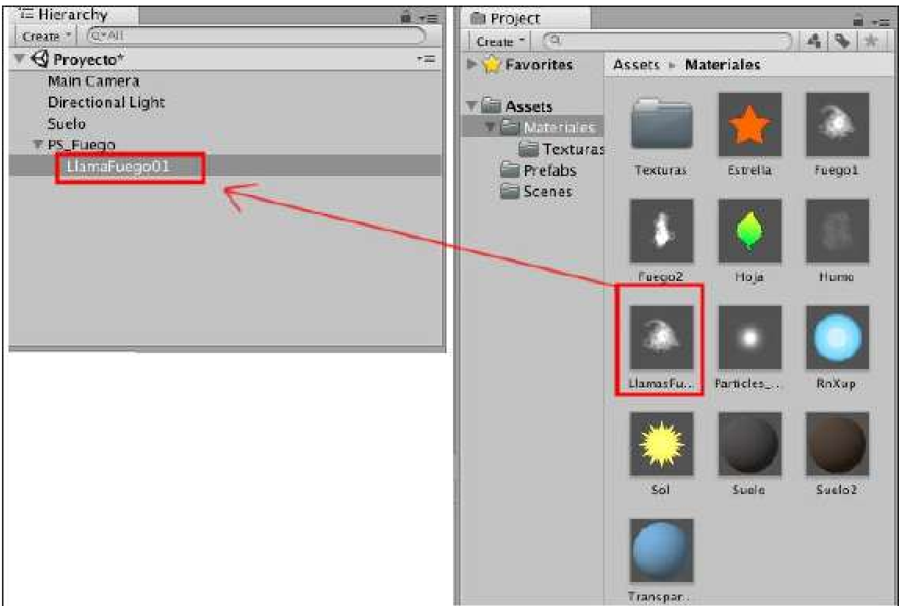


Fig. 13.50

Ahora si vemos en la ventana escena como queda nuestro sistema de partículas se vera de la siguiente forma.

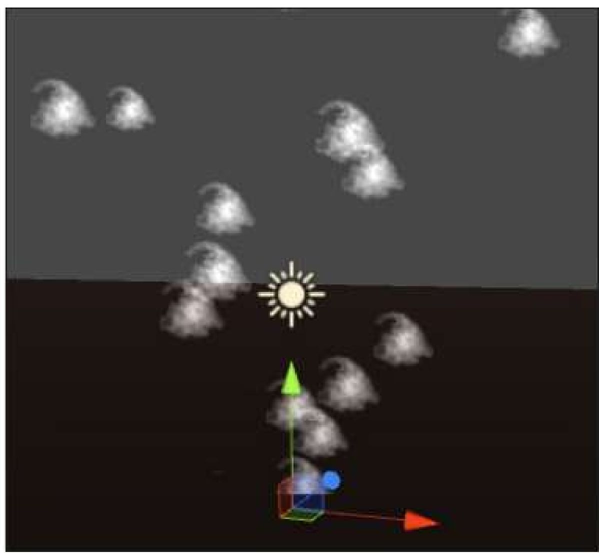


Fig. 13.51

Propiedades del sistema LlamaFuego01

En las opciones generales vamos a tener activada la opción Looping.

En el parametro Start Lifetime vamos a seleccionar la opción Random Between Two Constants que encontraremos accediendo a la flecha que tiene al lado. Le he dado los valores 1 y 2,5.

En el parámetro Start Speed también he seleccionado la opción Random Between Two Constants y le he dado los valores 1 y 2,5.

En el parámetro Start Size seleccionamos la opción Random Between Two Constants y le he dado los valores 0,1 y 2.

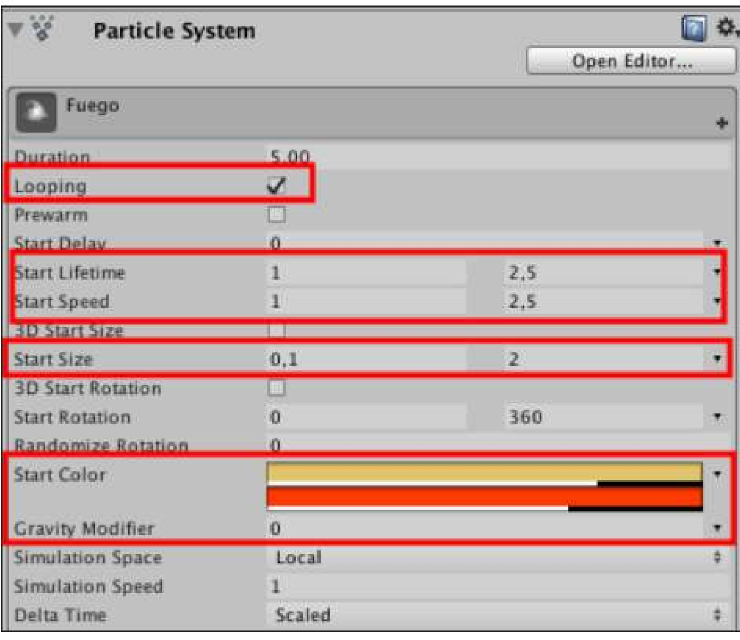


Fig. 13.52

El parámetro Start Color seleccionamos la opción Random Between Two Colors. En este caso puedes poner el color que más te guste en este caso en concreto utilizo el amarillo y rojo cercanos al naranja para recrear el color del fuego. A continuación nuestro sistema se vera de la siguiente manera.

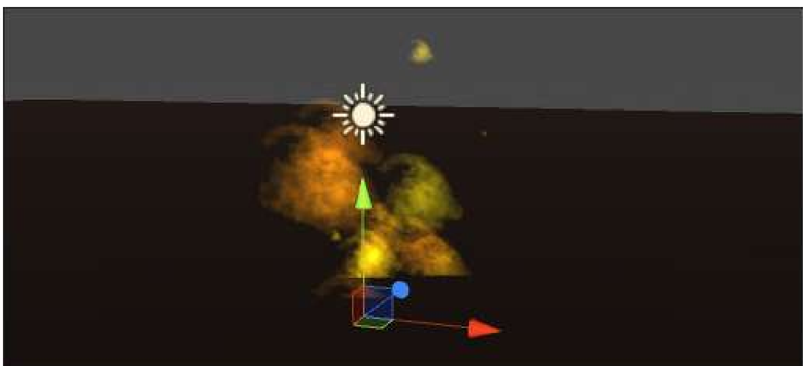


Fig. 13.53

En el modulo **Emission** vamos a ponerle el valor 100 en el parámetro **Rate over Time** de este modo aumentaremos la densidad de las partículas y tendremos un efecto más parecido al del fuego.

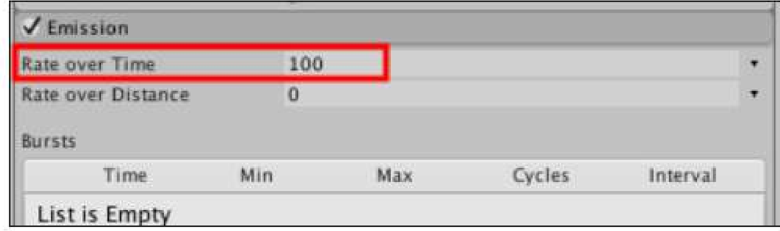


Fig. 13.54

En el modulo Shape solamente he cambiado el parámetro Radius con un valor de 0,35. Esto provoca que la continua salida de partículas se concentre un poco más. El resto de parámetros los he dejado con los valores por defecto.

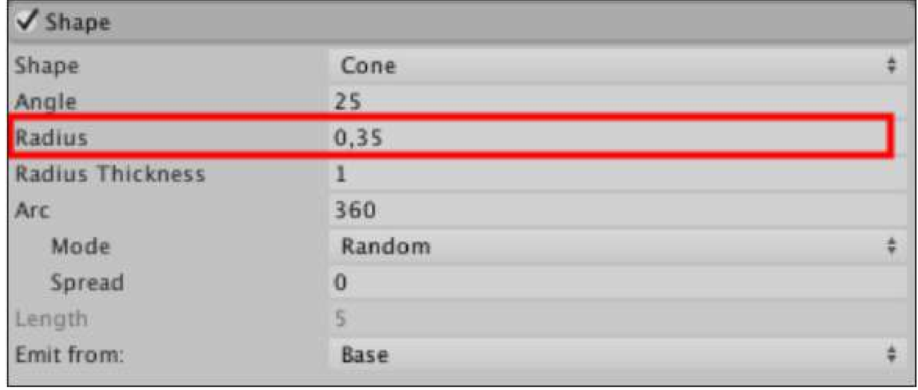


Fig. 13.55

El siguiente modulo que debemos activar es el Color over Lifetime al que podemos mediante un degradado de color podemos mejorar el aspecto de nuestro sistema de partículas. A continuación te muestro el degradado que he creado en el caso de que quieras representarlo.

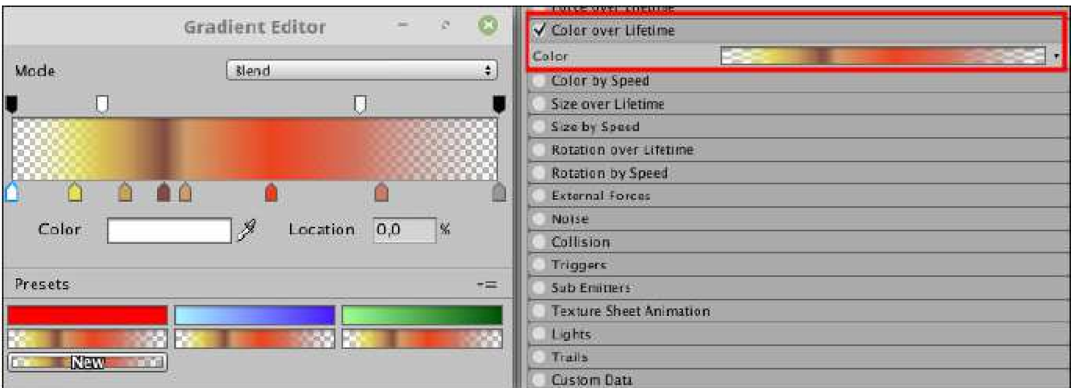


Fig. 13.56

El siguiente módulo a activar es el Size over Lifetime. En este parámetro haz clic encima de la curva gráfica y en la parte inferior de la ventana Inspector selecciona la curva que desciende como la que te muestro en la siguiente imagen.

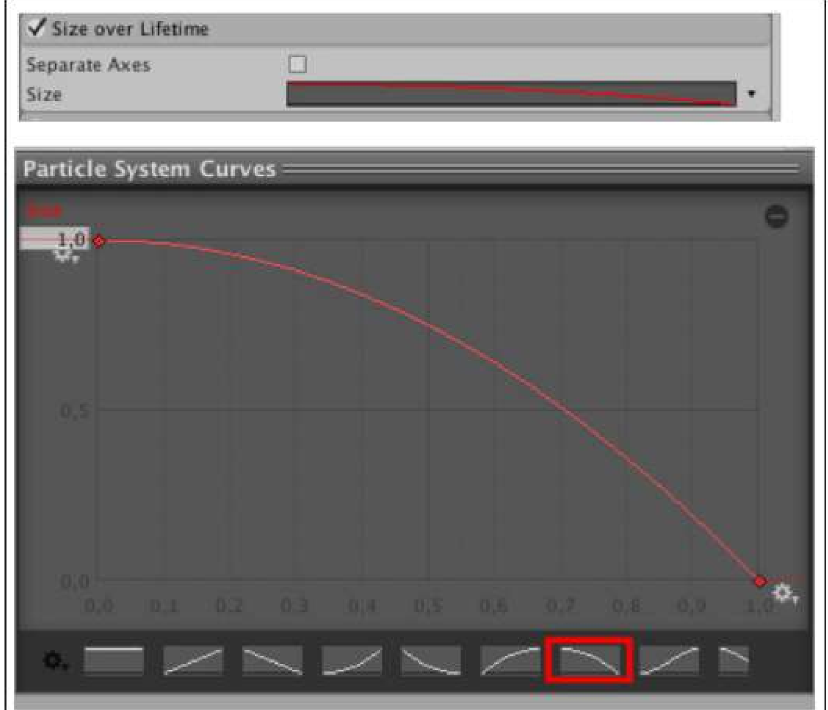


Fig. 13.57

Ahora activamos el modulo Rotation over Lifetime y le damos un valor de 15 en el parámetro Angular Velocity. Nuestro sistema debería verse como te muestro a continuación.

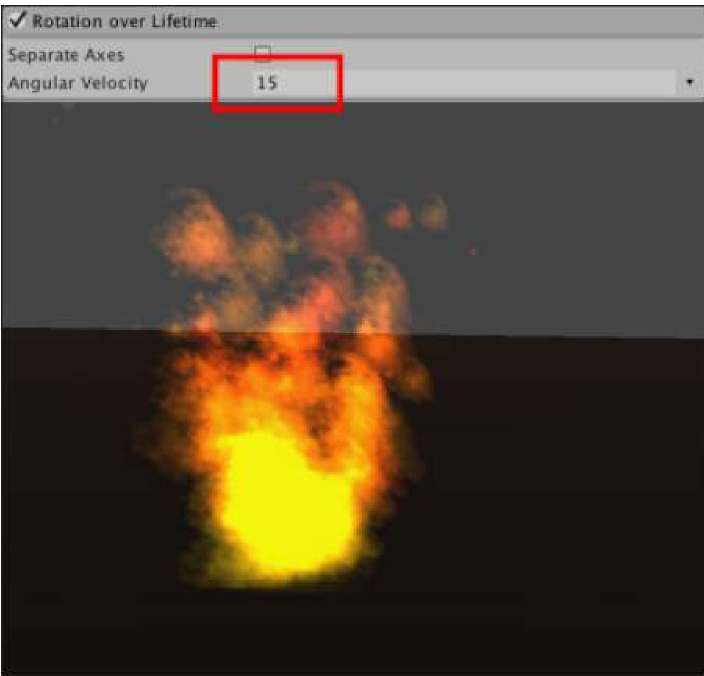


Fig. 13.58

El modulo Lights es un modulo que nos permite agregar luz a nuestras partículas, para no complicar el proyecto en la ventana Project > Prefabs tienes un prefab que es una luz que utilizaremos este modulo en concreto. Arrastra el prefab Luz de Partícula hacia la ventana Hierarchy para que se posicione por defecto.

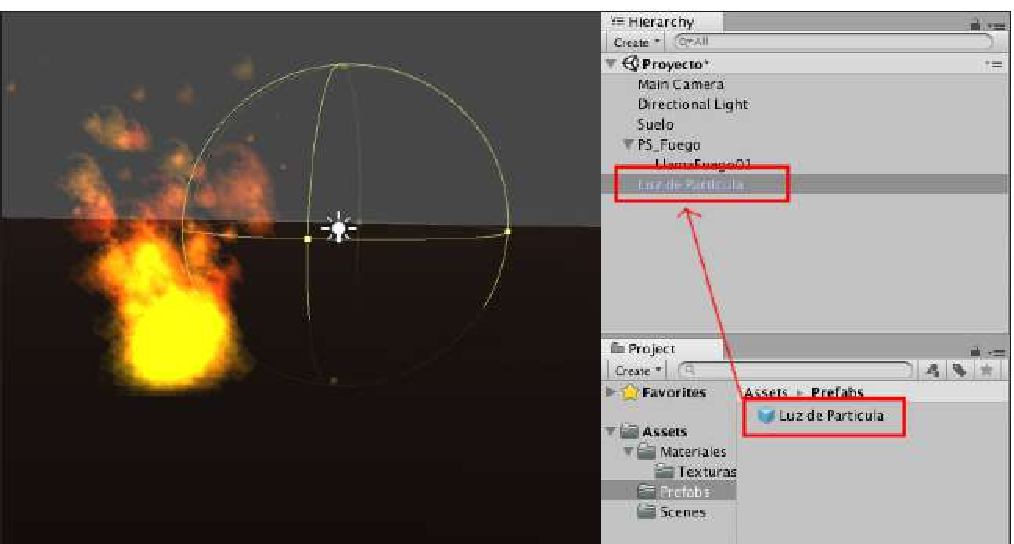


Fig. 13.59

Ahora seleccionamos nuestro sistema de partículas LlamaFuego01 y accedemos al modulo Lights para configurar los siguientes parámetros. El primero de los parámetros es Light a la cual debemos seleccionar el prefab que acabamos de poner en escena, es decir la luz de Particula.

El siguiente cambio es en el parámetro Range Multiplier en donde seleccionamos la opción Random Between Two Constants y le he dado los valores 0,8 y 2. Para el Intensity Multiplier también seleccionamos Random Between Two Constants y le he dado los valores 0,4 y 1.

Para finalizar en el parámetro Máximum Lights le damos un valor de 8. Si lo prefieres puedes aumentar este valor para que la iluminación sea más visual.

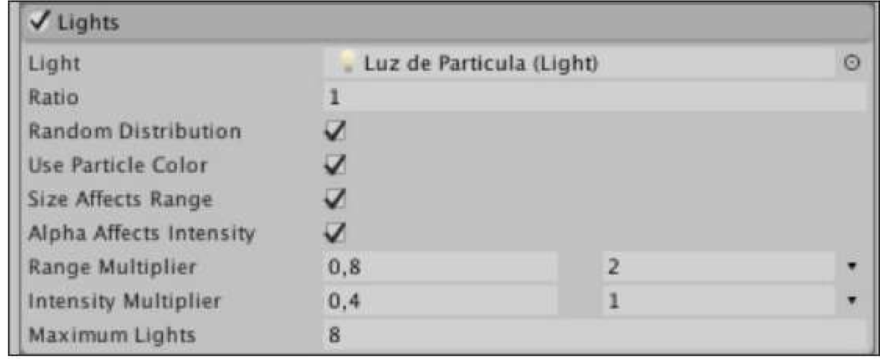


Fig. 13.60

Creación y Propiedades del sistema LlamaFuego02

Ahora vamos a crear otro tipo de llama de fuego pero vamos a reutilizar el que tenemos hecho. Para ello duplicamos el sistema de partículas LlamaFuego01, seleccionándolo primero y con el botón derecho del ratón seleccionamos la opción Duplicate.

Tenemos otro sistema de partículas completamente igual, así que vamos a cambiar primero el nombre por LlamaFuego02 y luego veremos sus propiedades.

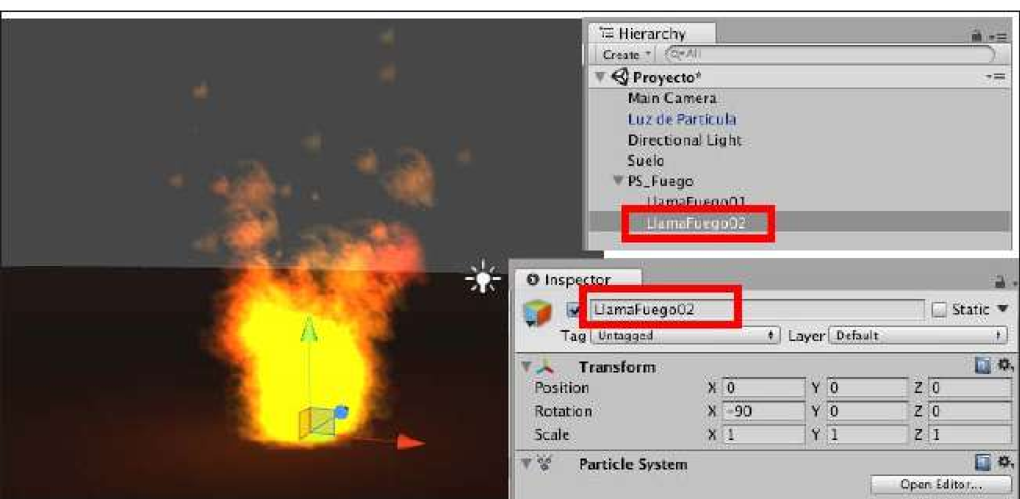


Fig. 13.61

En los módulos de este segundo sistema de particular amos a desactivar primero el modulo Lights , porque solamente quiero que se ilumine las partículas del primer sistema de partículas.

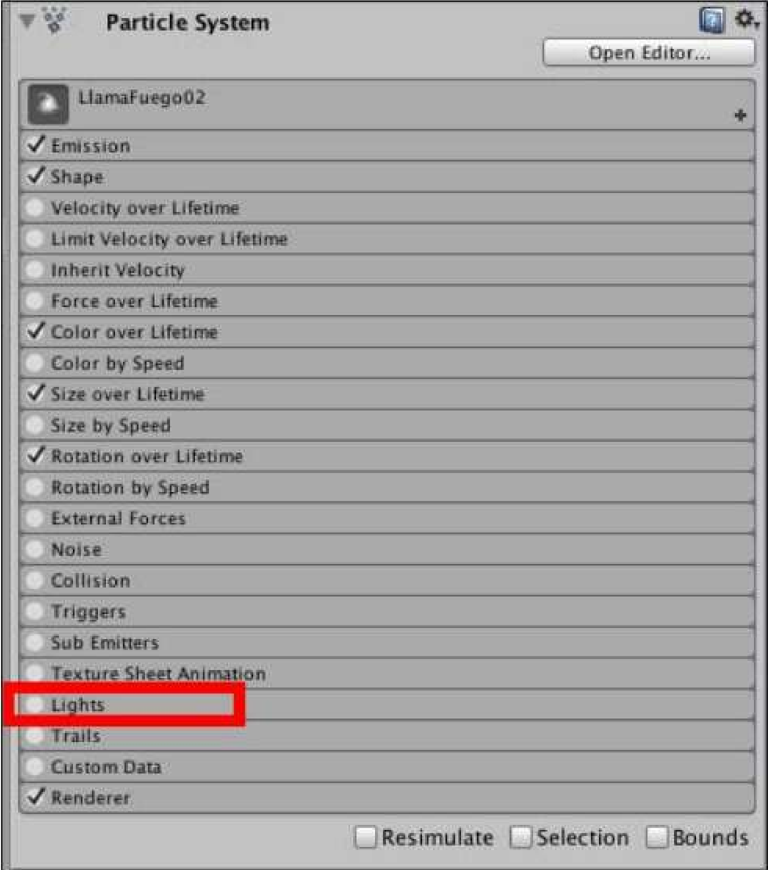


Fig. 13.62

Otro de los aspectos importantes a cambiar es que la textura sea una imagen diferente a la del primer sistema. Si vas a la ventana Project > Materiales antes hemos creado un material LlamasFuego01, ahora vamos a crear otro material Particles Additive exactamente igual como lo hemos hecho anteriormente pero en este caso le vamos a poner como nombre LlamasFuego02 y la textura Fuego2.

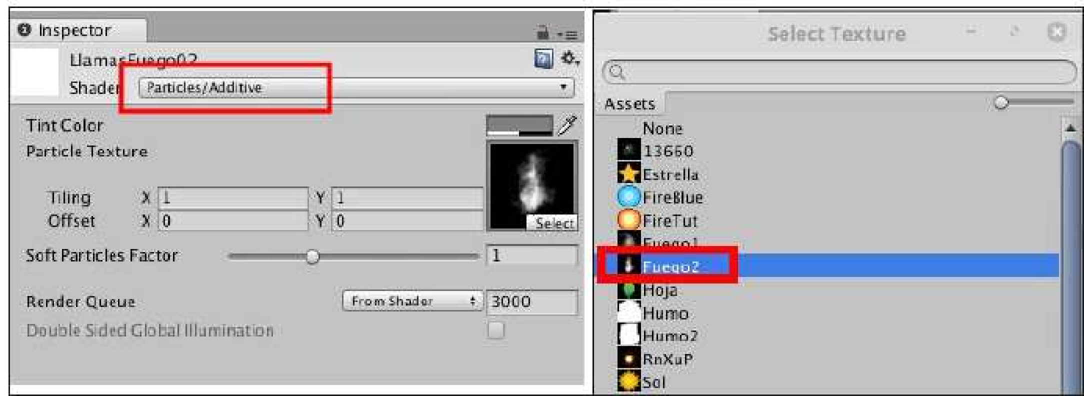


Fig. 13.63

Luego arrastramos el material LlamasFuego02 encima del objeto LlamasFuego02 para que tenga efecto.

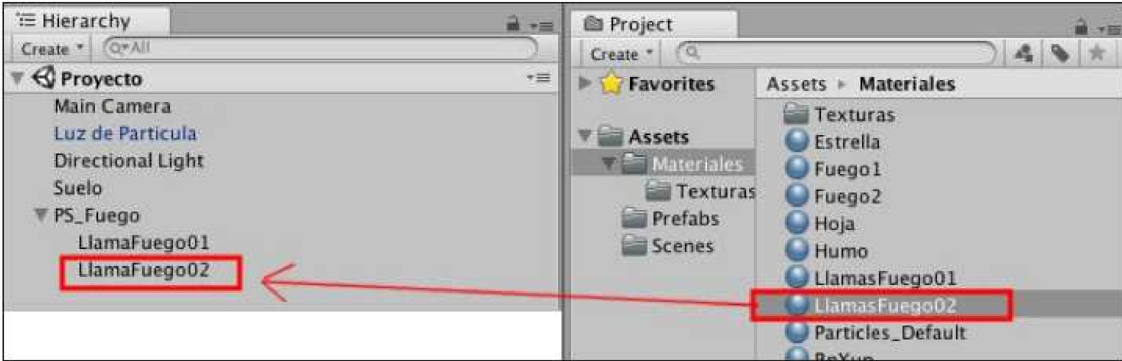


Fig. 13.64

Para las propiedades del sistema de partículas LlamaFuego02 vamos a cambiar los siguientes parámetros;

Del modulo General debemos cambiar los valores de Start Lifetime: 1 y 3, Start Speed:1 y 2,5, Start Size; 0,01 y 1. En este caso en Start Rotation activamos la opción Random Between Two Constants y le damos los valores 0 y 360.

Fig. 13.65

En el parámetro Start Color puedes apreciar que debajo de los colores existen unas lineas blancas y negras. Estas lineas son la cantidad de opacidad que tiene cada color. En este caso se le ha disminuido la opacidad, para que sean partículas más transparentes.

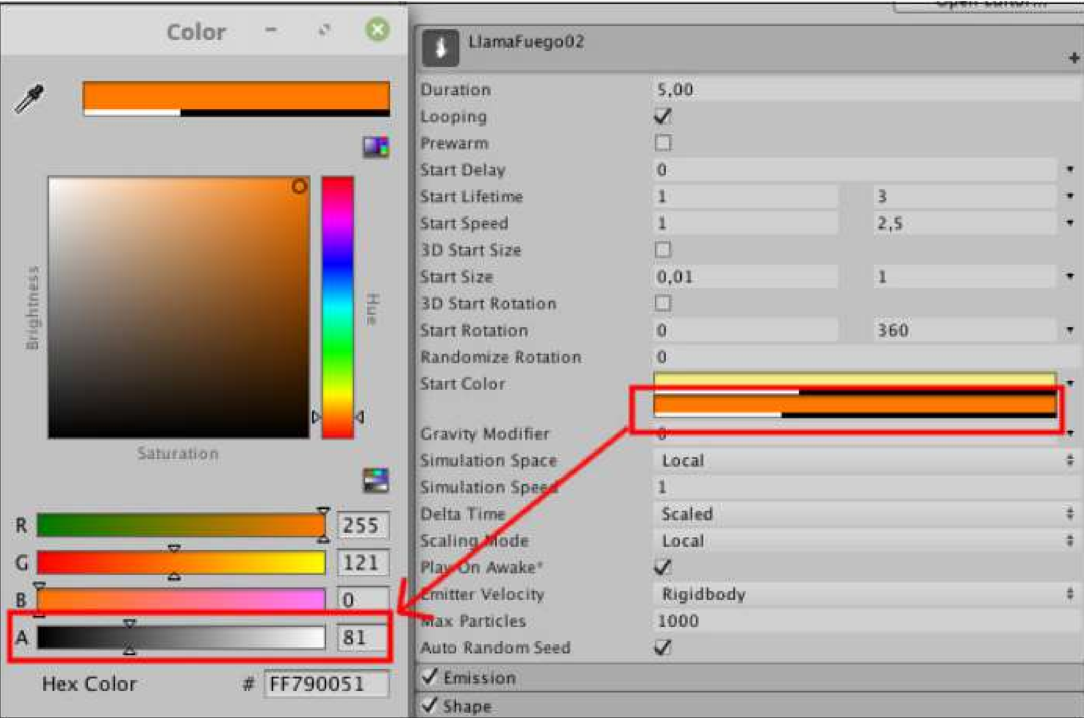


Fig. 13.66

Por ultimo vamos al módulo Shape y cambiamos el radio del cono por el 0,85 o si lo prefieres puedes ponerle un valor superior.

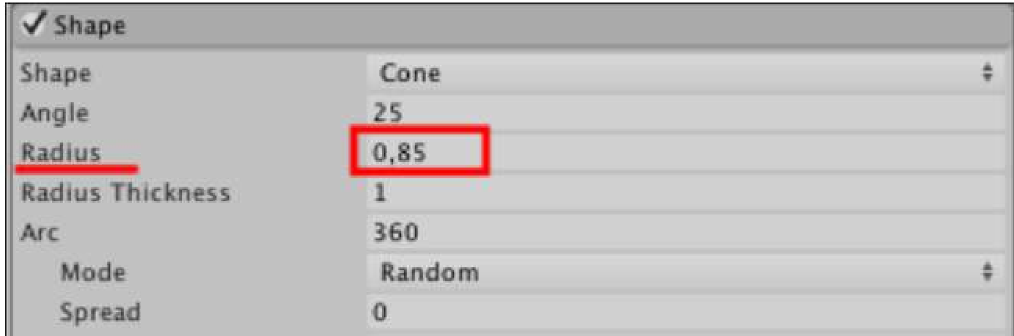


Fig. 13.67

Una vez hecho toda la configuración seleccionamos el sistema LlamaFuego02 en la ventana Hierarchy y lo arrastramos encima del sistema de partículas LlamaFuego01 es decir LlamaFuego02 pasa a ser hijo de LlamaFuego01.

Crear sistema de partículas para el Humo

Ahora vamos a crear un sistema de partículas para recrear el humo del fuego. Empezamos creando un nuevo sistema de partículas como hemos hecho hasta ahora. Este nuevo sistema de partículas será hijo del sistema LlamaFuego01 debajo del sistema LlamaFuego02 como te muestro en la siguiente imagen.

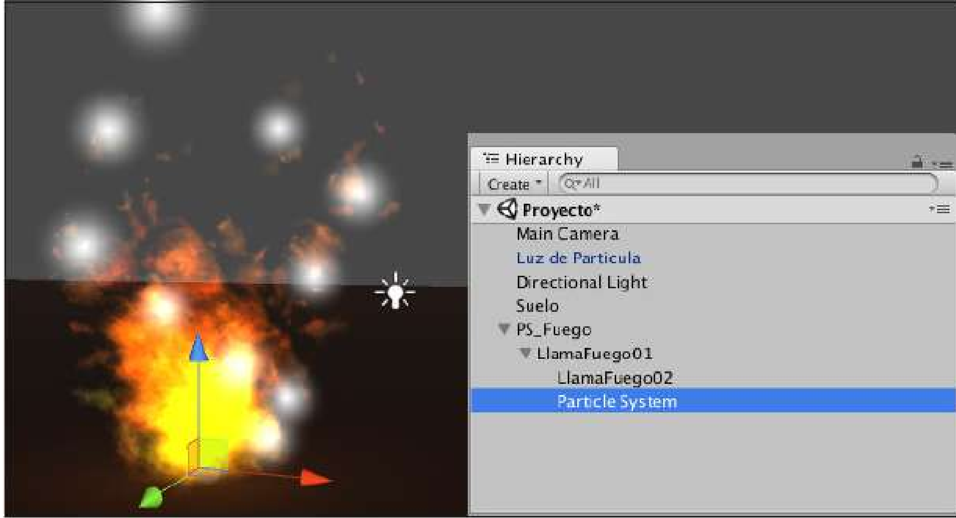


Fig. 13.68

Vamos a cambiarle el nombre de Particle System por el de Humo. El siguiente paso es crear un material para estas partículas. Creamos un material nuevo dentro de la carpeta Materiales con el nombre Humo02, porque si has descargado el material que acompaña el capitulo ya tiene este material creado. En las propiedades del material debemos seleccionar la opción de Particles Alpha Blend.

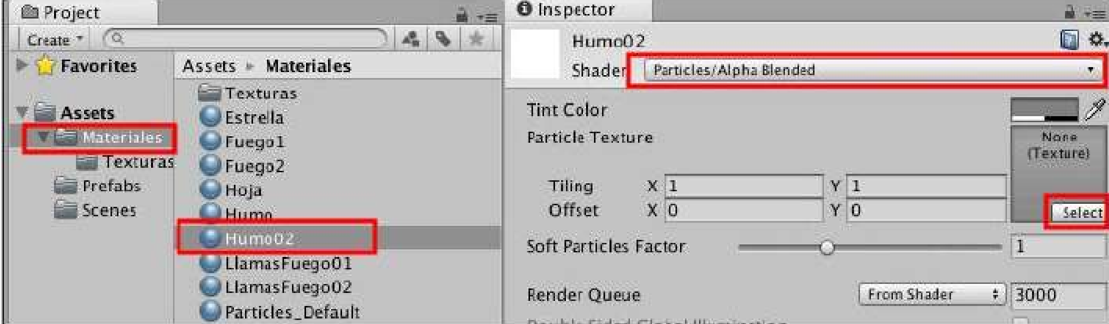


Fig. 13.69

La textura que he utilizado es la que se llama Humo2 y el parámetro Soft Particles Factor a 2,46.

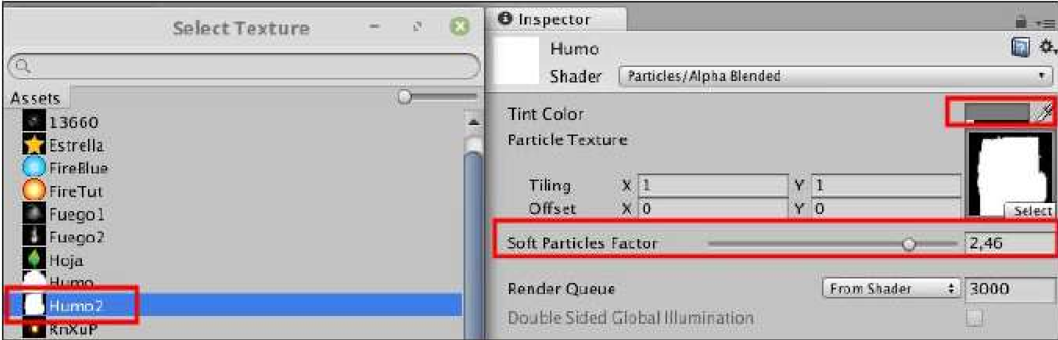


Fig. 13.70

Arrastramos el material encima del sistema de partículas Humo02. Si todo es correcto veras que el sistema toma un aspecto parecido a la que te muestro.

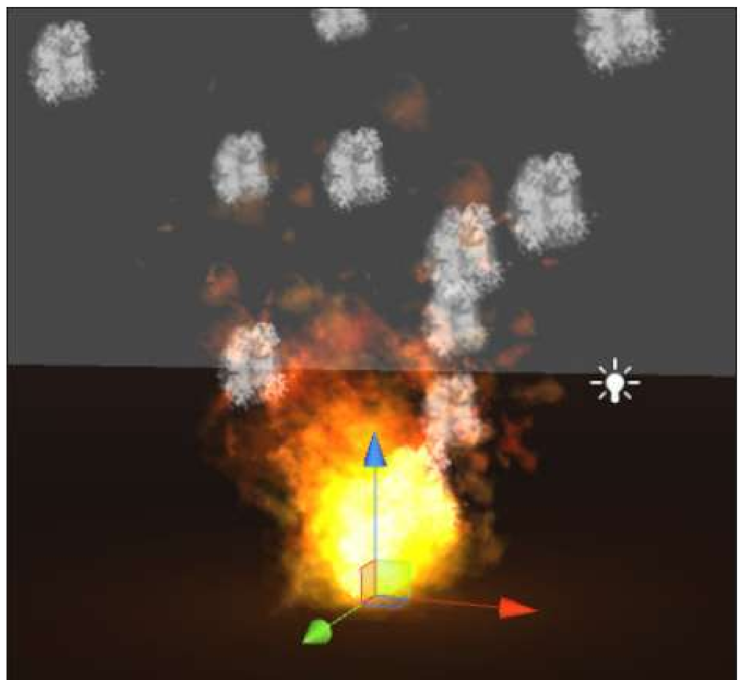


Fig. 13.71

Ahora vamos a las propiedades del sistema de partículas de Humo02 y en el módulo general vamos a cambiar los siguientes parámetros.

- Start Delay= 0,5.
- Start Lifetime = 1 y 4.
- Start Speed= 1 y 2
- Start Size=2 y 4
- Start Rotation = 0 y 360

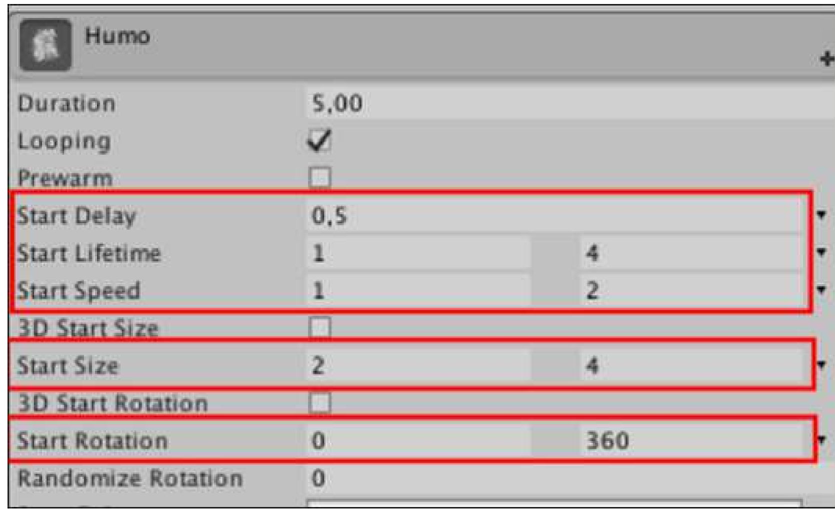


Fig. 13.72

En el módulo Emission en el parámetro Rate over Time le damos un valor de 100 y para el módulo Shape cambiamos el parámetro Radius con el valor de 0,85. Para finalizar la forma de nuestras partículas vamos a activar los módulos Size over Lifetime y Rotation overLifetime dejando sus parámetro por defecto. Nuestro humo debería verse como te muestro a continuación.

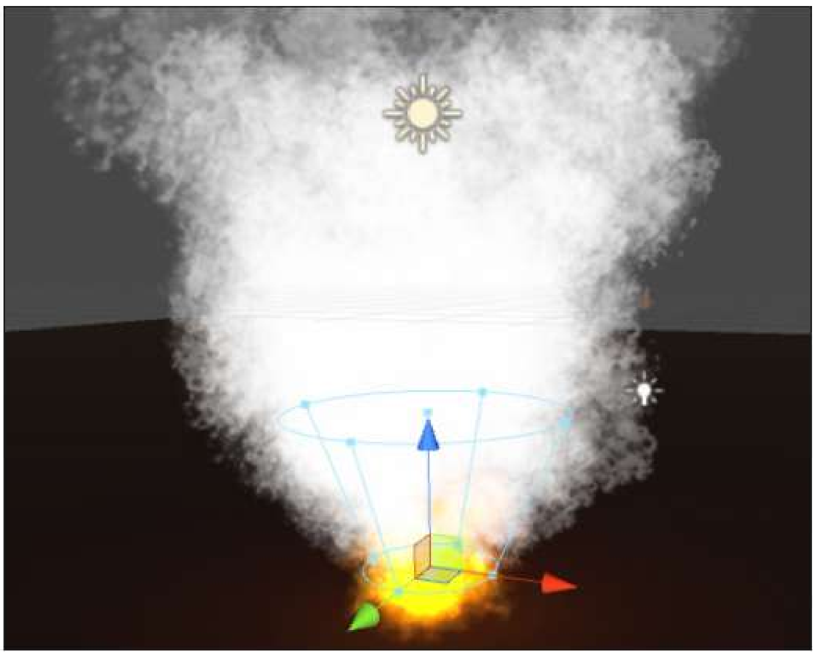


Fig. 13.73

Como puedes ver el humo no tiene un color muy acorde con el tipo de fuego que estamos creando , para ello vamos a activar el módulo Color over Lifetime. En este degradado intenta utilizar colores parecidos a los que te muestro a continuación.

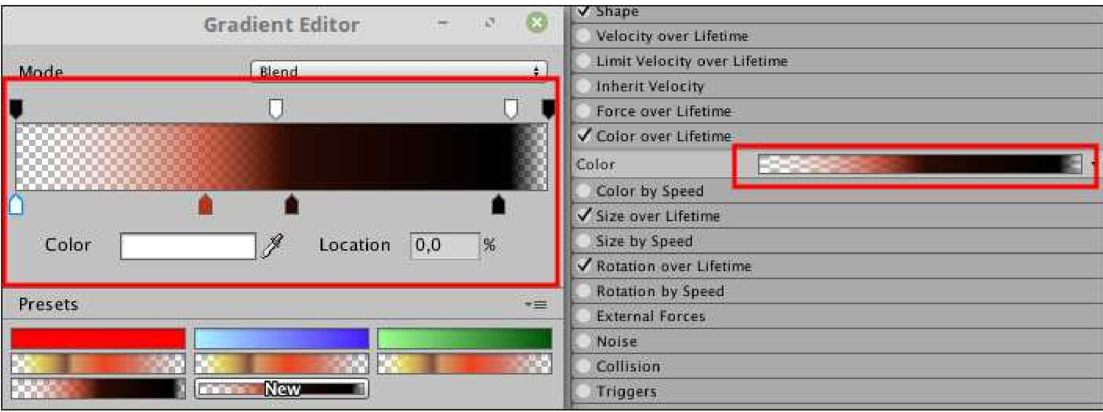


Fig. 13.74

Para que el humo no interfiera con las llamas del fuego este debería posicionarse un poco por encima del fuego como te muestro a continuación.

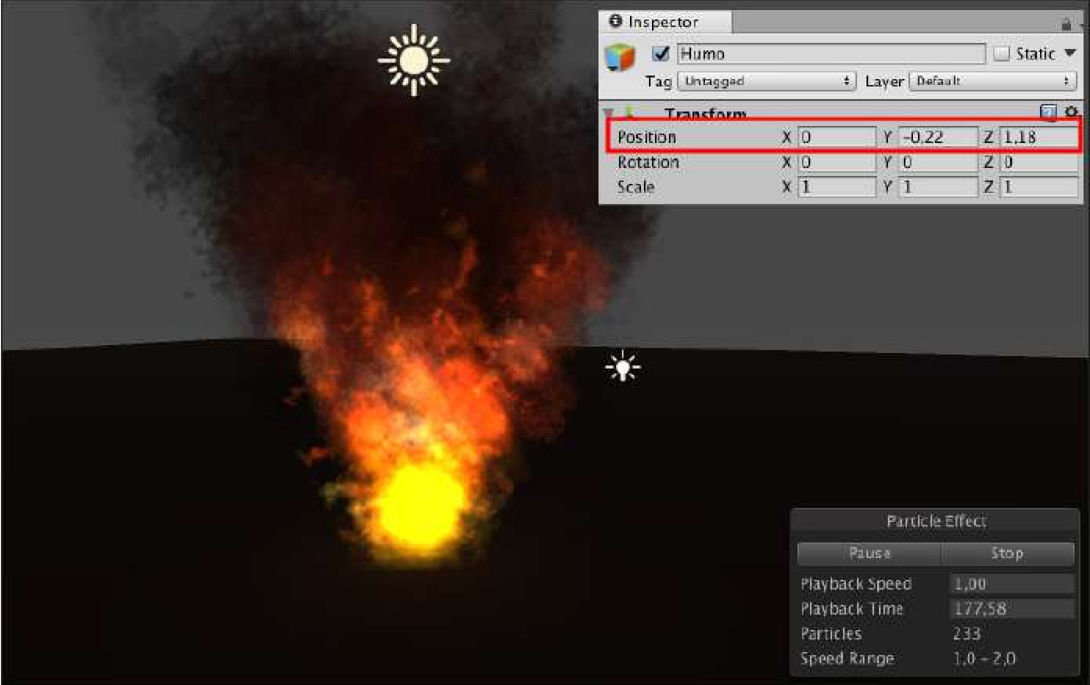


Fig. 13.75

Creación de destellos

Como es lógico cuando tenemos un fuego que quema carbón o madera suelen saltar destellos o chispas. El proceso es siempre el mismo, para no alargar más este proyecto te propongo que intentes crear un nuevo sistema de Partículas con el nombre Destellos e intentes crear esas partículas. Este proyecto lo tienes hecho completamente en el material que acompaña el capítulo.

Si decides probar de realizar este sistema Destellos te doy algunas pistas para ayudarte.

Los módulos que tienes que tener activados son; el Genera, Emission, Shape, Velocity over Lifetime, Color over Lifetime, Size over Lifetime, Collision, Trails y en Render deberemos utilizar dos texturas una para la partícula y otra para la cola del destello.

Este es uno de los capítulos en donde debes investigar y probar todos los parámetros posibles para ver como se comportan las partículas. Intentar mediante texturas, degradados de color emular elementos atmosféricos, de esta manera mejorarás en este sector.

Capítulo 14

Menús y sonido

- Introducción
- Vista general de sonido
- Empezar con el proyecto
- Creación de nuestro player
- Crear proyectiles para nuestro player
- Crear un enemigo
- Colisiones
- Explosiones
- Añadir sistema de puntos y vidas
- Escena principal y Game Over

1. Introducción

En este capitulo se pretende hacer una introducción a los componentes audio y crear un mini proyecto sencillo desde cero para ver como interactúan los menús para pasar de una escena a otra.

Como siempre para seguir correctamente este capitulo debes importar un paquete de assets que te permitirán empezar con la materia directamente. El paquete lo encontraras como siempre en la carpeta de proyectos del capítulo correspondiente con el nombre **Assets_Escena_Capitulo_14.unitypackage.** Recuerda que para importar este paquete debes acceder al menú principal **Assets > Import Package** y en la nueva ventana tener todos los archivos seleccionados y aceptar la importación. A continuación seguir las explicaciones siguientes.

Por otro lado en este capítulo también disponemos de un material adicional para desarrollar un proyecto propuesto desde cero. Este material se divide en tres carpetas una para imágenes, sonido y otra con el nombre UI, que corresponde a la parte en que se desarrolla el proyecto. Este proyecto también dispone de un paquete con nombre **Proyecto_Final.unitypackage**, que contiene el proyecto desarrollado para que puedas consultarlo.

2. Vista general de sonido

El sonido se emite mediante el aire desde un emisor a un receptor, el receptor percibe el sonido según una serie de factores que puede afectar a la transmisión; como la dirección, la distancia, el volumen la calidad del sonido y el entorno en el que se transmite.

Unity para simular efectos de sonido en una posición en concreto requiere de un objeto al que se le conecta fuente de audio (**Audio Source**), que emite sonidos. Estos sonidos son recogidos por otro objeto al que se le añade un audio Oyente (**Audio Listener**). Unity también puede simular la distancia y posición de un **Audio Source**, pero no puede calcular es el entorno de como se transmite el sonido a partir de la geometría de la escena, por el contrario se pueden agregar filtros al audio para emular estos sonidos.

El **Audio Listener** es un componente que se añade desde la ventana Inspector. Normalmente se suele poner en el objeto cámara, aunque se puede poner en cualquier objeto la única condición es que este tipo de componentes solo puede haber uno por escena.

Los **Audio Source** son también un componente que se añade desde la ventana Inspector y en este caso nos proporciona una fuente de sonido.

Vamos a crear un ejemplo sencillo para ir viendo como funciona el **Audio Listener** y los **Audio Source**. Si has importado el paquete **Pruebas_Escena_Capitulo_14.unitypackage** solamente debes ir a la carpeta escenas y hacer doble clic en **Escena1**. La escena solamente tiene un plano de color azul.

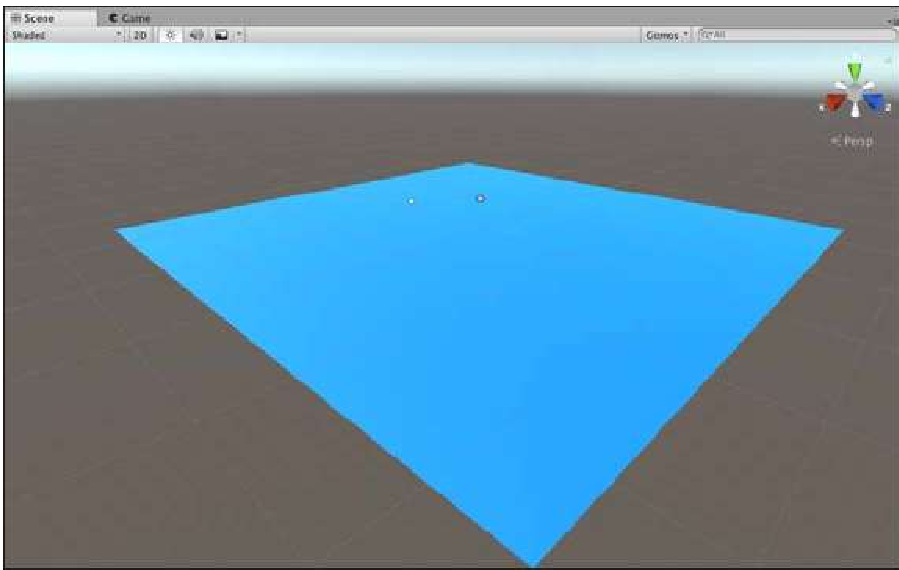


Fig. 14.1

Para empezar crearemos un objeto vacío al que le cambiaremos el nombre y le pondremos el nombre **Musica1**. Antes de añadir ningún audio tenemos que saber que por defecto el objeto **Main Camera** lleva un audio **Listener** por defecto. En una escena solamente debe haber un Audio **Listener**, que va a ser el objeto encargado de escuchar el resto de **Audio Source**.

Ahora seleccionamos nuestro objeto vacío que le hemos puesto nombre **Musica1** y accedemos a la ventana inspector para añadirle un componente **Audio Source** desde el botón **Add Component**.

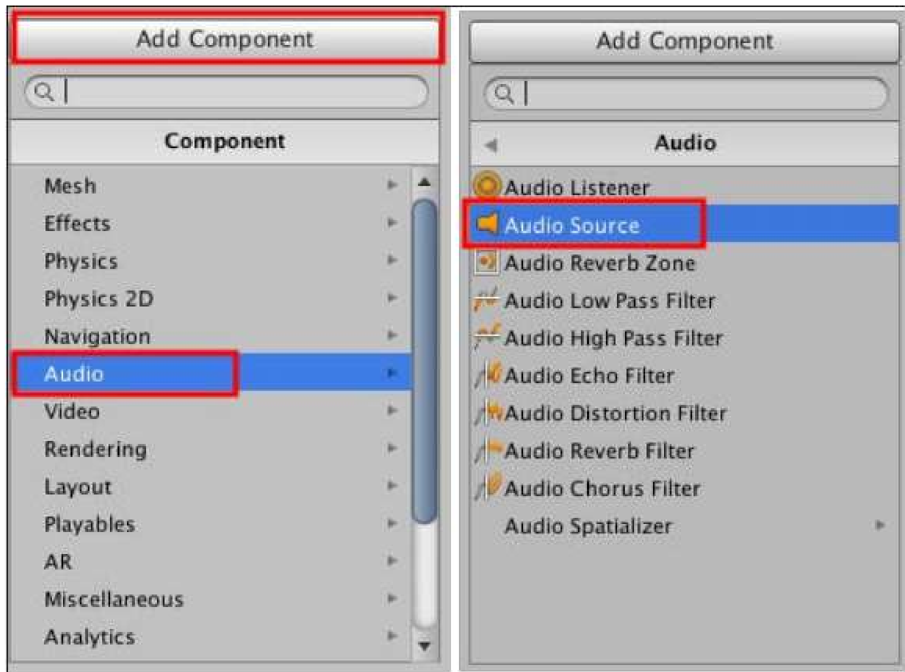


Fig. 14.2

Parámetros del componente Audio Source

Este componente nos permite cargar un clip de audio dentro del objeto y reproducirlo en la escena. Este clip de audio es escuchado por el componente **Audio Listener** que lo lleva la cámara. El componente **Audio Source** puede reproducir clips de audio en 2D, 3D.

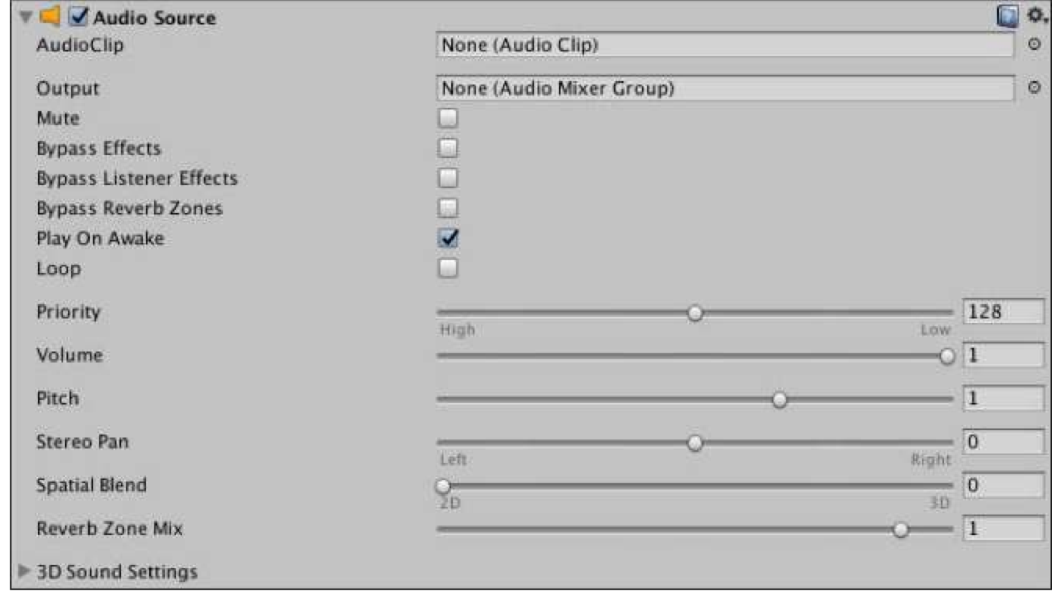


Fig. 14.3

- **Audio Clip**: Este parámetro hace referencia al clip de sonido que queremos utilizar.
- **Output**: Este parámetro nos permite emitir un sonido a través de un **listener** o un **audio mixer**.
- **Mute:** Si esta opción está habilitad, el sonido se reproducirá pero se silenciará.
- **Bypass Effects:** Esta opción es una forma sencilla de activar y desactivar todos los efectos.
- **Bypass Listener Effects: Esta opción** es para *activar* / *desactivar* todos los efectos de **Listener**.
- **Bypass Reverb Zones:** Esta opción es para *activar* / *desactivar* todas las zonas de reverberación rápidamente.
- **Play On Awake: Esta opción** si está habilitada, el sonido comenzará a reproducirse en el momento en que se inicie la escena. Si está deshabilitado, debe iniciarlo utilizando el comando **Play()** desde un script.
- **Loop: Esta opción** si está habilitada, el clip de audio vuelve a empezar cuando este finaliza una y otra vez.
- **Priority: Este** parámetro determina la prioridad de esta fuente de audio entre todas las que coexisten en la escena. Unity recomienda el valor 0 para las pistas de música para evitar que se intercambien en ocasiones. Los valores son prioridad=0 (más importante), prioridad = 256 (menos importante).
- **Volume**: Determina qué tan fuerte está el sonido a una distancia de un metro del **Audio Listener**.
- **Pitch:** Determina la cantidad de cambio en el tono debido a la ralentización / aceleración del clip de audio. El valor 1 es la velocidad de reproducción normal.
- **Stereo Pan**: Establece la posición en el campo estéreo de sonidos 2D.
- **Spatial Blend**: Establece cuánto afecta el motor 3D a la fuente de audio.
- **Reverb Zone Mix**: Establece la cantidad de señal de salida que se enruta a las zonas de reverberación.

- **3D sounds Settings**: Este parámetro engloba un conjunto de opciones con configuraciones que se aplican proporcionalmente al parámetro **Spatial Blend**.

Ahora que hemos visto por encima los parámetros del audio **source** vamos a la ventana **Project** en la carpeta audio y seleccionamos uno de los audios que tenemos y lo arrastramos encima del parámetro **Audio clip**.

Fig. 14.4

Si ejecutas ahora la escena veras como empieza a escucharse el clip de audio. Este clip si no le has activado la opción Loop, finalizará una vez se reproduzca por completo.

Clips de Audio

Ahora ya sabemos como utilizar los clips de audio vamos a ver como Unity gestiona los clips de audio. Unity puede importar varios formatos de audio, yo quiero destacar tres en concreto:

- **Wav:** Pierden muy poca calidad de sonido y ocupan mucho espacio y deberían ser utilizados para sonidos muy concretos y cortos.
- **Ogg:** Para canciones bandas sonoras, para comprimir en este formato se puede utilizar **Audacity** que es un programa gratuito.
- **Mp3:** Utilizarlo siempre que utilicemos plataformas para móviles

Cuando seleccionemos un archivo audio podemos ver la configuración del audio accediendo a la ventana Inspector.

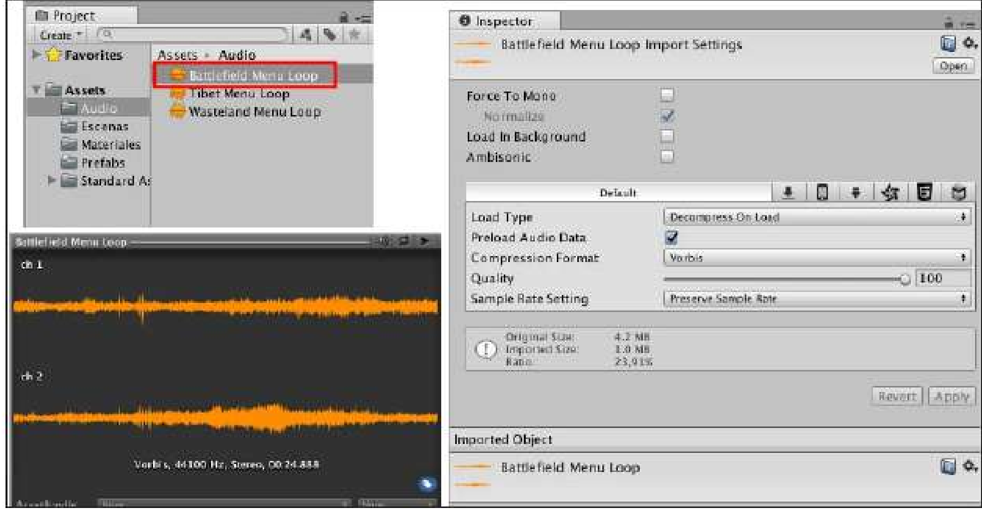


Fig. 14.5

En el inspector podemos ver que las opciones de un clip de audio se dividen en tres apartados concretos. En primer lugar tenemos una serie de opciones , en segundo propiedades de como se gestiona el clip de audio para las diferentes plataformas a las que va destinada, luego disponemos debajo del todo de la ventana Inspector un reproductor de audio en donde podemos reproducir el clip de audio seleccionado.

Opciones

- **Force to Mono**: Cuando esta opción está habilitada, el audio multicanal se mezclará en una pista mono antes de empaquetarse.
- **Normalize**: Cuando esta opción está habilitada, el audio se normalizará durante el proceso de mezcla "Force To Mono".
- **Load In Background:** Cuando esta opción está habilitada, la carga del clip se producirá en un momento diferido en una secuencia separada, sin bloquear el hilo principal.
- **Ambisonic**: Almacena el audio en un formato que representa un campo de sonido que se puede orientar hacia el objeto que lleva el **Audio Listener.** Esta opción se puede utilizar siempre que el clip de audio disponga de audio codificado en **Ambisonic.**

Propiedades

- **Load Type**: Es el método que Unity va a utilizar para cargar un audio.
  - **Decompress On Load:** Los archivos de audio se descomprimirán tan pronto como se carguen. Se recomienda esta opción para sonidos comprimidos de menor tamaño para evitar la sobrecarga de rendimiento al descomprimir sobre la marcha.
  - **Compressed In Memory:** Se mantienen los sonidos comprimidos en la memoria y los descomprime mientras se juega. Esta opción tiene una ligera sobrecarga.
  - **Streaming:** Decodifica sonidos sobre la marcha. Este método utiliza una cantidad mínima de memoria para almacenar datos comprimidos que se leen de forma ascendente en el disco y se decodifican sobre la marcha.
- **Compression Format:** Es el formato especifico que se usará para el sonido en tiempo de ejecución. Tenga en cuenta que las opciones disponibles dependen del objetivo de compilación seleccionado actualmente.
- **Quality:** Determina la cantidad de compresión que se aplicará a un clip comprimido.
- **Sample Rate Setting:** Los formatos de compresión **PCM** y **ADPCM** permiten una reducción de la frecuencia de muestreo automática o optimizada automáticamente.

El sonido 3D

Ahora vamos a abrir las escena 2 desde la carpeta Escenas de la ventana **Project**. En esta nueva escena disponemos de un **FPScontroler** que en este caso es el que lleva el componente **Listener**, y dos cubos de color rojo y verde. Estos cubos tienen un componente Audio **Source** al que se le ha añadido un Audio Clip distinto a cada uno.

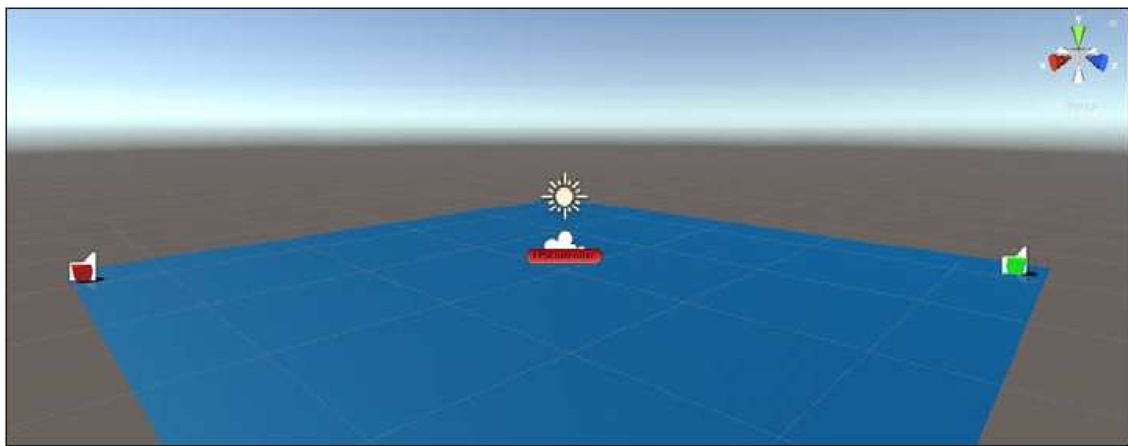


Fig. 14.6

Si ejecutas la escena tal y como está podrás escuchar los dos audio clips a la vez y en este caso lo que pretendemos es crear un audio clip 3d, que nos permita escuchar con nuestro **FPSController** en que dirección viene la música. Para entender mejor este concepto, imagínate que estás en una habitación donde estás escuchando música. Ahora decides salir de la habitación y la música se escucha con un volumen más bajo que cuando estabas dentro.

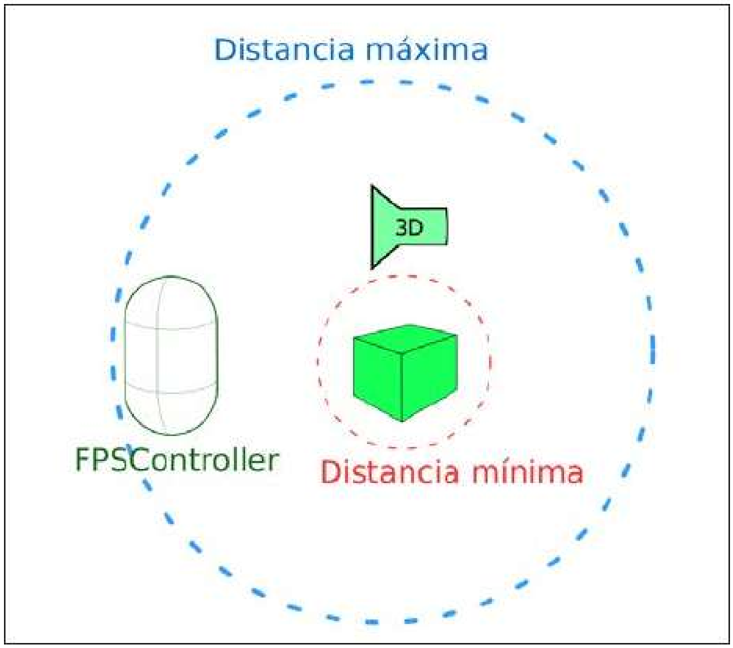


Fig. 14.7

En la imagen anterior te muestro como vamos a configurar los cubos con música. Primero activaremos dentro del componente **Audio Source** en el parámetro **Spatial Blend** que lleva cada cubo desplazaremos el tirador de 2d a 3d o cambiando el valor de **0** a **1**, como te muestro a continuación.

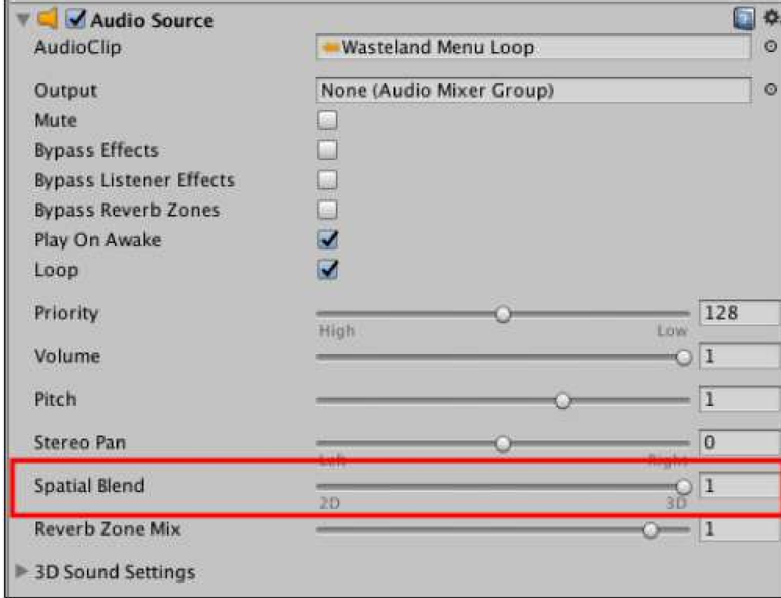


Fig. 14.8

Este parámetro nos permite que el clip de audio tenga una distancia máxima y una distancia mínima. Para configurar estas distancias debemos desplegar las opciones de 3D **Sound Settings** y darle valores a los parámetros **Min Distance** y **Max Distance**. Para este ejemplo introduce en **Max Distance** el valor de 20.

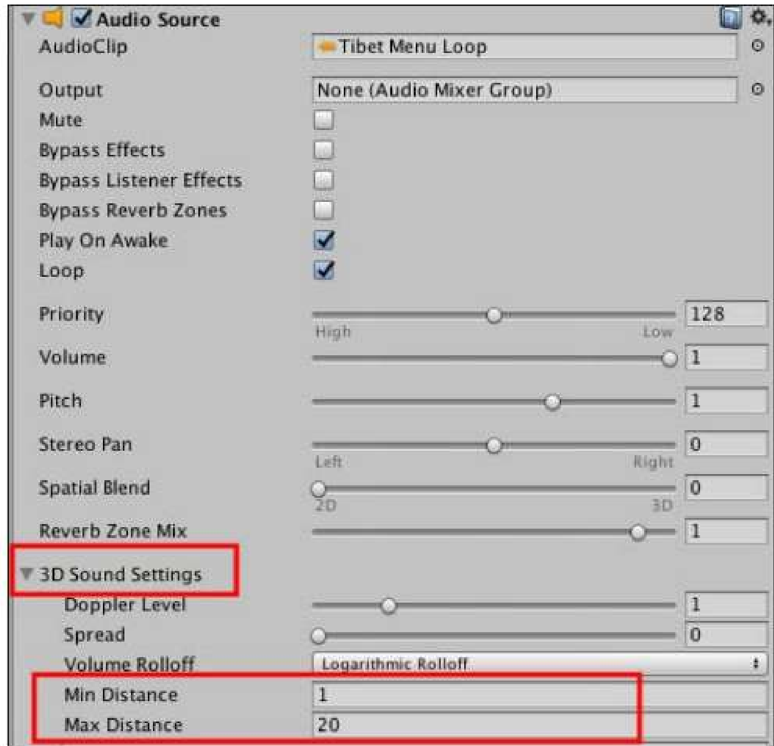


Fig. 14.9

Realiza este cambio en los dos cubos de la escena y ejecuta la escena para comprobar como cuando te acercas con el FPSController a uno de los cubos suena la música de este más fuerte y la del otro cubo deja de escucharse.

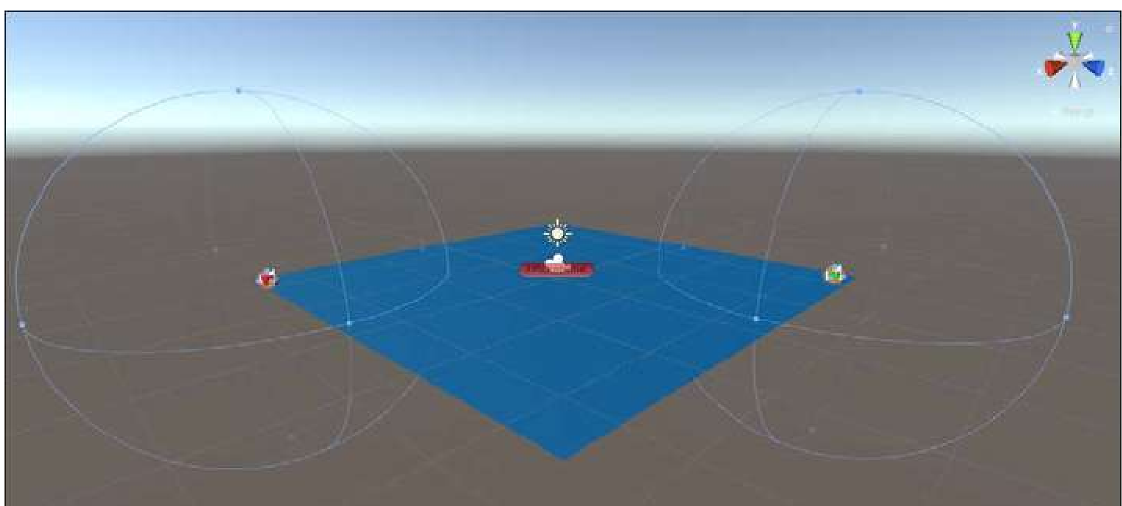


Fig. 14.10

3. Empezar con el proyecto

Este proyecto vamos a crearlo desde 0, para ello primero debemos crear un proyect 3D y le ponemos el nombre que quieras, en este caso lo voy a llamar Proyecto_Final. Una vez arrancado Unity y teniendo claro el esquema del proyecto vamos a crear las carpetas necesarias en la ventana **Project**.

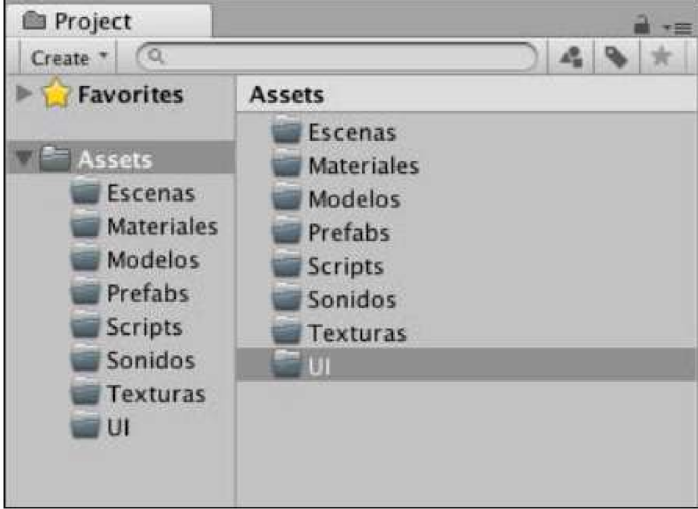


Fig. 14.11

Ahora vamos a crear la primera escena, accediendo al **menú** principal **File > Save Scene** y guardamos con el nombre **Nivel_1** en la carpeta **Escenas.**

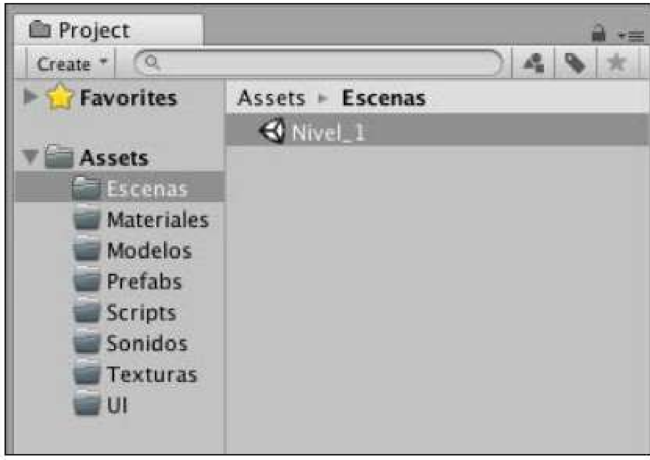


Fig. 14.12

Configurar la cámara es el siguiente paso para ello seleccionamos la cámara **Main Camera** desde la ventana **Hierrarchy** y vamos a configurar los siguientes parámetros desde la ventana **Inspector**.

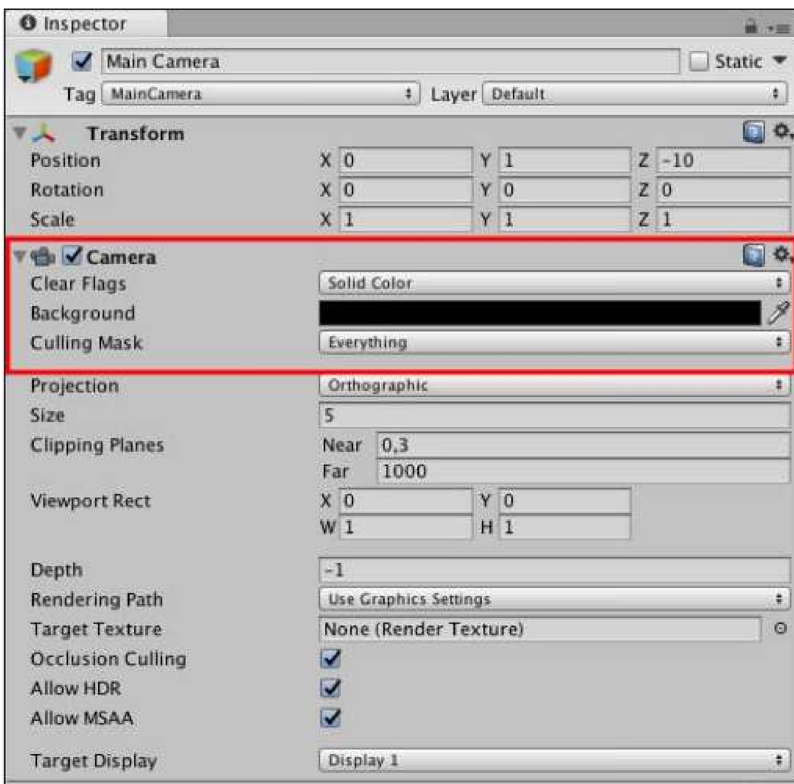


Fig. 14.13

La cámara que tomamos es por defecto la que viene con Unity y los parámetros que cambiamos son el **Clear Flags** que determinamos la opción **Solid Color**, el parámetro **Background** le ponemos el color negro y el parámetro **Projection** seleccionamos la opción **Orthographic**. También vamos a posicionar en el eje **y = 4,45**.

4. Creación de nuestro Player

Crearemos un cubo como player de momento como si estuviéramos creando un prototipo. Accedemos al menú **GameObejct** y seleccionamos la opción 3DObject Cube.

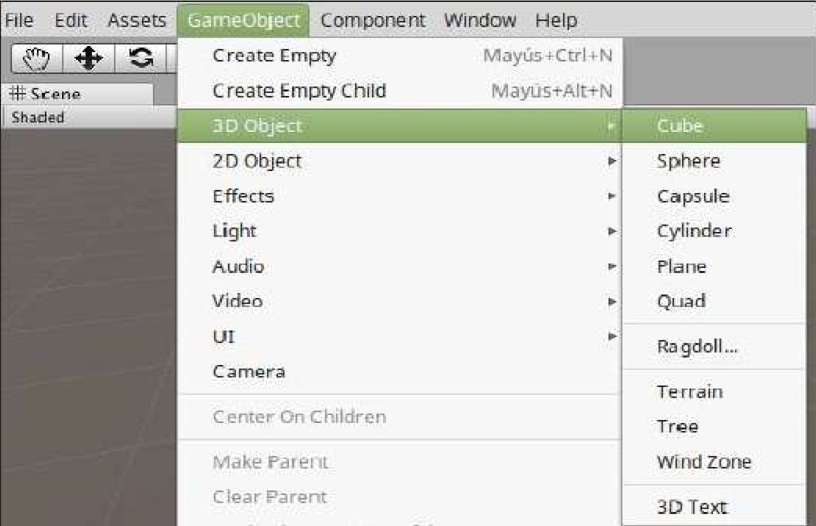


Fig. 14.14

Siempre que creamos un nuevo GameObject vamos a posicionar-lo en el centro de la escena, para ello seleccionamos el cube accedemos a la ventana Inspector pulsamos la rueda dentada y seleccionamos la opción **Reset**.

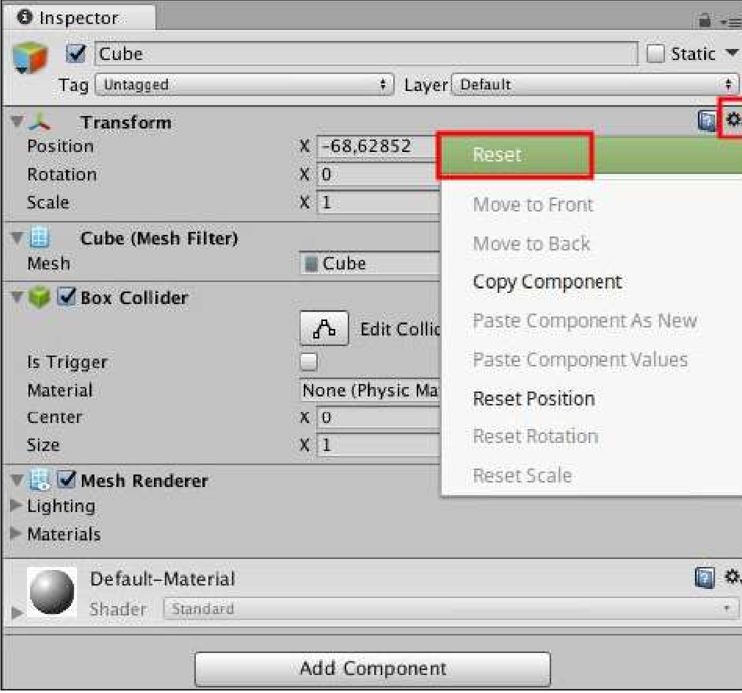


Fig. 14.15

Ahora cambiamos el nombre del cubo por el nombre de Player desde la caja de textos de la parte superior de la ventana Inspector.

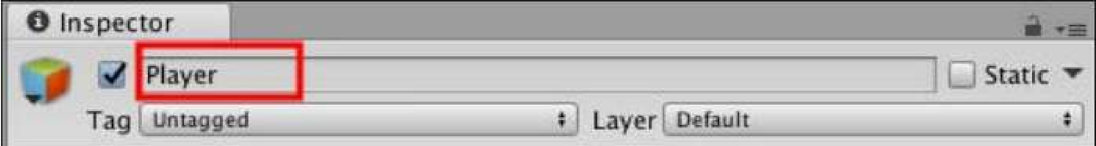


Fig. 14.16

Ahora vamos a crear un material para este Player. En este caso me gusta ponerle colores como habrás comprobado en todo el libro, por que me permite diferenciar bien en el caso de que utilice otros cubos en escena. Para crear un material voy a acceder a la carpeta Materiales que hemos creado dentro de la ventana **Project**. Para crear un material pulso dentro de la carpeta botón derecho del ratón y selecciono la opción **Create > Material.** A este material le pongo el nombre de **Player** y dentro de la ventana **Inspector** le doy un color Rojo.

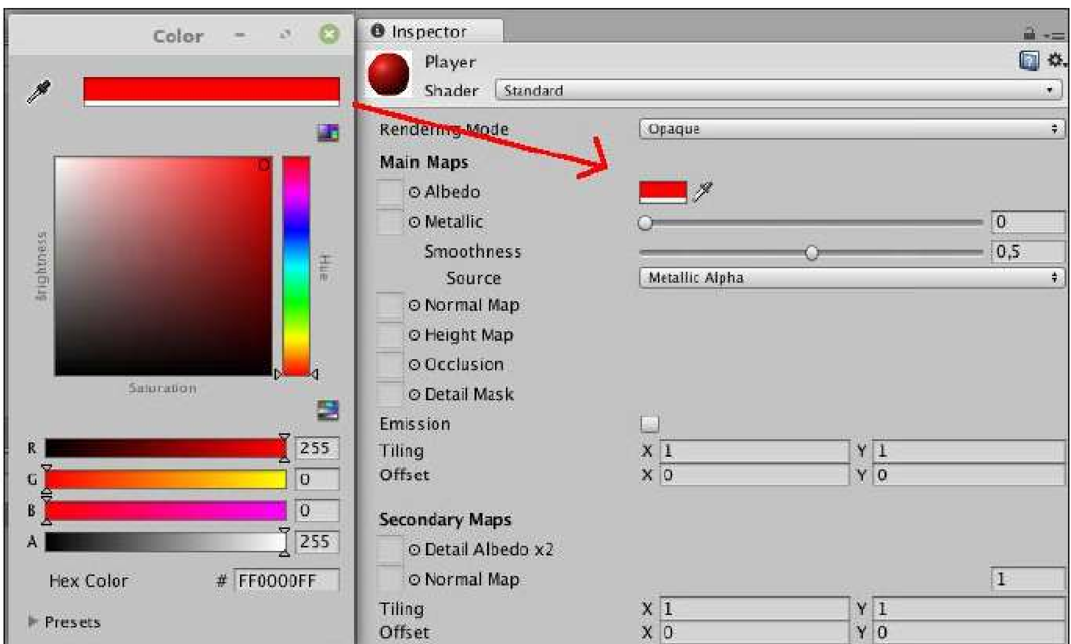


Fig. 14.17

Una vez tenemos el material creado lo arrastramos al objeto Player de la escena. Y nuestro prototipo de player se tintará de color rojo.

Scripting de nuestro player.

Para ello vamos a crear dentro de nuestra carpeta Scripts un **Script C#** dentro de la carpeta hacemos clic botón derecho del ratón y selecciono la opción **Create > C# Script** y a continuación le ponemos el nombre de **Control_Player.** Una vez creado el script lo arrastramos encima del objeto **Player.**

Script: Control_Player.cs

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
public class Control_Player : MonoBehaviour
{       public float playerSpeed;
        public float anchoMovimiento;
        public GameObject miplayer;
        void Awake ()
        {
                this.playerSpeed = 10f;
                this.miplayer = gameObject;
        }
        // Update is called once per frame
        void Update ()
        {
                // El ancho de movimiento
                this.anchoMovimiento = Input.GetAxisRaw ("Horizontal") * this.
playerSpeed * Time.deltaTime;
                // El movimiento del player
                this.miplayer.transform.Translate (Vector3.right * this.
anchoMovimiento);
                //Retorno infinito
                if (transform.position.x <= -7.5f)
                {
                        transform.position = new Vector3 (7.4f, transform.po-
sition.y, transform.position.z);
                }
                else if (transform.position.x >= 7.5f)
                {
                        transform.position= new Vector3(-7.4f,transform.posi-
tion.y, transform.position.z);
                }
        }
}
```

5. Crear proyectiles para nuestro player

Para el proyectil voy a utilizar una capsula. Creamos un objeto Capsule como hemos hecho con el cubo desde el menú **GameObject** > **3D Object** > **Capsule.** El siguiente paso es posicionar en el centro de la escena, escalar el objeto en 0,2 en todos los ejes y cambiarle el nombre por Proyectil.

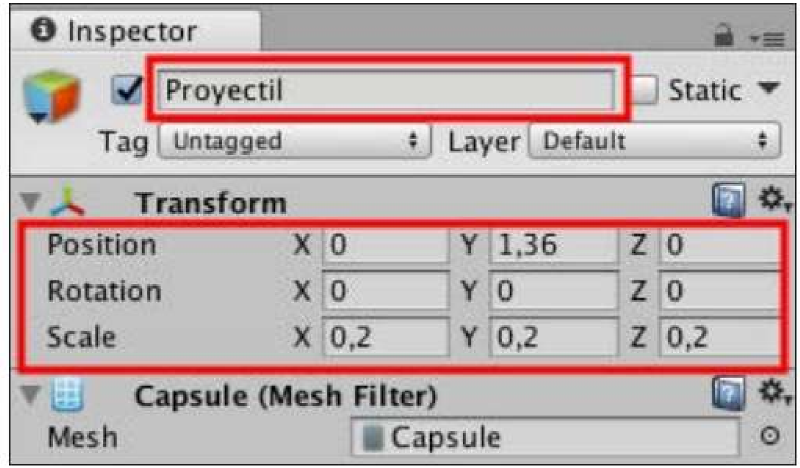


Fig. 14.18

A este objeto le añadimos otro material. Para ello debemos crear un material con nombre **Proyectil** y en el caso del ejemplo le he dado un color amarillo. Una vez creado el material lo arrastramos encima del objeto Proyectil.

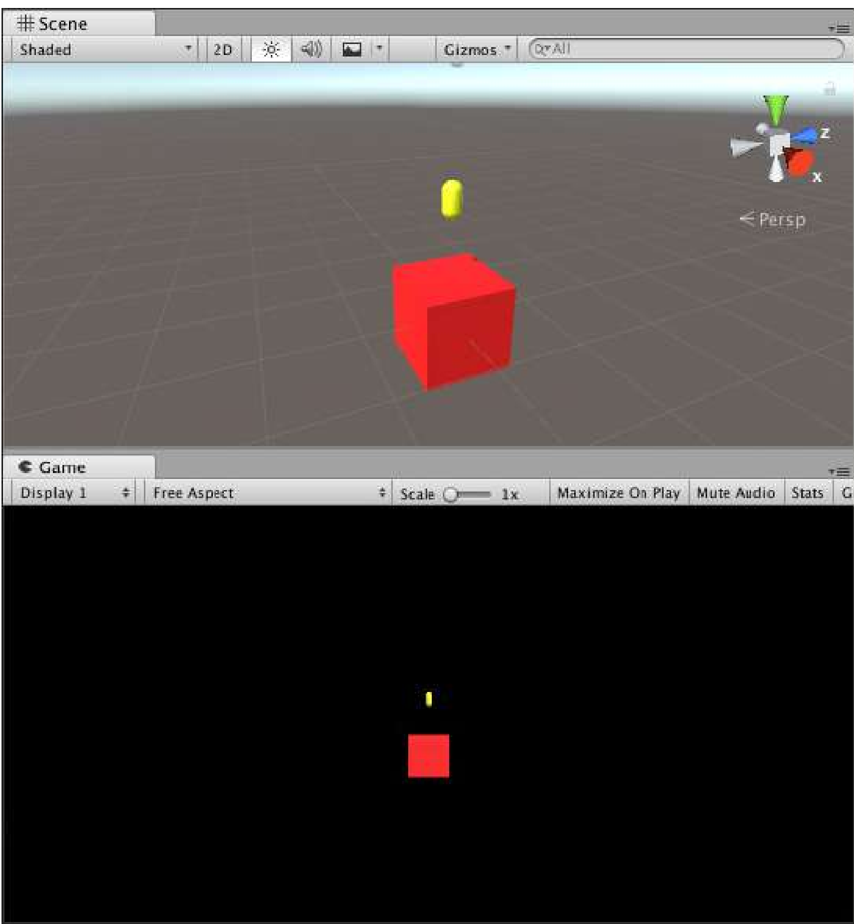


Fig. 14.19

Ahora con el proyectil seleccionado vamos a la ventana Inspector y le vamos a agregar un componente RigidBody. Para agregar este nuevo componente accedemos al botón **Add Component > Physics > Rigidbody**

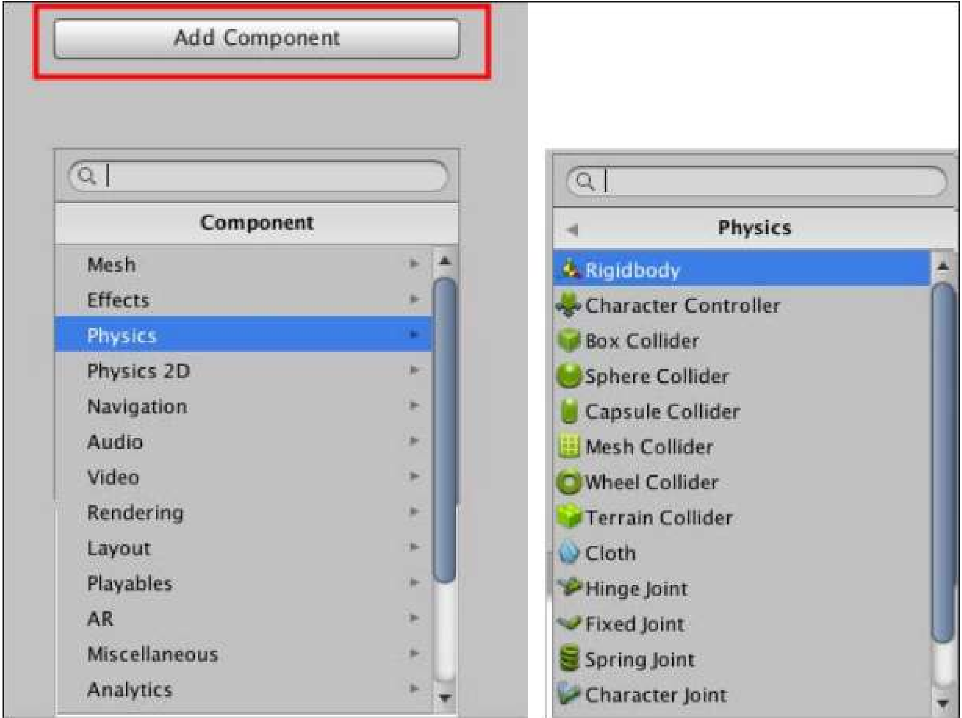


Fig. 14.20

Dentro de la ventana Inspector se nos añade el componente **RigidBody** a que debemos desactivar el parámetro Use **Gravity** y activamos el parámetro Is **Kinematic**

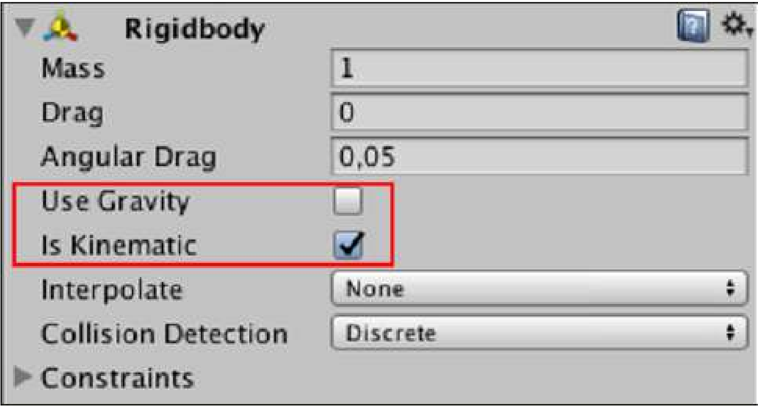


Fig. 14.21

Ahora vamos a crear un script para el proyectil, accedemos a la carpeta scripts y creamos un nuevo script con el nombre **Control_Proyectil** y lo añadimos al objeto **Proyectil.**

Script: Control_Proyectil.cs

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
public class Control_Proyectil : MonoBehaviour
{
        public float speedPro;
        public float velocidadPro;
        void Awake ()
        {
                this.speedPro = 10f;
        }
        void Update ()
        {
                this.velocidadPro = speedPro * Time.deltaTime;
                this.gameObject.transform.Translate (Vector3.up * this.
velocidadPro);
        }
}
```

Ahora si ejecutamos la escena veremos como el Proyectil sale disparado hacia el **eje z**, es decir hacia arriba según tenemos enfocada la cámara. Uno de los problemas a resolver es saber hasta que altura es visible en nuestra pantalla.

Para tener la misma resolución vamos a la ventana **Game** y en el menú superior seleccionamos la opción **Free Aspect** y en el menú que se despliega crearemos uno propio de 600x400 en este caso yo le he dado el nombre de **MiGame.**

Fig. 14.22

Ahora que ya tengo la pantalla determinada voy a seleccionar el proyectil y desde la ventana Inspector voy posicionar el proyectil hasta que desaparece de la pantalla y voy a tener este valor muy presente. En este caso del ejemplo el valor es en y =10, este valor es el que voy a utilizar para que cuando el objeto Proyectil llegue hasta ese valor se destruya.

Script: Control_Proyectil.cs

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
public class Control_Proyectil : MonoBehaviour
{
        public float speedPro;
        public float velocidadPro;
        void Awake ()
        {
                this.speedPro = 10f;
        }
        void Update ()
        {
                    this.velocidadPro = speedPro * Time.deltaTime;
                    this.miPro.transform.Translate (Vector3.up * this.
velocidadPro);
                    if (gameObject.transform.position.y>10f)
                    {
                        Destroy (gameObject);
                    }
        }
}
```

Reutilizar el Proyectil

A continuación vamos a crear un prefab de nuestro proyectil. Para ello localiza la carpeta con el nombre Prefabs que hemos creado dentro de la ventana **Project** y arrastramos el objeto Proyectil desde la ventana **Hierarchy** hasta la carpeta **Prefabs**, Unity automáticamente creará un objeto prefab. Una prueba de que se ha creado el objeto es que en la ventana **Hierarchy** el nombre del objeto toma un color Azul.

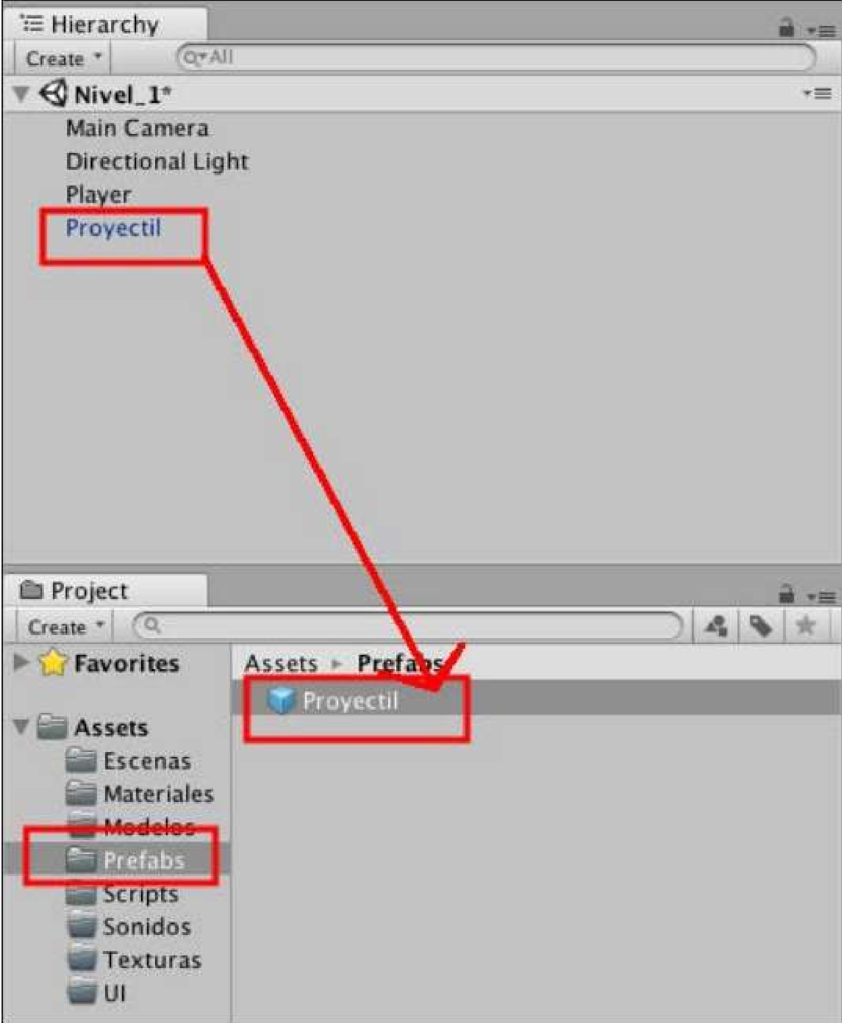


Fig. 14.23

Ahora vamos a crear una variable para poder acceder al proyectil desde el player. Para eso debemos acceder al script **Control_Player** y añadimos la siguiente linea: ***public GameObject proyectil;***

Al añadir esta variable si guardamos el script y volvemos a Unity en la ventana Inspector , dentro del componente Script se nos aparece una caja de texto en donde debemos arrastrar dentro el prefab proyectil.

Instanciar el Proyectil

Para Instanciar un proyectil debemos acceder al script del player y añadir la siguientes lineas de código dentro de la función Update().

Script: Control_Player

```
//añadir una variable de tipo Vector3 con nombre posicionProyectil;
Vector3 posicionProyectil;
//Esta parte del codigo va dentro de la función Update() al final de todo.
if (Input.GetKeyDown(KeyCode.LeftControl))
{
        this.posicionProyectil=this.miplayer.transform.position;
        this.posicionProyectil.y+=this.posicionProyectil.y/2;
        Instantiate(this.proyectil,this.miplayer.transform.position,Quater-
nion.identity);
}
```

Esto nos permite decirle a nuestro player que cada vez que pulsemos la tecla **Ctrl** del lado izquierdo del teclado, creará un nuevo objeto Proyectil o mejor dicho se instancia un proyectil al que le hemos dado el argumento de que siempre se cree desde donde esta el objeto player. La posición de este proyectil en el eje y se le ha añadido la posición del player y luego se le ha dividido por la mitad para que el proyectil salga por delante del player.

Sonido para el proyectil

El material para este proyecto lo vas a tener dentro de la carpeta del capitulo en una carpeta Proyecto final. Dentro de esta tienes los sonidos imágenes y todo lo necesario para continuar.

Selecciona los archivos de la carpeta sonidos y arrástralos dentro de Unity en la carpeta que se llama también sonidos.

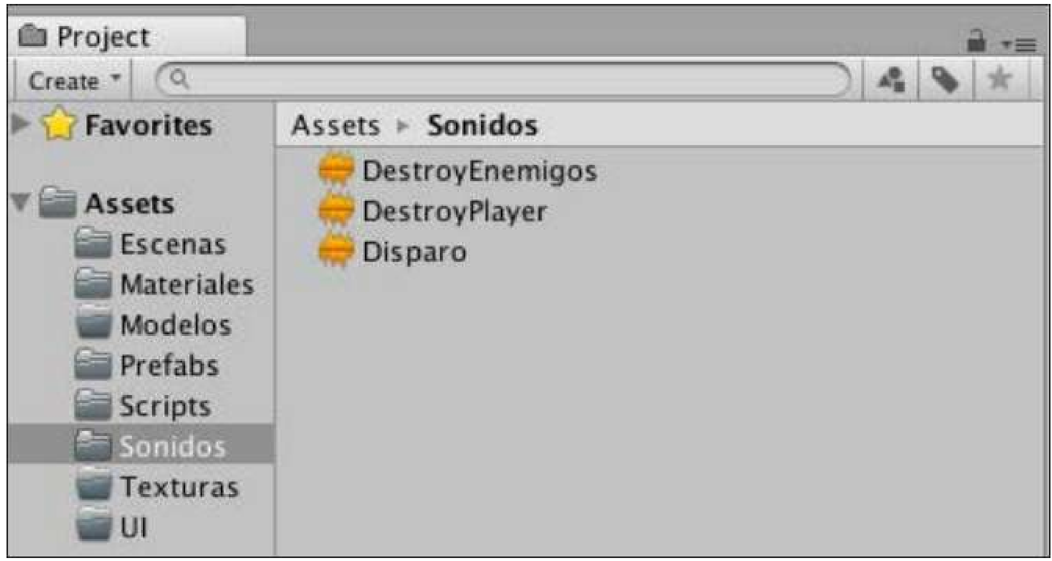


Fig. 14.24

Ahora que tenemos los audios vamos a la carpeta **prefabs** seleccionamos el proyectil y dentro de la ventana inspector vamos añadirle un componente **AudioSource**.

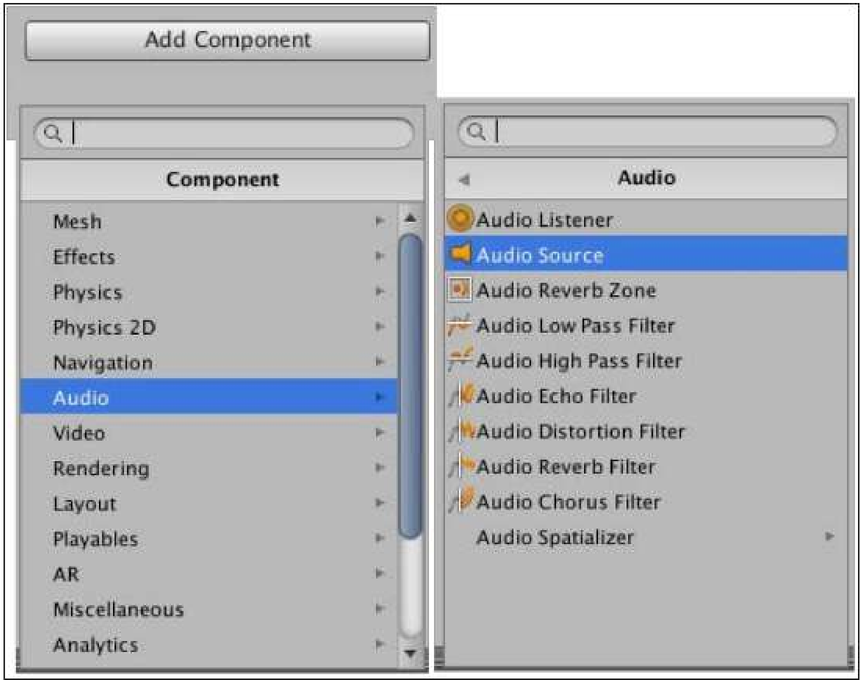


Fig. 14.25

En este nuevo componente arrastramos el archivo clip con el nombre Disparo, dentro del parámetro **AudioClip**.

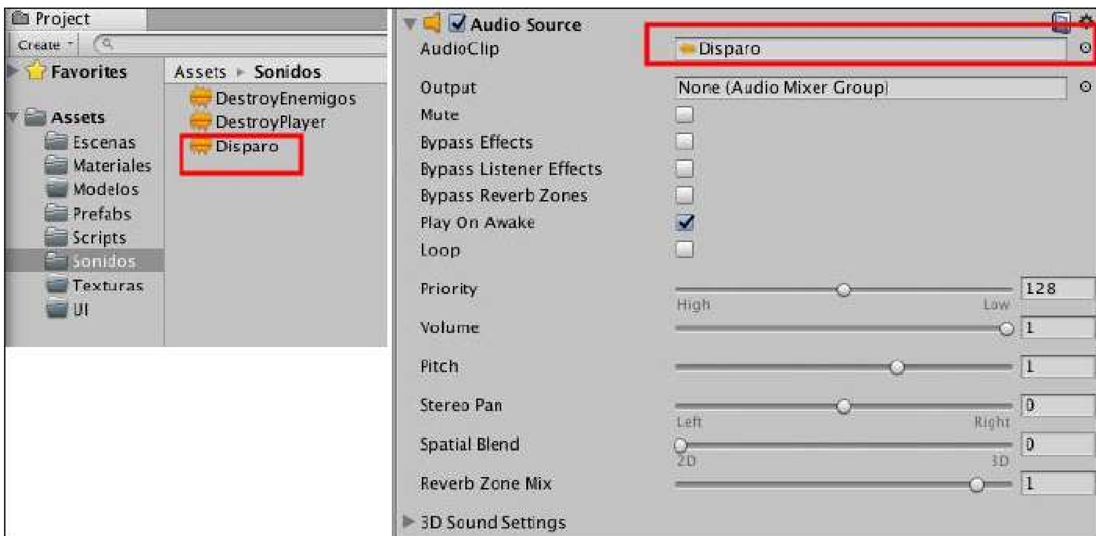


Fig. 14.26

Ahora si ejecutas la escena veras como cada vez que disparas un proyectil se escucha el sonido disparo.

6. Crear un enemigo

Para crear el prototipo de enemigo voy a crear una esfera y le voy a cambiar el nombre por el de Enemigo. El siguiente paso es posicionar el objeto enemigo en la parte alta de la pantalla en donde no se pueda ver. La idea es que el objeto Enemigo venga de arriba hacia abajo. A continuación te muestro los parámetros del enemigo.

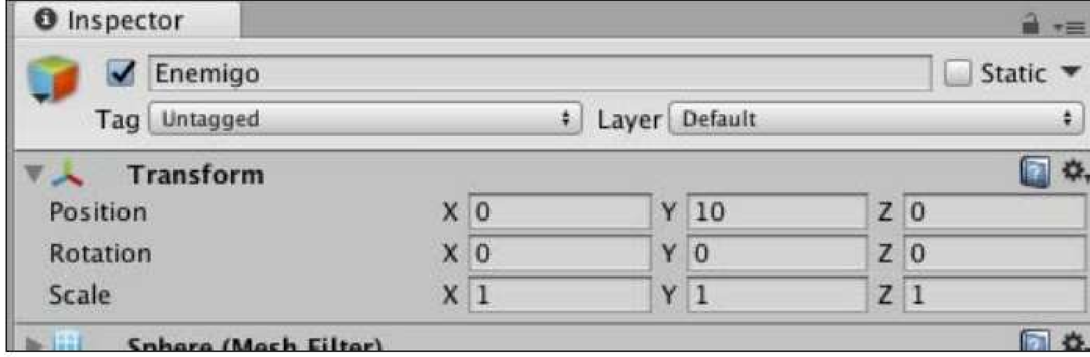


Fig. 14.27

El siguiente paso que debemos realizar es crear un material para el enemigo como hemos hecho hasta ahora. En el caso del ejemplo le he dado un color verde. Ahora es el momento de crear un **Script** para que controle el enemigo. Crea un script con nombre **Control_Enemigo**.

Script: Control_Enemigo.cs

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
public class Control_Enemigo : MonoBehaviour
{
        public float minSpeed=2;
        public float maxSpeed=8;
        private float actualSpeed;
        private float velocidadEnemigo;
        GameObject miEnemigo;

        private float x, y, z;
        void Awake ()
        {
                this.miEnemigo = gameObject;
                this.AlternarEnemigo ();
        }

        public void AlternarEnemigo()
```

```
        {
                this.actualSpeed = Random.Range (minSpeed, maxSpeed);
                x = Random.Range (-6f, 6f);
                y = 10f;
                z = 0f;
                this.miEnemigo.transform.position = new Vector3 (x, y, z);
        }
        void Update ()
        {
                this.velocidadEnemigo = this.actualSpeed * Time.deltaTime;
                this.miEnemigo.transform.Translate (Vector3.down * this.
velocidadEnemigo);
                if (miEnemigo.transform.position.y <-4)
                {
                        this.AlternarEnemigo ();
                }
        }
}
```

La idea es que cuando el enemigo desaparezca por abajo de la pantalla se posicione otra vez en la parte superior de una forma distinta cada vez y de este modo reutilizar el enemigo.

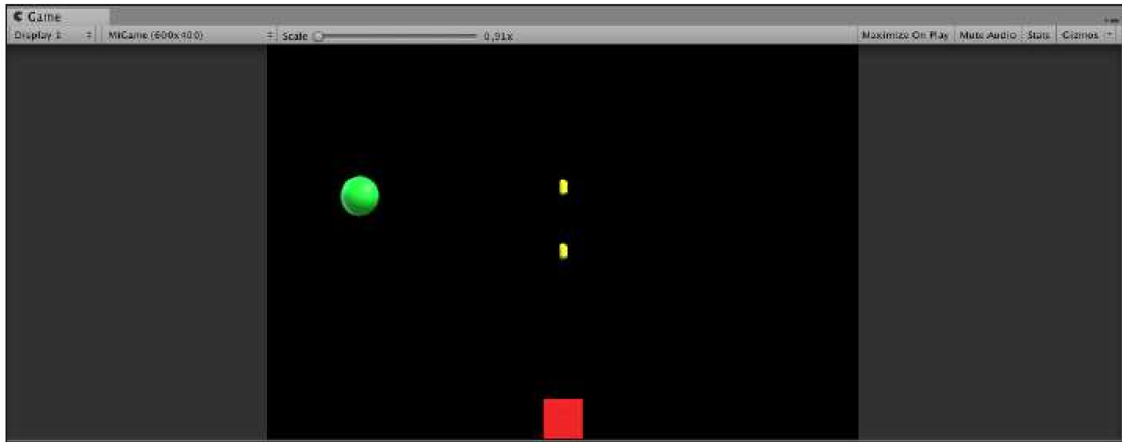


Fig. 14.28

7. Colisiones

Primero seleccionamos el objeto Enemigo y en la ventana Inspector en el apartado **Tag** vamos a crear una etiqueta con el nombre de Enemigo.

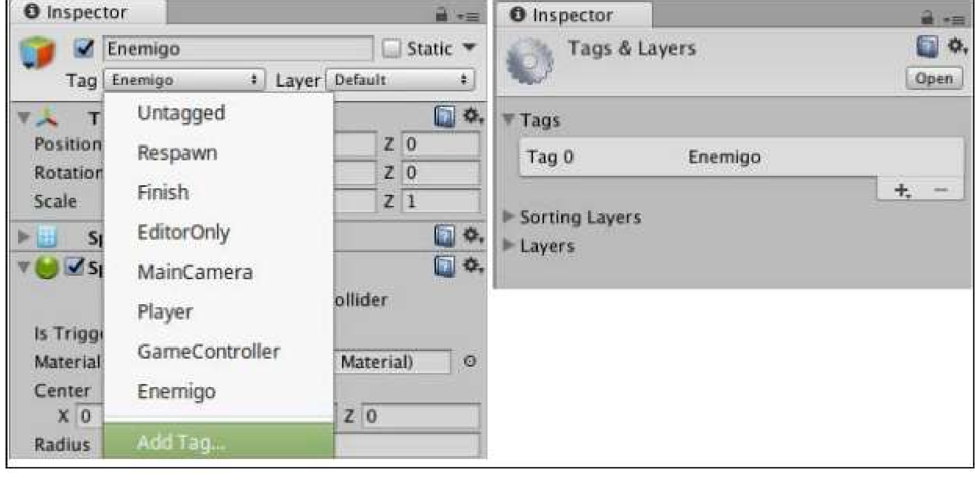


Fig. 14.29

Una vez le hemos puesta la etiqueta tag Enemigo al objeto Enemigo en su componente Collider activamos la opción **Triggers**.

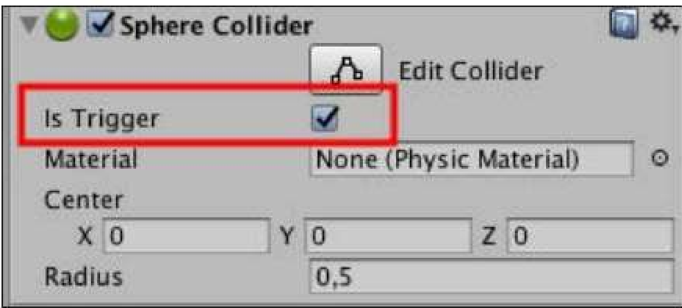


Fig. 14.30

Ahora accedemos al script **Control_Proyectil.cs** y añadiremos al final el siguiente código para que los proyectiles detecten a los enemigos, los destruyan y se destruya el proyectil cuando colisiona con un enemigo.

Script: Control_Proyectil.cs

```
        void OnTriggerEnter(Collider otro)
        {
           if (otro.tag == "Enemigo")
           {
                otro.SendMessage ("AlternarEnemigo",true, SendMessageOptions.
DontRequireReceiver);
                GameObject.Destroy (this.gameObject);
           }
        }
```

8. Explosiones

Ahora que nuestros proyectiles detectan a los enemigos y se destruyen, vamos a crear una explosión, cuando se produce el impacto entre estos dos objetos (proyectil y enemigo).

Primero tenemos que crear una explosión, para ello accedemos al menú principal **GameObject > Effects > Particle System.**

Fig. 14.31

Ahora tenemos un sistema de partículas al que le vamos a cambiar el nombre, por el nombre de **"Explosion"** desde la ventana Inspector y vamos a configurar varios parámetros para darle un aspecto de explosión.

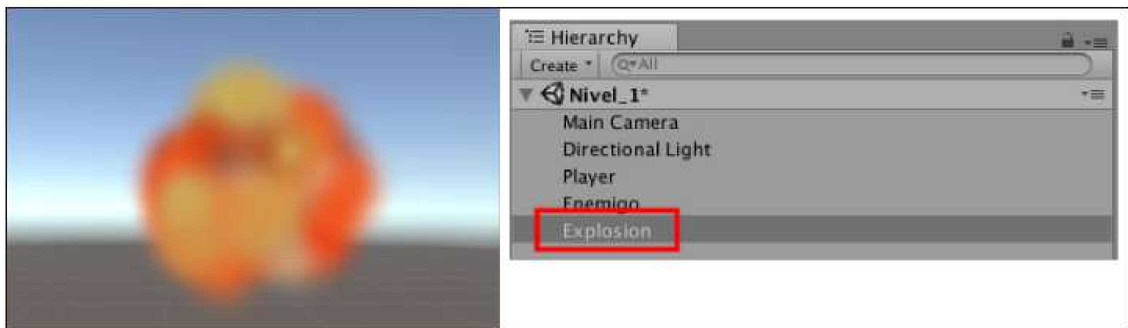


Fig. 14.32

Si accedemos a los componentes de **Particle System** podemos configurar en el primer apartado las siguientes características que te muestro a continuación.

En este caso el color lo dejo a tu juicio en el caso del ejemplo he seleccionado la opción Gradient para simular una explosión de fuego. En la siguiente imagen te muestro como acceder , en el caso de que tengas dudas revisa el capitulo 13.

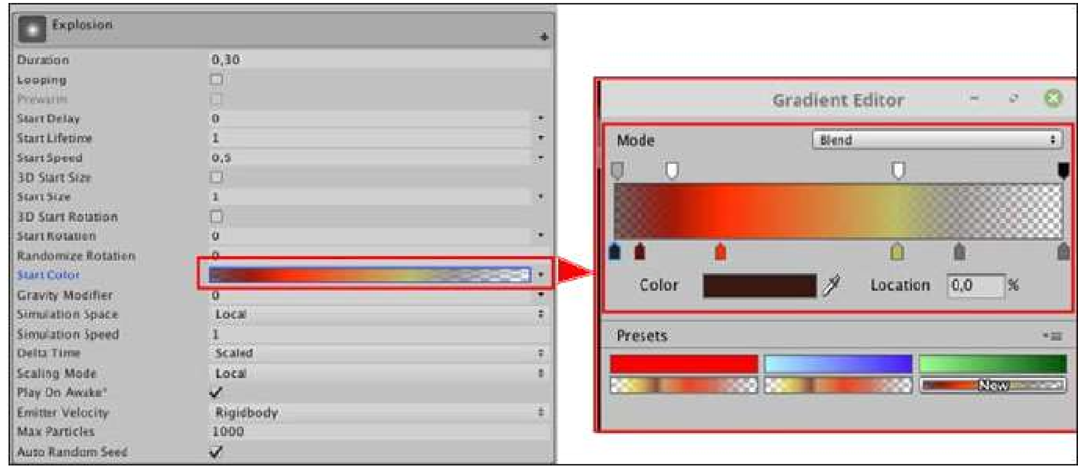


Fig. 14.33

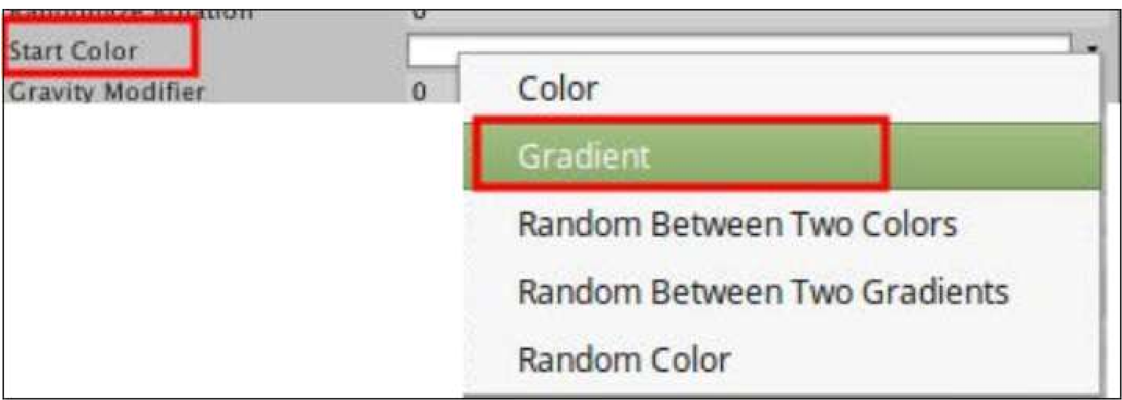


Fig. 14.34

- **Duration** = 0,30.
- **Looping**: Desactivado
- **Start Delay** = 0
- **Start Lifetime** = 1
- **Start Speed** = 0,5
- **3D Start Size**: Desactivado
- **Start Size**= 1
- **3D Start Rotation**, **Start Rotation**,**Randomize Rotation**, **Gravity Modifier** = 0
- **Simulation Space**: Local
- **Simulation Speed**:1
- **Delta Time:** Scaled
- **Scaling Mode**: Local
- **Play On Awake:** Activado
- **Emiter Velocity:** Rigidbody
- **Max Particles** = 1000
- **Auto Random Seed**: Activado

En el apartado **Emission** en el parámetro **Rate over Time** le valor es de 150 y el **Rate over Distance** el valor es 0.

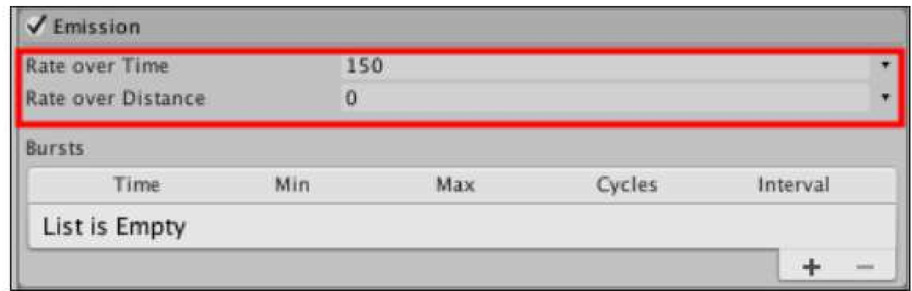


Fig. 14.35

Para finalizar el ultimo cambio que he realizado es en el apartado Shape en donde le he dado una forma de Cono al sistema de partículas, con un angulo de 60. El Radio y el arco y el resto de parámetros tienen los valores por defecto.

✓ Shape						
Shape	Cone					
Angle	60					
Radius	0,3883279					
Radius Thickness	1					
Arc	360					
Mode	Random					
Spread	0					
Length	5					
Emit from:	Base					
Position	X	0	Y	0	Z	0
Rotation	X	0	Y	0	Z	0
Scale	X	1	Y	1	Z	1
Align To Direction	☐					
Randomize Direction	0					
Spherize Direction	0					
Randomize Position	0					

Fig. 14.36

Una vez tenemos la configuración de nuestro sistema de partículas creado con el nombre **Explosion**, vamos a crear un prefab para reutilizar esta misma explosión tantas veces como necesitemos. Una de las formas más rápidas de crear un prefab es arrastrando el objeto **Explosion** desde la ventana **Hierarchy** hasta la ventana **Project** dentro de la carpeta Prefabs. Unity crea un prefab del objeto automáticamente. Ahora borra el objeto de la escena, seleccionando o bien el objeto en la propia escena o bien el nombre del objeto en la ventana **Hierarchy** y pulsar Suprimir.

Crear explosiones con los proyectiles

Debemos acceder al script **Control_Proyectil** en donde vamos a crear una variable de tipo **GameObject** para añadirle el prefab Explosión como te muestro a continuación.

Script: Control_Proyectil.cs

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
public class Control_Proyectil : MonoBehaviour
{
        public float speedPro;
        public float velocidadPro;
        public GameObject explosion;
```

Primero añadimos la variable **Explosion** guardamos el script y volvemos a Unity. Para que no te confundas, selecciona el prefab **Proyectil** y en la ventana Inspector podrás ver en el componente Script un parámetro nuevo llamado **Explosion**, ahora simplemente debes arrastrar el prefab **Explosión** encima de este parámetro para que tome la referencia el script.

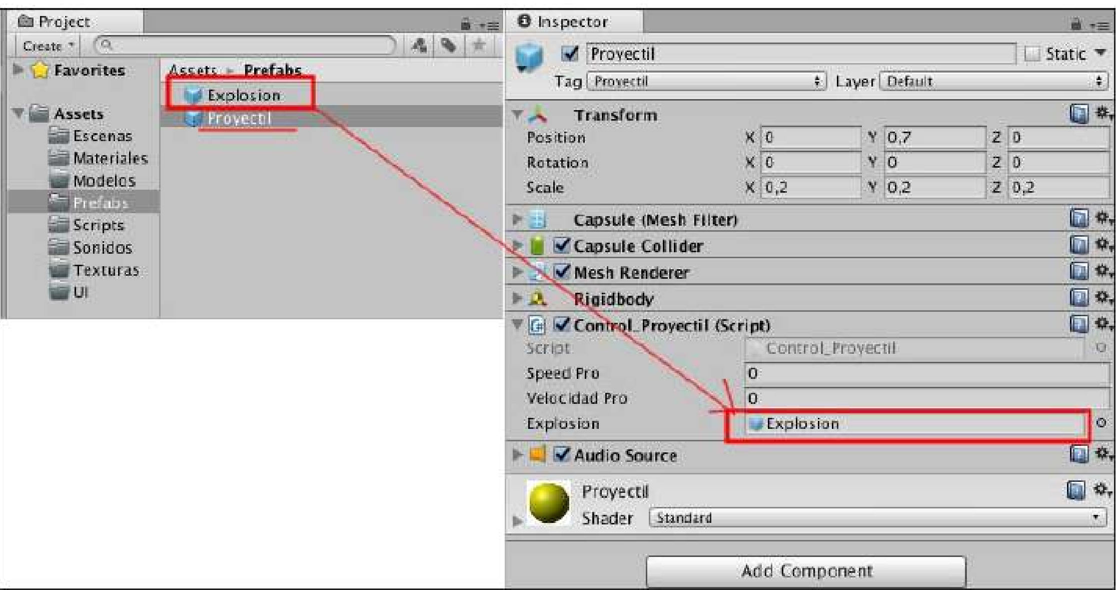


Fig. 14.37

Ahora que tenemos referencia-da la explosión con el prefab "**Explosion**" volvemos al script del proyectil y añadimos dentro de la función **OnTriggerEnter** una instancia de este prefab y luego su destrucción para que desaparezca.

Script: Control_Proyectil.cs

```
void OnTriggerEnter(Collider otro)
{
   if (otro.tag == "Enemigo")
   {
        otro.SendMessage ("AlternarEnemigo", true, SendMessageOptions.
DontRequireReceiver);
        this.explosion=Instantiate(this.explosion,gameObject.transform.posi-
tion,   this.explosion.transform.rotation) as GameObject;
        GameObeject.Destroy (this.explosion.gameObject, 1,0f);
        GameObeject.Destroy (this.gameObject);
   }
}
```

Guardamos el script y ahora al volver a Unity y ejecutar la escena podemos comprobar que cuando disparamos un proyectil contra un enemigo este crea una explosión, como te muestro en la siguiente imagen.

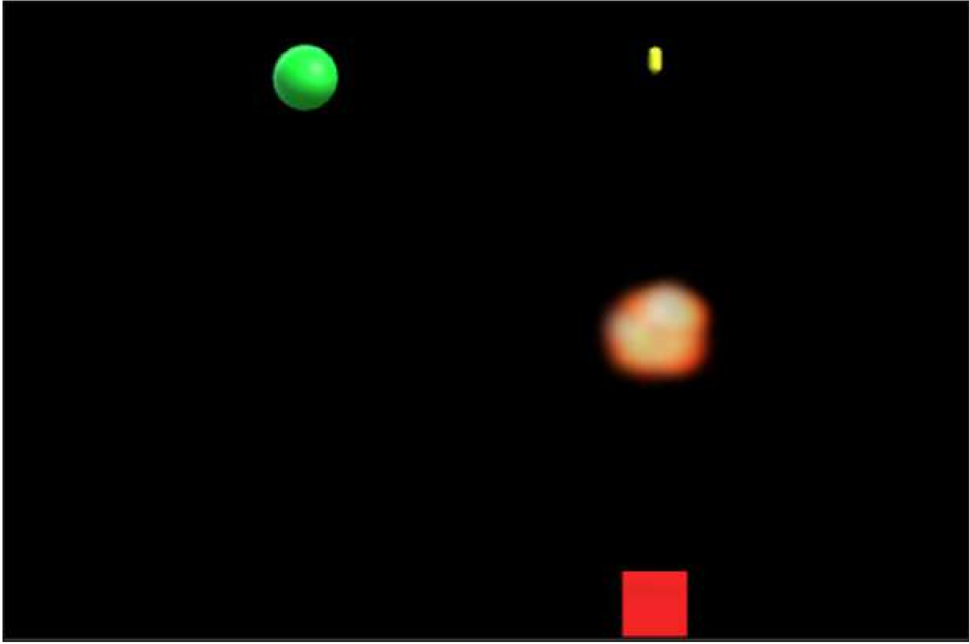


Fig. 14.38

Añadir audio a la explosión

Como la explosión solo se va a activar cuando el proyectil impacte con el enemigo y luego va a ser eliminado, el audio se lo vamos a añadir al prefab Explosión. Accedemos a la carpeta Prefabs en la ventana **Project** y seleccionamos **Explosion** y accedemos a la ventana Inspector.

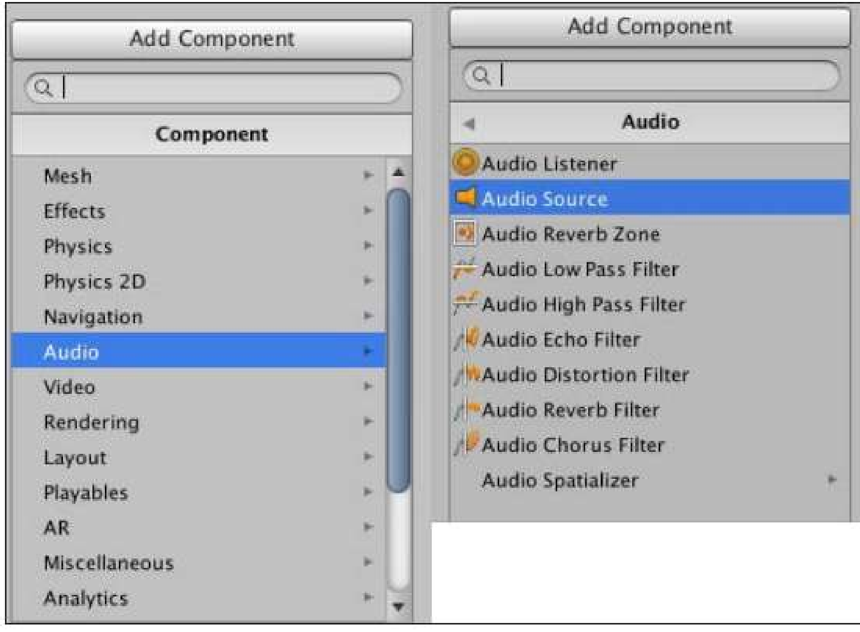


Fig. 14.39

En el componente **AudioSource** arrastramos dentro del parámetro audio clip el audio **DestroyEnemigos**, que encontraras en la carpeta Sonidos.

Fig. 14.40

Ahora si ejecutas la escena veras como al impactar un proyectil con un Enemigo este desaparece, se ve y se escucha el audio de explosión.

9. Añadir sistema de puntos y vidas

Para crear un sistema de puntos vamos a crear un Canvas para poder representar en pantalla los valores. Primero accedemos a la ventana **Hierarchy** y con el botón **Create** seleccionamos la opción **UI > Canvas.**

En las propiedades del canvas dentro de la ventana **Inspector,** cambiamos los siguientes parámetros. En el apartado **Canvas Scale** en el parámetro **UI Scale Mode** seleccionamos la opción **Scale With Screen Size.** La **reference Resolution** en el caso concreto de este ejemplo es de 800x600.

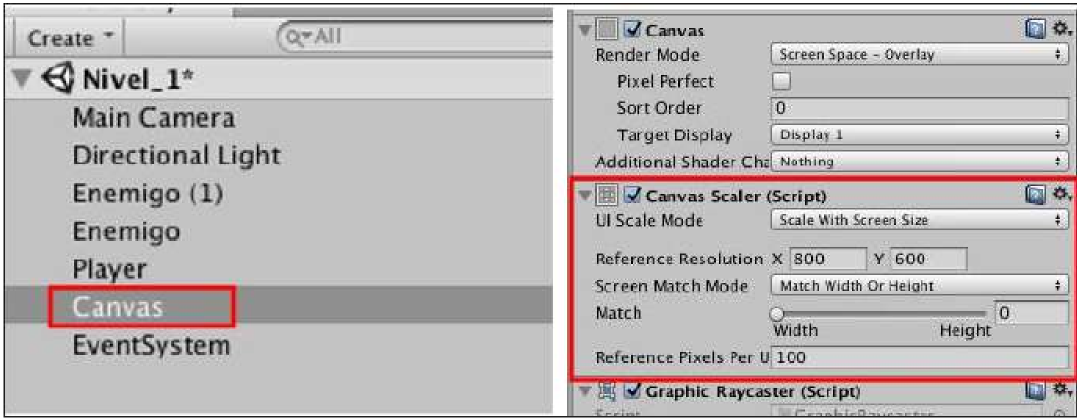


Fig. 14.41

Ahora dentro del canvas vamos a crear dos objetos de tipo texto. En uno nos va a decir los puntos que llevaremos y en el otro las vidas que nos quedan. Para crear este tipo de objeto seleccionamos el canvas y luego desde el botón **Create** de la ventana **Hierarchy** seleccionamos **UI> Text.** Esta acción la realizaremos dos veces y re nombraremos los objetos con los nombres **Score y Lives.**

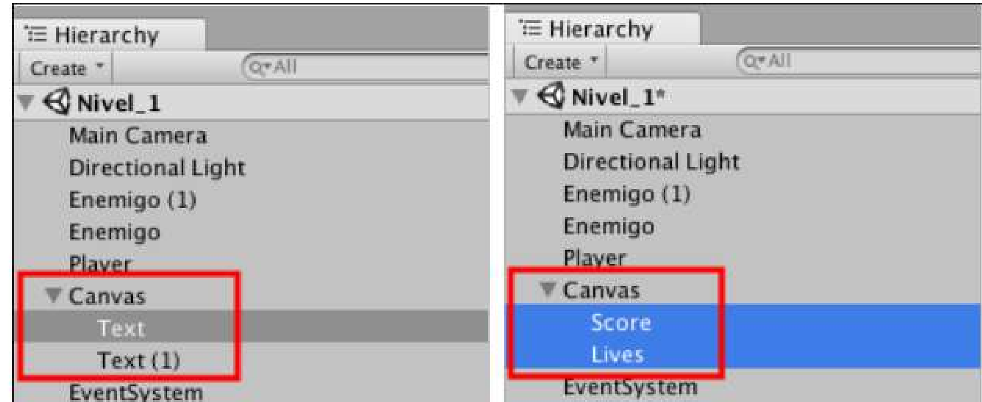


Fig. 14.42

Ahora seleccionemos el objeto text **Score** y vamos a acceder a sus parámetros para configurar-lo de la siguiente manera.

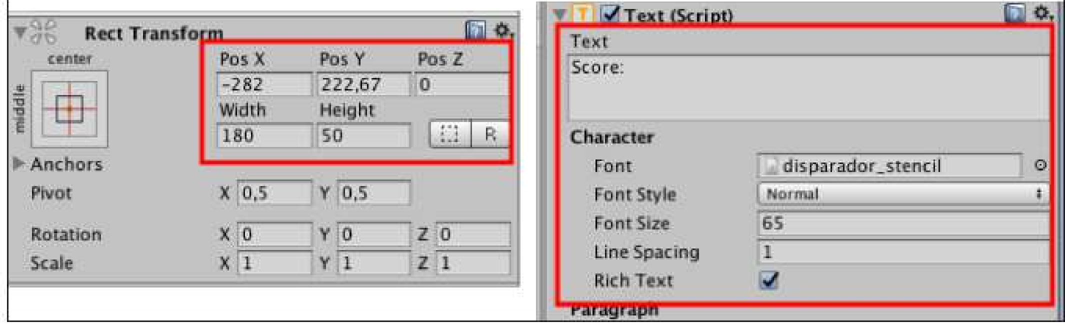


Fig. 14.43

En el apartado **Rect Transform** he puesto los valores en **X= -282, Y=222,67 Z=0** , en el parámetro **Width = 180** y en **Height= 50.**

En el apartado **Text** en la caja de texto he escrito el nombre **Score** seguido de dos puntos. En las características del carácter he seleccionado una tipografía que encontraras dentro de la carpeta UI que acompaña el material del capítulo. El tamaño de la fuente le he dado el valor de 65 . Otro punto importante que no está marcado en la imagen anterior es el color del texto que se le ha puesto en blanco para que se pueda ver bien con el fondo negro que tenemos en la escena.

El objeto **Lives** que también es de tipo texto vamos a darle una posición por debajo de **Score c**on los siguientes valores X= -282, Y=173 Z=0 en el parámetro Width = 180 y en Height= 50. En la caja de texto del apartado **Text** le ponemos el nombre de **Lives** seguido de dos puntos y al igual que en **Score** la fuente es la que encontraremos dentro de la carpeta UI y el tamaño es de 65.

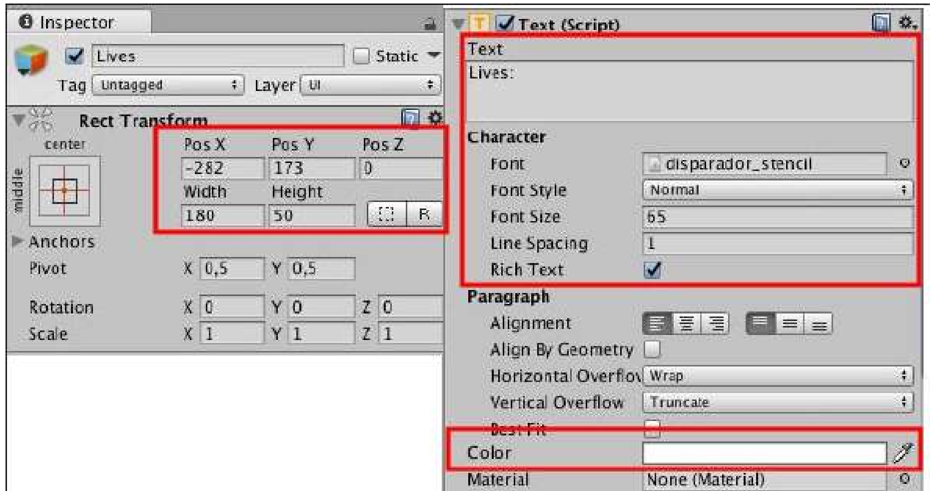


Fig. 14.44

Si todo está correctamente configurado la escena debería verse como te muestro en la siguiente imagen.

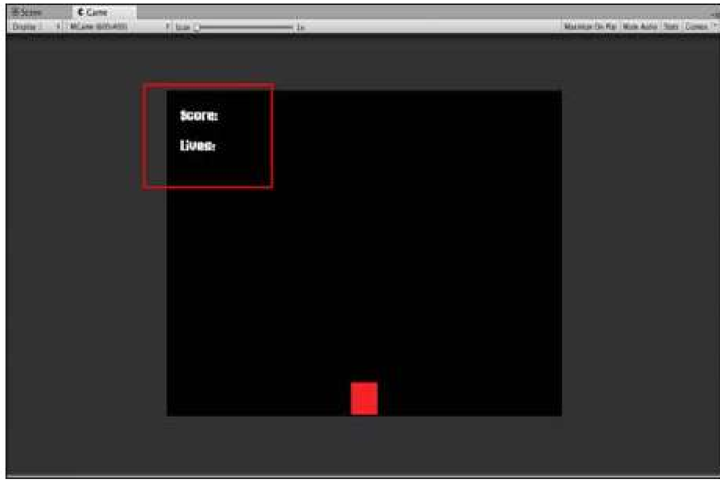


Fig. 14.45

Contadores de puntos y de vida

Ahora que tenemos los textos bien colocados en la escena vamos a darles funcionalidad. En el script Control_Player vamos a crear las variables y una función para que nuestros textos reaccionen a las acciones del juego.

Script: Control_Player.cs

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.UI;
public class Control_Player : MonoBehaviour {
	public float playerSpeed;
	public float anchoMovimiento;
	public GameObject miplayer;
	public GameObject proyectil;
	Vector3 posicionProyectil;
	public static int score = 0;
	public static int lives = 3;
	public Text vidas;
	public Text puntos;

	void Awake (){
		this.playerSpeed = 10f;
		this.miplayer = gameObject;
	}
	void Update () {
		this.anchoMovimiento = Input.GetAxisRaw ("Horizontal") * this.
playerSpeed * Time.deltaTime;
		this.miplayer.transform.Translate (Vector3.right * this.
anchoMovimiento);
		if (transform.position.x <= -7.5f) {
		  transform.position=new Vector3 (7.4f, transform.position.y,
transform.position.z);
		}

		else if (transform.position.x >= 7.5f) {
			transform.position= new Vector3(-7.4f,transform.posi-
tion.y, transform.position.z);
		}
		this.Disparar ();
		this.Contadores ();
	}
```

```
        public void Disparar()
        {
                if (Input.GetKeyDown(KeyCode.LeftControl)){
                        this.posicionProyectil=this.miplayer.transform.
position;
                        this.posicionProyectil.y+=this.posicionProyectil.y/2;
                        Instantiate(this.proyectil, this.miplayer.transform.
position,                                       Quaternion.identity);
                }
        }

        void Contadores()
        {
                this.puntos.text="Score : "+ Control_Player.score.ToString();
                this.vidas.text="Lives : "+ Control_Player.lives.ToString();
        }

}
```

En el **sript** hemos utilizado la extensión UI en el inicio del script para que nos permita Unity utilizar textos. Las variables que vamos a utilizar primero son de tipo **int** para valores enteros, para **score** le damos el valor 0 y para **lives** le damos el valor 3. Para **referenciar** que textos son los que utilizaremos creamos dos variables de tipo **Text** y sobretodo publicas para poder arrastrar el objeto texto que queremos **referenciar** hacia la ventana **Inpector** dentro del componente script con el parámetro con nombre de la variable. En este caso le hemos puesto el nombre de vidas y puntos. En la siguiente imagen te muestro como se verá en el inspector y que elementos debemos arrastrar a los parámetros vidas y puntos.

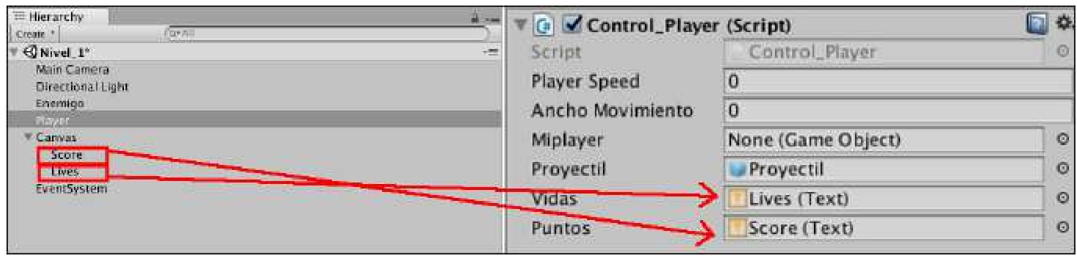


Fig. 14.46

Volviendo al script **Control_Player** al final de este script creamos una función con nombre **Contadores**() en donde accedemos a las variables de tipo texto a su método texto en donde les daremos los valores de **Score** para puntos y utilizaremos al final el método **ToString** para que los valores que son enteros convertirlos en texto. Luego esta función la tenemos que llamar dentro de la función **Update**(). Una vez tenemos completado el script debemos vincular estas variables a las acciones del juego y para ello para que el texto **Score** vaya sumando por ejemplo 25 puntos cada vez que se elimina un enemigo debemos acceder al script de **Control_Proyectiles** y añadir un valor para score cada vez que un proyectil destruye un enemigo.

Script: Control_Proyectil.cs

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

public class Control_Proyectil : MonoBehaviour {

        public float speedPro;
        public float velocidadPro;
        public GameObject explosion;

        void Awake (){
                this.speedPro = 10f;
        }

        void OnTriggerEnter(Collider otro)
        {
                if (otro.tag == "Enemigo")
                {
                        otro.SendMessage ("AlternarEnemigo", true,
                                                        SendMessageOptions.
DontRequireReceiver);
                        this.explosion = Instantiate (this.explosion, gameOb-
ject.transform.position,                 this.explosion.transform.rota-
tion)as GameObject;
                        GameObject.Destroy (this.explosion.gameObject, 1.0f);
                        GameObject.Destroy (this.gameObject);
                        Control_Player.score += 25;
                }
        }

        void Update ()
        {
                this.velocidadPro = speedPro * Time.deltaTime;
                this.gameObject.transform.Translate (Vector3.up * this.
velocidadPro);
                if (gameObject.transform.position.y>10f)
                {
                        Destroy (gameObject);
                }
        }
}
```

Si todo es correcto guardamos los scripts y volviendo a Unity, si ejecutamos la escena y destruimos algunos enemigos, veremos que el texto S**core** va aumentando su valor en 25 unidades.

Fig. 14.47

Restar vida

Ahora vamos a ver como hacer desaparecer nuestro player cuando impacta con un enemigo y como esta acción resta vidas al Player. Primero seleccionamos el objeto Player y luego en la ventana Inspector, en el componente **Box Collider** activamos la opción **IsTrigger**.

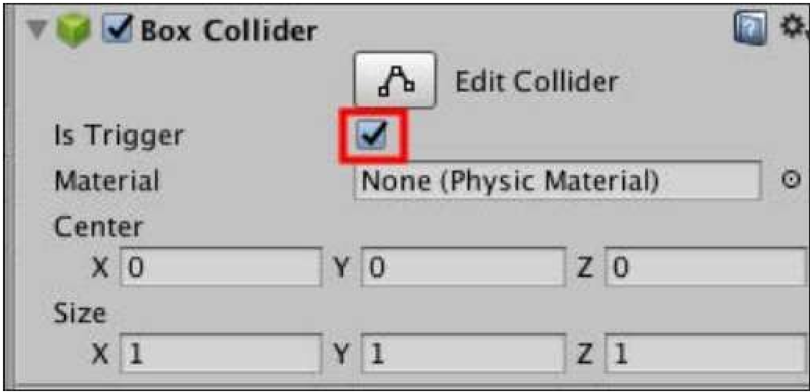


Fig. 14.48

Al activar el **Trigger** vamos a poder configurar mediante código cuando se produce el impacto del Player con el objeto. Cuando esto se produzca desactivaremos el Player y después de 1,5 segundos volveremos a reactivar el **Player** mediante el método **Invoke**.

Script: Control_Player.cs

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.UI;
public class Control_Player : MonoBehaviour {
        public float playerSpeed;
        public float anchoMovimiento;
        public GameObject miplayer;
        public GameObject proyectil;
        Vector3 posicionProyectil;
        public static int score = 0;
        public static int lives = 3;
        public Text vidas;
        public Text puntos;
        public GameObject explosion;
        public bool activado=true;
        void Awake ()
        {
                this.playerSpeed = 10f;
                this.miplayer = gameObject;
        }
        void OnTriggerEnter(Collider otro)
        {
                if(otro.tag=="Enemigo")
                {
                        Control_Player.lives-=1;
                        this.explosion= Instantiate(this.explosion, gameOb-
ject.transform.x,                                gameObject.transform.y, gameOb-
ject.transform.z);
                        this.activado=false;
                }
        }
        void Update ()
        {
                this.anchoMovimiento = Input.GetAxisRaw ("Horizontal") * this.
playerSpeed *                   Time.deltaTime;
                this.miplayer.transform.Translate (Vector3.right * this.
anchoMovimiento);
}
```

```
            if (transform.position.x <= -7.5f)
            {
                    transform.position=new Vector3 (7.4f, transform.posi-
tion.y,                                       transform.position.z);
            }

            else if (transform.position.x >= 7.5f)
            {
                    transform.position= new Vector3(-7.4f,transform.posi-
tion.y,                                       transform.position.z);
            }
            if (this.activado==false)
            {
                    gameObject.SetActive(false);
                    Invoke("ActivaAhora",1,5f);
            }

            this.Disparar ();
            this.Contadores ();
        }

        void ActivaAhora()
        {
            gameObject.SetActive(true);
            this.activado=true;
        }

        public void Disparar()
        {
            if (Input.GetKeyDown(KeyCode.LeftControl)){
                    this.posicionProyectil=this.miplayer.transform.
position;
                    this.posicionProyectil.y+=this.posicionProyectil.y/2;
                    Instantiate(this.proyectil, this.miplayer.transform.
position,                                     Quaternion.identity);
            }
        }

        void Contadores()
        {
            this.puntos.text="Score : "+ Control_Player.score.ToString();
            this.vidas.text="Lives : "+ Control_Player.lives.ToString();
        }
```

En el código anterior vemos como tenemos una variable de tipo GameObject con el nombre **explosion**, que nos servirá para arrastrar el prefab explosión que tenemos en la carpeta Prefabs para que cuando el Enemigo impacte con el Player se vea una explosión. En mi caso he creado otro prefab exactamente igual, pero con un sonido distinto, en este caso lo importante es que cuando volvamos a Unity, con el objeto player seleccionado debemos arrastrar el prefab **Explosion** encima del parámetro **Explosion** dentro del componente script **Controll_Player** como te muestro a continuación.

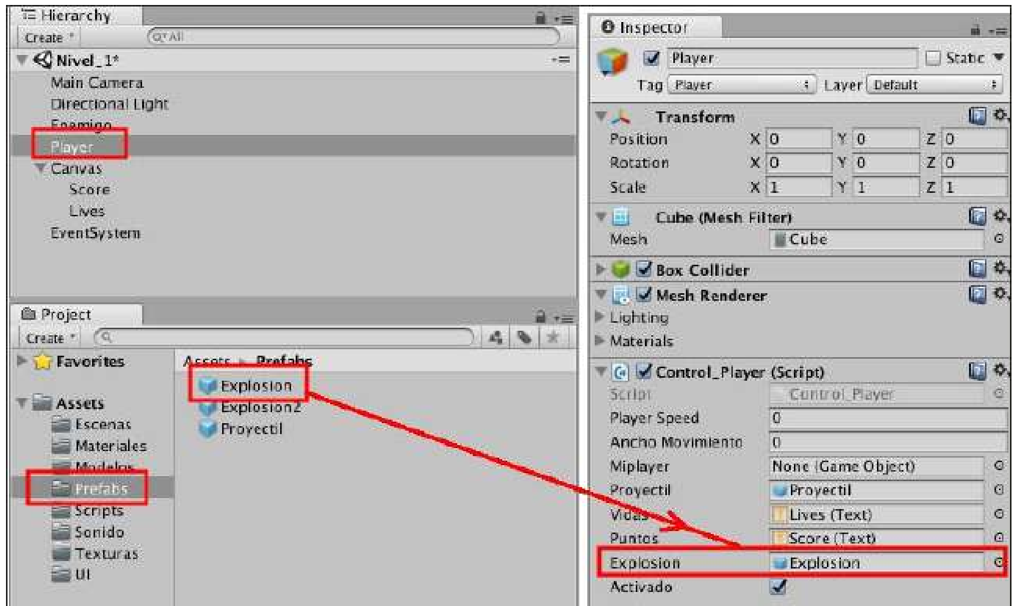


Fig. 14.49

Cuando utilizamos el método **Invoke** nos permite llamar a un método pasado cierto tiempo. Si quieres ver un ejemplo puedes acceder a la documentación de Unity desde el enlace que te proporciono a continuación.

https://docs.unity3d.com/2018.1/Documentation/ScriptReference/MonoBehaviour.Invoke.html

Una vez hemos añadido el prefab al parámetro **Explosion** podemos comprobar como funciona la escena.

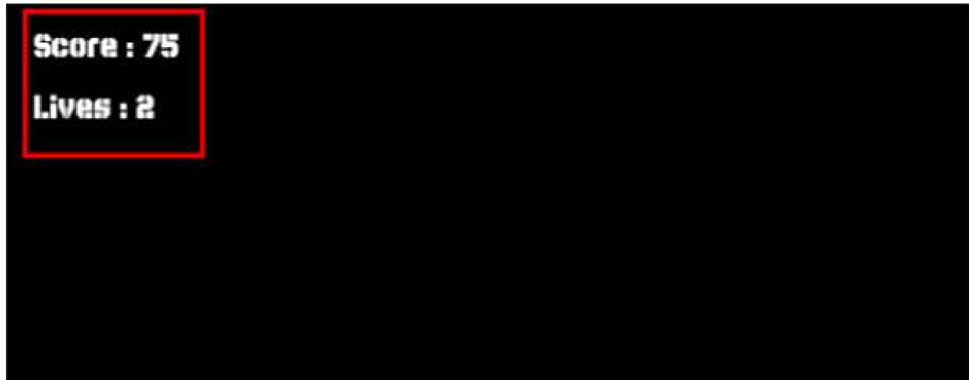


Fig. 14.50

10. Escena principal y GameOver

Para continuar con este pequeño proyecto, necesitamos de una escena principal que tenga un menú para iniciar el juego o quitar el juego. Después necesitaremos también crear una escena para cuando nuestro player se le terminen las vidas aparezca la escena **GameOver** para finalizar.

Escena principal

Para esta escena vamos a crear un menú con un Panel para poner una imagen con un titulo y dos botones, uno para empezar y el otro para quitar el juego. La escena que vamos a crear debe quedar como te muestro a continuación.

Fig. 14.51

Para empezar a montar el menú debemos acceder a la barra principal en **GameObject > UI > Canvas**. En la ventana **Hierarchy** aparecerá el objeto **Canvas** y el objeto **EventSystem**.

En la ventana inspector dejamos la siguiente configuración en los distintos componentes de Canvas.

Para el componente canvas te dejo la siguiente imagen con la configuración utilizada.

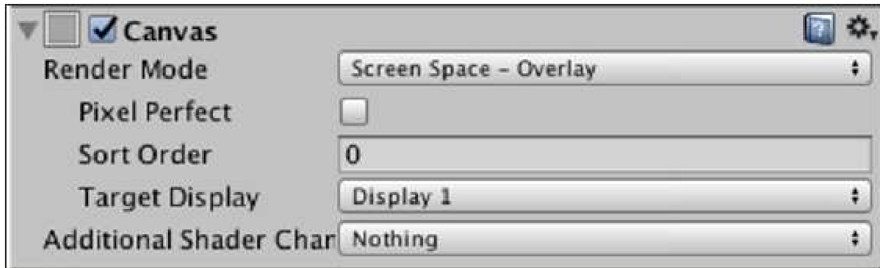


Fig. 14.52

En el componente **Canvas Scaler** (**Script**) que se encarga del tamaño en que se va a mostrar el canvas utilizamos la configuración que te muestro en la siguiente imagen.

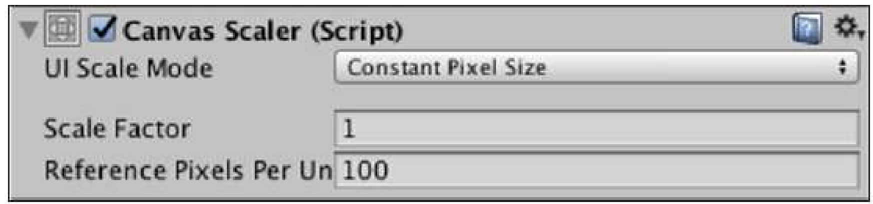


Fig. 14.53

Para finalizar los componentes en el **Graphic Raycaster** (Script) utilizamos la siguiente configuración.

Fig. 14.54

Ahora vamos a crear una serie de funciones dentro del canvas mediante un Script que utilizaremos después con los botones. Creamos un script añadiendo un componente Script desde el botón **Add Component** > **New Script** > Ponemos el nombre de **Main_ Menu** y pulsamos el botón **Create and Add**. Recuerda que si añades un script de esta manera en la ventana **Project** el script se encontrará fuera de la carpeta **scripts,** te recomiendo que la arrastres dentro de la ventana **Scripts** para tener organizado el proyecto. Dentro de este script **Main_Menu** escribiremos el siguiente código.

Script: Main_Menu

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine.SceneManagement;
using UnityEngine;
public class Main_Menu : MonoBehaviour {
        public void LoadLevel(string name)
        {
                SceneManager.LoadScene(name);
        }

        public void QuitRequest()
        {
                Application.Quit ();
        }
}
```

En el script anterior creamos dos funciones. Primero de todo debemos escribir la clase **SceneManager** en la parte superior. La primera función es de tipo publico con el nombre **LoadLevel**. Esta función va a tener un argumento que es una variable de tipo "**string**" con el nombre "**name**". Dentro de la función utilizamos el método **LoadScene** del **SceneManager** con el atributo **name**. Este método nos permite cargar una escena a partir del nombre de la escena.

La otra función también publica, la llamamos **QuitRequest().** Dentro de la función utilizamos la función **Application.Quit()** para que cuando se utilice salga de la aplicación. Este script lo utilizaremos con los botones cuando los tengamos creados.

Creación del Menú

Dentro del canvas vamos a crear una serie de objetos UI de tipo Panel Imagen y **Button**. El objeto **Button** lleva un como hijo un objeto texto. La disposición debe ser la siguiente en la ventana **Hierarchy** y a continuación vamos a ver las propiedades cada elemento en orden descendente.

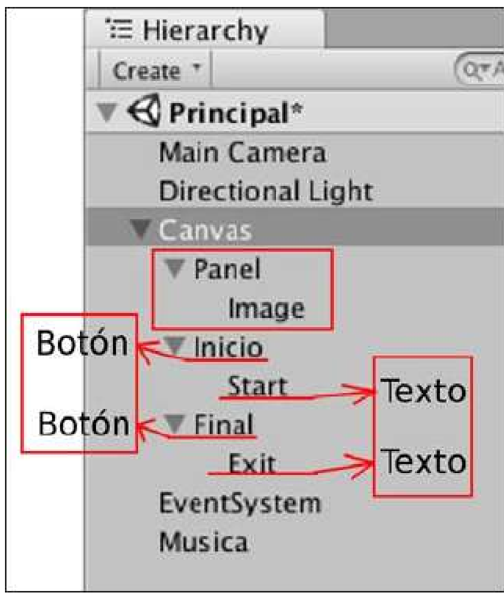


Fig. 14.55

Panel

Para crear el panel accedemos dentro de la ventana **Hierarchy** al botón **Create** y seleccionamos la opción **UI > Panel**. Este lo arrastramos encima de Canvas y ya tenemos el panel en la escena.

Seleccionamos el objeto Panel y configuramos las propiedades de sus componentes desde la ventana Inspector. El primer componente es el **Rect Transform** a continuación tienes una imagen con los valores que he utilizado. Si lo deseas puedes introducir los valores que creas convenientes.

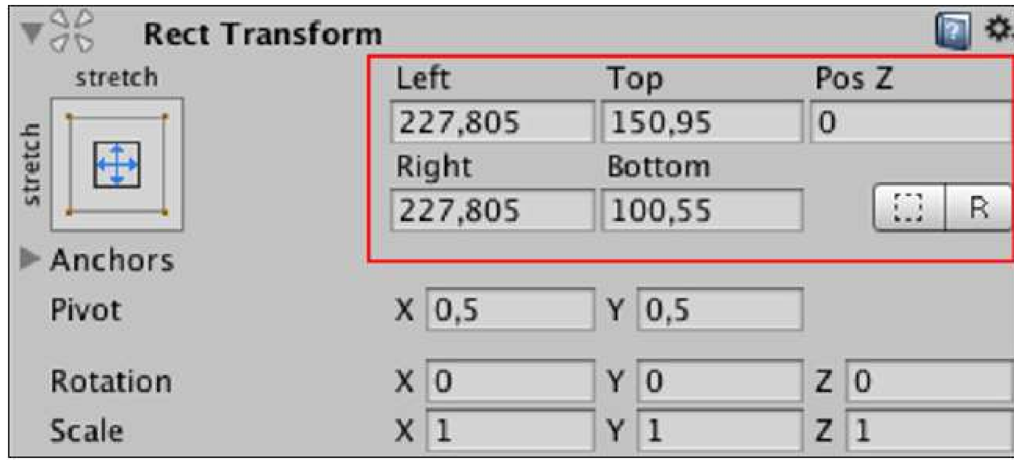


Fig. 14.56

El siguiente componente que debemos configurar es el de **Image**(**Script**) que por defecto lleva una imagen **Background** y en este caso he cambiado el color y he bajado la opacidad desde el canal alfa que se puede ver representada en la siguiente imagen con la linea negra que se ve debajo del parámetro color.

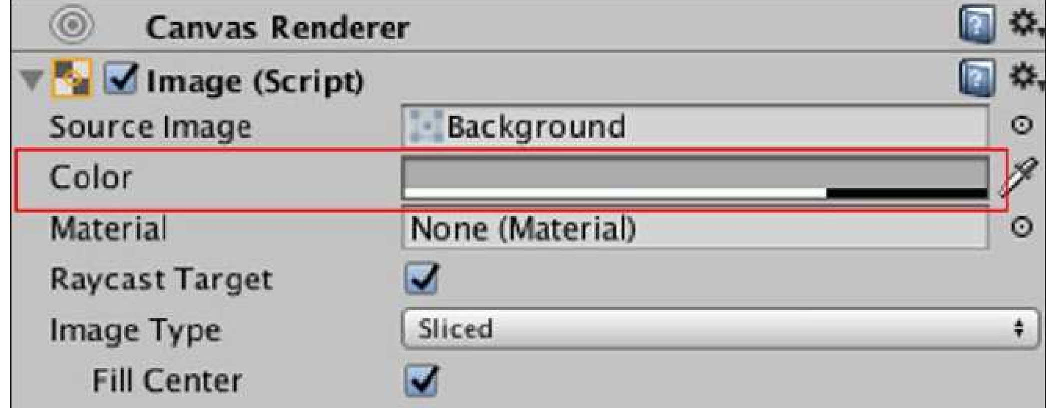


Fig. 14.57

Image

Para crear una imagen accedemos dentro de la ventana **Hierarchy** al botón **Create** y seleccionamos la opción **UI > Image**. Este lo arrastramos encima de Panel y automáticamente nuestra imagen pasará a ser hija de Panel.

Antes de pasar a configurar los parámetros de la Imagen debemos arrastrar la imagen con el nombre Titulo que encontrarás en la carpeta UI del proyecto de este capitulo dentro de Unity en la ventana **Project** encima de la carpeta UI. Al realizar esta acción ya tendremos acceso a esta imagen.

El siguiente paso es configurar la imagen para que sea un Sprite. Debes seleccionar la imagen accediendo a la ventana Inspector en el parámetro **Texture Type** seleccionamos la opción **Sprite(2D and UI)** y pulsamos la opción **Apply**

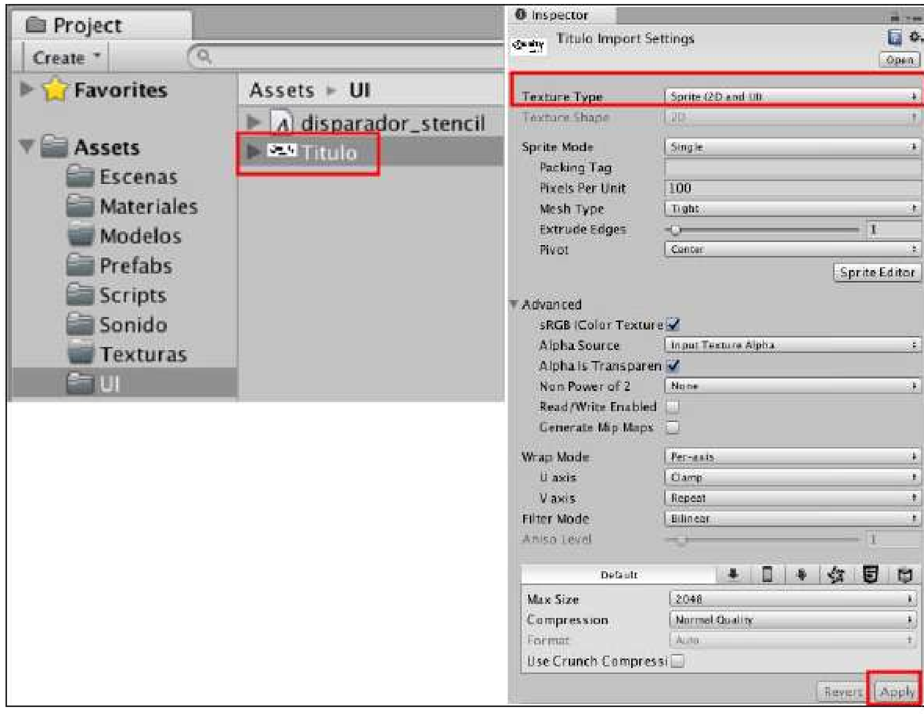


Fig. 14.58

Una vez tenemos la imagen preparada vamos a configurar las opciones del **Objeto Image**, para ello seleccionamos la **Image** desde la ventana **Hierarchy** y accedemos a la ventana inspector. En el **Rect Transform** te pongo una imagen con los valores que he puesto para colocar la imagen. Te recomiendo que pongas tus propios valores y que pruebes a crear tu propio menú.

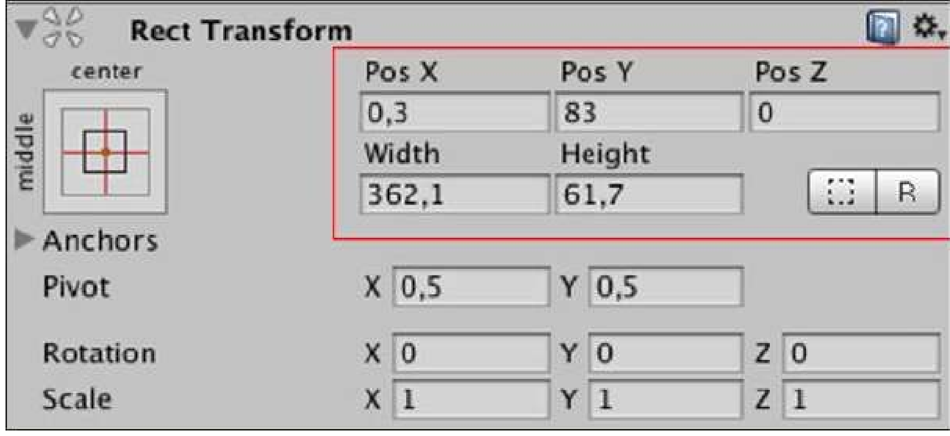


Fig. 14.59

En el componente **Image(Script)** ponemos en la imagen Titulo dentro del parámetro **Source Image** con el color blanco. Como te muestro en la siguiente imagen.

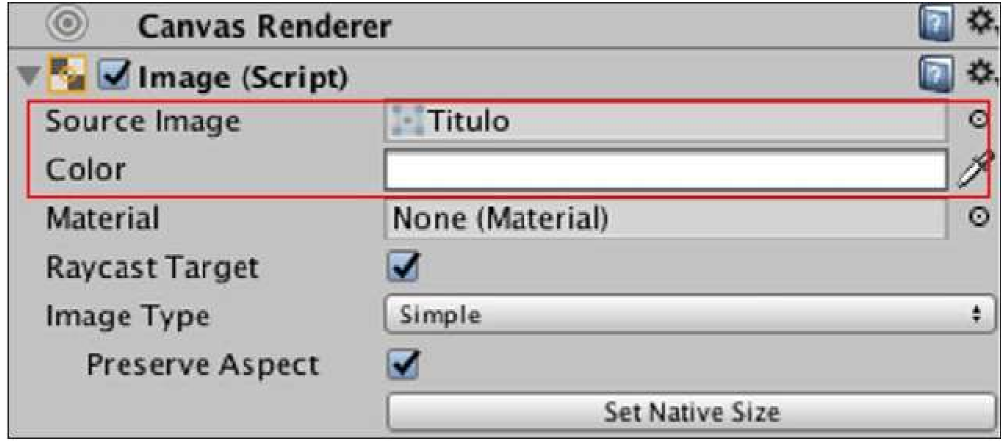


Fig. 14.60

Nuestro menú va tomando forma de manera que deberías poder ver algo parecido a la imagen que te muestro a continuación.

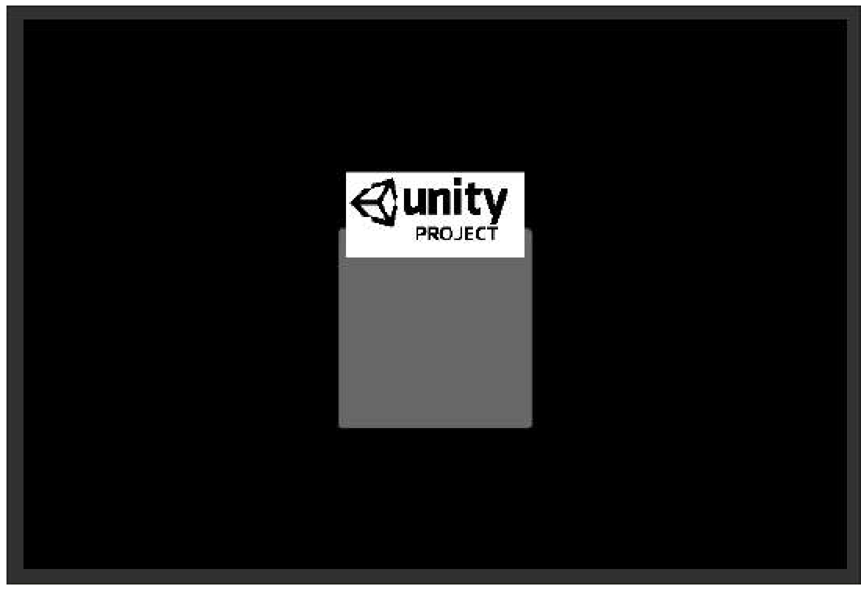


Fig. 14.61

Los botones

Para crear un botón accedemos dentro de la ventana **Hierarchy** al botón **Create** y seleccionamos la opción **UI > Button**. Este objeto botón lo arrastramos encima de Canvas y automáticamente nuestro **button** pasará a ser hijo de Canvas.

Dentro de la ventana Inspector vamos a cambiar el nombre de **Button** por el de Inicio y vamos a posicionar el botón con los valores que te muestro a en la siguiente imagen.

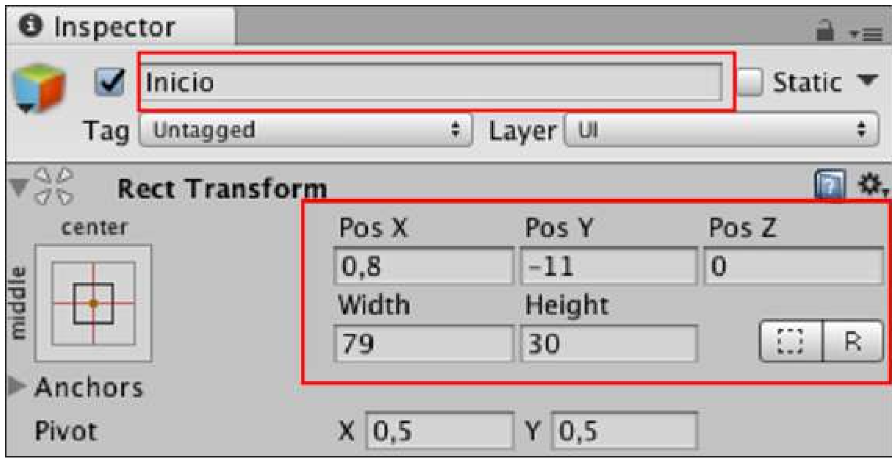


Fig. 14.62

El resto de parámetros los dejamos tal y como vienen por defecto. El siguiente objeto que vamos a configurar se encuentra dentro del objeto **Button** que es un objeto de tipo texto. Si accedemos a los componentes del Texto podemos cambiar el nombre y en el componente **Text** ponemos el nombre que se va a ver en el botón, en este caso **Start**. A continuación te dejo los valores que he utilizado.

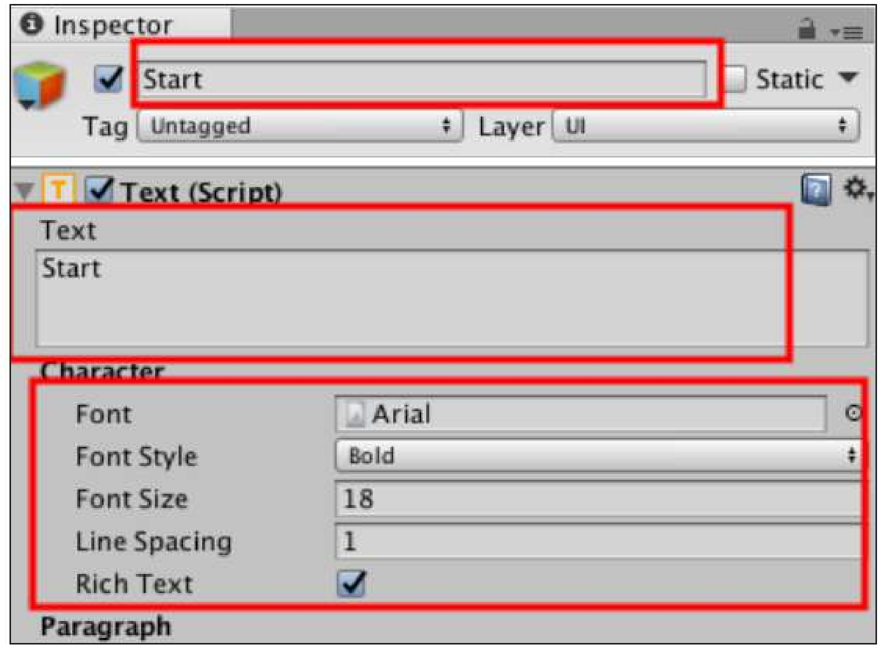


Fig. 14.63

Para crear el siguiente botón de una forma muy sencilla seria duplicar el que ya tenemos hecho y cambiarle los nombres y la posición. Para duplicar el botón dentro de la ventana **Hierarchy** seleccionamos **Inicio (button)** hacemos clic encima con el botón derecho del ratón y seleccionamos **Duplicate**. Ahora solamente debemos cambiar los nombres y la posición del nuevo botón que se ha creado.

Interactividad de los botones

Los botones tienen una componente dentro de la ventana Inspector que nos permite configurar con un método o función para interactuar mediante la selección de un script. Para entender mejor que estoy diciendo vamos a seleccionar el botón con nombre Inicio desde la ventana **Hierarchy** y luego accedemos a la ventana Inspector. Al final de la ventana Inspector dentro del componente **Button(Script)** encontramos un parámetro **On Click**() si pulsamos en el símbolo **(+)** se nos aparecerá un menú con la opción **Runtime Only** y debajo un caja de textos.

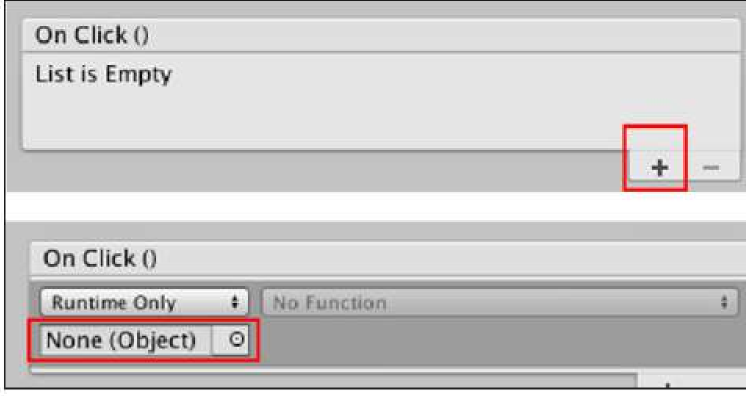


Fig. 14.64

En la caja donde pone **None** (**Object**) debemos hacer clic encima del símbolo con forma de diana o punto que encontramos al lado. Se nos abrirá una nueva ventana donde vamos a seleccionar el objeto que contiene el script con los métodos que queremos utilizar, en este caso es el objeto Canvas que contiene el script **Main_Menu**.

Una vez seleccionado el objeto debemos seleccionar que función queremos, para ello hacemos clic encima del menú con el nombre **No function**. Dentro del menú debemos seleccionar en este caso el script **Main_Menu >LoadLevel(string).**

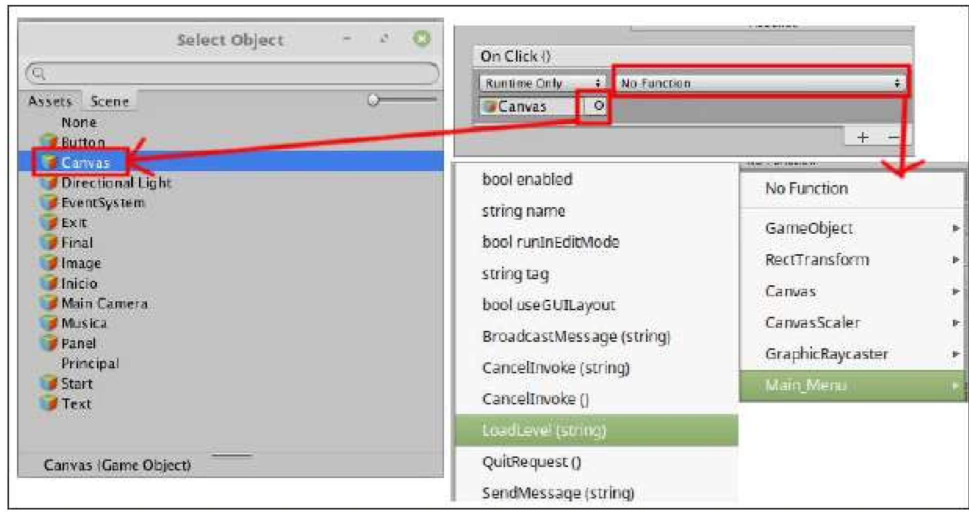


Fig. 14.65

Una vez seleccionado el método ahora solo tenemos que poner el nombre de la escena que queremos cargar. En otras palabras el botón inicio, tiene un evento **On Click** que reacciona cambiando de escena cuando pulsamos el botón. Debemos poner el nombre de la escena en la caja de texto como te muestro a continuación.

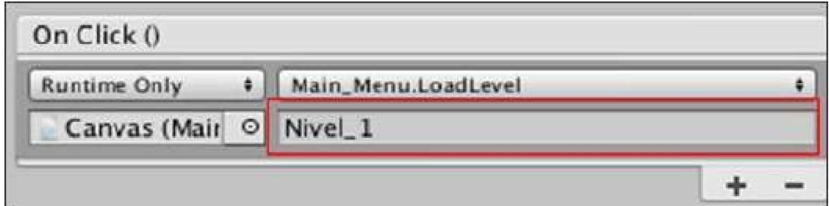


Fig. 14.66

Ahora vamos a realizar la misma acción para el segundo botón (Final). En este caso el método que debemos seleccionar no es **LoadLevel(string)**, debemos seleccionar **QuitRequest()** para que cuando pulsemos el botón cerremos la aplicación.

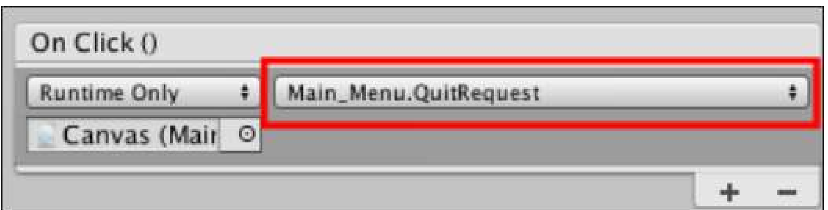


Fig. 14.67

Una vez hemos configurado los botones si pruebas o ejecutas la escena, comprobaras que no funcionan, porque todavía necesitamos configurar otra opción.

Primero accedemos al menú principal **File > Building Settings** o puedes acceder pulsando **Mayusculas + Ctrl + B.**

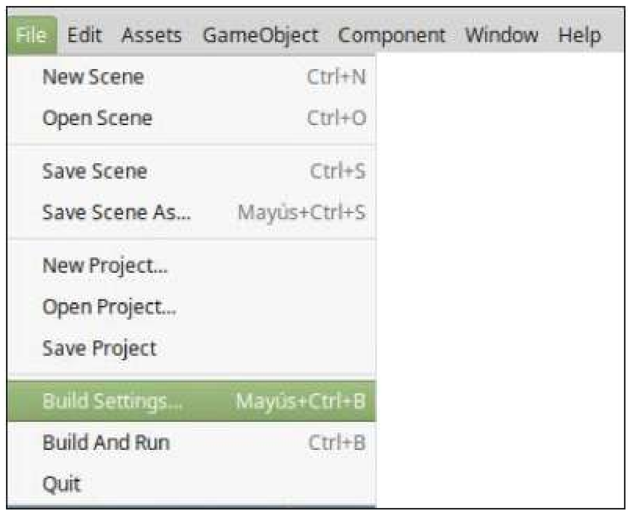


Fig. 14.68

Se nos abrirá una ventana en donde debemos arrastrar por orden, las escenas que tenemos en el proyecto. En este caso en la siguiente imagen veras tres escenas la que Escena Principal que es donde estamos, la escena **Nivel_1** que es donde tenemos el juego y una escena **GameOver** que todavía no hemos creado y veremos más adelante.

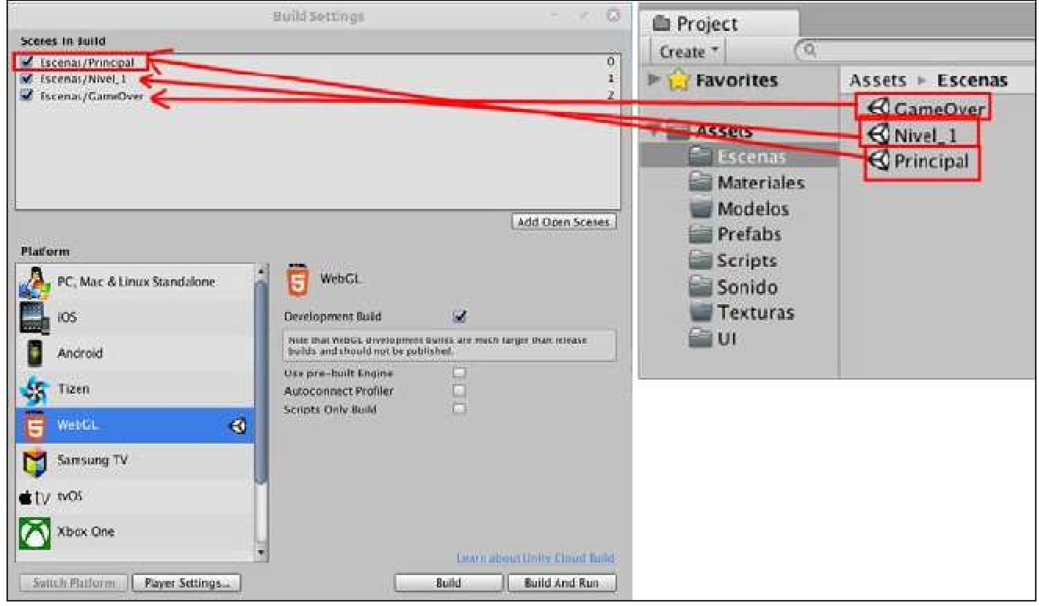


Fig. 14.69

Una vez arrastradas las escenas cierra la ventana **Build Settings,** Cuidado no pulses los botones **Build ni Build and Run** porque eso compilaría todo el proyecto para crear el juego que todavía no esta terminado.

Antes de comprobar que todo funciona correctamente vamos a seleccionar la cámara **Main Camera** desde la ventana **Hierarchy** y en los componentes de la ventana Inspector vamos a cambiar los siguientes parámetros. En el parámetro **Clear Flags** seleccionamos la opción **Solid Color** y en el parámetro **Background** seleccionamos el color negro.

Fig. 14.70

Ahora si ejecutamos la escena podemos comprobar como cuando pulsamos en el botón Start pasamos a la escena **Nivel_1**

Poner música al menú

Para poner música al menú en la escena debemos crear un **gameObject** vació al que le ponemos el nombre de **Musica** y le añadimos un componente **Audio Source.** En el parámetro **AudioClip** le añadimos el clip con el nombre **Musica.** Todo el material lo encontrarás en el material del capítulo en la carpeta **Sonidos.** A continuación te muestro una imagen con los parámetros que he configurado para el **GameObject Musica.**

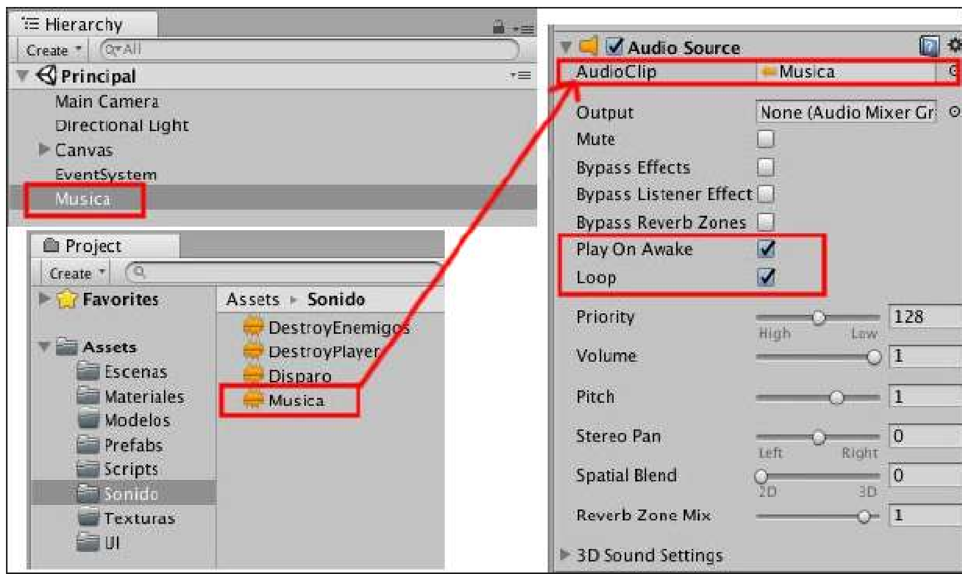


Fig. 14.71

Recuerda a guardar la escena y el proyecto. Ahora ya podemos ejecutar la escena y ver como suena la música.

Escena GameOver

Crea una escena nueva si no la tienes creada todavía o puedes guardar la escena Principal con otro nombre y eliminar todos los objetos de dentro del Canvas y el objeto música.

En el caso de que hayas creado una escena nueva, ponle el nombre de **GameOver**. Esta nueva escena va a contener un Canvas con un texto que diga **Game Over** en el centro de la escena y un botón donde ponga **Return** y nos lleve devuelta a la escena Principal. A continuación te muestro como debería quedar.

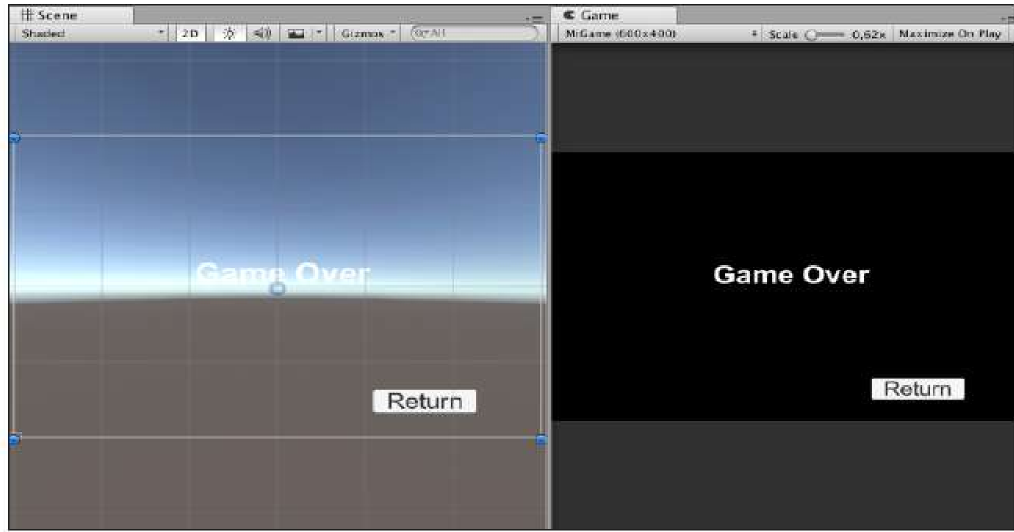


Fig. 14.72

En esta ocasión no es muy diferente de lo que hemos hecho en la escena Principal. Para empezar vamos a configurar la cámara **Main Camera** como lo hemos hecho en la escena principal, para que quede con un fondo oscuro.

Para crear nuestro menú **GameOver**, crearemos un Canvas primero y luego añadiremos los elementos **Button** y **Text. Al Canvas también debemos añadirle el script Main_Menu.**

Button (Volver)

A continuación te muestro como he configurado cada elemento de **Button**. En primer lugar he renombrado el elemento **Button** por el nombre Volver y en la ventana Inspector he configurado sus parámetros de la siguiente manera.

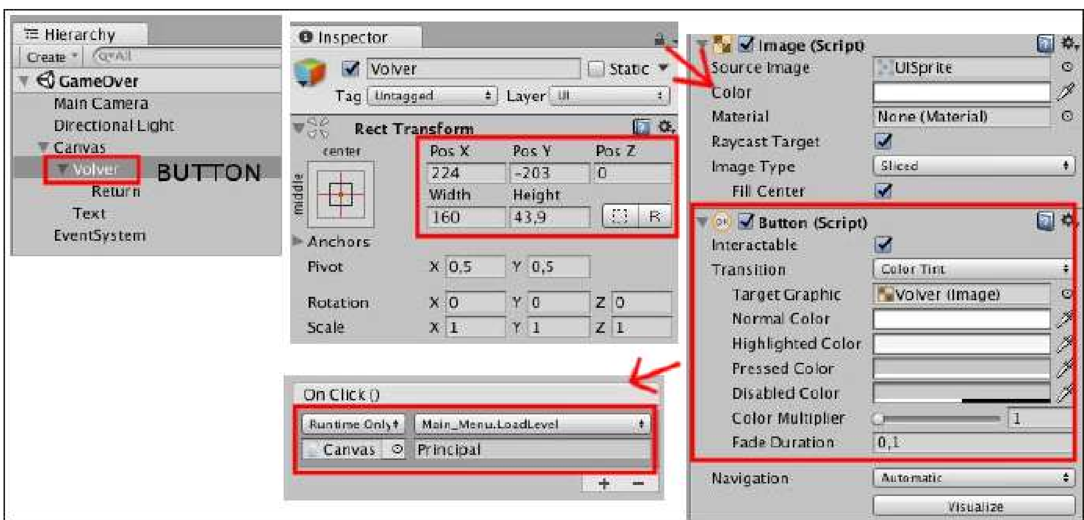


Fig. 14.73

Recuerda que en el evento **On Click()** debemos buscar primero el objeto que tiene nuestro script en este caso el Canvas y luego acceder desde el menú al método **LoadLevel** que hemos creado anteriormente y escribir el nombre de la escena a la que queremos acceder. También debes asegurarte de añadir esta escena en la ventana **Build Settings** accediendo desde el menú principal **File > Building Settings** o con el atajo de teclado **Mayusculas + Ctrl + B.**

En el elemento hijo del botón tenemos un texto que en la imagen he renombrado por **Return** a continuación te muestro los parámetros que he configurado en la siguiente imagen.

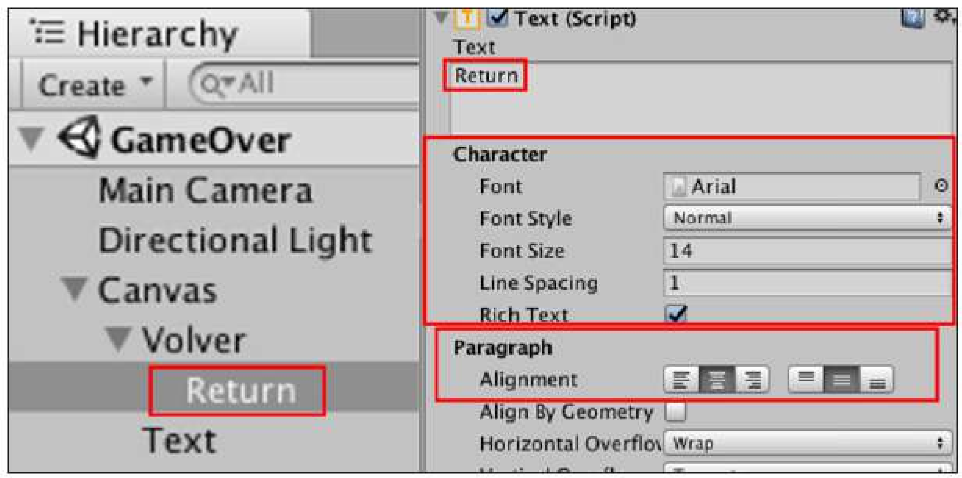


Fig. 14.74

Para finalizar tenemos que crear otro elemento en el canvas UI de tipo **Text.** En este caso no le he cambiado el nombre. Una vez creado accedemos a la ventana **Inspector** y configuramos sus parámetros como te muestro en la siguiente imagen.

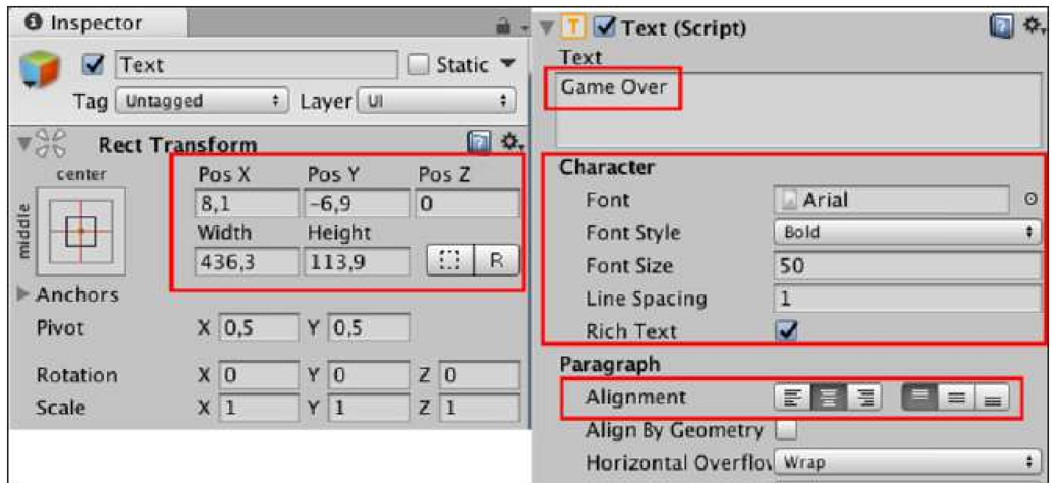


Fig. 14.75

Si todo es correcto cuando ejecutes la escena y pulses encima de el botón, se te debería cargar la escena Principal.

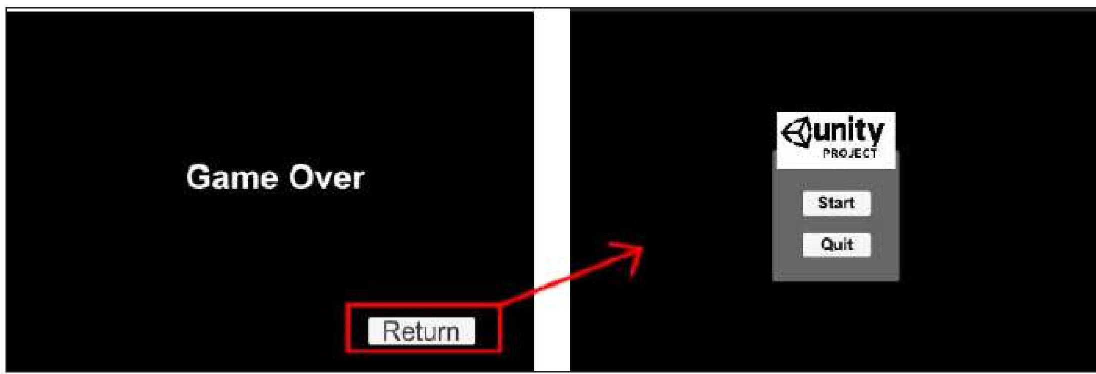


Fig. 14.76

De la escena Nivel_1 a la escena GameOver

Ahora que tenemos una escena **GameOver** debemos configurar la escena **Nivel_1** para que cuando nuestro player se le terminen las vidas pase directamente a la escena **GameOver**.

Esta parte es sencilla, primero la responsabilidad de que el juego termine recae en el player, así pues debemos abrir el script **Control_Player**. Dentro del script debemos crear una función que determine una condición que cumpla los siguiente:

```
public void GameOver()
{
        if (Control_Player.lives == 0)
        {
                SceneManager.LoadScene (“GameOver”);
                Control_Player.lives = 3;
                Control_Player.score = 0;
        }
}
```

Esta función la he llamado GameOver() y la condición es que cuando las vidas del player sean 0 que primero cargue la escena con el nombre “GameOver”, seguidamente le digo que las vidas vuelvan a ser 3 por si queremos volver a jugar y finalmente ponemos el valor de score a 0.
Esta función la debemos llamar dentro de la función **Update()**.
Otro aspecto muy importante a tener en cuenta es que si utilizamos SceneManager debemos cargar en la parte superior el **(using UnityEngine.SceneManagement;)**

```
void Update ()
{
        this.anchoMovimiento = Input.GetAxisRaw (“Horizontal”) * this.pla-
yerSpeed *      Time.deltaTime;
        this.miplayer.transform.Translate (Vector3.right * this.
anchoMovimiento);
```

```
        if (transform.position.x <= -7.5f) {
                transform.position = new Vector3 (7.4f, transform.position.y,
                        transform.position.z);
        }
        else if (transform.position.x >= 7.5f)
        {
                transform.position= new Vector3(-7.4f,transform.position.y,
                                transform.position.z);
        }

        if (this.activado == false)
        {
                gameObject.SetActive (false);
                Invoke ("ActivaAhora", 1.5f);
        }
        this.Disparar ();
        this.Contadores ();
        this.GameOver ();
}
```

En el codigo de arriba te muestro donde debes llamar a la función **GameOver();** dentro de la función Update()

Ahora guardamos el script, volvemos a Unity y guardamos la escena y el proyecto. Si todo es correcto puedes ejecuta la escena del juego y cuando pierdas todas las vidas automáticamente aparecerá la escena **GameOver**.

Compilar el proyecto

Unity te facilita mucho la vida para crear la aplicación de tus proyectos si volvemos a la ventana **Build Settings** accediendo desde el menú principal **File > Building Settings** o con el atajo de teclado **Mayusculas + Ctrl + B.**

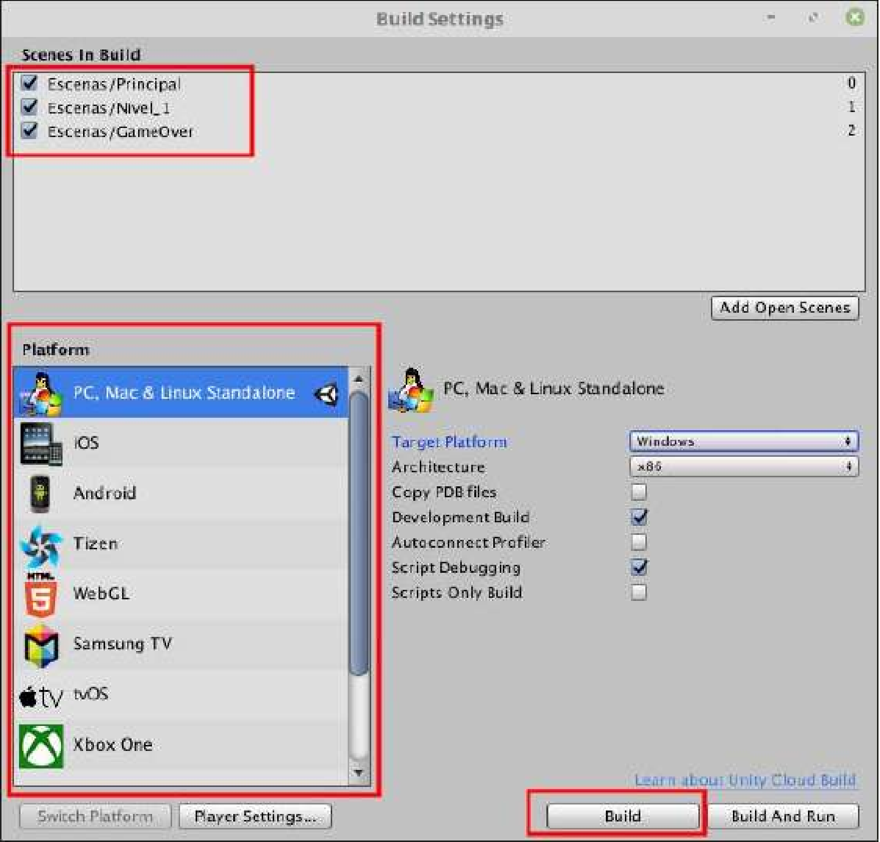


Fig. 14.77

En esta ventana verás una área llamada **Platform** en donde puedes seleccionar la plataforma para la que va destinado tu proyecto o juego. Una vez selecciones la plataforma que deseas, en el área derecha aparecerán las características correspondientes al tipo de plataforma, como son el sistema operativo, la arquitectura si es una CPU de 32-bits (x86) o de 64-bits (x86_64) etc..

Para compilar todo el proyecto solo debes pulsar el botón Build y guardar en tu sistema operativo el proyecto. Unity realizará unos cálculos durante un periodo de tiempo y una vez finalizado se creará un ejecutable de tu proyecto en la carpeta o lugar donde lo hayas guardado.

Ahora solamente debes probar el ejecutable y comprobar que todo funciona bien en mi caso personal lo he compilado en Linux y funciona perfectamente. Espero que el resultado sea el mismo para ti.

Conclusión

El objetivo principal de esta obra es acercarte al programa Unity de una forma sencilla intentando abarcar todos los sectores. Soy consciente de que es imposible llegar a profundizar en todos los temas que ofrece Unity, es por ese motivo que he intentado resumir de la forma en que un usuario novel empieza a descubrir el mundo de la programación.

Si es la primera vez que utilizas un programa como Unity, seguramente te sentirás perdido y a la vez emocionado, por ello se ha intentado crear un temario progresivo que a la vez sirva para ir asentando bases.

Si has seguido el temario por orden habrás descubierto que los proyectos en Unity son como una obra teatral en donde los escenarios y los actores son **gameObjects**, que tu como programador eres el director de una obra y tienes que escribir el guión (scripts) para cada actor que interactúa en ella.

Sabrás que cada actor tiene sus propias características (componentes) y que ese aspecto es el que diferencia unos de otros. Has aprendido a crear Actores principales (**players**) y aspectos que pueden interferir en su actuación (**Raycast, Navegacion, Triggers,** etc..).

En toda obra debes tener cuidado con la iluminación, la música y los efectos especiales (Partículas), que te proporcionarán un efecto visual impactante y crearán sensaciones a los espectadores.

En definitiva creo el desarrollo de un proyecto consta de muchas habilidades y la magia de Unity es que podemos contar historias de una forma interactiva, visual, auditiva y sensorial que con otras herramientas no podemos. Espero haberte ayudado a entender mejor este programa y a motivarte para seguir aprendiendo.

UNITY 3D

Si quieres aprender a crear tus propios videojuegos, con o sin conocimientos previos de programación, este libro es ideal para ti.

Los 14 capítulos comprendidos en este manual se centran en cada uno de los módulos básicos de Unity y te preparan de forma progresiva para comprender el programa.

Todo el contenido del libro se ve reforzado con prácticas explicadas paso a paso para que puedas seguir la lección sin problemas y construir videojuegos desde cero gracias a:

- El editor de Unity.
- La programación orientada a objetos con el lenguaje C#.
- Los scripts que permiten crear interacción.

Además, si quieres aprender sobre el modelaje en 3D, en el interior del libro encontrarás las instrucciones para descargar de forma gratuita los contenidos adicionales del libro.

No esperes más: consigue este libro y haz realidad todos tus proyectos con Unity 3D.

www.alfaomega.com.mx
atencionalcliente@alfaomega.com.mx

ÁREA: Computación
SUBÁREA: Programación. Lenguajes y Técnicas

Te acerca al conocimiento