Universidad de Valladolid

E. U. de Informática (Segovia)

Ingeniería Técnica en Informática de Gestión

UN ESTUDIO ACERCA DEL DESARROLLO DE VIDEOJUEGOS MEDIANTE EL MOTOR GRÁFICO UNITY 3D

Alumna: Eva Gil Romero

Tutor: Fernando Díaz Gómez

Luis Mª Fuentes García

Agradecimientos

En primer lugar, doy las gracias a mis padres por todo el apoyo que me han dado a lo largo de todos estos años de estudios, en los buenos y en los malos momentos y a mis abuelos por estar siempre ahí y darme su cariño.

En segundo lugar a mi tutor de Proyecto Fin de Carrera Fernando Díaz por toda la ayuda que me ha proporcionado, por su amabilidad y su disponibilidad siempre que le he requerido y a Luis Mª Fuentes García por su ofrecimiento e interés demostrados.

También dar las gracias a todos los profesores que he tenido a lo largo de la carrera por su dedicación y su ayuda que han sido muy importantes para mí y a todos mis compañeros por colaborar a que hubiera un buen ambiente de estudio y por su solidaridad.

Por último hacer una mención especial y con mucho cariño a todas aquellas amistades fuera del ámbito de la facultad con las que paso muy buenos ratos y se han interesado mucho por la evolución de mi proyecto (María, Elena, Marisa, Toñi, Loli, etc.).

Índice

1.-Introducción 1
1.1.-Identificación del proyecto 1
1.2.-Organización de la memoria 1
1.3.-Motivación 2
1.4.-Objetivos 3
2.-Motor de videojuegos (*Game Engine*) 5
2.1.-Definición 5
2.2.-Breve historia de los motores de videojuegos 5
2.3.-Motores más populares 10
3.-Motor Unity 3D 11
3.1.-Historia de Unity 11
3.2.-Licencias 11
3.3.-Iniciativas de democratización 12
3.4.-Juegos hechos con Unity 13
4.- Descripción del motor gráfico Unity 3D 15
4.1.- Instalando Unity 15
4.2.- La interfaz de usuario de Unity 19
4.2.1.- Vista de escena 20
4.2.2.- Vista de proyecto 22
4.2.3.-Vista de jerarquía 22
4.2.4.-Inspector 23
4.2.5.-Vista de juego 23
4.3.-Primeros pasos 25
4.3.1.-Crear un proyecto 25
4.3.2.-La primera escena. 26
4.3.3.-Localizando objetos 26
4.3.4.-Crear Objetos 26
4.3.5.-Mover, rotar y redimensionar objetos 27
4.3.6.-Utilizar recursos 28
4.3.7.-Añadir componentes 29
4.3.8.-Duplicar objetos 29

4.4.-Terrenos 30
4.4.1.-Creando un terreno 30
4.4.2.-Configuración básica de un terreno 31
4.4.3.-Herramientas de terreno 33
4.4.4.-Añadir un cielo 48
4.4.5.-Añadir niebla 49
4.4.6.-Color de ambiente 50
4.4.7.-Añadir luces 50
4.4.8.-Añadir Agua 51
4.4.9.-Añadir un Controlador 53
4.5.-Game Objects 54
4.5.1.-Concepto de objeto de juego 54
4.5.2.-Objetos vacíos 54
4.5.3.-Resto de objetos básicos 54
4.5.4.-Mover y alinear con la vista 71
4.6.-Físicas 71
4.6.1.-Rigidbody 71
4.6.2.-Fixed Joint e Hinge Joint 73
4.7.- Shaders 77
4.7.1.-Ejemplo Cuero 77
4.8.-Animación 86
4.8.1.-La vista de animación 86
4.8.2.-Animar de forma gráfica 90
4.8.3.-Curvas de animación 91
4.8.4.-Animaciones de cámara 92
4.9.-Programación en Unity 94
4.9.1.-Introducción a la programación orientada a objetos. 94
4.9.2.-Introducción a los *scripts* de Unity 3D 95
4.9.3.-Introducción al *scripting* con Javascript 99
4.9.4.-Tutorial Transform 103
4.9.5.-Tutorial Input 117
4.9.6.-Tutorial GUI 128
4.9.7.-Tutorial Object 147
4.10.-*Triggers* y colisiones 154

4.10.1.-Tutorial *Triggers* 154
4.10.2.-Tutorial Colisiones 156
5.-Prototipo videojuego de disparos (*Shooter*) 159
5.1.-Especificación 159
5.2.-Pasos seguidos en el desarrollo del juego 159
5.2.1.-Definir la escena (elementos pasivos) 159
5.2.2.-Definir los elementos activos de la escena 163
5.2.3.-Control global de la escena 165
5.3.-Controlador del juego 167
5.4.-Compilar nuestro juego para PC 174
6.-Posibles ampliaciones 177
7.-Conclusiones 179
Bibliografía 181
Glosario de términos 183
Estructura del CD 186

1.-Introducción

1.1.-Identificación del proyecto

Título: Un estudio acerca del desarrollo de videojuegos mediante el motor gráfico Unity 3D.

Autor: Eva Gil Romero.

Tutor oficial o principal: Fernando Díaz Gómez

Departamento: Informática

Área: Ciencias de la Computación e Inteligencia Artificial

Tutor 1: Luis Mª Fuentes García

Departamento: Física Aplicada

Área: Física Aplicada

1.2.-Organización de la memoria

1. Introducción: El primer apartado recoge la identificación del proyecto, la explicación de los puntos principales del proyecto, es decir, la organización de la memoria, los motivos que me impulsaron a desarrollar este proyecto y por último, los objetivos del mismo.

2. Motor de videojuegos (Game Engine): El segundo apartado recoge la definición de motor de videojuegos o game engine, una breve historia de los motores de videojuegos y por último cuáles son los motores más populares.

3. Motor Unity 3D: El tercer apartado recoge un resumen de la historia de Unity, la descripción de las dos principales licencias de Unity, las iniciativas de democratización y ejemplos de juegos hechos con unity.

4. Descripción del motor gráfico Unity 3D:

Este apartado, el más extenso, engloba los siguientes puntos:

- Instalando Unity: Como su propio nombre indica, en este apartado se describen los pasos necesarios para instalar Unity.
- La interfaz de usuario de Unity: En este apartado se describen detalladamente cada una de las vistas de la interfaz de Unity 3D.
- Primeros pasos: En este apartado se pretende introducir al usuario en el manejo de las herramientas y funcionalidades básicas de Unity tales como crear un proyecto, crear una escena, localizar objetos, crear objetos, etc.
- Terrenos: En este apartado se explica cómo crear un terreno, su configuración básica, se describe cada una de las herramientas de edición de terrenos y se recogen algunos

tutoriales sobre elementos que podemos añadir a un terreno, como cielo, niebla, agua, luces, etc.

- Game Objects: En este apartado se describe lo que es un Game Object y los principales Game Objects que incorpora Unity y además un pequeño tutorial de cómo mover los objetos y alinearlos con la vista.
- Físicas: En este apartado se ilustra mediante ejemplos la utilización de las distintas físicas que podemos encontrar en Unity.
- Shaders: En este apartado se explica que es un shader y su utilización mediante un ejemplo.
- Animación: En este apartado se describe la vista de animación, como animar de forma gráfica, las curvas de animación y las animaciones de cámara.
- Programación en Unity: En este apartado se hace una introducción a la programación orientada a objetos, a los scripts en Unity 3D y al scripting con JavaScript, además una descripción detallada de algunas de las clases más importantes de Unity.
- Triggers y colisiones: En este apartado se explica que son los Triggers y las colisiones y cómo utilizarlos.

5. Prototipo Videojuego de disparos (Shooter): Apartado donde se recoge una breve descripción del prototipo y se explica cómo se han desarrollado las distintas partes de este.

6. Posibles ampliaciones: Como su nombre indica, las posibles ampliaciones que se pueden añadir al proyecto.

7. Conclusiones: Las conclusiones a las que se ha llegado después de haber realizado este proyecto y del aprendizaje de Unity.

1.3.-Motivación

Durante los últimos 30 años, los videojuegos han pasado de ser aquellas máquinas que ofrecían un solo juego a "juegos de rol en línea multijugador" en los que participan millones de jugadores. Actualmente, el mundo de los videojuegos es un extraordinario negocio mediático, y los productos que copan las listas de más vendidos generan por las ventas durante la primera semana ingresos superiores a los de las películas más taquilleras.

De acuerdo con las estimaciones de los estudios llevados a cabo por Gartner, la industria mundial de los videojuegos —incluidos los programas, los equipos y los juegos en línea— pasará de generar 74.000 millones de dólares en 2011 a 112.000 millones en 2015, cifras que superan con creces los ingresos totales de la industria cinematográfica, que en 2010 recaudó 31.800 millones de dólares. Gartner aventura asimismo que, en 2015, el gasto en videojuegos superará el gasto en equipos informáticos.

De ser un ámbito dominado por chicos y jóvenes, el público de los videojuegos se ha ampliado y actualmente incluye a chicas, mujeres jóvenes y personas de edad. El aumento en el número de jugadores dará cada vez más relevancia al análisis del impacto de los videojuegos no sólo en el desarrollo económico, sino también en la propia sociedad.

Crear un videojuego es una tarea compleja y requiere años de trabajo a equipos de decenas de personas, con mucha experiencia y con un gran presupuesto. No existen libros, atajos, trucos ni herramientas para hacer un juego. Hasta el juego más sencillo hoy día exige un buen conocimiento en diferentes disciplinas si se pretende llevar a cabo por uno mismo, además de un gran esfuerzo y voluntad de aprender constantemente.

Un motor de videojuegos o *engine* es una herramienta creada para facilitar y agilizar en gran medida el proceso de creación de un videojuego. La oferta actual de motores es bastante amplia y no es

difícil encontrar un motor que se ajuste a las necesidades particulares del proyecto. A la hora de escoger hay que tener en cuenta ciertos aspectos como el presupuesto, el tipo de proyecto, la plataforma de destino (consola, PC, *smartphone*…) o las intenciones de comercializar el producto final. Supongamos que nuestro presupuesto inicial es bajo (o nulo) y no podemos permitirnos desembolsar una gran suma de dinero. Por suerte, tenemos a nuestra disposición algunos motores de última generación valorados en millones de dólares por un precio de 0 euros**.**

Dentro de esos motores tenemos Unity 3D. Unity es un motor desarrollado por *Unity Technologies* que se está abriendo paso en la comunidad por su sencillez de uso. Uno de sus mayores fuertes es la facilidad para desarrollar en diversas plataformas, permitiendo crear aplicaciones para consolas, iOS, Android, Linux, Windows…

El desarrollo con Unity es sencillo gracias a su interfaz y que el motor está diseñado para un uso muy amigable. Además la comunidad de desarrolladores es enorme y hay multitud de foros de ayuda, así como una extensa documentación.

1.4.-Objetivos

El objetivo principal de este proyecto se centra en el estudio del desarrollo de videojuegos mediante el motor gráfico Unity 3D. En este estudio no se recogen ciertos conocimientos sobre física, matemáticas, etc., que los usuarios de esta herramienta deberían tener ya que no quería extender el trabajo en exceso.

Este objetivo principal se divide en tres puntos:

- La descripción de las funcionalidades y capacidades de cada una de las herramientas incluidas en el motor.
- La descripción del papel que juega cada herramienta en el proceso de desarrollo de un videojuego.
- La ilustración de su utilización en este contexto, mediante el desarrollo de un pequeño prototipo de videojuego con este motor.

2.-Motor de videojuegos (*Game Engine*)

2.1.-Definición

Un motor de videojuego es un término que hace referencia a una serie de rutinas de programación que permiten el diseño, la creación y la representación de un videojuego. La funcionalidad básica de un motor es proveer al videojuego de un motor de renderizado para los gráficos 2D y 3D, motor físico o detector de colisiones, sonidos, *scripting*, animación, inteligencia artificial, redes, *streaming*, administración de memoria y un escenario gráfico. El proceso de desarrollo de un videojuego puede variar notablemente por reutilizar o adaptar un mismo motor de videojuego para crear diferentes juegos.

2.2.-Breve historia de los motores de videojuegos

Antiguamente los juegos solían programarse desde cero, sin software intermediario. Esto era posible por la menor complejidad de los juegos, pero a menudo resultaba poco práctico: había muy pocas herramientas de desarrollo específicas.

Figura 2.1 Herramienta de creación de juegos SEUCK (Shoot'Em-Up Construction Kit)

Fueron los juegos de disparos en primera persona (*shooters*) los que impulsaron el desarrollo de los primeros motores gráficos 3D y motores de juegos. A mediados de los ochenta, Incentive Software creaba Freescape, un motor 3D que usó para sus juegos.

Figura 2.2 Iglesia creada con 3D Construction Kit que usaba el motor gráfico Freescape

Algo más tarde, a principios de los noventa, aparecieron *shooters* 3D muy famosos, como Wolfenstein 3D y Doom. No se hablaba todavía de 3D, sino de 2.5D, mundos aparentemente tridimensionales, pero cuya esencia era bidimensional.

Figura 2.3 Imagen de Wolfenstein 3D

Los motores más famosos de la época fueron el id Tech 1 -usado por Doom, Heretic y Hexen-, y el Build Engine, usado por Duke Nukem 3D, Blood y Shadow Warrior, entre otros. Otro motor popular fue Jedi Engine, usado por el clásico Dark Forces**.**

Figura 2.4 Imagen de Dark Forces

Por aquel entonces los juegos 3D ya se habían popularizado, pero muchos de ellos no usaban un motor gráfico concreto. En 1996**,** fue de nuevo id Software quien revolucionó el panorama de los motores gráficos 3D con Quake y su motor 3D**.**

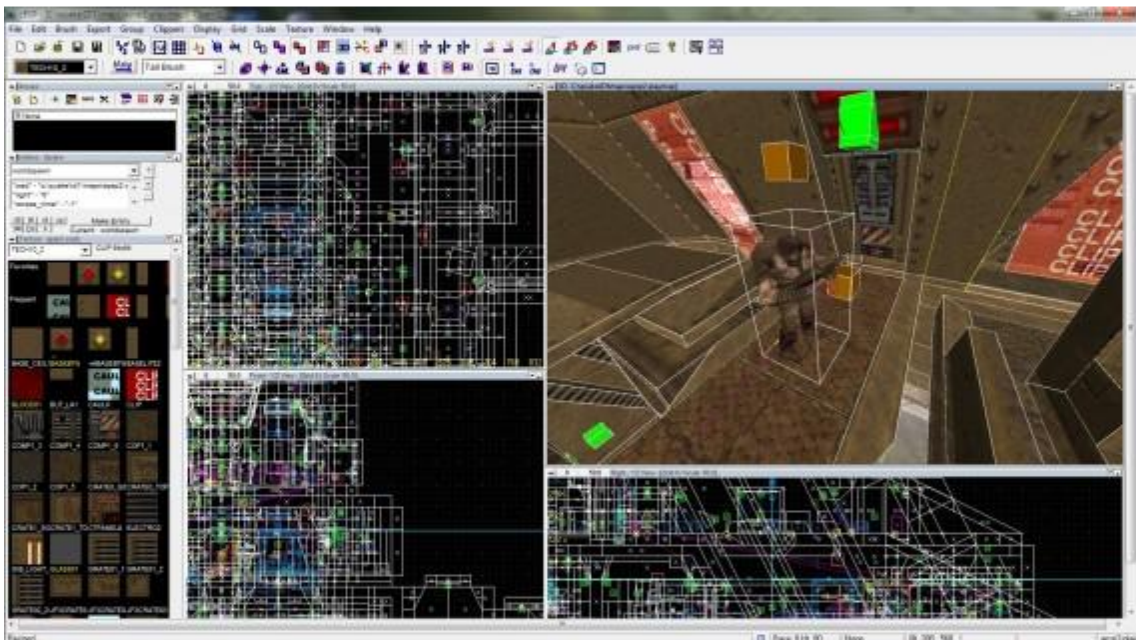


Figura 2.5 Quake engine

El motor de Quake fue de los primeros en aprovechar las primeras tarjetas aceleradoras 3D**,** como las míticas 3dfx. En 1998, una versión derivada del motor de Quake, GoldSrc, se usó para dar vida a otro clásico, Half-Life.

Figura 2.6 Counter Strike usaba el motor GoldSrc

A finales de los noventa el rey de los motores gráficos fue Unreal Engine, el motor 3D que se hallaba bajo el "capó" de Unreal y Unreal Tournament. Una de sus características más revolucionarias fue la posibilidad de crear modificaciones.

Figura 2.7 Diferencia de calidad entre las versiones 1, 2 y 3 del motor Unreal

Los motores 3D ya establecidos –Unreal Engine, id Tech- siguieron luchando durante años por la supremacía. Juegos como Doom 3 supusieron saltos enormes para los gráficos 3D en los videojuegos. Pero entonces llegaron dos nuevos contendientes.

Figura 2.8 Imagen de Half-Life 2

Valve lanzó en 2004 un motor propio, Source Engine, usado por Half-Life 2 y alabado por la calidad de las animaciones faciales. Ese mismo año llegó el increíble motor CryEngine, usado para crear Far Cry**.**

2.3.-Motores más populares

A día de hoy, hay muchos motores gráficos y de juego, pero solo un puñado gozan de gran popularidad. Entre ellos, cabe destacar:

Id Tech 5 (id Software)

- Rage (2011)
- Wolfenstein*:* The New Order (2014)
- Doom 4 (sin fecha)

CryEngine 3 (CryTek)

- Crysis 2 (2011)
- Crysis 3 (2013)
- State of Decay (2013)

Anvil (Ubisoft)

- Assassin's Creed Revelations (2011)
- Assassin's Creed III (2012)
- Assassin's Creed IV (2013)

RAGE (Rockstar Games)

- Grand Theft Auto IV (2008)
- Max Payne 3 (2012)
- Grand Theft Auto V (2013)

Source (Valve)

- Half-Life 2 (2004)
- Portal 2 (2011)
- Dota 2 (2013)

Unity (Unity Technologies)

- Slender: The Arrival (2013)
- Surgeon Simulator 2013
- Kerbal Space Program (2013)

Unreal Engine (Epic Games)

- Gears of War 3 (2011)
- Borderlands 2 (2012)
- Mass Effect 3 (2012)

3.-Motor Unity 3D

3.1.-Historia de Unity

La aventura de Unity Tecnhologies empezó en el año 2004 cuando David Helgason, Nicholas Francis y Joachim Ante decidieron dar un vuelco a su compañía de desarrollo de videojuegos tras el fracaso de 'GooBall'. El juego no había tenido el éxito esperado pero en su desarrollo habían creado unas herramientas muy potentes que sirvieron como semilla para una idea que rondaba la cabeza del equipo: democratizar el desarrollo de videojuegos.

Crear un motor de videojuegos que pequeñas y grandes empresas pudieran utilizar por igual. Un entorno amigable para programadores, artistas y diseñadores que llegase a diferentes plataformas sin obligar a programar el juego específicamente para cada una de ellas. Sí, una especie de utopía hace diez años y que sin embargo, a día de hoy, se ha convertido en realidad.

El motor llegaría solo a Mac en principio y se presentaría en dos versiones, *Indie* y Profesional. La primera representaba un punto de acceso económico para los pequeños estudios que estaban empezando y no podían permitirse un dispendio considerable, tenía muchas funciones y su precio empezaba en 300 dólares. La versión Pro, de 1.500 dólares de coste, traía todas las funciones del motor. Es importante señalar que incluso en la versión *Indie* del motor se permitía vender el producto resultante. De momento el motor funcionaba bien pero estaba lejos de ser una alternativa al mismo nivel de los grandes.

En 2008 llegaría el gran salto al aprovechar el lanzamiento del iPhone, la fiebre que desató y la compatibilidad con la plataforma. Un poco más tarde llegaría la versión para Android. El auge de Unity era imparable. En 2009 la compañía dio el espaldarazo definitivo a su estrategia. La versión *Indie* desaparecía como tal y se convertía en gratuita para todo aquel que quisiera iniciarse en la plataforma. Incluso con esa versión, la gratuita, se permitía distribuir el juego.

Los desarrolladores independientes no tardaron en abrazar la idea y convirtieron a Unity en uno de los motores gráficos más usados. Eso sí, todavía faltaba dar un salto de calidad para conquistar a los estudios de desarrollo que exigían más calidad gráfica. Un salto que llegó con la versión 4.0 y que demuestran los acuerdos cerrados con Sony, Microsoft y Nintendo para que el motor sea compatible con sus sistemas.

La versión actual es compatible con muchísimas plataformas (PC, Mac, Linux, iOS, Android, BlackBerry, PlayStation, Xbox, Wii, Wii U, Web…) y el sueño de desarrollar una sola vez, darle a publicar y olvidarte de problemas está más cerca que nunca.

3.2.-Licencias

Hay dos licencias principales para desarrolladores: Unity y Unity Pro, que está disponible por un precio ya que la versión Pro no es gratis. Originalmente costaba alrededor de 200 dólares (ahora $1500). La versión Pro tiene características adicionales, tales como *render to texture o texture baking*, determinación de cara oculta, iluminación global y efectos de post-procesamiento. La versión gratuita, por otro lado, muestra una pantalla de bienvenida (en juegos independientes) y una marca de agua (en los juegos web) que no se puede personalizar o desactivar.

Tanto Unity como Unity Pro incluyen el entorno de desarrollo, tutoriales, ejemplos de proyectos y de contenido, soporte a través de foros, wiki, y las actualizaciones futuras de la misma versión principal (es decir, la compra Unity Pro 3 obtiene todas las futuras actualizaciones de Unity Pro 3.x gratis).

Unity para Android, Unity para iOS, Unity para Adobe Flash Player, y pronto Unity para teléfonos con Windows 8 son complementos para una compra de Unity. Es obligatorio el certificado de Unity Pro para comprar licencias Pro para Android o iOS. Las licencias normales de Android e iOS se pueden utilizar con la versión gratuita de Unity.

El código fuente, PlayStation 3, Xbox 360, Wii y licencias se negocian caso por caso.

Las licencias educativas son proporcionadas por Studica con la estipulación de que es para la compra y uso de las escuelas, exclusivamente para la educación.

Desde la versión 4.0, un nuevo modelo de licencia se puso en marcha para organizaciones de juegos de azar. Deben ponerse en contacto con Unity directamente para obtener una licencia de distribución. Esta licencia se encuentra en el nivel de la distribución, no el nivel de desarrollador.

3.3.-Iniciativas de democratización

Para reforzar su democratización en el desarrollo del juego, Unity Technologies invierte en iniciativas que las considera como vías para ayudar a capacitar a los desarrolladores mediante la ampliación de sus capacidades y el alcance del cliente.

Asset Store

En noviembre de 2010 se lanzó el *Unity Asset Store* que es un recurso disponible en el editor de Unity. Más de 150.000 usuarios de Unity pueden acceder a la colección de más de 4.400 paquetes de *Assets* (recursos) en una amplia gama de categorías, incluyendo modelos 3D, texturas y materiales, sistemas de partículas, música y efectos de sonido, tutoriales y proyectos, paquetes de scripts, extensiones para el editor y servicios en línea.

La *Store* es el hogar de muchas extensiones, herramientas y paquetes de *assets*, como el paquete NGUI*:* Next-Gen UI por Tasharen Entertainment, y la extensión de scripting visual uScript por los estudios de Detox. Tile Mapper Tidy, creador de juegos 2D/3D basado en tiles de Doppler Interactive y los paquetes de scripts de entrada de FingerGestures.

inXile Entertainment ha sido vocal en el uso de la *Asset Store* para la producción de Wasteland 2.

Union

Union es una división de Unity Technologies dedicada a la sindicación de los juegos de Unity para teléfonos móviles, tiendas de aplicaciones, tabletas, decodificadores, televisores conectados y otras plataformas emergentes. Con el objetivo de democratizar la distribución de juegos, *Union* trabaja con desarrolladores de Unity en licencias de juegos para el lanzamiento en los nuevos dispositivos.

Union incluye una gama de más de 125 juegos que ha generado un total acumulado de 120 millones de descargas en sus comunicados colectivos. Ejemplos de títulos de *Union* incluyen Shadowgun, Super Crossfire HD, Forever Frisbee, Falling Fred, y Cordy.

Union ofrece a los socios de la plataforma acceder a los juegos mientras fortalecen a los desarrolladores de Unity con nuevas oportunidades de distribución. *Union* es libre de unirse y proporciona el 80% de cuota de ingresos de sus desarrolladores.

Las plataformas de *Union* incluyen Intel, LGTV, Roku, BlackBerry, Nokia, Sony y Lenovo.

3.4.-Juegos hechos con Unity

Dead Frontier (2008)

Three Kingdoms Online (2008)

Cartoon Network Universe: FusionFall (2009)

Max & the Magic Marker (2010)

Thomas Was Alone (2010)

Battlestar Galactica Online (2011)

Family Guy Online (2011)

I Am Playr (2011)

Rochard (2011)

Nihilumbra (2012)

Dead Lab (2012)

No Heroes (2013)

Shadowgun (2011)

Triple Town (2011)

Bad Piggies (2012)

Dead Trigger (2012)

Endless Space (2012)

Escape Plan (2012)

Guns of Icarus Online (2012)

MechWarrior Tactics (2012)

Prime World (2012)

Slender: The Eight Pages (2012)

Temple Run (2012)

Among the Sleep (2013)

Game of Thrones: Seven Kingdoms (2013)

Hearthstone: Heroes of Warcraft (2013)

Interstellar Marines (2013)

République (2013)

Wasteland 2 (2013)

Project Eternity (2014)

Rust (Alpha 2013, TBD)

Castle Story (TBD)

War for the Overworld (2013)

Dreamfall Chapters: The Longest Journey (2014)

Kerbal Space Program (TBD)

Shadowrun Online (TBD)

Kartrider Dash (Facebook)

Red Crucible 2 (Facebook)

Torment: Tides of Numenera

Unearthed: Trail of Ibn Battuta (TBD)

Slender: The Arrival (2013)

4.- Descripción del motor gráfico Unity 3D

4.1.- Instalando Unity

En primer lugar, si queremos usar el motor Unity 3D tendremos que instalarlo. Entraremos en la página oficial de Unity: http://unity3d.com/

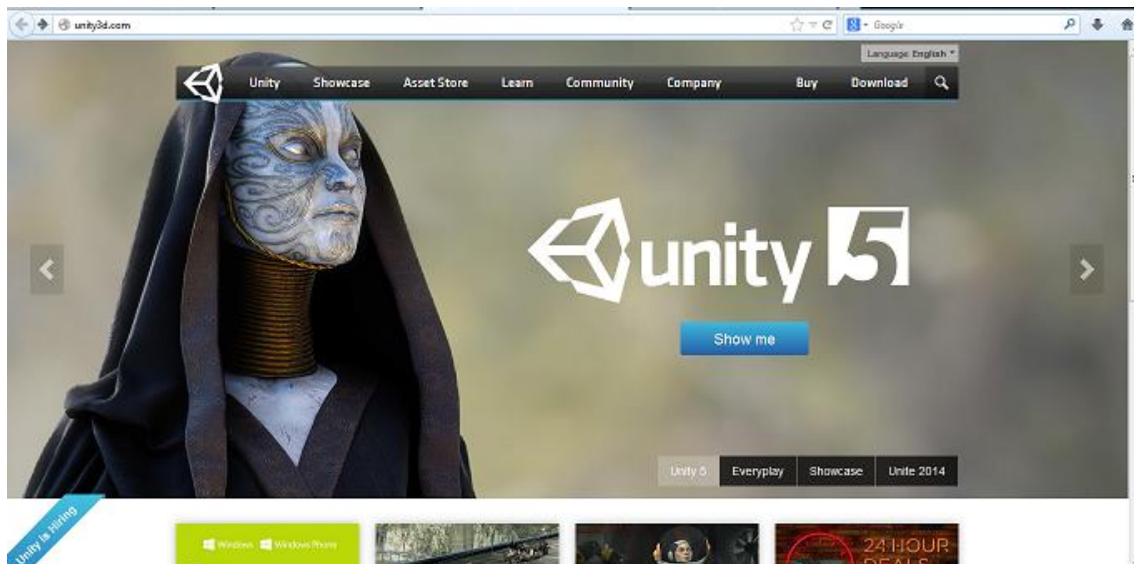


Figura 4.1 Página oficial de Unity

Por un lado está la versión pro que es de pago y por otro lado tenemos la versión gratuita que es la que he utilizado para este proyecto. Para bajarnos esta última tendremos que ir a `Dowload`:

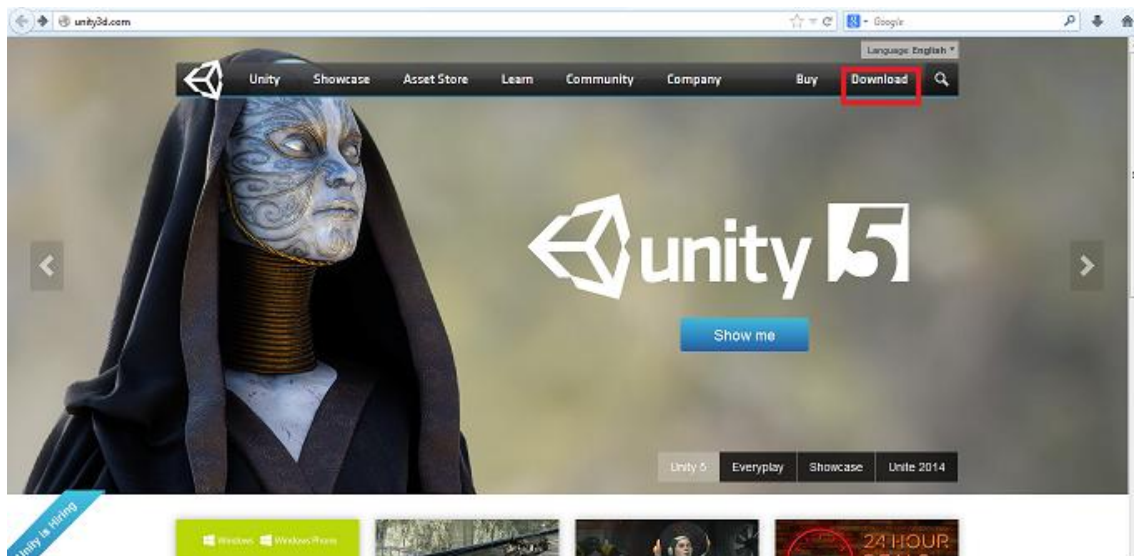


Figura 4.2 Botón `Dowload`

Dentro de la pantalla de `Dowload` nos bajaremos la versión de Unity más reciente:

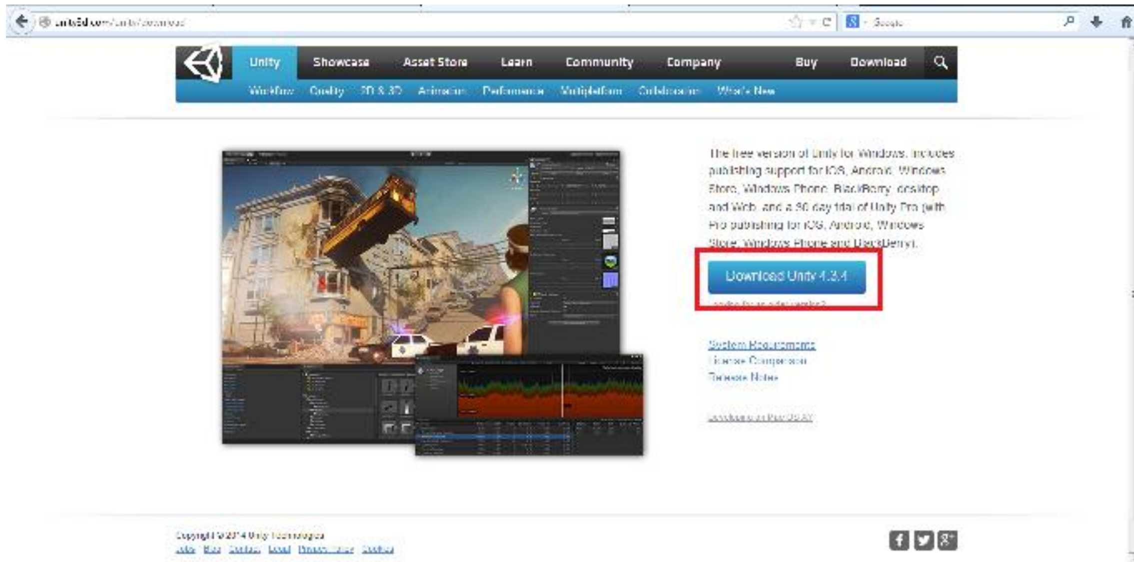


Figura 4.3 Pantalla `Dowload`

En ese momento comenzará a bajarse un instalador y una vez descargado, lo ejecutamos y seguimos los pasos.

Cuando se termine de instalar Unity y abramos por primera vez el programa, nos saldrá una ventana pidiéndonos que lo activemos. Marcamos "activate the free version of unity" y a continuación pulsaremos "ok":

Figura 4.4 Ventana "Activate your Unity license"

Nos aparecerá otra ventana donde se nos pedirá nuestro correo electrónico y una contraseña.

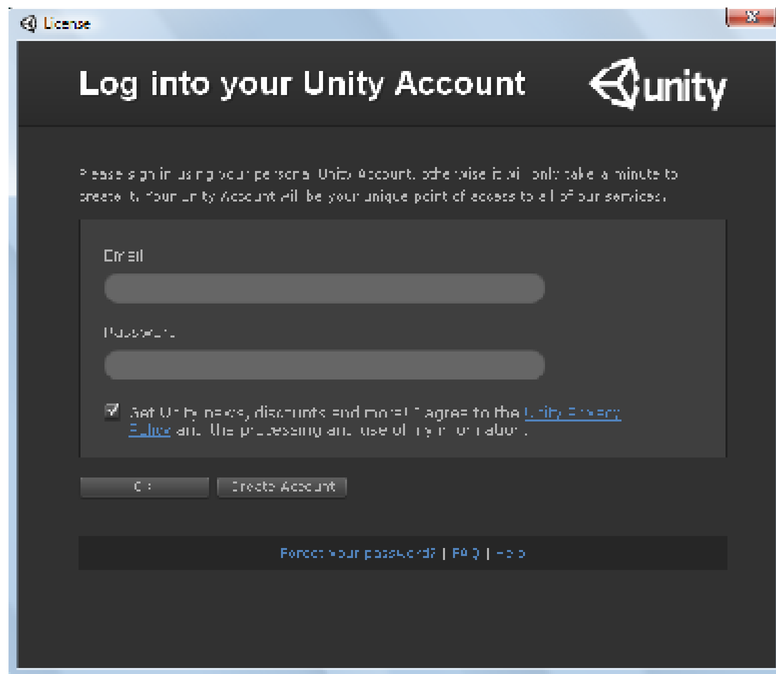


Figura 4.5 Ventana “Log into your Unity Account”

Después de introducir una dirección válida de correo y una contraseña, le daremos clic en “create account”. Nos llevará a otra ventana, donde tendremos que introducir nuestro nombre, nuestro email de nuevo, la contraseña y otra vez la misma para confirmarla:

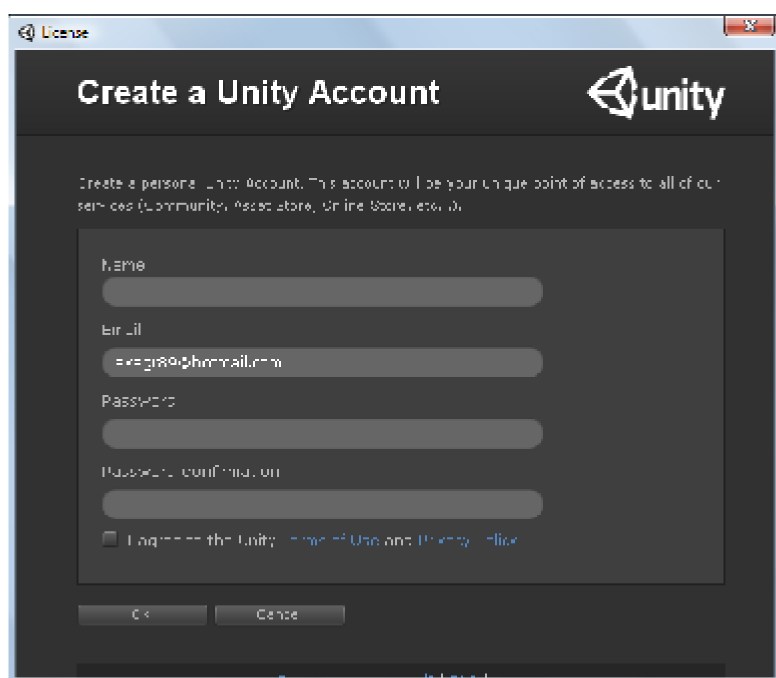


Figura 4.6 Ventana “Create a Unity Account”

Introducimos los datos, marcamos la casilla de aceptar los términos de uso y la política de privacidad de Unity, pulsamos “ok” y ya podemos usar Unity:

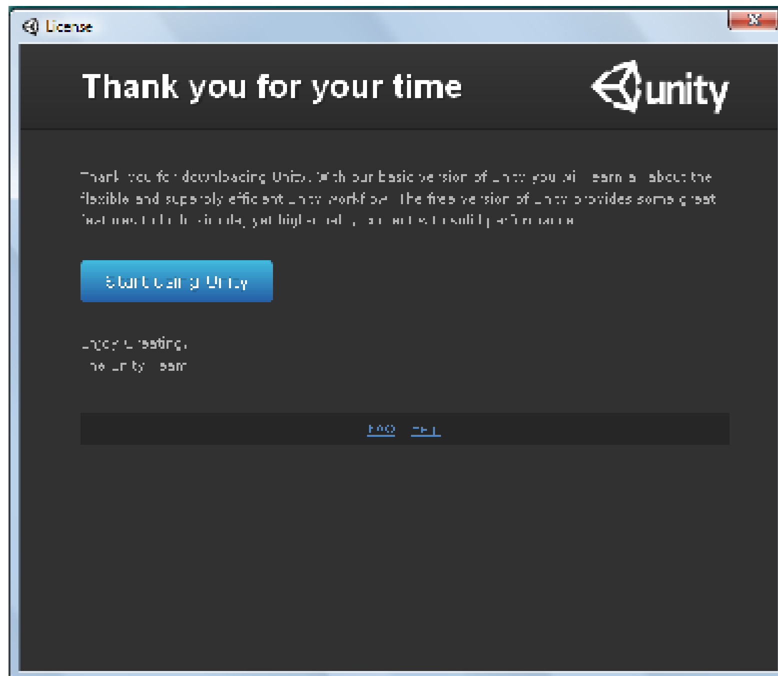


Figura 4.7 Ventana “Thank you for your time”

4.2.- La interfaz de usuario de Unity

Existen principalmente 5 áreas en la interfaz de Unity, numeradas en la siguiente imagen:

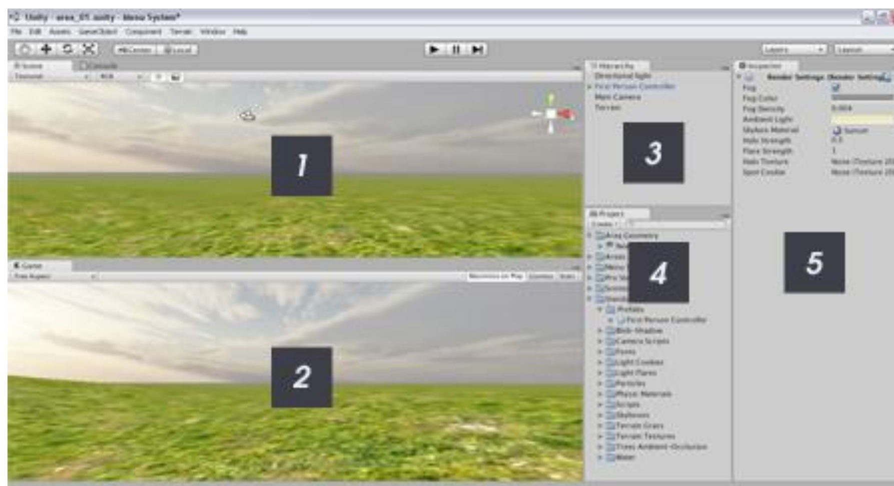


Figura 4.8 Interfaz de usuario de Unity

1) Vista de escena: La escena es el área de construcción de Unity donde construimos visualmente cada escena de nuestro juego.
2) Vista de juego: En la vista de juego obtendremos una previsualización de nuestro juego. En cualquier momento podemos reproducir nuestro juego y jugarlo en esta vista.
3) Vista de jerarquía: La vista de jerarquía contiene todos los objetos en la escena actual.
4) Vista de proyecto: Esta es la librería de *assets* para nuestro juego. Podemos importar objetos 3D de distintas aplicaciones a la librería, podemos importar texturas y crear otros objetos como *Scripts* o *Prefabs* (tipo de *asset* que nos permite definir las propiedades de un objeto y además poder instanciar una o más veces el mismo) que se almacenarán aquí. Todos los *assets* que importemos en nuestro juego se almacenarán aquí para que podamos usarlos.

   Ya que un juego normal contendrá varias escenas y una gran cantidad de *assets* es una buena idea estructurar la librería en diferentes carpetas que harán que nuestros *assets* se encuentren organizados y sea más fácil trabajar con ellos.

5) Vista de inspector: La vista de inspector sirve para varias cosas. Si seleccionamos objetos entonces mostrará las propiedades de ese objeto donde podemos personalizar varias características del objeto. También contiene la configuración para ciertas herramientas como la herramienta de terrenos si tenemos el terreno seleccionado.

Los *scripts* los creamos en MonoDevelop, que es el editor que nos salta por defecto. Cuando creamos un *script*, podemos saltar al editor de *scripts* desde Unity, guardar el *script* y usarlo en el juego.

Al igual que muchas aplicaciones podemos modificar la interfaz de Unity. Todas sus ventanas pueden redimensionarse, cambiarse de posición y anclarse arrastrando y soltando sus pestañas superiores. En el menú `Layout`, al que se accede a través del botón `Window` en el menú superior de la interfaz, o a través del botón `Layout` en la esquina superior derecha, además de haber configuraciones de interfaz predeterminadas, podemos salvar configuraciones personalizadas, algo que nos será muy útil para guardar nuestra configuración favorita a la hora de animar, editar terrenos, etc.

En las siguientes páginas vamos a ver de forma más detallada cada uno de estos componentes para entenderlos mejor.

4.2.1.- Vista de escena

Los juegos creados en Unity están divididos en “escenas”. La vista de escena es un entorno 3D para crear cada escena. Trabajar con la vista de escena, en la forma más sencilla, sería arrastrar un objeto desde la vista de proyecto a la vista de escena que colocará el objeto en la escena; entonces podremos posicionarlo, escalarlo y rotarlo sin salir de esta vista.

La vista de escena es también el lugar donde editamos los terrenos (esculpiéndolos, pintando texturas y colocando elementos), colocamos luces y cámaras y otros objetos.

Modos de visualización

Por defecto, la vista de escena tiene una perspectiva 3D de la escena. Podemos cambiar esto por una serie de vistas: *top down, side* y *front*. En la parte superior derecha de la vista de escena veremos un *Gizmo* que parece una caja con conos que salen de ella.

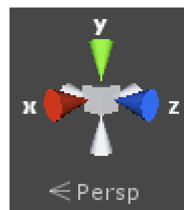


Figura 4.9 *Gizmo*

Podemos usar este *gizmo* para cambiar la perspectiva:

- Hacer click sobre la caja nos llevará al modo perspectiva.
- Hacer click en el cono verde (y) nos llevará al modo *Top*.
- Hacer click sobre el cono rojo (x) nos llevará al modo *Right* (derecha).
- Hacer click sobre el cono azul (z) nos llevará al modo *Front* (frontal).
- También tenemos tres conos grises que nos llevarán a los siguientes modos: *Back, Left*, y *Bottom*, en castellano, Atrás, Izquierda y Abajo.

Configuración de la visualización

En la parte superior de la vista encontraremos un conjunto de botones para cambiar la configuración general de la visualización. Vamos a ver cada uno de ellos, de izquierda a derecha.

Figura 4.10 Menú para configurar la visualización

Render Mode

La primera opción es Render Mode o modo de renderización en castellano. Por defecto aparecerá en Textured. Si hacemos clic aparecerá una lista desplegable con un número de diferentes opciones de renderizado:

- Textured: Las texturas se renderizan en la vista.
- Wireframe: Las superficies no se renderizan, solo vemos la malla.
- Textured Wireframe: Las texturas se renderizan, pero también vemos la malla.

Color Modes

La segunda opción es el modo de color, que aparece como RGB por defecto. Si hacemos clic sobre este botón aparecerá una lista desplegable que mostrará los modos de color disponibles:

- RGB: Todos los colores son renderizados.
- Alpha: El modo es cambiado a Alpha.
- Overdraw: El modo es cambiado a Overdraw.
- Minimaps: El modo es cambiado a Minimaps.

Interruptor de luces

El siguiente botón enciende o apaga la iluminación del escenario. Apagar la iluminación dará lugar a una escena mostrada sin luces, lo que puede ser útil para el rendimiento y también si no hay luces en la escena.

Encender la luz provocará que las luces tengan efecto sobre la escena. Si no tienes luces en la escena, esta será oscura, ya que no hay luz.

Interruptor de skybox, lense flare y niebla

El último botón activa y desactiva estos tres efectos. Esta opción es útil para desactivar los efectos por razones de rendimiento o visibilidad al trabajar sobre una escena.

Botones de control

En la parte superior izquierda de la interfaz de Unity, hay una fila con cuatro botones:

Figura 4.11 Botones de control

Se puede usar Q, W, E, R para alternar entre cada uno de los controles, que se detallan debajo:

Hand Tool (Q): Este control nos permite movernos alrededor en la vista de escena. Con este control podemos:

Rotar: Clic derecho del ratón y arrastrar o también mantener pulsado ALT y arrastrar con el botón izquierdo del ratón.

Desplazarse: Clic central (presionar la rueda del ratón) y arrastrar. También con las teclas direccionales.

Zoom: Rueda del ratón, aunque es recomendable hacerlo por desplazamientos. Al hacer zoom con la rueda del ratón se modifica la precisión de desplazamiento y abusar del zoom provocará que a veces el desplazamiento sea demasiado lento o demasiado rápido. Si es posible, es mejor evitar usar la rueda del ratón para hacer zoom.

`Translate Tool` (W): Nos permite mover cualquier objeto seleccionado en la escena en los ejes X, Y y Z.

`Rotate Tool` (E): Nos permite rotar cualquier objeto seleccionado en la escena.

`Scale Tool` (R): Nos permite escalar o cambiar el tamaño de cualquier objeto seleccionado en la escena.

4.2.2.- Vista de proyecto

La vista de proyecto es esencialmente una librería de *assets* para el proyecto de nuestro juego.

Todos los componentes del juego que creemos desde el editor y todos los objetos que importemos como modelos 3D, texturas, efectos de sonido, música, etc. Se guardarán ahí.

Como este panel contiene todos los *assets* de nuestro juego y no solo los que están en la escena actual, es importante mantener una buena estructura.

Podemos crear carpetas y colocar los objetos dentro de esas carpetas para crear una jerarquía de carpetas. Puesto que un proyecto de juego completo contendrá muchos *assets* (en algunos juegos una cantidad muy grande de éstos), es una buena idea el decidir y crear una estructura que sea fácil de usar por el equipo con el que trabajemos antes de empezar a producir nuestro juego.

Ya que la vista de proyecto es muy sencilla y funciona como cualquier gestor de ficheros no será necesario más que explicar algunas notas de uso:

- En la parte superior de la vista de proyecto veremos un botón `Create` que mostrará una lista desplegable con varias opciones de creación. Podremos crear carpetas, *scripts*, *shaders*, animaciones y otros tipos de objetos usando este panel.

- Hacer dos clics sobre un ítem en la librería nos permitirá renombrarlo.

- Podemos hacer clic y arrastre de carpetas e ítems para organizar la estructura.

- Podemos importar y exportar paquetes (colecciones de *assets*) simplemente haciendo clic derecho sobre la vista del proyecto y seleccionando la opción apropiada.

- Podemos importar *assets* como archivos de audio, texturas, modelos, etc. Con hacer clic derecho y seleccionar `Import asset`.

- Podemos buscar por la librería usando la casilla de búsqueda al lado del botón `Create`.

4.2.3.-Vista de jerarquía

La vista de jerarquía contiene una lista de todos los objetos usados en la escena actual. Cualquier objeto que coloquemos en la escena aparecerá como una entrada en la jerarquía.

La jerarquía también sirve como método rápido y fácil para seleccionar objetos en la escena.

Si queremos por ejemplo, seleccionar un objeto de la escena podemos seleccionarlo desde la jerarquía en lugar de movernos por la escena, encontrarlo y seleccionarlo.

Cuando un objeto es seleccionado en la jerarquía también lo es en la vista de escena, donde podemos moverlo, escalarlo, rotarlo, borrarlo o editarlo. El inspector también mostrará las propiedades del objeto seleccionado; de esta forma la jerarquía sirve como una herramienta útil para seleccionar rápidamente objetos y editar sus propiedades.

4.2.4.-Inspector

El inspector es esencialmente un panel de propiedades; si seleccionamos un objeto en la escena, todas las propiedades editables aparecerán en el inspector.

Por ejemplo, si seleccionamos una luz o cámara, el inspector te permitirá editar varias propiedades de la luz o de la cámara. Adicionalmente el inspector sirve como panel de herramientas para ciertos tipos de objetos. Por ejemplo, si seleccionamos un terreno el inspector mostrará las opciones de terreno y también el editor con herramientas como esculpir, texturizar, etc.

Como el inspector cambia según el objeto que seleccionemos, la mejor forma de aprenderlo es probarlo.

4.2.5.-Vista de juego

En Unity podemos ejecutar nuestro juego sin salir del editor. En la imagen de debajo, se muestran los controles de reproducción que están localizados en la parte superior del editor:

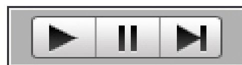
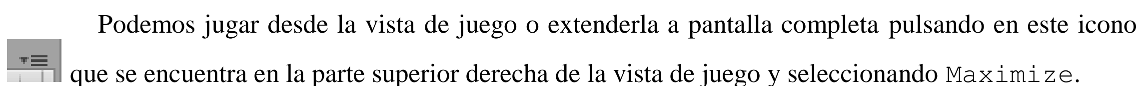


Figura 4.12 Controles de reproducción

Se puede entrar en la previsualización del juego en cualquier momento pulsando el botón de reproducción (el primero por la izquierda), pausar usando el botón de pausa (botón central) o saltar adelante usando el botón derecho.

Podemos jugar desde la vista de juego o extenderla a pantalla completa pulsando en este icono  que se encuentra en la parte superior derecha de la vista de juego y seleccionando `Maximize`.

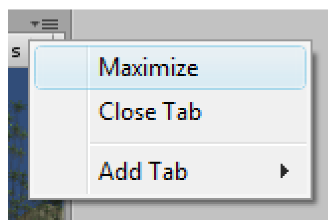


Figura 4.13 Opción `Maximize`

El panel situado en la parte superior de la vista de juego tiene un menú a la izquierda que aparece como `Free Aspect` por defecto. De esta lista podremos seleccionar un número de proporciones para el juego, lo que es ideal para probar nuestro juego en distintas pantallas y plataformas.

Si no podemos ver el mundo de nuestro videojuego en la vista del juego es porque nuestra cámara no está mirando en la dirección correcta.

Para que mire en la dirección correcta hacemos clic en la cámara principal y veremos que una malla en forma de pirámide blanca sale desde ella. Se denomina campo de visión de la cámara y representa lo que esta puede ver. Si el campo de visión no apunta directamente a los objetos de nuestro videojuego, navegamos por la escena hasta localizarlos. Una vez hecho esto, volvemos a seleccionar la cámara en la jerarquía y hacemos clic en la pestaña superior: `Game Object->Align with view`. De esta manera, la cámara del videojuego se alineará exactamente en esa posición y orientación, así la vista de videojuego coincidirá con la vista de escena.

Para casos concretos, podemos ajustar manualmente la configuración de la cámara usando las herramientas de Mover y Rotar o, cambiando sus valores directamente en el panel de la vista de inspector.

4.3.-Primeros pasos

4.3.1.-Crear un proyecto

Empecemos por crear un nuevo proyecto. Para crear un nuevo proyecto vamos a `File->New Project`:

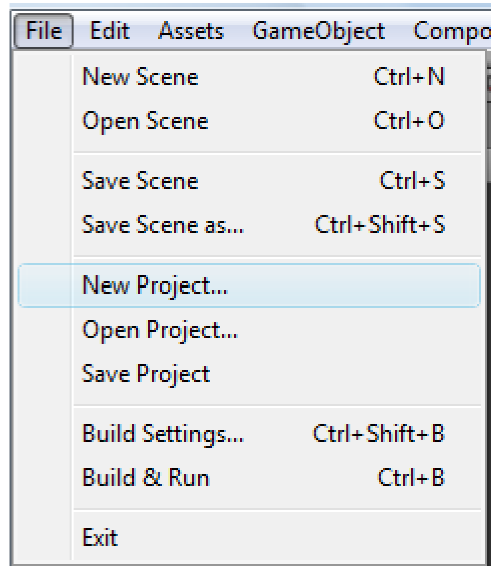


Figura 4.14

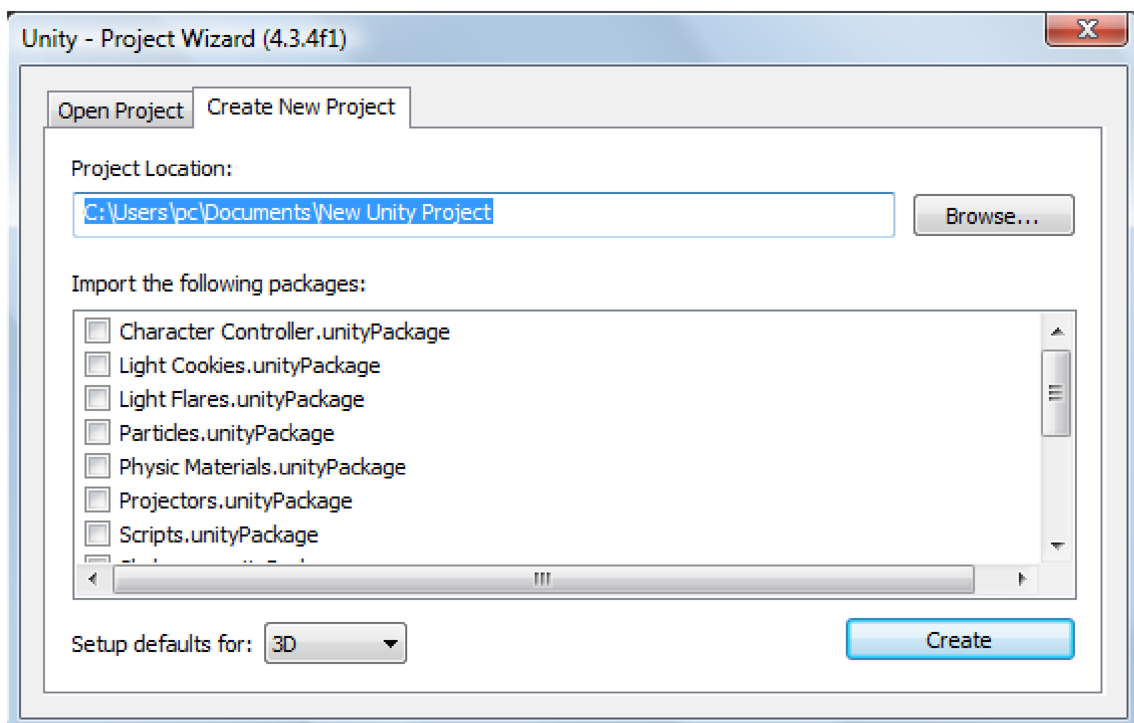


Figura 4.15 Ventana para crear un nuevo proyecto

Lo primero que debemos hacer es elegir la localización de nuestro proyecto. Para eso hacemos clic sobre "Browse" y seleccionamos la carpeta donde queremos crear nuestros proyectos y crear una carpeta: "Primeros pasos".

Lo siguiente que hay que hacer es seleccionar los paquetes que queremos incluir en el proyecto. Los paquetes se pueden importar en cualquier momento desde Unity si queremos usar paquetes adicionales que hemos descargado o no hemos seleccionado en este punto.

En esta ocasión vamos a seleccionar todos los paquetes y luego hacemos clic en "Create".

Cuando hagamos esto Unity se reiniciará, creará la estructura del proyecto en la carpeta especificada e importará los paquetes seleccionados al proyecto. Una vez termine se mostrará una escena en blanco con una cámara principal que podemos ver en la vista de escena y en la jerarquía.

También veremos los *assets* que hemos importado en la vista del proyecto, listos para ser usados.

4.3.2.-La primera escena.

Unity ha creado nuestra primera escena automáticamente al crear el proyecto. Si queremos crear una escena nueva, vamos a `File->New Scene`.

Esta nueva escena es un espacio sin título. Habrá que guardarla y para ello vamos a `File->Save Scene as…`, la ponemos un nombre y la guardamos en algún lugar en la carpeta de *Assets* de este proyecto, por ejemplo dentro de una nueva carpeta que creemos llamada "Escenas". La escena aparecerá en la vista de proyecto una vez hagamos esto.

4.3.3.-Localizando objetos

Para familiarizarnos un poco con la localización de objetos, vamos a hacer unos sencillos ejercicios. Vamos a renombrar la cámara principal. Para ello hacemos clic sobre su nombre en la vista de jerarquía y a continuación otro clic más que nos permitirá cambiarlo. Otra opción es hacer clic derecho y seleccionar `Rename`. Le ponemos como nombre "Cámara" y pulsamos intro para terminar.

Ahora comprobaremos la Vista de escena. Si no podemos ver la cámara en ella, podemos hacer clic en ella en la Vista de Jerarquía. Una vez hecho eso, si movemos el cursor dentro de la vista de escena, pulsando la tecla F realizaremos la opción de seleccionar el *Frame* (`Frame Select`), que se encarga de centrar en la escena el objeto seleccionado. Esta es la técnica que debemos seguir siempre que queramos localizar un objeto en la escena. También se puede hacer doble clic en su nombre en la Vista de Jerarquía para centrarla de la misma manera y así con cualquier objeto.

Observemos que todos los objetos pueden ser seleccionados tanto haciendo clic sobre ellos en la Vista de Escena como en la Vista de Jerarquía. Si seleccionamos un objeto en la Vista de Escena podemos comprobar si hemos seleccionado el objeto correcto porque se mostrará también iluminado en la Vista de Jerarquía.

4.3.4.-Crear Objetos

A continuación vamos a añadir unos cuantos objetos en el mundo de nuestro videojuego.

Seleccionamos la pestaña superior: `Game Object->Create Other->Plane`. Esto creará una superficie bidimensional (un plano).

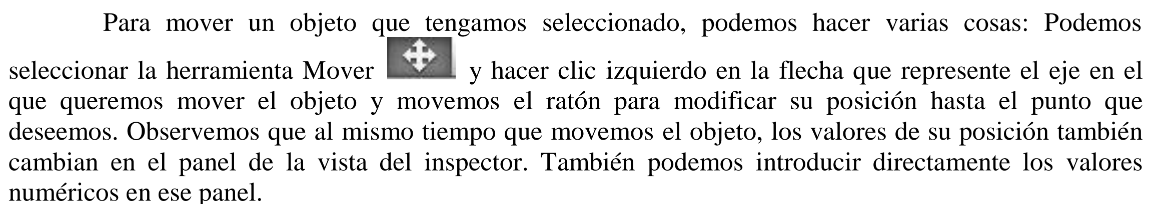
Añadimos un cubo al mundo del videojuego seleccionando: `Game Object->Create Other->Cube`.

Y por último, añadimos una luz puntual seleccionando: `Game Object->Create Other->Point Light`.

Todos los objetos que hemos creado aparecen por defecto en la misma posición, con una rotación neutra y sus valores iniciales por defecto.

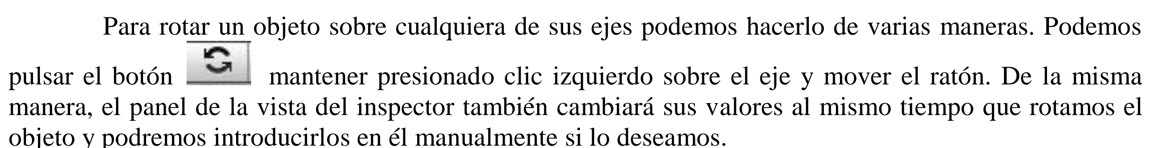
4.3.5.-Mover, rotar y redimensionar objetos

Mover objetos

Para mover un objeto que tengamos seleccionado, podemos hacer varias cosas: Podemos seleccionar la herramienta Mover  y hacer clic izquierdo en la flecha que represente el eje en el que queremos mover el objeto y movemos el ratón para modificar su posición hasta el punto que deseemos. Observemos que al mismo tiempo que movemos el objeto, los valores de su posición también cambian en el panel de la vista del inspector. También podemos introducir directamente los valores numéricos en ese panel.

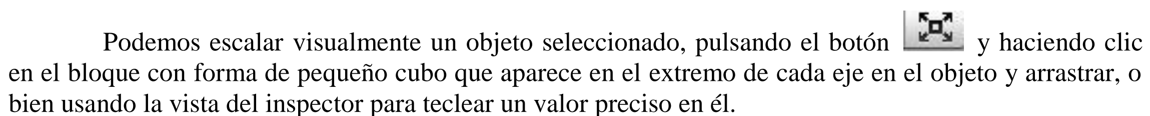
Otra forma más avanzada de modificar la posición de un objeto del videojuego es hacer clic sobre él para seleccionarlo y a continuación mover la vista de escena de forma que mire a la posición donde queremos mover el objeto. Si hacemos clic en las pestañas superiores y seleccionamos `Game Object->Move to view`, el objeto se moverá a la posición a la que mira la cámara actualmente.

Rotar objetos

Para rotar un objeto sobre cualquiera de sus ejes podemos hacerlo de varias maneras. Podemos pulsar el botón  mantener presionado clic izquierdo sobre el eje y mover el ratón. De la misma manera, el panel de la vista del inspector también cambiará sus valores al mismo tiempo que rotamos el objeto y podremos introducirlos en él manualmente si lo deseamos.

Otra forma más avanzada de modificar la rotación es mover la vista de escena de forma que mire en la dirección con la que queremos alinear el objeto. Si hacemos clic en las pestañas superiores y seleccionamos: `Game Object->Align with view`, el objeto se quedará alineado en la dirección a la que mira la cámara de la vista de escena. Esto nos será sobre todo útil para orientar luces direccionales o cámaras para que apunten a donde queremos.

Redimensionar objetos

Podemos escalar visualmente un objeto seleccionado, pulsando el botón  y haciendo clic en el bloque con forma de pequeño cubo que aparece en el extremo de cada eje en el objeto y arrastrar, o bien usando la vista del inspector para teclear un valor preciso en él.

Vamos a reescalar algunos objetos de nuestro videojuego:

Seleccionamos el plano y ponemos a 10 los valores de escala tanto en el eje X como en el Z usando La vista de Inspector. Serán el primer y el tercer valor de la escala, ya que el segundo correspondería al eje Y y los planos como tales no pueden agrandarse “hacia arriba”.

Seleccionamos el cubo y lo redimensionamos visualmente a lo largo del eje Y subiendo hacia arriba el bloque con forma de cubo en ese eje de forma que el cubo termine pareciéndose a un pilar.

Renombramos el cubo como “Pilar”. Esto lo podemos hacer seleccionándolo en la Vista de la Jerarquía, haciendo clic derecho sobre él y seleccionando la opción `Rename` (Renombrar). Tras escribir el nombre, pulsamos Enter para guardarlo. Es recomendable que los nombres de los objetos del videojuego comiencen con letra mayúscula.

4.3.6.-Utilizar recursos

Un videojuego está compuesto por modelos 3D, texturas, archivos de sonido, código fuente, etc.

Estos elementos se denominan Recursos (*Assets*). Unity 3D viene con muchos recursos y también pueden descargarse gran cantidad de ellos en internet.

Vamos a añadir a nuestro videojuego uno de los personajes prefabricados que podemos encontrar en los recursos. En la vista de Proyecto expandimos el directorio llamado *Standard Assets* (recursos estándar) y a continuación expandimos el subdirectorio *Character Controllers* (Controles de personaje).

Todos los *Standard Assets* que vienen con Unity son colecciones de objetos de videojuego (*Scripts*, modelos, texturas…) que han sido previamente ensamblados entre ellos con una agrupación lógica. Estas agrupaciones de objetos se denominan *Prefabs* y son fundamentalmente para trabajar con Unity de forma eficiente. En el subdirectorio *Character Controllers* veremos un *Prefab* llamado *First Person Controller* (Controlador de primera persona).

Vamos a arrastrar el *First Person Controller* en la vista de escena. Observemos como la vista de videojuego cambia, esto es porque el *FPS Controller* prefabricado viene con su propia cámara, la cual toma precedencia frente a la cámara principal que existía desde un principio. Nos aseguramos de que el *first person controller* no está atrapado dentro del plano, para ello lo movemos de forma que quede sobre el plano.

Borramos la cámara principal (*Main Camera*) ya que no la vamos a volver a necesitar, podemos hacerlo pulsando la tecla Supr con ella seleccionada o bien haciendo clic derecho sobre ella en la vista de jerarquía y seleccionando la opción `delete.` La vista de videojuego a partir de ahora será lo que el jugador puede ver dentro del juego.

Arrancamos el juego clicando el botón Play que podemos ver en la parte superior central de la `GUI` de Unity (Figura 4.16).

Figura 4.16

Observamos que la pantalla permanece igual, de forma que podemos ver el videojuego dentro de la vista de escena y la vista de videojuego. Usamos las teclas cursor para mover a nuestro personaje por el entorno, usamos el ratón para mirar a nuestro alrededor y la tecla espacio para saltar.

Podemos obtener la ejecución del videojuego en cualquier momento pulsando nuevamente el botón play o bien hacer una pausa utilizando el botón pausa.

4.3.7.-Añadir componentes

Los objetos de nuestro videojuego pueden tener cierto número de componentes (también llamados comportamientos) asociados. Al hacer clic en un objeto del videojuego, podemos ver los componentes asociados en su vista de Inspector.

Vamos a hacer clic en el "Pilar" y miramos sus componentes asociados en la vista de inspector. Añadimos otro cubo más a la escena: `Game Object->Create Other->Cube`. Lo renombramos como "Cubo con gravedad" en la vista de jerarquía.

Ahora añadimos un *Rigidbody* (Cuerpo rígido) al cubo. Este tipo de componente permite al objeto del videojuego comportarse como si estuviese dentro de un mundo con gravedad. Por ejemplo, el objeto caerá hacia abajo hasta que golpee una superficie que tenga un componente asociado de tipo `Collider`, por ejemplo el Plano.

Nos aseguramos de que tenemos el Cubo con gravedad seleccionado y seleccionamos: `Component->Physics->Rigidbody`. Esto añadirá el componente *Rigidbody* al objeto del videojuego que tengamos seleccionado. Observamos que en la vista de Inspector el componente también aparecerá añadido.

Movemos el cubo con gravedad de forma que se encuentre sobre el pilar, a un poco de distancia sobre él.

Queremos configurar la escena de forma que, cuando pulsemos Play, el cubo con gravedad colisione con el pilar y después caiga al suelo.

Pulsamos el botón Play, el cubo con gravedad debería colisionar contra el pilar y a continuación caer sobre el plano comportándose como si hubiese gravedad.

4.3.8.-Duplicar objetos

Duplicar un objeto del juego es la característica más potente de Unity. Cuando duplicamos un objeto, todas las características y comportamientos del objeto son a su vez copiados. Es una forma muy rápida de crear escenas complejas con multitud de objetos. Ahora vamos a añadir más cubos con gravedad a nuestra escena.

Hacemos clic en el cubo con gravedad para asegurarnos de que lo tenemos seleccionado.

Pulsamos Ctrl+D (también podemos hacerlo desde el menú `Edit`, o simplemente haciendo clic derecho sobre él en la vista de la jerarquía y seleccionando la opción `duplicate`). Observemos como aparece una nueva entrada con un cubo con gravedad adicional en la vista de jerarquía. Observamos que sin embargo, no podemos verlo, ya que ha sido creado exactamente sobre la misma posición que el anterior.

Movemos el cubo con gravedad seleccionando la herramienta mover (o pulsando la tecla w) y lo movemos hacia arriba en el eje Y.

Repetimos este proceso de forma que haya varios cubos con gravedad en la escena, unos sobre otros formando alguna estructura que cuando demos al play se derrumbe.

Ejecutamos el juego y los cubos con gravedad interactuarán entre ellos comportándose con total naturalidad.

4.4.-Terrenos

4.4.1.-Creando un terreno

Para crear un terreno vamos a ir a `Game Object->Create Other->Terrain`:

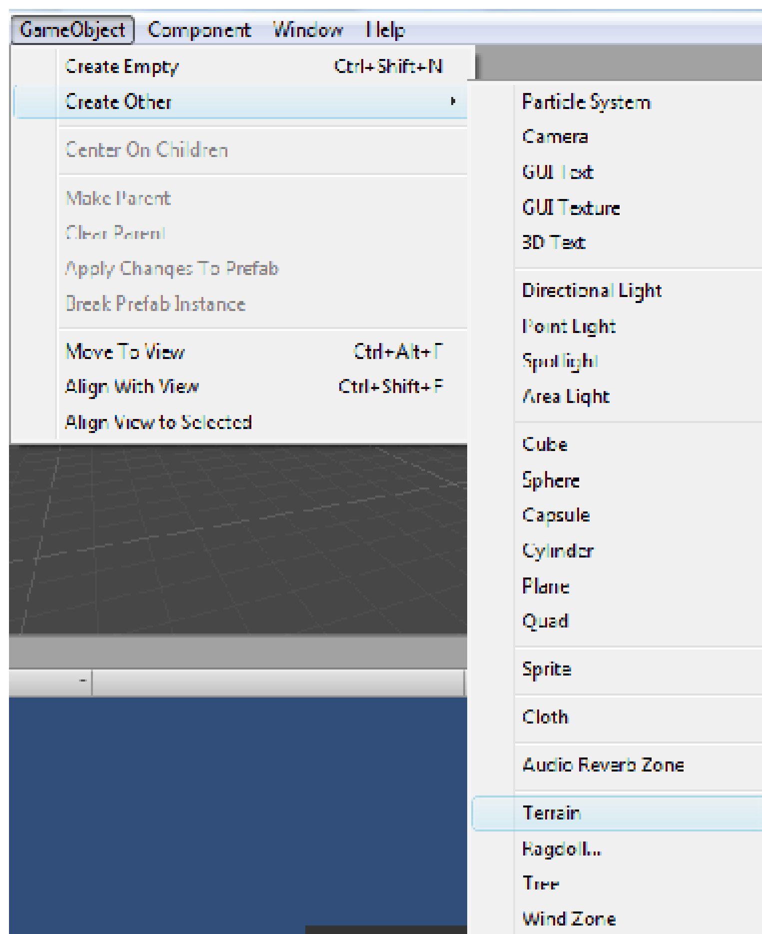


Figura 4.17

En este momento vemos que el terreno ha aparecido en la vista de escena, en la vista de jerarquía y en la de proyecto.

En la vista de jerarquía, vamos a hacer doble clic en el objeto "Terrain" para verlo centrado en la vista de escena y hacemos clic en el centro del icono situado en la parte superior derecha de la vista de escena (*Gizmo*). Así vemos el terreno desde una perspectiva natural.

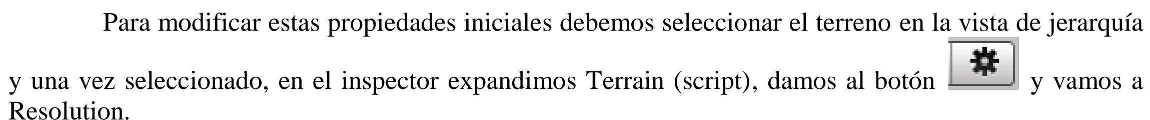
Recordamos que se había generado un recurso en la vista del proyecto. Es recomendable que lo movamos a una nueva carpeta llamada “Terrenos” y le demos un nombre propio al terreno, por ejemplo: “Terreno 1”. Así tenemos los recursos bien organizados y será más fácil trabajar con ellos.

Para renombrar un recurso en la vista de proyecto hacemos clic sobre él y volvemos a hacer otro clic o también podemos renombrar de la misma manera pulsando la tecla F2. Después de escribir el nombre, damos a intro para finalizar.

Hacemos clic derecho en una zona vacía de la vista del proyecto y seleccionamos: `Create->Folder` y la ponemos el nombre: “Terrenos”. Renombramos el recurso “New Terrain” como “Terreno 1” y lo arrastramos dentro de la carpeta de terrenos.

4.4.2.-Configuración básica de un terreno

Ahora que tenemos nuestro terreno en la escena, debemos definir algunas propiedades importantes, como la longitud del terreno o como de detallado debe ser.

Para modificar estas propiedades iniciales debemos seleccionar el terreno en la vista de jerarquía y una vez seleccionado, en el inspector expandimos Terrain (script), damos al botón  y vamos a Resolution.

Nos aparecerán las siguientes propiedades:

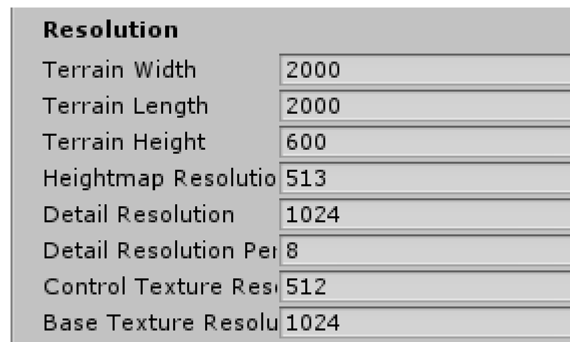


Figura 4.18

Vamos a definir cada una de ellas:

`Terrain Width`: El ancho en metros de nuestro terreno.

`Terrain Length`: La longitud en metros del terreno.

`Terrain Height`: La máxima altura en metros del terreno.

`Heightmap Resolution`: La resolución del Heightmap.

`Detail Resolution`: La resolución del mapa de detalles. Cuánta más resolución, más precisión a la hora de dibujar los detalles sobre el terreno y colocar objetos.

`Control Texture Resolution`: La resolución de las texturas pintadas sobre el terreno.

`Base Texture Resolution`: Esta es la resolución base de la textura que se renderiza desde la distancia.

Vamos a practicar un poco dándoles nuevos valores:

`Width`: 500 m

`Lenght`: 500 m

`Height`: 500 m

`Heightmap Resolution`: 513

`Detail Resolution`: 1024

`Control Texture resolution`: 512

`Base Texture Resolution`: 1024

Nuestro terreno debería haber quedado así:

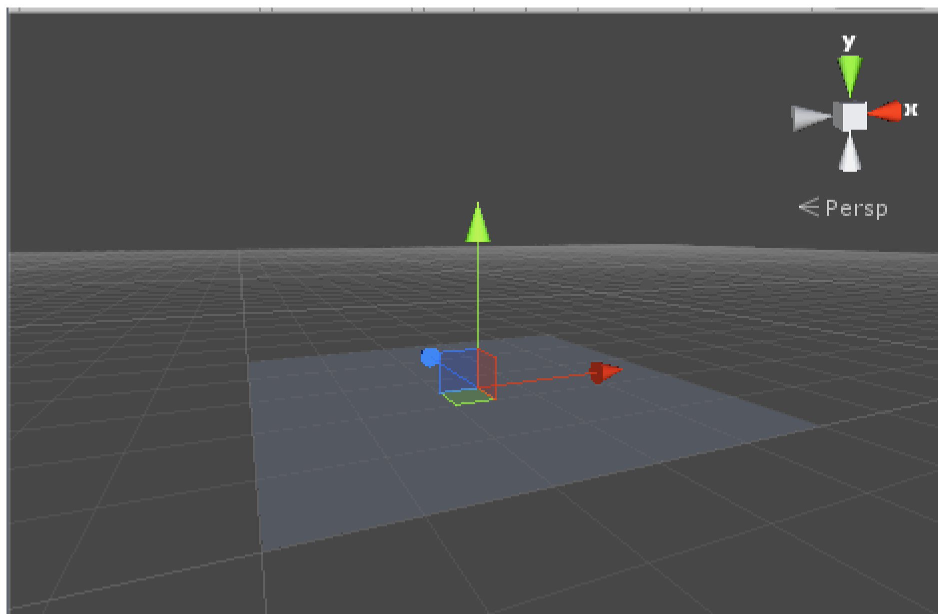
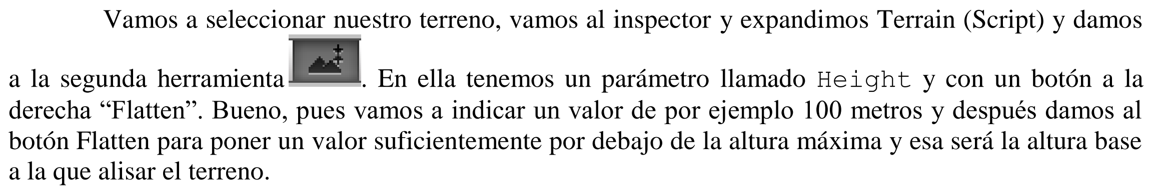


Figura 4.19

Acabamos de ajustar una altura máxima de terreno, pero debemos tener en cuenta que el terreno recién creado está totalmente aplanado a una altura de 0 metros y Unity nos permitirá crear montañas por encima de esa altura, pero no hoyos por debajo.

Por eso es interesante establecer desde el principio una altura base inicial. Así crearemos montañas por encima de esa altura base y hoyos donde queramos para hacer por ejemplo, un lago.

Vamos a seleccionar nuestro terreno, vamos al inspector y expandimos Terrain (Script) y damos a la segunda herramienta . En ella tenemos un parámetro llamado `Height` y con un botón a la derecha "Flatten". Bueno, pues vamos a indicar un valor de por ejemplo 100 metros y después damos al botón Flatten para poner un valor suficientemente por debajo de la altura máxima y esa será la altura base a la que alisar el terreno.

Observaremos que el terreno aparece un poco por encima de su posición anterior debido al cambio de elevación inicial.

4.4.3.-Herramientas de terreno

Como vemos, nuestro terreno es fino y no muy bonito. Para modificar nuestro terreno vamos a hacer uso de las herramientas de edición de terrenos. Para acceder a las herramientas seleccionamos nuestro terreno en la jerarquía. Cuando lo seleccionemos, notaremos que el inspector cambia para mostrar las herramientas de edición de terrenos:

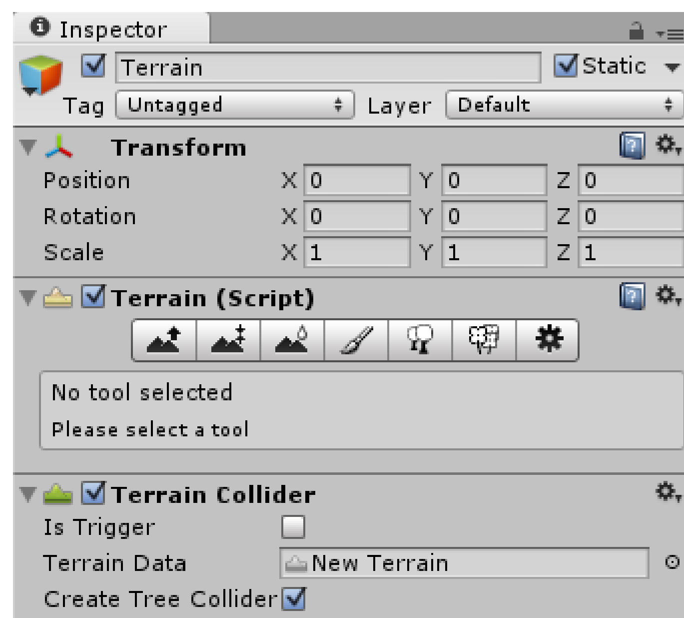


Figura 4.20 Herramientas de edición de terrenos

El inspector está dividido en tres partes:

Transform: Permite mover y escalar el terreno sobre los ejes x, y, z.

Terrain Script: Contiene varias herramientas y propiedades para el terreno.

Terrain Collider: Contiene las propiedades de colisión para el terreno.

En el panel Terrain Script veremos una fila de botones; estos son los botones editores de terreno y cada uno puede realizar diferentes tareas.

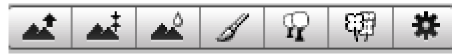


Figura 4.21 Botones editores de terreno

Vamos a describir lo que puede hacer cada uno de los botones, en orden de izquierda a derecha:

Raise/Lower: Nos permite levantar y hundir la geometría del terreno usando un pincel.

Set Height: Pintamos el terreno con una altura límite.

Smooth: Nos permite suavizar un terreno para eliminar esquinas, por ejemplo.

Paint Texture: Nos permite pintar texturas sobre la superficie del terreno.

Place Trees: Nos permite colocar árboles.

Paint Detail: Nos permite dibujar los detalles del terreno como la hierba.

Terrain Settings: Accede a las propiedades del terreno donde podemos cambiarlas.

Vamos a detallarlos a continuación:

Botón Raise/Lower o subir/bajar

Nos ofrece un conjunto de brochas con las cuales podemos realizar elevaciones y hoyos a mano alzada haciendo clic en el terreno.

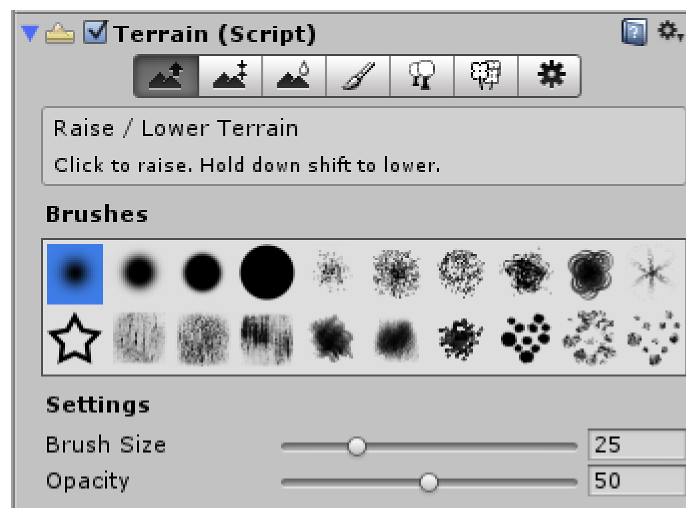


Figura 4.22 Botón Raise/Lower seleccionado

Las brochas (*brushes*) determinan la geometría de elevación. Es mejor evitar brochas circulares o con geometrías que normalmente no se encuentran en la naturaleza para conseguir más realismo.

El parámetro de tamaño (`Brush size`) determina el diámetro de elevación medida en metros aproximadamente.

El parámetro de opacidad (`Opacity`) determina la velocidad con la que se realizará la elevación al hacer clic. Para trabajar con precisión hay que evitar los valores de opacidad elevados, generalmente 50% es más que suficiente.

Si pulsamos la tecla Shift a la vez que hacemos clic, la herramienta en vez de elevaciones realizará perforaciones. Como hemos establecido una altura base, la herramienta no nos permitirá bajar más allá de cero ni subir más allá de la altura máxima permitida.

Vamos a dedicar unos minutos a diseñar a mano alzada un terreno rodeado de montañas elevadas que conste de lagos, grietas y otros accidentes geográficos.

Si en algún momento realizamos alguna modificación y queremos volver atrás, podemos hacer clic en la pestaña superior `Edit->Undo`, o bien utilizar un atajo de teclado Ctrl+Z para deshacer la última acción. Esta técnica se aplica a cualquier modificación realizada en la Vista de Escena de Unity y nos será muy útil.

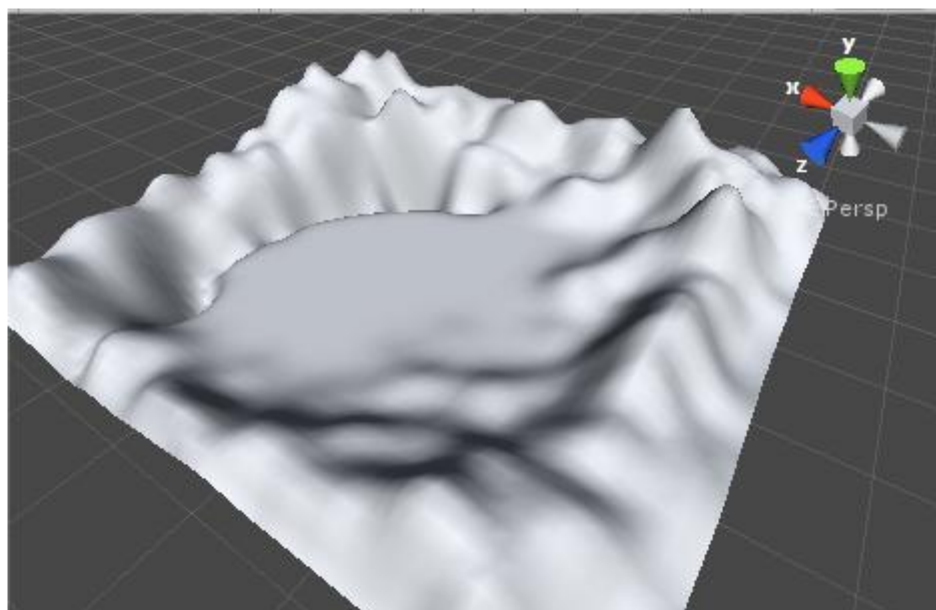


Figura 4.23

Botón Set Height o aplanar

Un inconveniente que tiene el botón anterior es que las elevaciones se realizan a mano alzada y en ocasiones resultan excesivas para la capacidad de movimiento de nuestro personaje o producen lagos con demasiada profundidad.

Si tenemos en cuenta la altura base que dimos para nuestro terreno, podemos utilizar este botón para realizar elevaciones y hoyos sobre el terreno con una altura objetivo, de manera que podamos controlar exactamente cuál es la altura de nuestras montañas o profundidad de nuestros lagos, así como crear mesetas, valles, escalones, etc.

Este botón es muy similar al anterior ya que nos ofrece el mismo conjunto de brochas y opciones, pero este tiene un parámetro más: `Height`, que es la altura objetivo.

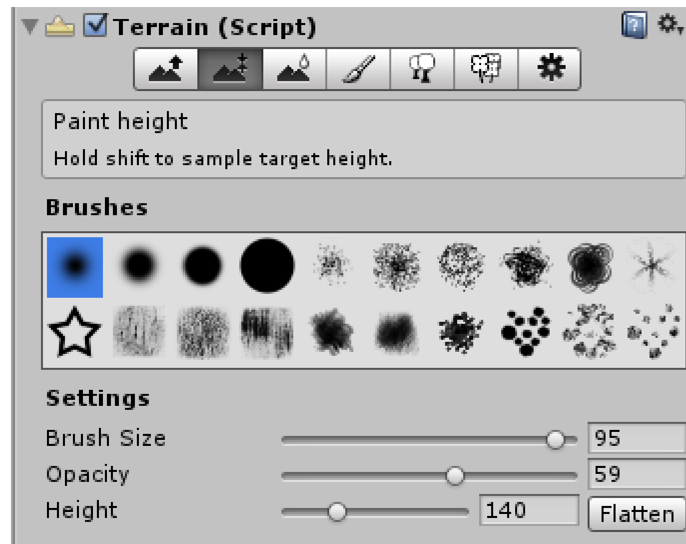


Figura 4.24 Botón de aplanar seleccionado

A este parámetro adicional: `Height` o altura objetivo, se le puede dar un valor cualquiera entre cero y la altura máxima que hayamos configurado para el terreno.

Vamos a utilizar esta herramienta para generar una gran superficie plana a cierta altura, por ejemplo 120 metros.

Generamos también una meseta 20 metros por encima de ella y un camino que la atraviese.

Después vamos a generar sobre la planicie alrededor de la meseta una grieta en zig-zag a modo de río de 5 metros de profundidad y vamos a tener en cuenta aquí el tamaño de la brocha y la opacidad para conseguir mayor realismo.

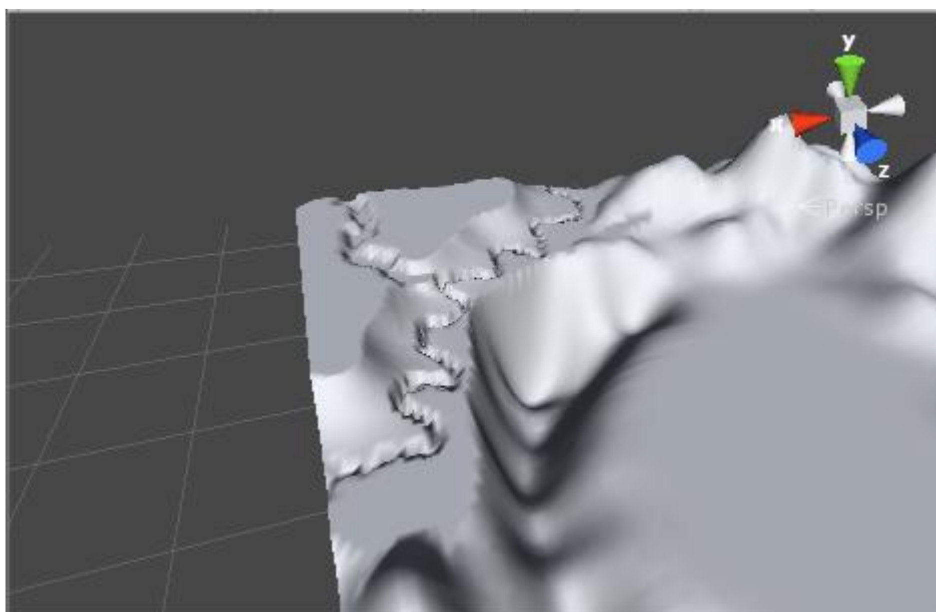


Figura 4.25

Botón de suavizado

Las deformaciones que hemos realizado sobre el terreno son almacenadas en forma de pequeños triángulos unidos por sus aristas que conforman la geometría. Al trabajar a mano alzada, uno de los problemas más habituales es la presencia de picos sobre el terreno que restan realismo a nuestro trabajo.

Este botón tiene la utilidad de difuminar la altura de los triángulos que generan el terreno consiguiendo una sensación de suavidad y eliminando los picos que hayamos generado durante el proceso de diseño.

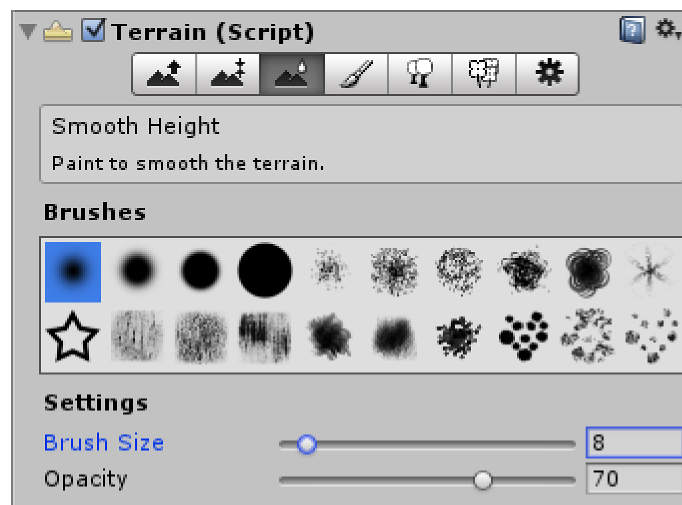


Figura 4.26 Botón de suavizado seleccionado

Esta brocha debe utilizarse con un tamaño y opacidad baja y solamente sobre aquellas zonas en las que haya picos y su objetivo es redondear y suavizar para conseguir una apariencia más realista. También nos permite generar caminos ascendentes y descendentes cuando la aplicamos sobre escalones generados con la brocha de aplanado.

Vamos a recorrer todo el terreno de cerca buscando picos y a utilizar la brocha para suavizarlos.

Después vamos a generar con la brocha de aplanado una subida escalonada hacia la meseta que hemos diseñado y hacemos uso de la brocha de suavizado para convertir los escalones en una pendiente lo más suave posible.

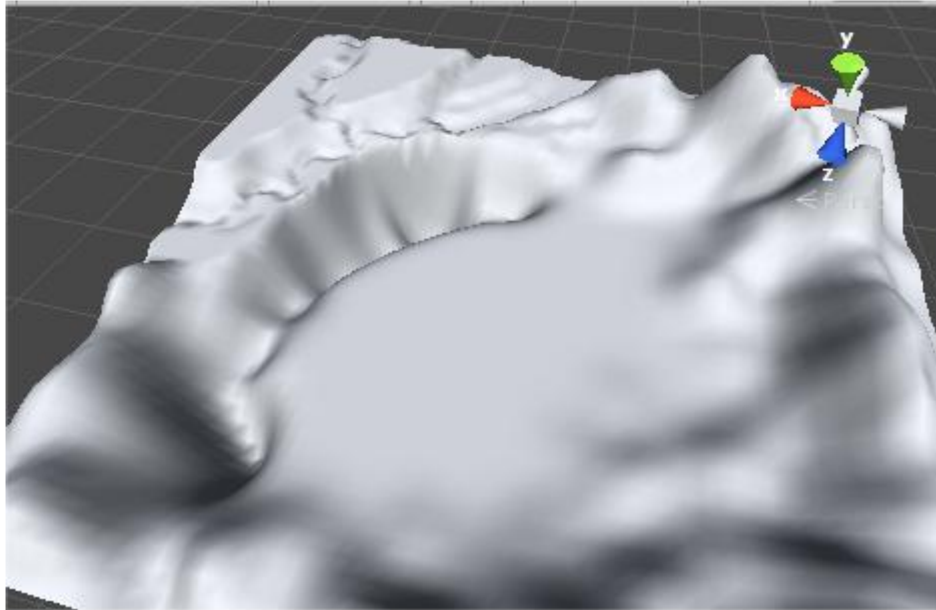


Figura 4.27

Botón de texturizado

Hasta ahora hemos trabajado con un terreno en escala de grises. Vamos a empezar a añadirle texturas a nuestro terreno.

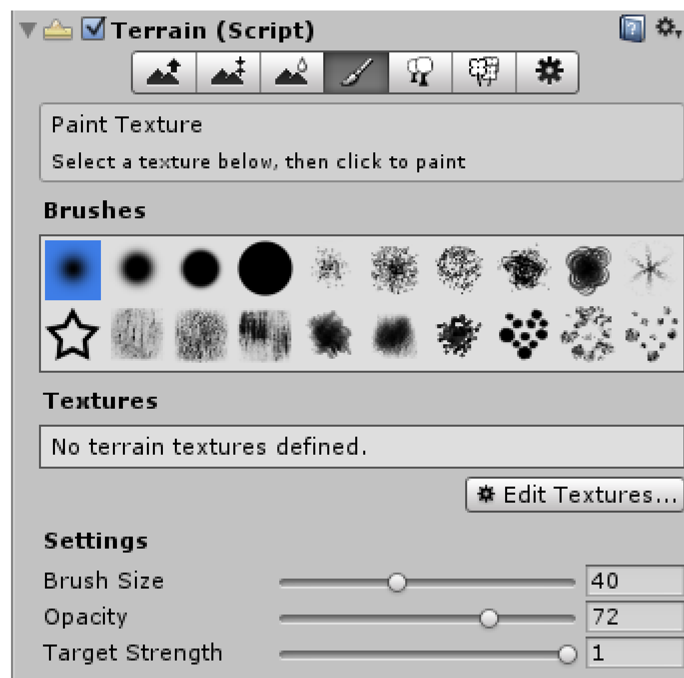


Figura 4.28 Botón de texturizado seleccionado

En este menú volvemos a tener los diferentes tipos de brocha, el tamaño y la opacidad, que en este caso afecta a la velocidad con la que teñimos con una textura sobre la anterior.

`Target Strength` o fuerza objetivo nos indica en tanto por uno el máximo teñido que se aplicará finalmente.

La diferencia más notable en este menú es el botón `Edit Textures` que nos permitirá añadir nuevas texturas a nuestra paleta de pintura.

La primera textura recubrirá todo el terreno y será la base sobre la que pintaremos las demás. Hay que seleccionarla con cuidado ya que al ser la base, determinará si el terreno es arenoso, rocoso, de bosque, etc.

Vamos a hacer clic en el botón `Edit textures->Add texture`:

Figura 4.29 Ventana “Add Terrain Texture”

Esta ventana que se ha abierto, nos permitirá seleccionar una textura de entre los recursos de nuestro proyecto y también configurar algunos de sus aspectos básicos de tamaño.

En el primer recuadro “Texture”, podemos seleccionar una textura. Hacemos clic en “Select” y nos aparece la siguiente ventana:

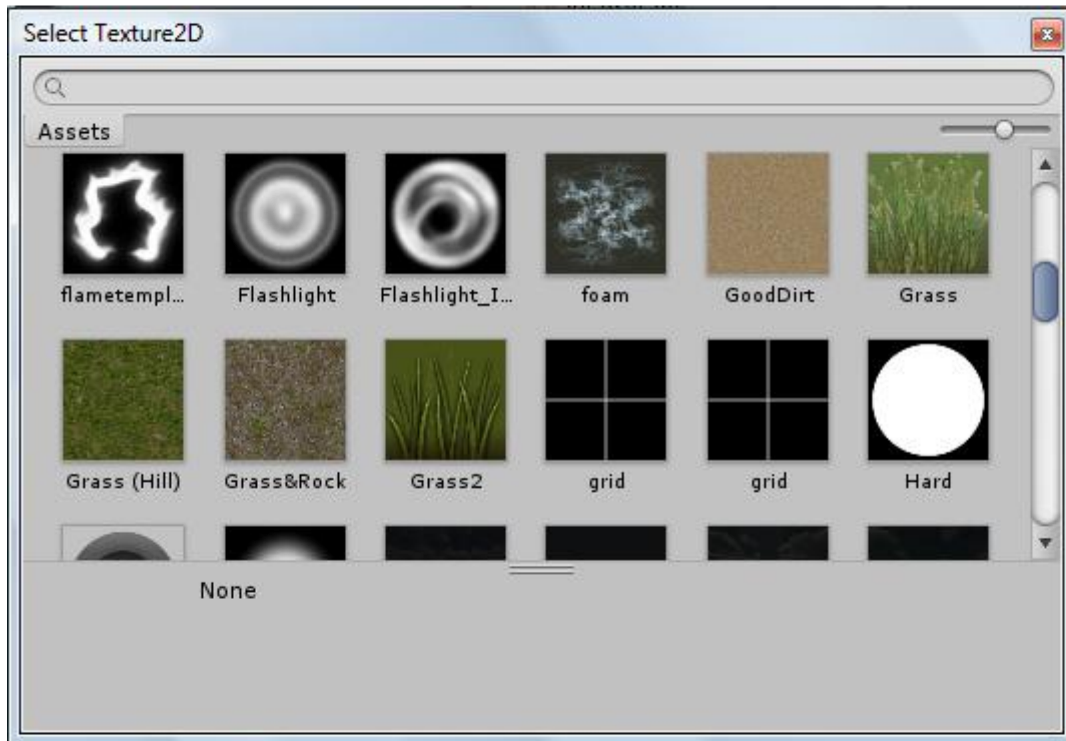


Figura 4.30 Ventana de selección de textura

Donde aparecen todas las texturas que tenemos. En la barra superior de búsqueda, podemos escribir parte del nombre de la textura para encontrarla rápidamente. Además podemos utilizar la barra de desplazamiento en la parte superior derecha para ajustar el tamaño con el que se previsualizan las texturas.

Vamos a seleccionar como textura base, por ejemplo la que tiene por nombre: “Grass (Hill)” y dejamos el ancho y el largo de la textura como están.

Pulsamos finalmente “Add” para añadir la textura a la paleta de texturizado. Al ser la primera textura que seleccionamos, esta cubrirá todo el terreno.

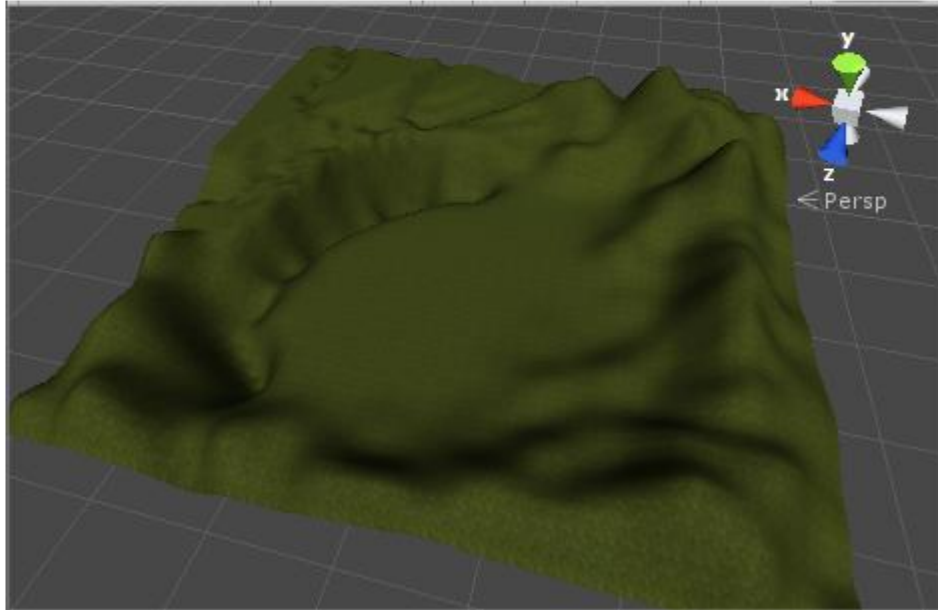


Figura 4.31 Terreno de ejemplo con textura base

Para cargar más texturas en la paleta, hacemos lo mismo de antes, pero en esta ocasión, para pintar con ellas deberemos seleccionar la textura con la que queremos pintar en la vista del inspector y después clicar sobre el terreno para pintar sobre esa zona con la brocha seleccionada.

Vamos a usar la brocha con las nuevas texturas que hayamos cargado para pintar sobre el terreno y a familiarizarnos con los valores de opacidad y fuerza objetivo para conseguir que las texturas se fundan entre ellas con naturalidad.

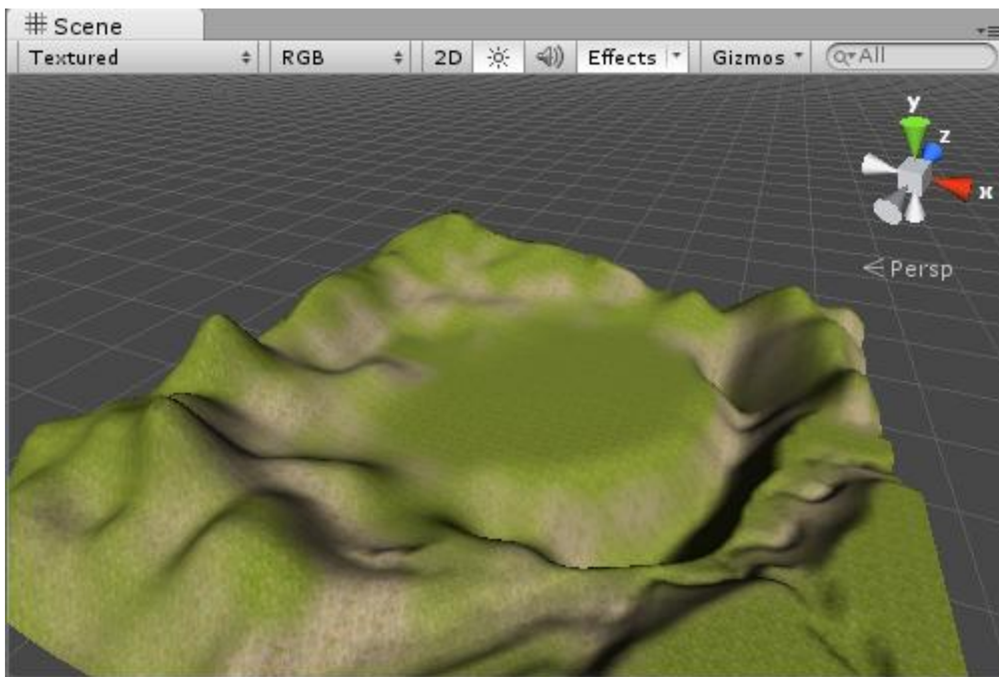


Figura 4.32

Botón de árboles

Vamos a pasar a ocuparnos de los elementos de la naturaleza que se sitúan encima del terreno, en este caso los árboles.

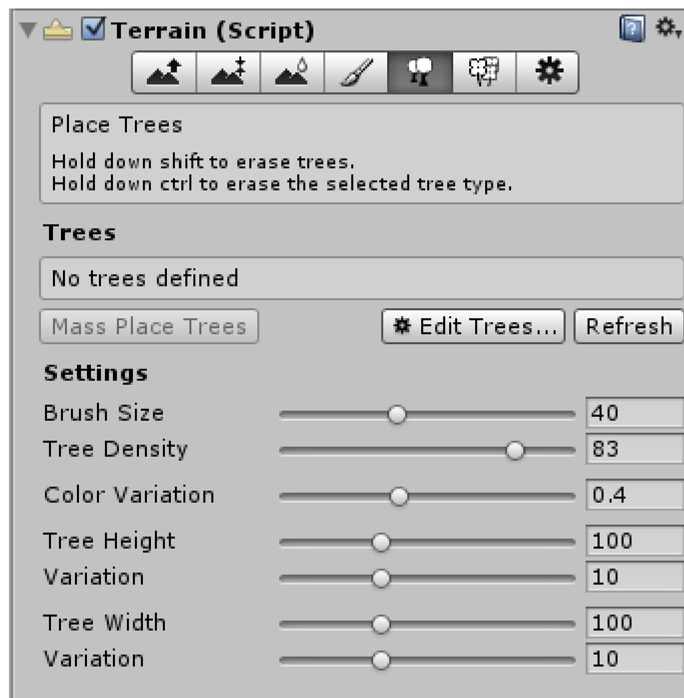


Figura 4.33 Botón de árboles seleccionado

Este menú es similar al de texturizado, ya que nos permite cargar los árboles de entre los recursos de nuestro proyecto, además de ofrecernos una serie de barras de desplazamiento para configurar su aspecto final.

En este menú no hay brochas, por lo que a la hora de colocar árboles lo haremos con la brocha circular por defecto, pudiendo cambiar sólo su tamaño (`Brush Size`).

`Tree Density` o densidad de árboles: indica la densidad de árboles que se dibujarán al hacer clic sobre el terreno. Es similar a la opacidad que hemos visto anteriormente.

`Color variation` o variación de color: nos indica cómo de diferentes serán unos árboles de otros en cuanto a color se refiere. Siempre interesa que los árboles tengan alguna pequeña diferencia entre sí, así que es mejor mantener este valor por encima de 0 pero sin excederse.

`Tree Height` o altura de los árboles y su variación (`Variation`): Establece una altura de base en tanto por ciento respecto del árbol original y una variación, también porcentual, en torno a ella para cada uno de los árboles.

`Tree width` y su variación: establece una anchura base en tanto por ciento respecto del árbol original y una variación, también porcentual en torno a ella para cada árbol dibujado.

Vamos a poner árboles en nuestro terreno:

En la vista del inspector, hacemos clic en el botón `Edit Trees->Add Tree` y se abrirá una ventana donde podremos seleccionar un árbol de los recursos de nuestro proyecto y asignarle un factor de doblado (`Bend factor`), que son los grados máximos que podrá doblarse el árbol en caso de que añadamos una zona de viento.

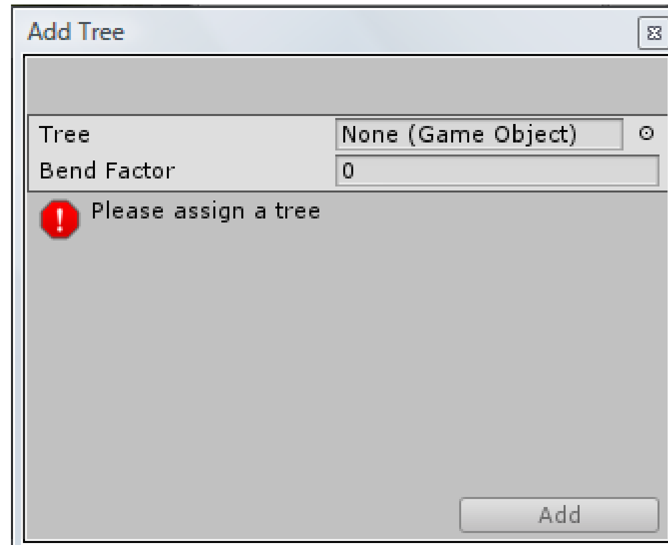


Figura 4.34

Vamos a hacer clic en el botoncito con forma de disco situado a la derecha de `Tree` para abrir la ventana con los recursos disponibles en el proyecto.

Hacemos clic en el árbol que queramos y pulsamos intro para confirmar. Establecemos un factor de doblado (`Bend factor`), por ejemplo 5 grados y pulsamos "Add" para añadirle a nuestra paleta de árboles.

En la vista del inspector configuramos como queramos los parámetros: tamaño de la brocha, densidad, variaciones de color, etc. Y pintamos el bosque sobre nuestro terreno, generalmente añadiremos los árboles en las zonas lisas.

Si en algún momento queremos borrar árboles, podemos hacerlo, manteniendo pulsado Shift mientras hacemos clic.

Si en algún momento queremos sustituir todos los árboles dibujados por otro distintos, podemos hacerlo, seleccionando el árbol en cuestión de la paleta, pulsando el botón `Edit Trees->Edit Tree` y seleccionando otro. También podemos borrar todos los árboles de un tipo determinado de la misma manera con `Edit Trees->Remove Tree`.

Si queremos poner árboles uno a uno, podemos hacerlo estableciendo el tamaño de la brocha a uno en la vista del inspector.

Figura 4.35 Terreno con palmeras

Botón de vegetación

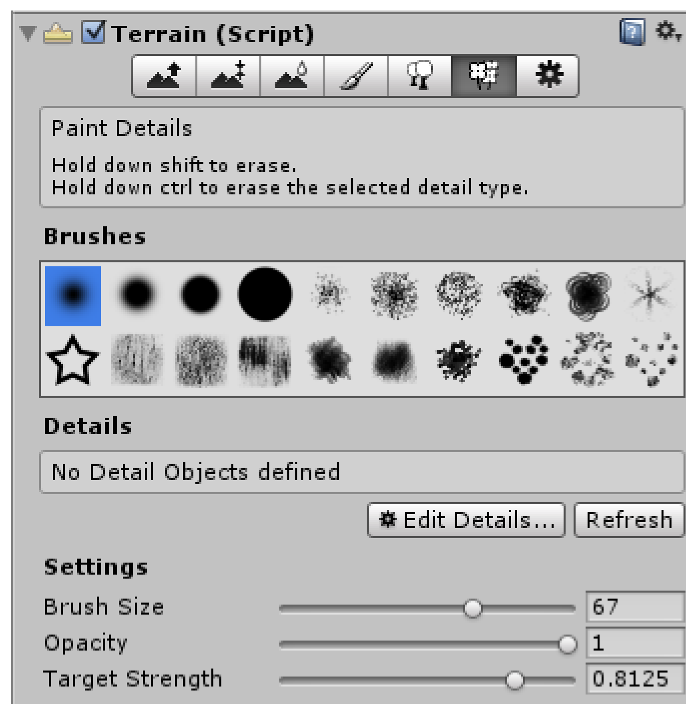


Figura 4.36 Botón de vegetación seleccionado

Este menú es muy similar al de árboles, ya que nos permite cargar los elementos de vegetación de entre los recursos de nuestro proyecto y además nos ofrece una serie de barras de desplazamiento para configurar su aspecto final.

En este menú en la vista del inspector volvemos a tener brochas y podemos cambiar su tamaño (`Brush size`) y opacidad (`Opacity`) que indica la cantidad de vegetación que se dibujará sobre el terreno cuando hagamos clic.

Al igual que con el texturizado, también tenemos el factor de fuerza objetivo (`Target Strength`) con el que podemos indicar en tanto por uno para los sucesivos tipos de césped hasta qué punto queremos que se sustituya el césped actual por el anterior sobre el que se pinta.

En la vista del inspector vamos a hacer clic en `Edit Details->Add Grass Texture` y se abrirá un menú donde podemos seleccionar la textura del césped.

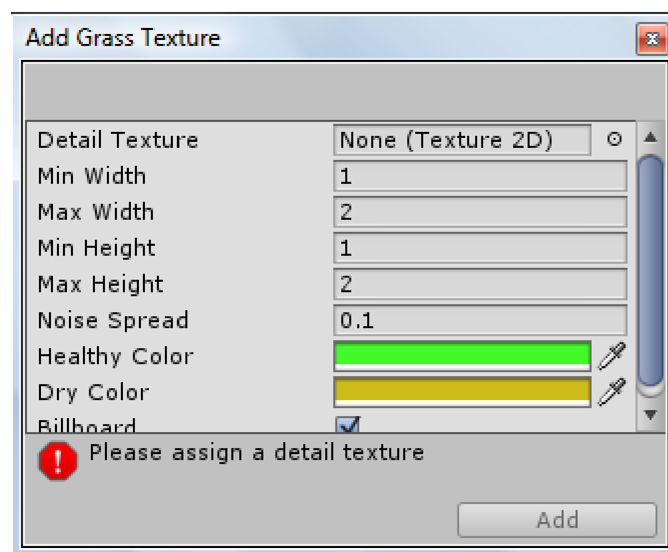


Figura 4.37 Ventana “Add Grass Texture”

Además de la textura a utilizar, podemos ajustar parámetros como la anchura (`width`) mínima y máxima medida en metros del césped plantado y de la misma forma con su altura (`Height`).

`Noise Spread` o factor de ruido: Nos permite establecer un factor en tanto por uno para deformar aleatoriamente cada césped, para que no sea siempre igual.

Las dos últimas casillas son celdas de color que pueden abrirse haciendo clic sobre ellas. Podemos asignar un color para el césped saludable (`Healthy color`) y otro para el césped seco (`Dry color`). Entre esos colores se teñirá aleatoriamente el césped por zonas.

Una vez hayamos configurado los parámetros del césped vamos a hacer clic en el icono con forma de disco a la derecha del menú de selección de textura para abrir los recursos de textura disponibles en el proyecto. Se abrirá la ventana de selección. Al aparecer las mismas texturas que en el menú de texturizado del terreno tendremos que distinguir cuáles son válidas como texturas de césped y cuáles no.

Generalmente, las texturas de césped muestran la vista frontal de una planta dibujada sobre un fondo de un solo color, mientras que las texturas de terreno son perfectamente cuadradas y con un relleno homogéneo.

El césped es uno de los recursos gráficos que mayor consumo de memoria supone, por lo tanto la distancia a la que podemos verlo es relativamente baja. Al pintar césped sobre el terreno no se aprecia que aparezca, pero si acercamos la cámara lo máximo posible en la vista de escena veremos cómo a partir de los 30 metros de distancia aproximadamente comienza a verse.

Hay que tener en cuenta que a la hora de pintar el césped, queda más estético pintar cada césped sobre un suelo texturizado con un color similar.

Figura 4.38 Terreno con césped

Configuración del terreno

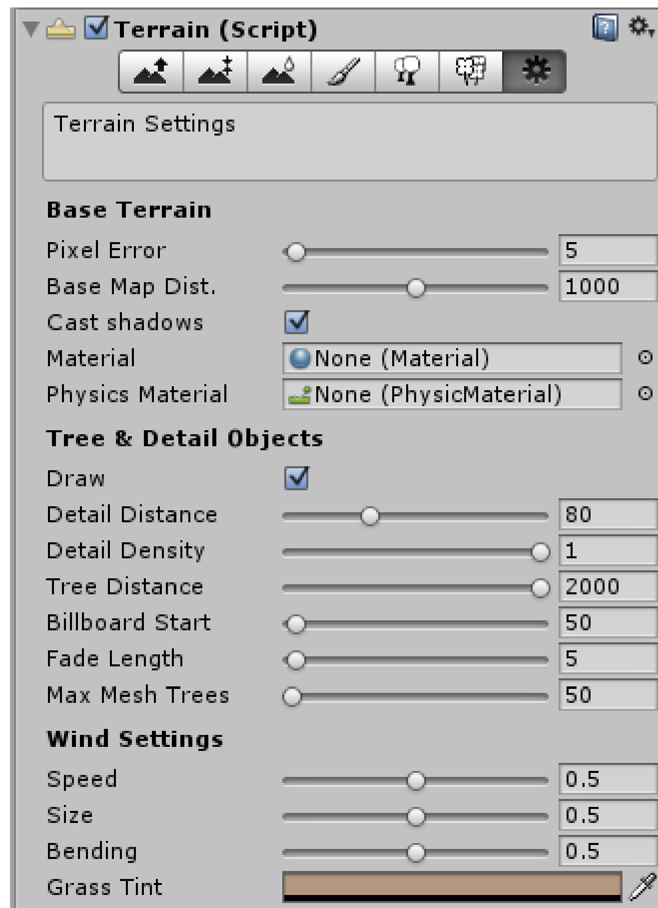


Figura 4.39 Botón de configuración seleccionado

Este menú afecta sobre todo a la calidad del resultado gráfico final de nuestro terreno y también al consumo de memoria, por lo que debe de utilizarse con precaución. Vamos a tratar sólo los factores más importantes.

`Cast shadows` o la proyección de sombras: es una casilla que indica si las elevaciones realizadas sobre el terreno proyectan sombra unas sobre otras. Por defecto está activada pero se puede desactivar para mejorar el rendimiento.

`Detail Distance` o distancia de detalle: indica la distancia en metros desde la cámara a partir de la cual podremos ver el césped.

`Tree Distance` o la distancia de árboles: indica la distancia en metros desde la cámara a partir de la cual podremos ver los árboles. Los árboles no consumen excesiva memoria así que podemos dejar este parámetro como está o reducirlo si nuestro equipo muestra problemas de rendimiento.

`Billboard Distance` o la distancia de cartel: indica la distancia en metros desde la cámara a partir de la cual los árboles serán visibles en baja calidad. Esto nos permite tener la sensación de que los árboles se ven a una gran distancia cuando realmente en la lejanía no se muestra más que un árbol plano en lugar de uno en 3D. La distancia de Billboard nos marca el punto en el cual sucede la transición entre una y otra calidad.

4.4.4.-Añadir un cielo

En Unity hay varias opciones para añadir un cielo o *skybox*. En este ejemplo vamos a añadirlo de la siguiente manera:

Vamos a ir a `Edit->Render Settings` para mostrar las propiedades de renderizado en el inspector.

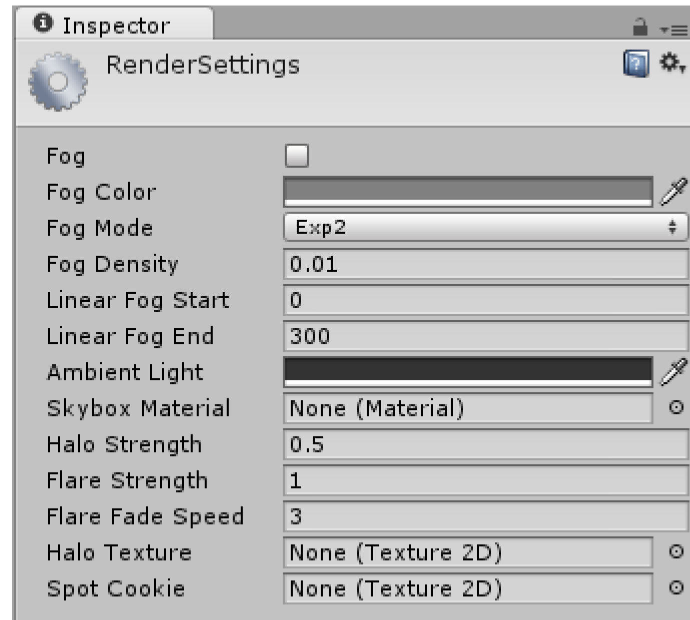


Figura 4.40 Propiedades de renderizado

Entre estas propiedades, veremos una entrada llamada `Skybox Material`. Al hacer clic en el icono con forma de circulito podremos seleccionar un material para el cielo.

Como podremos observar, se muestran, no sólo los materiales para el cielo, si no todos. Si queremos que se muestren sólo los materiales para el cielo, en la barra de búsqueda, en la parte superior de la ventana de materiales, teclearemos: “sky”.

Una vez hayamos seleccionado un material para el cielo lo veremos aparecer en la vista de escena inmediatamente. Si no se puede ver, debemos asegurarnos de que tenemos `Skybox`, `Fog` y `Flares` activados. El botón para activarlos se encuentra en la parte superior de la vista de escena, en `Effects`.

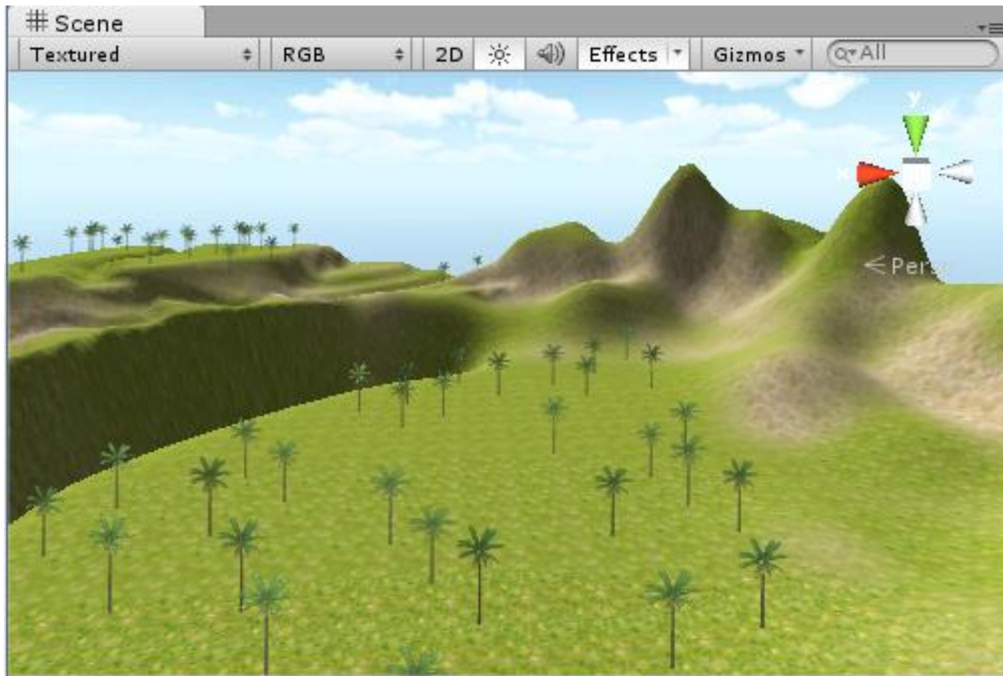


Figura 4.41

Con el cielo en la escena hemos avanzado bastante, ahora parece un mundo para un juego.

4.4.5.-Añadir niebla

Unity incluye un sistema de niebla, que también se encuentra en las opciones de renderizado (`Fog`).

Si activamos la niebla en el *checkbox*, la escena se vuelve brumosa. Debajo del *checkbox* hay algunas propiedades para la niebla.

La primera es el color que por defecto es gris. Se puede cambiar según las necesidades, por ejemplo si trabajamos en un área desértica podemos cambiar el color a amarillo o naranja, si es un mundo mágico, verde, etc.

Está también la densidad de niebla. Una densidad mayor disminuirá la visibilidad y una densidad menor la aumentará.

Figura 4.42

4.4.6.-Color de ambiente

Debajo de las propiedades de la niebla podemos ver `Ambient Light`, por defecto como gris. Si queremos dar a la escena un esquema diferente de iluminación, cambiar la luz ambiental es un buen comienzo.

Para que la luz ambiental se muestre, debemos tener la iluminación activada en la vista de escena.

4.4.7.-Añadir luces

Para añadir una luz tenemos que ir a `Game object->Create other`. Podemos elegir entre cuatro tipos de luces: `Directional light`, `Point Light`, `spotlight` y `area light`. Vamos a seleccionar una luz direccional.

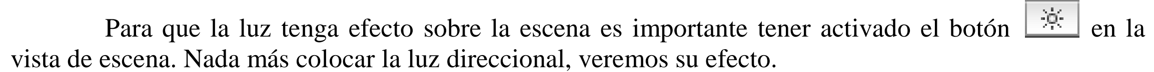
Para que la luz tenga efecto sobre la escena es importante tener activado el botón  en la vista de escena. Nada más colocar la luz direccional, veremos su efecto.

Figura 4.43

Este tipo de luz no varía cambiando su tamaño, ni su posición ya que lo que importa de ella es su dirección y para variarla a nuestro antojo tendríamos que rotar la luz con la herramienta de rotación.

4.4.8.-Añadir Agua

Este elemento podremos añadirlo a nuestra escena gracias a un paquete llamado “Water (basic)” que añadimos al principio, cuando creamos el proyecto. Si en ese momento no lo añadimos, podemos añadirlo ahora haciendo clic derecho en la vista de proyecto, en *assets*, `import package->water (basic)` y damos a import.

Aparecerá una carpeta llamada water (basic). En ella tendremos dos tipos de agua: Daylight Simple Water y Nighttime Simple Water, que son dos tipos de agua muy sencillos y poco realistas que vienen con la versión free de Unity.

Para este ejemplo seleccionaremos Daylight Simple Water. Para poner el agua en la escena, simplemente haremos clic izquierdo en ella y arrastraremos a la escena.

Veremos un pequeño círculo de agua que podremos mover, rotar y escalar como queramos.

Figura 4.44

Para personalizar nuestro agua, tenemos varias opciones en el inspector:

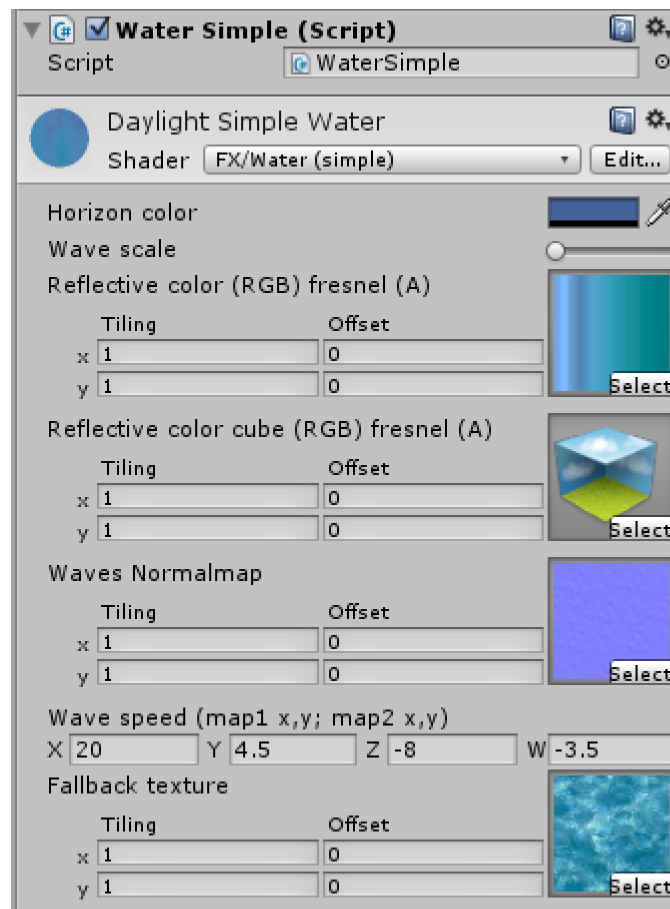


Figura 4.45 Opciones para personalizar el agua

4.4.9.-Añadir un Controlador

Vamos a añadir un controlador para poder movernos a través de nuestro escenario en la vista de juego.

Al crear nuestro proyecto, importamos un paquete llamado “Character controller”. Unity nos ofrece dos tipos de controladores: uno en tercera persona y otro en primera persona.

Vamos a utilizar el controlador en primera persona. Para ello hacemos clic con el botón izquierdo del ratón sobre este y lo arrastramos a la vista de escena.

Este controlador está representado por una cápsula que será el jugador, en este caso, nosotros.

Ahora la vista de juego nos muestra lo que ve el controlador en primera persona. Una vez añadido nuestro controlador en primera persona, cuando probemos el juego dando a play podremos mover la cámara con el ratón, podemos caminar con las teclas de dirección, o con w, s, a, d y saltar con la tecla espacio, de esta manera podremos probar nuestro escenario e interactuar con él como si estuviéramos en un videojuego en primera persona.

Es importante que situemos el controlador totalmente por encima de nuestro escenario, porque si lo colocamos penetrando en el suelo, al darle al play, caería al vacío.

Figura 4.46 Controlador en primera persona

4.5.-Game Objects

4.5.1.-Concepto de objeto de juego

Las escenas del videojuego están compuestas de objetos de juego. Son entidades que podemos ver en la vista de jerarquía y que están constituidas por una serie de atributos que podemos configurar en la vista del inspector.

Hay objetos de juego muy básicos, como un cubo o una luz puntual, pero también podemos añadir componentes a un objeto de juego básico para hacer de él una entidad más compleja.

En este punto trabajaremos con los objetos de juego más básicos que ofrece Unity. Todos éstos se encuentran disponibles en la pestaña superior `GameObject`.

4.5.2.-Objetos vacíos

En la pestaña `GameObject`, la primera opción es `Create Empty` (Crear objeto vacío) y esta nos permite añadir un objeto vacío a nuestra escena.

Los objetos vacíos, no tienen utilidad por sí solos, pero son útiles para establecer jerarquías de varios objetos. Por ejemplo, si tenemos un escuadrón de naves jerarquizado dentro de un mismo objeto vacío, podremos controlar todas las naves desde ese objeto en lugar de una a una.

Otra utilidad que tienen los objetos vacíos es la composición de objetos más complejos mediante adicción de componentes. Los objetos de juego básicos de Unity, a pesar de ser básicos, en ocasiones tienen campos que se pueden modificar en el inspector. Por ejemplo, todas las cámaras vienen con un componente que escucha el audio encargado de ajustar el volumen de cada sonido en función de la distancia de la cámara. Por ejemplo, pongámonos en la situación de que queremos tener una cámara que se ocupe de renderizar un retrovisor, en ese caso, no nos hará falta ese componente y a lo mejor nos interesa componer nuestra propia cámara, más sencilla, a partir de un objeto vacío.

4.5.3.-Resto de objetos básicos.

En la pestaña `GameObject`, debajo de `Create empty`, tenemos la opción `Create other` (crear otro). Esta opción, nos da acceso a la lista de objetos de juego básicos de Unity.

Vamos a verlos uno por uno en profundidad:

Sistema de partículas

Los sistemas de partículas consisten en una fuente de emisión de planos texturizados que siempre muestran sus caras hacia la cámara. Permiten generar efectos como fuego, humo, lluvia, hechizos, etc.

El sistema de partículas de Unity se llama Shuriken y es uno de los más utilizados en videojuegos de última generación.

Vamos a realizar un ejemplo: Crearemos un pequeño fuego, muy útil en los juegos si queremos usar antorchas, hogueras, etc.

Empezaremos por una textura base. Esta puede ser de cualquier tipo, desde una imagen de fuego buscada directamente en internet, a un dibujo que podamos hacer nosotros en photoshop, lo único que tiene que tener es un fondo negro y que tenga la transparencia en el canal alpha de nuestra textura.

Una vez que tenemos la imagen, vamos a crear tres carpetas que vamos a usar: "Prefabs", donde guardaremos la partícula final, "textures", donde guardaremos esta textura que vamos a usar y "materials", donde guardaremos el material asociado a estas partículas.

Primero, importaremos esta textura que vamos a usar de base para el sistema de partículas en la carpeta Textures. Después vamos a la carpeta Materials y ahí con el botón derecho elegimos crear un nuevo material. Una vez creado, lo elegimos, por ejemplo de tipo `mobile->particles->additive`, así podremos usarla tanto en dispositivos móviles como en aquellos que no lo sean.

Una vez creado el material arrastramos directamente nuestra textura al material para tenerlo, quedándonos algo de este estilo:

Figura 4.47

En el que podemos observar como ya nuestro material tiene transparencia donde no se encuentra el fuego.

Vamos a empezar a crear el sistema de partículas en sí. Para ello, creamos un nuevo sistema de partículas en nuestra escena: `Game Object->Create other->Particle system` y lo posicionamos en el suelo de nuestra escena. Una vez posicionado en la escena, veremos que nos da toda una lista de opciones, que veremos y comentaremos a continuación:

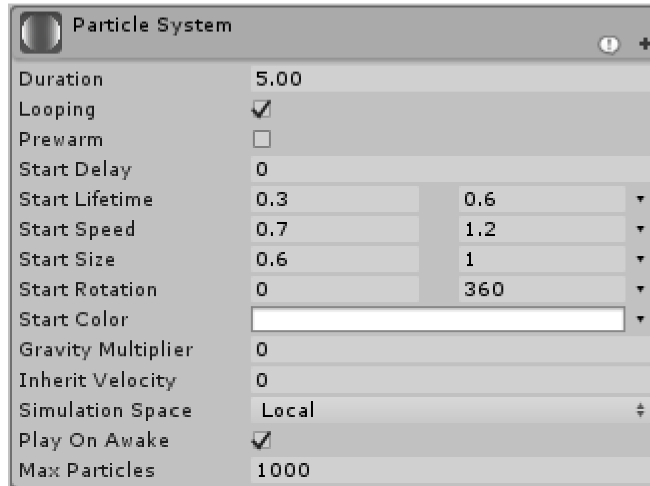


Figura 4.48 Parámetros de un sistema de partículas

En esta parte tenemos que meter unos valores variables en `Start LifeTime`, `Start speed`, `start size` y `start rotation` para conseguir un patrón de creación de partículas para el fuego en el que tengan distintas velocidades a la hora de moverse desde su nacimiento hasta su extinción, así como distintos tamaños para dar más sensación aleatoriedad (entre 0.6 y 1, por ejemplo) y de giro (entre 0 y 360). Hay que pensar que el fuego es una de las partículas que más aleatoriamente se generan, así que aquí se puede jugar directamente con los valores hasta encontrar los que mejor queden, pero por ejemplo una configuración como la de la imagen anterior estaría perfecta como base para empezar a trabajar.

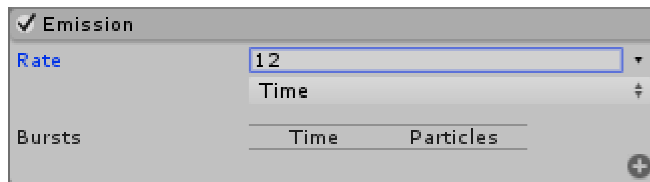


Figura 4.49

En `Emission` (ver Figura 4.49) marcamos la cantidad de partículas que queremos emitir, en este caso por unidad de tiempo, así que una cantidad de 10 ó 12 es un buen valor de base. Siempre se puede cambiar ese valor según la intensidad del fuego que queramos crear y no hay que olvidar marcar que queremos que esta emisión se haga a lo largo del tiempo, no de la distancia.

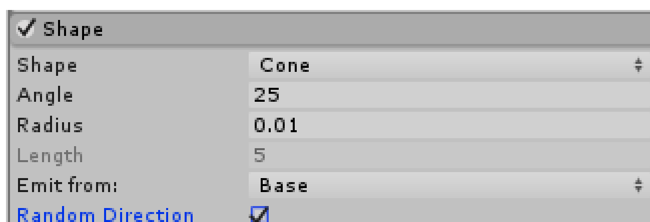


Figura 4.50

En `Shape` (ver Figura 4.50) controlamos la forma en la que se van a dispersar estas partículas desde su creación hasta su extinción. Si queremos un fuego de una hoguera o de una antorcha pondremos en `shape` que queremos un cone, en el que añadiremos un ángulo medio de unos 25 grados y un radio lo más pequeño que podamos, porque este valor controla el punto de emisión de partículas al nacer, con lo que cuanto más alto lo tengamos, más dispersión encontraremos en la base y para fuegos del estilo de antorchas o de hogueras lo mejor es tener un punto cuanto más concentrado mejor. Finalmente le diremos que queremos que emita esas partículas desde la base.

Otro valor bastante importante es la variable de color `over lifetime`. Con este valor podemos conseguir que las partículas, tanto en su nacimiento como en su muerte no tengan siempre la misma opacidad. En casos como el fuego, siempre partimos de unas partículas que se crean con poca opacidad, alcanzan su cota máxima de color y luego según siguen subiendo se van destruyendo y haciéndose cada vez más transparentes. La mejor manera de conseguir esto es usando un gradiente en esta opción y para ello nada mejor que ilustrar este comportamiento con colores. Como lo que queremos es que las partículas nazcan con cierta transparencia, hacemos que empiece en un color gris medio, poco a poco va pasando a un color blanco puro, que alcanza al poco de crearse y finalmente, al final le ponemos un color negro que hará que la partícula, según va avanzando el tiempo vaya haciéndose cada vez más transparente hasta que finalmente desaparezca, de ahí el usar un color negro al final.

Nos tendría que quedar algo de este estilo:

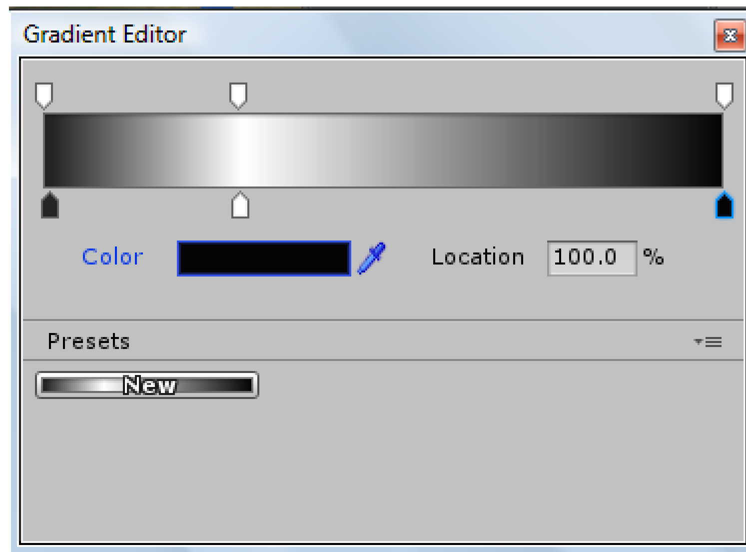


Figura 4.51 Editor de gradiente

El siguiente valor sería `Size over lifetime`. De este poco hay que explicar, ya que lo que queremos hacer es que las partículas a lo largo del tiempo pasen de su tamaño de creación a que desaparezca completamente, así que para eso podemos usar unos valores de este estilo para conseguir el efecto que buscamos:

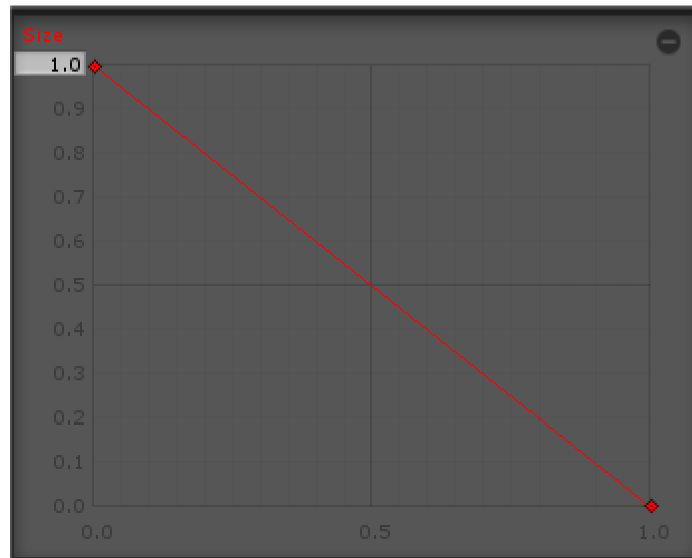


Figura 4.52

Otro valor que puede servirnos para darle más realismo a estas partículas de fuego es usar `rotation over lifetime` (ver Figura 4.53), que lo que hace es entre los valores que le pongamos (Random between two constants) hacer que las partículas a lo largo de su vida vayan realizando un giro. Aquí hay que tener cuidado, porque si bien puede quedar bastante bien un giro pequeño (por ejemplo entre -60 y 60) el usar valores muy altos puede hacer que las llamas finales queden totalmente irreales. También comentar que aquí también es bueno usar, siempre que se pueda, valores en el rango de los valores negativos y positivos, porque esos dos valores no indican en ningún momento el valor de rotación en valores absolutos, sino que lo que hacemos poniendo a un lado valores negativos y a otro positivos es decirle que puedan girar tanto a la izquierda como a la derecha a partir de su posición de creación.

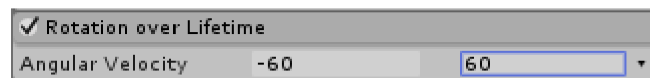


Figura 4.53

Finalmente y para terminar, nos ocupamos de la parte más importante, la de `Renderer` (ver Figura 4.54), donde le vamos a decir cómo queremos que esa partícula se muestre finalmente y donde le vamos a poner nuestro material para que finalmente se vea como queremos. En primer lugar, como `Render mode` vamos a elegir Billboard, que es la manera que tenemos de decirle a nuestro sistema de partículas que nos genere planos que directamente van a estar enfocados a la cámara, que es la mejor manera para representar nuestro fuego en pantalla.

En segundo lugar, donde pone `material`, simplemente tenemos que agregar nuestro material que anteriormente creamos para poder ver el aspecto definitivo de nuestras partículas. Decir que este paso se puede hacer el primero de todos a la hora de empezar a configurar los valores del sistema de partículas, ya que así ya vamos viendo de una manera más directa como está quedando nuestro fuego o cualquier otra cosa que estamos creando.

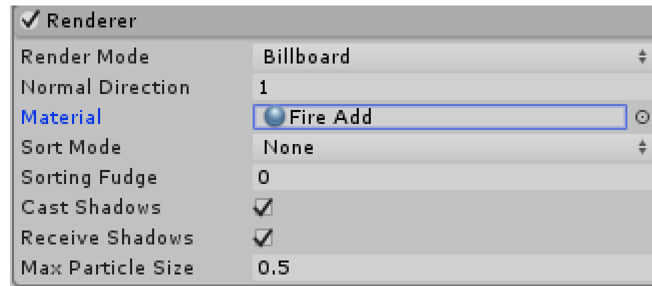


Figura 4.54

Finalmente, una vez hechos los cambios necesarios en nuestro sistema de partículas, simplemente arrastrándolo de la jerarquía de nuestro proyecto en Unity, a la carpeta que creamos anteriormente en el proyecto, llamada "prefabs" podremos tenerlo disponible para usarlo siempre que queramos.

Figura 4.55 Pequeño fuego hecho a partir de un sistema de partículas

Cámaras

Las cámaras son las encargadas de renderizar, es decir, dibujar el resultado de la escena de juego en la pantalla. Aunque al crear una nueva escena Unity siempre aparece en ella por defecto una cámara, es posible añadir más.

Las cámaras no sólo sirven para el dibujado total de la escena, pueden ser usadas para simular retrovisores, pueden proyectar lo que capturan sobre un objeto para simular una cámara de videovigilancia, pueden usarse en conjunto para que cada una de ellas renderice en una sección distinta de la pantalla consiguiendo el efecto de pantalla partida para videojuegos de varios jugadores.

Vamos a crear una nueva cámara en la escena. En primer lugar, en la vista de jerarquía, localizamos la cámara principal que viene incluida en toda nueva escena Unity. Hacemos clic derecho sobre ella y seleccionamos `delete` para eliminarla de la escena.

Comprobaremos que la vista de juego ahora no muestra nada, ya que no existe en la escena ninguna cámara que pueda renderizar el entorno.

Ahora seleccionamos en la pestaña superior `Game Object->Create Other->Camera` y observaremos que aparece en la vista de jerarquía y que además la vista de juego vuelve a mostrar un renderizado.

Vamos a seleccionar la cámara haciendo clic sobre ella en la vista de jerarquía y a observar sus parámetros en la vista del inspector.

El primero se llama `clear flags`. Digamos que las `clear flags` son las encargadas de limpiar la cámara cada vez que unity manda dibujar a nuestra tarjeta de video y cambia el dibujo por así decirlo. Tiene varias opciones: `skybox`, `solid color` o color sólido, `depth only` o sólo con profundidad y `don´t clear`, que sería no limpiar.

Luego tenemos el `Background` o color de fondo. Si hacemos clic en él podremos seleccionar el color que queramos.

`Culling mask`: Es lo que queremos que se renderice basado en *layers* o capas. Esto sirve para optimizar el juego. Si queremos que no renderice cierta cámara alguna parte, podemos decirle que ignore una de las *layers*.

`Projection`: Qué tipo de proyección queremos, esto de proyección se refiere a la capacidad de la cámara para simular la perspectiva. Hay dos tipos de proyecciones, `perspective` en el que hay perspectiva simulada, imitando a la realidad y `orthographic` donde no hay una simulación de perspectiva y los objetos parece que estén a la misma distancia.

`Field of view` o campo de visión: Es el ángulo de visión de la cámara, es decir, lo que puede renderizar la cámara.

`Clipping planes`: Tenemos dos, `Near` y `far`, `near` es el plano que está más cerca de la cámara y `far` el que está más lejos.

En el campo `Viewport Rect`, los apartados numéricos `X` e `Y` indican la coordenada de pantalla inicial donde la cámara mostrará su contenido. Por defecto sus valores son cero y cero e indican que el dibujado se realiza desde la parte inferior izquierda. Si modificamos esos valores haciendo clic sobre ellos y ajustándolos a 0,5 y 0,5 comprobaremos que en la vista de juego el dibujado ahora sólo se realiza en el cuadrante superior derecho.

`Depth`: Es para dar un orden de renderizado a las cámaras cuando tenemos varias.

`Rendering Path`: Cómo queremos que renderice nuestra cámara la luz, la iluminación.

Textos GUI

Los textos de GUI (*Graphic user interfaz* o interfaz gráfica de usuario) son uno de los elementos Unity que no se renderizan en 3D.

Los elementos de GUI en general se renderizan en 2D y siempre en primer plano, por delante del resto de elementos en 3D. Generalmente se utilizan para mostrar información al jugador.

Para crear un objeto GUI Text hay que seleccionar: `Game Object->Create Other->GUI text`. Comprobaremos que el nuevo objeto aparece en la vista de jerarquía y que además tanto en la vista de escena como en la vista de juego podemos observar un texto simple en el centro.

Descripción de las propiedades de Gui Text:

`Text`: Es una variable que contiene el texto que se muestra en el GUI.

`Anchor`: Permite indicar dónde se colocará el ancla o fijación del texto.

`Alignment`: La alineación del texto.

`Pixel Offset`: El desplazamiento en pixeles del texto desde su posición inicial en base a los valores contenidos en un Vector2.

`Line Spacing`: El multiplicador del espacio de interlineado. Esta cantidad será multiplicada por el espacio de línea definido en la fuente.

`Tab Size`: El multiplicador de la anchura del tab. Esta cantidad se multiplicará con la anchura de tab definida en la fuente.

`Font`: La fuente usada para el texto. Podemos asignar una de entre las que tengamos disponibles.

`Material`: El material usado para renderizar el texto.

`Font size`: El tamaño de fuente a usar (para fuentes dinámicas).

`Font style`: Lo mismo que la anterior para el estilo de Fuentes dinámicas.

Texturas de GUI

Las texturas de GUI también se renderizan en 2D y por delante del resto de elementos 3D de la escena. A diferencia de los textos, se trata de simples imágenes que pueden hacer la vez de iconos o marcos dentro de los cuales dibujar los textos de GUI.

Para crear una GUI texture: En las pestañas superiores seleccionamos `Game Object->Create other->GUI texture`.

Observaremos como aparece en la Vista de jerarquía y haremos clic sobre ella para seleccionarla. También comprobaremos que tanto en la vista de escena como en la vista de juego aparece una textura dibujada en la parte central.

Propiedades:

`Texture`: La textura usada para el dibujo.

`Color`: El color de la textura de la GUI.

`Pixel Inset`: Inserción en pixels usada para ajustar el tamaño y la posición.

Luces

En Unity tenemos varios tipos de luces:

- Luz direccional o Directional light: Este es un tipo de luz en la que lo que importa es la dirección, no importa su posición, si no, hacia donde está ubicada. Si la rotamos cambia la iluminación y vemos que si la trasladamos e incluso si la escalamos no variaría nada la iluminación.

  Este tipo de luz tiene varias propiedades en el inspector:

  `Color`: Podemos ajustar el color de la luz para crear diferentes ambientes, por ejemplo un ambiente frío con un color azul.

  `Intensidad`: Si queremos que nuestra luz brille más o menos.

  `Shadow type`: En este menú desplegable podemos seleccionar qué tipo de sombras queremos que cree nuestra luz sobre el terreno o por el contrario que no cree sombras. Las `Hard Shadows` son menos pesadas que las `Soft Shadows` por lo tanto también son más rápidas.

  `Draw halo`: Si queremos que nuestra luz tenga un halo esférico de luz.

  `Flare`: Si queremos que nuestra luz tenga un efecto *flare*, como el sol.

- Spot light: Esta luz es el equivalente a un foco, solamente ilumina un área y no provoca sombras. De esta luz sí que importa su posición y también su rotación.

  Si nos fijamos en la vista del inspector, veremos que tiene un rango con el cual podemos especificar a cuánta distancia puede iluminar nuestra luz.

  También tiene un ángulo, que es el ángulo de apertura de la luz.

  Los demás parámetros serían los mismos que los de la luz direccional.

- Point light o luz puntual: Como su propio nombre indica, es un punto que emana luz.

  Sus propiedades son las mismas que las de la luz direccional y la luz spot.

Cube, Sphere, Capsulle, etc.

Son figuras básicas en 3D, compuestas por un componente de Malla (`Mesh filter`) que determina su geometría poligonal, un renderizador de malla (`Mesh Renderer`) que se encarga de mostrarlo si está visible para la cámara y un colisionador (`Collider`) que se encarga de que no sea un objeto traspasable a nivel físico.

Clase `Meshfilter`

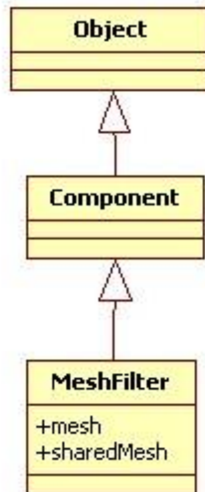


Figura 4.56 Diagrama de clases de la clase MeshFilter

Un mesh filter (o filtro de malla) toma una malla de la carpeta de *assets* y se la pasa a mesh renderer para renderizarla en la pantalla. Es, para entendernos, la estructura de alambre de la malla.

VARIABLES:

mesh:

var mesh : Mesh

Devuelve la instancia de malla asignada al mesh filter. Si no se ha asignado ninguna malla al mesh filter se creará y asignará una nueva.

Si la malla asignada al mesh filter es compartida por otros objetos, automáticamente se duplicará y la malla instanciada será retornada.

Usando las propiedades de malla podemos modificar la malla para un solo objeto solamente. Los otros objetos que usan la misma malla no serán modificados.

sharedMesh:

var sharedMesh : Mesh

Devuelve la malla compartida del mesh filter.

Es recomendable usar esta función sólo para leer datos de malla y no para escribirlos, dado que podríamos modificar los *assets* importados y todos los objetos que usan esta malla podrían ser afectados. Tengamos en cuenta también que no será posible deshacer los cambios hechos a esta malla.

Clase MeshRenderer

Un *render* vendría a ser el proceso por el que un ordenador muestra una imagen. De ahí que sea incuestionable la importancia de esta clase dentro de Unity. Tanto los *gameobjects* como algunos componentes tiene una propiedad renderer a la que podemos acceder/modificar, o que incluso se puede deshabilitar para hacer dicho *gameobject* o componente invisible.

VARIABLES:

`enabled`:

`var enabled : boolean`

Hace el objeto visible (true) o invisible (false).

`castShadows`:

`var castShadows : boolean`

¿Proyecta sombras este objeto?

`receiveShadows`:

`var receiveShadows : boolean`

¿Recibe sombras este objeto?

`material`:

`var material : Material`

El material de nuestro objeto. Modificar esta variable sólo cambiará el material para este objeto. Si el material que asignamos a nuestro objeto está siendo usado para otros *renders*, se clonará el material compartido y se asignará una copia de dicho material para nuestro objeto.

`sharedMaterial`:

`var sharedMaterial : Material`

Hace referencia al material que nuestro objeto comparte con otros. Modificar esta variable cambiará la apariencia de todos los objetos que usen este material y las propiedades del mismo que estén almacenadas en el proyecto también, razón por la que no es recomendable modificar materiales retornados por `sharedMaterial`. Si queremos modificar el material de un *renderer* usaremos la variable `material` en su lugar.

`sharedMaterials`:

`var sharedMaterials : Material[]`

Devuelve un array con todos los materiales compartidos de este objeto, a diferencia de `sharedMaterial` y `material`, que sólo devuelven el primer material usado si el objeto tiene más de uno. Unity soporta que un objeto use múltiples materiales.

Al igual que la variable anterior y por la misma razón, no es aconsejable modificar materiales devueltos por esta variable.

`materials`:

`var materials : Material[]`

Devuelve un array con todos los materiales usados por el *renderer*.

bounds:

var bounds : Bounds

Es una variable de sólo lectura que indica los límites del volumen del *renderer*. A cada *renderer* Unity le asigna una caja invisible que contiene (que envuelve) al objeto y cuyos bordes están alineados con los ejes globales. De esta manera a Unity le resulta más sencillo hacer cálculos de desplazamiento y situación. Por ejemplo, `renderer.bounds.center` normalmente es más preciso para indicar el centro de un objeto que `transform.position`, especialmente si el objeto no es simétrico. Pensemos, para aproximarnos intuitivamente al concepto, en esas rosas que vienen dentro de cajas transparentes de plástico.

lightmapIndex:

var lightmapIndex : int

El índice del *lightmap* aplicado a este *renderer*. El índice se refiere al array de *lightmaps* que está en la clase `LightmapSettings`. Un valor de 255 significa que no se ha asignado ningún *lightmap*, por lo que se usa el que está por defecto.

Una escena puede tener varias *lightmaps* almacenados en ella, y el *renderer* puede usar uno o varios. Esto hace posible tener el mismo material en varios objetos, mientras cada objeto puede referirse a un *ligthmap* diferente o diferente porción de un *lightmap*.

isVisible:

var isVisible : boolean

Variable de sólo lectura que indica si el *renderer* es visible en alguna cámara.

Hay que tener presente que un objeto es considerado visible para Unity cuando necesita ser renderizado en la escena. Podría por lo tanto no ser visible por ninguna cámara, pero todavía necesitar ser renderizado (para proyectar sombras, por ejemplo).

FUNCIONES:

OnBecameVisible:

function OnBecameVisible () : void

Esta función (del tipo mensaje enviado) es llamada cuando el objeto se vuelve visible para alguna cámara. Este mensaje es enviado a todos los *scripts* vinculados al *renderer*. Es útil para evitar cálculos que son sólo necesarios cuando el objeto es visible.

Recordemos que la sintaxis de este tipo de funciones es distinta, así que si quisiéramos por ejemplo que un *gameobject* fuera visible cuando fuera a salir en una cámara, escribiríamos esto:

```
function OnBecameVisible() {
enabled = true;
}
```

OnBecameInvisible:

function OnBecameInvisible () : void

Es llamada cuando el objeto ya no es visible por ninguna cámara.

Clase Collider

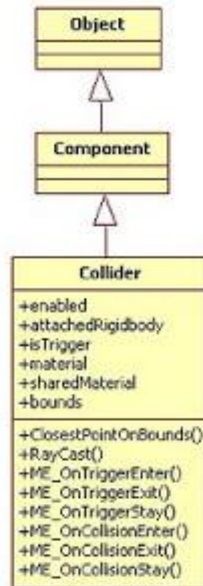


Figura 4.57 Diagrama de clases de la clase Collider

Esta es la clase base para todos los tipos de *colliders* (que en castellano traduciríamos por "colisionadores"). Las clases BoxCollider, SphereCollider, CapsuleCollider y MeshCollider derivan de ella.

Un *collider* vendría a ser la estructura que hace sólidos a los objetos. Seleccionemos en Unity la esfera. En el inspector observamos que tiene un *Sphere Collider*, que no es más que un *collider* con forma de esfera. Si desmarcamos el *checkbox* de dicho *collider* y le damos al play, automáticamente la esfera deja de ser sólida y, por efecto de la gravedad que le da el *rigidbody*, atraviesa el suelo y cae.

Podemos seleccionar el cubo. Obviamente, el cubo no tiene una *sphere collider*, sino una *box collider*. Y si importáramos un *gameobject* capsule, tendría un *capsule Collider*.

El problema viene cuando importamos un *game object* que no tiene una de estas formas primitivas. A veces se puede "encajar" una *capsule collider* en un árbol, o una *box collider* en un coche. A veces no necesitamos que el colisionador de un objeto coincida al 100% con dicho objeto y uno de estos *colliders* básicos nos puede hacer el apaño.

Pero hay veces en que, bien por la importancia del *game object* en el juego, bien por la forma compleja que tiene dicho *game object*, bien en la mayoría de casos por ambas cosas (pensemos en el ninja protagonista de nuestro juego, por ejemplo) necesitamos un *collider* que sea completamente fiel a la forma del *gameobject*. Para ello tenemos el *mesh collider*, que es meramente la malla del objeto convertida en la estructura sólida del mismo que interactúa con el resto del mundo.

Existe bastante confusión respecto la diferencia entre un *collider* y un *rigidbody*. Un *collider*, es meramente la superficie de nuestro objeto. Un *rigidbody* en cambio implica la aplicación de las leyes físicas a dicho objeto. Un objeto puede tener un *collider* y no un *rigidbody* (chocará con otros objetos, aunque no podremos controlar sus reacciones a las colisiones y será Unity quien se encargue de ellas automáticamente), y puede tener un *rigidbody* y no un *collider* (aunque entre otras cosas tendremos que

desactivarle la gravedad, para que no atraviese los suelos). Obviamente, un objeto puede tener un *rigidbody* y un *collider*, y de hecho Unity recomienda que si tu objeto es previsible que vaya a intervenir en muchas colisiones, además de un *collider* es recomendable añadirle un *rigidbody* kinemático.

VARIABLES:

`enabled`:

`var enabled : boolean`

Si el *collider* está habilitado (true), colisionará con otros *colliders*. Se corresponde al *checkbox* que está en el inspector en el apartado del *collider*.

`attachedRididbody`:

`var attachedRigidbody : Rigidbody`

Esta variable hace referencia al *rigidbody* vinculado a este *collider*, y permite acceder a él. Devuelve nulo si el *collider* no tiene *rigidbody*.

Los *colliders* son automáticamente conectados al *rigidbody* relacionado con el mismo *game object* o con algún *game object* padre.

`isTrigger`:

`var isTrigger : boolean`

Si esta variable está habilitada (true) el *collider* se convierte en un *trigger* (lo podríamos traducir por "desencadenante" o "disparador"). Un *trigger* no colisiona con *rigidbodies*, y en su lugar envía los mensajes `OnTriggerEnter`, `OnTriggerExit` y `OnTriggerStay` cuando un *rigidbody* entra o sale de él. De esta manera nos permite a nosotros diseñar de manera específica su comportamiento o las consecuencias de entrar en contacto con un *trigger* (pensemos en una puerta que nos lleva a otra dimensión, que se puede atravesar pero dispara un evento que nos teletransporta, por ejemplo).

`material`:

`var material : PhysicMaterial`

El material usado por el *collider*. Si el material es compartido por varios *colliders*, al ser asignado a esta variable se hará una copia de dicho material que le será asignada al *collider*.

La variable es del tipo `PhysicMaterial` que es una clase que nos permite manejar aspectos de los materiales como la fricción o la capacidad de rebote y el grado de la misma.

`sharedMaterial`:

`var sharedMaterial : PhysicMaterial`

El material compartido de este *collider*. Modificando este material cambiaremos las propiedades de la superficie de todos los *colliders* que estén usando el mismo material. En muchos casos es preferible modificar en su lugar el `Collider.material`.

`bounds`:

`var bounds : Bounds`

Los límites/bordes del *collider* en coordenadas globales.

FUNCIONES:

`ClosestPointOfBounds:`

`function ClosestPointOnBounds (position : Vector3) : Vector3`

Devuelve el punto más cercano de nuestro *Collider* con respecto a un punto dado. Esto puede ser usado para calcular puntos de choque cuando se aplique daño derivado de una explosión.

`Raycast:`

`function Raycast (ray : Ray, hitInfo : RaycastHit, distance : float) : boolean`

Proyecta un rayo que devuelve true si tropieza con algún *collider* en la dirección y distancia indicadas.

Consta de varios parámetros. El primero es de tipo `Ray`, y nos vamos a detener un momento en él.

La estructura `Ray` nos permite crear una línea con un origen y una dirección. Consta de dos variables -de nombre `origin` y `direction`- que no son sino sendos `Vector3` para representar el inicio y la dirección de dicha línea.

El segundo parámetro de la función `Raycast` -`HitInfo`- contendrá información sobre aquello con lo que golpeó el *collider* para el caso de que la función devuelva true, esto es, tropiece con algo.

El tercer parámetro -`distance`- es obviamente la longitud del rayo.

`OnTriggerEnter:`

`function OnTriggerEnter (other : Collider) : void`

Esta función y las que restan forman parte de la categoría de "mensajes enviados".

Al estudiar la variable `isTrigger` decíamos que si nuestro *collider* habilitaba dicha variable, se convertía en un *trigger* (disparador) y dejaba de colisionar con otros *rigidbodies*, y que en lugar de responder a las físicas, al entrar en contacto (o dejar de tenerlo) con ellos lanzaba una serie de mensajes.

`OnTriggerEnter`, así, es llamada cuando nuestro *trigger* entra en contacto con otros *colliders*, de tal manera que podemos escribir dentro de dicha función el código que defina lo que queramos que ocurra cuando se produce dicho contacto.

Este tipo de funciones no pertenecen propiamente a la clase, no usan un operador punto para vincularse a un objeto de una clase, sino que se usan de manera independiente, aunque para su activación requieren que efectivamente un *collider* con el *trigger* habilitado entre en contacto con algo. Es decir, su inclusión en la clase `collider` es más por "afinidad" que porque realmente formen parte de aquella.

`OnTriggerExit:`

`function OnTriggerExit (other : Collider) : void`

Es llamada cuando el *collider* "other" deja de tocar con nuestro *trigger*.

`OnTriggerStay:`

`function OnTriggerStay (other : Collider) : void`

Es llamada casi todos los *frames* que el *collider* other esté tocando nuestro *trigger*.

`OnCollisionEnter:`

`function OnCollisionEnter (collisionInfo : Collision) : void`

Es llamada cuando nuestro *collider/rigidbody* empiece a tocar otro *rigidbody/collider*.

`OnCollisionExit:`

`function OnCollisionExit (collisionInfo : Collision) : void`

Es llamada cuando nuestro *collider/rigidbody* deja de tocar otro *rigidbody/collider*.

`OnCollisionStay:`

`function OnCollisionStay (collisionInfo : Collision) : void`

Es llamada una vez por *frame* por cada *collider/rigidbody* que esté tocando nuestro *rigidbody/collider*.

<u>Clothes</u>

Son los objetos en 3D que se comportan como tejidos tanto a nivel visual como de interacciones físicas (Deformaciones, rupturas, etc.).

La configuración de los parámetros del tejido viene dada por un componente de tejido interactivo (*Interactive Cloth*).

Éstos se pueden crear de dos maneras:

1. `GameObject->Create Other->Cloth`: Creará un plano que podemos usar como *Cloth*.
2. `GameObject->Create Empty`, `Component->Physics->Interactive Cloth & Cloth Render`. De esta manera podemos seleccionar cualquier *mesh* que esté dentro del project y usarlo como *Cloth*.

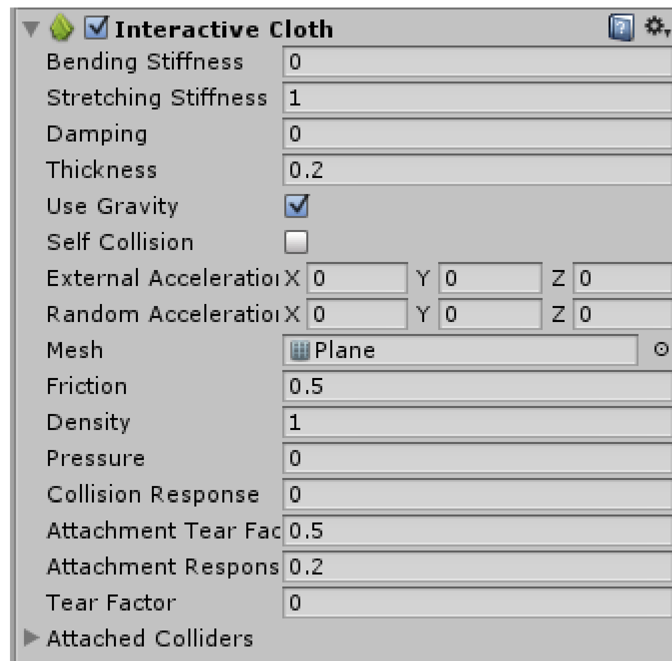


Figura 4.58 Parámetros del componente *Interactive Cloth*

Permite anclar el *cloth* a cualquier objeto dentro de la escena para que quede colgado.

Nota: El objeto a anclar debe de estar lo suficientemente cerca para poder anclarlo.

Vamos a describir cada uno de los parámetros;

`Bending`: Permite simular mayor espesor en la tela.

`Streching`: Permite que la tela se arrugue más al doblarse.

`Damping`: Permite que la tela brinque de acuerdo a las fuerzas aplicadas.

`Thickness`: Permite trasladar la posición a la que se encuentra anclada el *cloth*.

`Use Gravity`: Permite aplicar una fuerza constante de gravedad.

`Self Collision`: Permite que el mismo *cloth* pueda colisionar consigo mismo.

`External Acceleration`: Es la aceleración de una fuerza constante directa aplicada al *cloth*.

`Random Acceleration`: Es la aceleración de una fuerza aleatoria aplicada al *cloth*.

`Mesh`: Asigna un *Mesh* como *interactive cloth* (*Empty Objects*).

`Friction`: Cantidad de fricción del *cloth*, esto evita demasiado rebote o movimiento.

`Density`: La densidad del objeto para hacerlo más pesado.

`Pressure`: Hace que los *Mesh-Cloth* (*softbodies*) se inflen o desinflen al colisionar.

`Collision Response`: Fuerza con la que hará rebotar a los objetos que colisionen con él.

`Attachment Tear Factor`: Cuan lejos tiene que estar pegado el *rigidbody* para romperse.

`Attachment Response`: Cuánta fuerza debe de aplicarse a los *rigidbodies* pegados.

`Tear Factor`: Permite que el *cloth* se pueda romper.

4.5.4.-Mover y alinear con la vista

En numerosas ocasiones nos interesará llevar un objeto a la posición exacta de otro, por ejemplo para ponerle un arma a nuestro personaje o para añadir una luz a una farola.

Una manera fácil de mover un objeto a la posición de otro es la siguiente:

Crearemos un cubo haciendo clic en `Game Object->Create other->Cube`. Nos desplazaremos por la escena hasta un lugar lejano a ese cubo.

Creamos un sistema de partículas haciendo clic en `Game Object->Create Other->Particle system`. En la vista de jerarquía, hacemos doble clic en el cubo para centrar en él la vista de escena.

Seleccionamos el sistema de partículas sin hacer doble clic y por último hacemos clic en `Game Object-> Move to view`.

Observaremos que el sistema de partículas ha sido movido exactamente a la misma posición que el cubo.

Otra opción es `Align with view`, que alineará el objeto seleccionado con la cámara de la vista de escena. Es útil sobre todo a la hora de posicionar cámaras o luces direccionales.

La última opción es `Align with selected` y tiene la misma funcionalidad que hacer doble clic sobre el objeto en la vista de jerarquía.

4.6.-Físicas

4.6.1.-Rigidbody

Vamos a continuar con las físicas. Veremos cómo los objetos reaccionan a diferentes elementos como la gravedad, una fuerza constante, la interacción con otro objeto que lleva también a una colisión, etc.

Partiremos de un terreno bastante simple, con algunos árboles, una luz que será el sol y colinas. Crearemos una pelota: `Game Object->Create Other->Sphere`, la introduciremos una escala: `X`=5, `Y`=5, `Z`=5 y la posicionamos sobre una colina para que cuando la añadamos el componente *rigidbody* caiga por efecto de la gravedad y veamos cómo reacciona. A esta pelota le asignaremos un

material y le añadiremos un *rigidbody* de la siguiente manera: `Component->Physics->Rigidbody`.

Figura 4.59 Pelota sobre colina

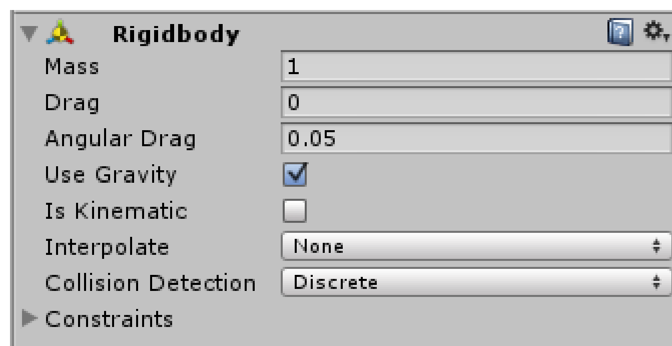


Figura 4.60 Propiedades del componente *rigidbody*

Cuando añadimos un componente *rigidbody* a un objeto lo que hacemos es dotar a ese objeto de propiedades físicas tales como el peso que va a tener ese objeto (`mass`), como va a reaccionar con el aire (`Drag`), como va a rotar ese objeto en contacto con el aire (`angular drag`).

`Use gravity` o usar gravedad: si se activa, el objeto se verá afectado por la gravedad. Daremos al play y veremos que la pelota está afectada por la gravedad ya que tenemos este campo activado y caerá. Además como entrará en colisión con el terreno con las diferentes colinas que hay y otras cosas, veremos cómo va reaccionando de acuerdo a ese contacto o fricción. Evidentemente si borráramos el componente *rigidbody* y diésemos al play, la pelota no se comportaría de ninguna forma.

Otra opción interesante es `Is Kinematic` que es una variable de tipo `booleana`, es decir, se puede activar o desactivar. Esta opción lo que hace es, si está activada, el elemento al cual el *rigidbody* está unido no actuará de acuerdo a las leyes de la física, solamente actuará si nosotros hacemos una transformación sobre ese objeto. Un ejemplo claro de lo que puede ser esta variable activada es que por ejemplo imaginemos impulsores de una mesa de pinball en el cual la bola choca con éstos. Los impulsores no deberían verse afectados por el choque de esa pelota, sin embargo los impulsores si

actuarían con una fuerza determinada sobre esa pelota para volverla a impulsar otra vez hacia arriba. Por tanto esos impulsores no deberían de reaccionar ante la física de esa pelota y sí deberían reaccionar ante la transformación que nosotros introduzcamos vía *scripting*.

Un ejemplo referente a la propiedad `Drag`: Vamos a introducir 200 en ella, recordemos que es cómo influye el aire a ese objeto. Damos al play y veremos que la pelota no se mueve.

Dejamos otra vez la propiedad a 0 y veremos otra cosa bastante interesante referente a las físicas: si nos fijamos en la esfera tiene un componente *sphere collider* (colisión) que viene añadida a ese objeto y veremos que una de las propiedades de este es la opción `material` que no tiene nada que ver con el material que nosotros introducimos y que ya conocemos, si no que es un material con una serie de propiedades físicas. Cuando creamos el proyecto si hemos importado el paquete de materiales con propiedades físicas, estaría dentro de la carpeta "standard assets" y "physic materials". Veremos que aparecen cinco ejemplos de material y evidentemente nosotros podemos crear uno.

Vamos a ver cómo actúa esto. Introduciremos el material bouncy de la carpeta standard assets-> physic materials en la casilla `material` de *sphere collider*. Damos play y veremos cómo se le dota a la esfera de una propiedad de rebote. El material bouncy tiene una propiedad `Bounciness` a 1 y esto es lo que dota de rebote a la esfera o pelota. Más opciones que aparecen: la fricción estática que es aquella que impide que un objeto inicie un movimiento y la fricción dinámica que es una fuerza constante que se opone al movimiento una vez que este ya comenzó.

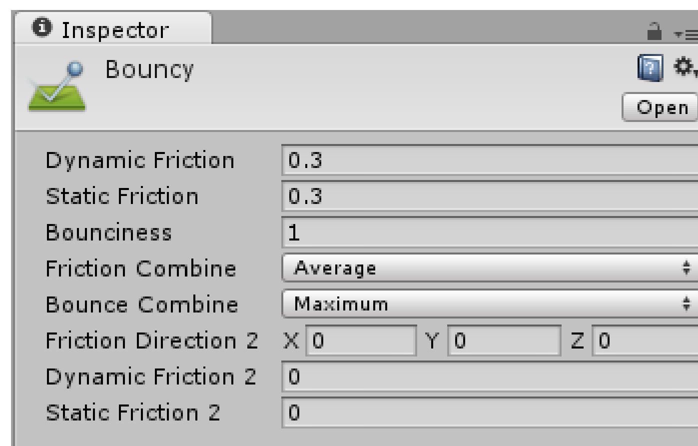


Figura 4.61 Propiedades del material bouncy

Luego aparece una combinación de la fricción del objeto con el *collider* con el que entra en contacto, en este caso dos *colliders* entran en contacto, la pelota y el terreno. Aparece también un coeficiente de combinación de rebote de la pelota también con el terreno.

Si queremos crear un material físico nuevo, simplemente iríamos a la vista de proyecto, `create->physic material` o bien en el menú `Assets->Create->physic material`.

Vamos a crearlo y le llamaremos: "mi material" y veremos cómo aparecen en la vista de inspector las opciones que hemos visto antes.

Para asignarlo a un objeto simplemente seleccionamos el objeto, vamos a su componente *collider* y arrastramos el material al hueco de la propiedad `material`.

4.6.2.-Fixed Joint e Hinge Joint

Para explicar este apartado vamos a realizar un ejemplo, que será un péndulo usando estas dos opciones: *Hinge Joint* y *Fixed Joint*.

Empezaremos creando una esfera, para ello iremos a `Game Object->Create Other->Sphere`. A continuación en la vista del inspector, en Scale introduciremos estos valores: `X`=4, `Y`=4, `Z`=4.

Le añadiremos a la esfera un *Rigidbody*: `Component->Physics->Rigidbody`.

Después crearemos un cilindro: `Game Object->Create Other->Cylinder`. Lo situaremos encima de la esfera y en la vista del inspector, en Scale introduciremos unos valores para que nos quede más o menos con estas dimensiones:

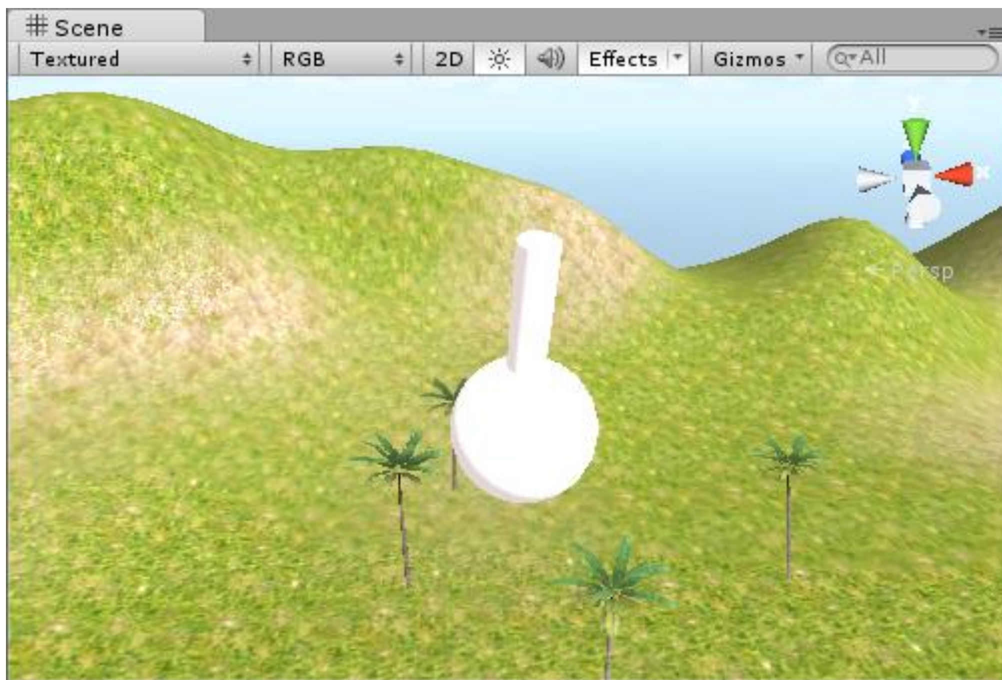


Figura 4.62 Esfera con cilindro

Vamos a introducir a la esfera un *Fixed Joint*. El *Fixed Joint* lo que va a hacer es unir a la esfera con el cilindro.

Seleccionamos la esfera y vamos a `Component->Physics->Fixed Joint`. Nos fijaremos en que el *Fixed Joint* que habrá aparecido en la vista del inspector en el elemento esfera, nos pide un *rigidbody*, por lo tanto necesitaremos que el cilindro sea un *rigidbody* para incluirlo aquí.

Vamos a seleccionar el cilindro y le añadiremos un *rigidbody*: `Component->Physics->Rigidbody`. Ahora sí podemos, con la esfera seleccionada, arrastrar el cilindro a `Connected Body` en el apartado Fixed Joint.

Como curiosidad: Si la esfera no tuviese en ese parámetro ningún *rigidbody*, esta estaría conectada con el resto del mundo del videojuego.

Ya tendríamos una unión entre la esfera y el cilindro. Ahora si diéramos al play, simplemente, la esfera unida al cilindro caería al suelo.

Ahora vamos a hacer la primera cadena: `Game Object->Create Other->Capsule`. A esta cápsula, la pondremos como nombre: “Cadena1”. La posicionamos encima del cilindro y la escalamos de manera que quede más o menos de esta forma:

Figura 4.63 Esfera con cilindro y cápsula

Lo que haremos es que la cadena actúe en relación a la bola y el cilindro. Esta actuaría de otra forma, tipo bisagra que es la traducción de *Hinge*.

La unión de tipo bisagra también nos pedirá un *rigidbody*, por tanto necesitamos que la cápsula sea un *rigidbody*. La seleccionamos y después: `Component->Physics->Rigidbody` y al cilindro le vamos a dar: `Component->Physics->Hinge Joint`.

Una vez insertado el *Hinge Joint* en el cilindro, lo que haremos es arrastrar la Cadena1 donde pone `Connected Body` en la parte de Hinge Joint. Si no introdujéramos ningún cuerpo en este apartado, se interpretaría que el cilindro se une al universo, por así decirlo. Esto lo haremos con el último elemento de la cadena para que el péndulo se mantenga suspendido en el aire. El *Hinge Joint* en este caso no estaría unido a ningún cuerpo.

Vamos a seleccionar Cadena1, con Ctrl+D lo duplicamos y llamaremos al elemento duplicado “Cadena2”. Situaremos Cadena2 encima de Cadena1.

Figura 4.64 Esfera con cilindro y dos cápsulas

Ya que Cadena2 es un duplicado de Cadena1 y Cadena1 ya tenía el componente *Rigidbody*, no hará falta añadírselo a Cadena2.

Seleccionamos Cadena1 y le introduciremos un *Hinge Joint*: `Component->Physics->Hinge Joint`. Este nos pide un *Rigidbody* por lo tanto le introduciremos Cadena2.

Duplicamos Cadena2 también y la llamamos Cadena3, la situamos encima de Cadena2 y como en el caso anterior tenemos que unir Cadena3 con Cadena2, por tanto a la Cadena2 le introducimos una unión tipo bisagra (*Hinge Joint*) de nuevo (ya sabemos cómo se hace) y le introducimos a este componente un *Rigidbody* que en este caso sería Cadena3.

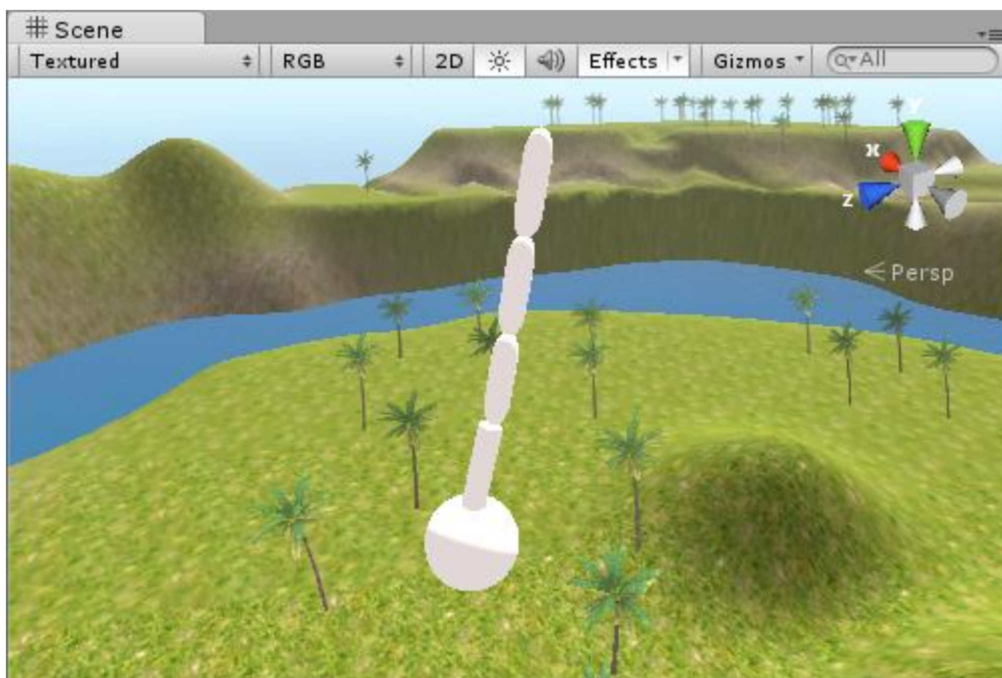


Figura 4.65 Esfera con cilindro y tres cápsulas

Ahora en Cadena3 introduciremos un elemento tipo bisagra y en este caso no le conectaremos con ningún cuerpo, dejaremos ese elemento vacío. De esta manera Cadena3 se quedaría conectada con el mundo y el péndulo se quedará suspendido en el aire.

Para finalizar, seleccionaremos los elementos: esfera, cilindro y las tres cadenas y rotaremos el péndulo de esta manera:

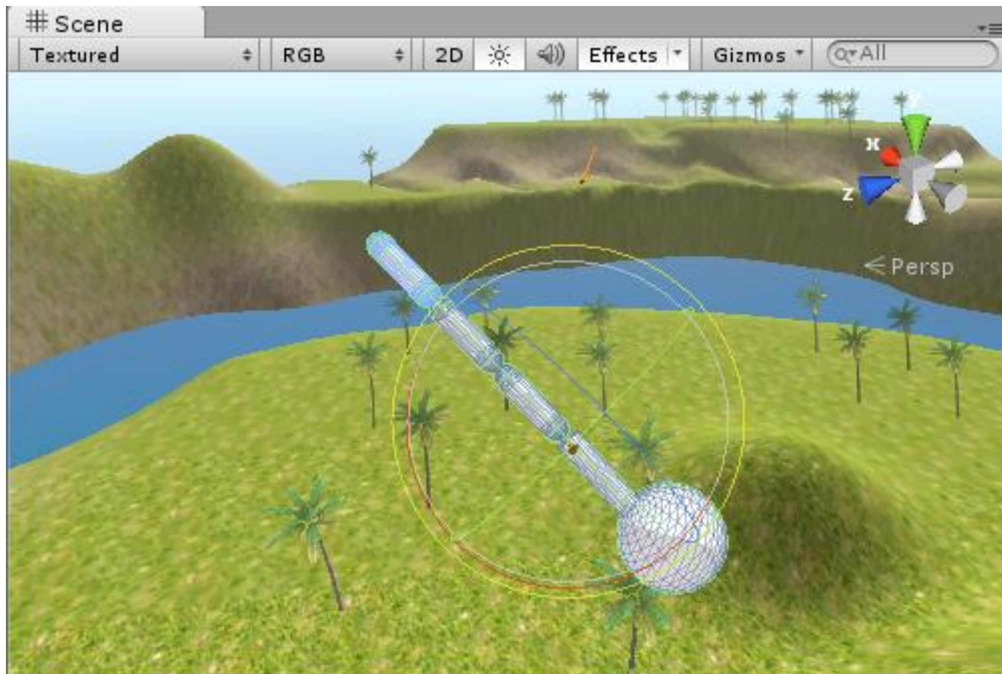


Figura 4.66 Péndulo completo

Daremos al play y veremos que el péndulo oscila debido a la acción de la gravedad.

4.7.- Shaders

4.7.1.-Ejemplo Cuero

Definición de *Shaders*:

Son efectos gráficos asociados a una o varias texturas que ofrecen el resultado visual final de un material renderizado.

Vamos a realizar unos simples ejercicios para aprender a trabajar con los *shaders*.

En primer lugar vamos a añadir texturas como recurso en nuestro proyecto. Para ello vamos a ir a la vista de proyecto, haremos clic derecho en una zona vacía y seleccionaremos `Create->Folder` para crear una nueva carpeta a la que llamaremos “Texturas”.

Abrimos nuestro navegador web y accedemos a Google Imágenes. Podemos acceder simplemente a Google y seleccionar Imágenes en la parte superior de la web.

En el buscador vamos a escribir la palabra “texture”, en inglés para asegurarnos más resultados y a esperar los resultados de búsqueda. Podemos concretar más el tipo de textura que buscamos añadiendo alguna palabra más, como por ejemplo *leather* (cuero), si queremos buscar texturas de cuero.

Haremos clic en el resultado de búsqueda que más nos guste. Se abrirá una previsualización en la que es importante que en la columna lateral derecha hagamos clic en Tamaño completo, ya que lo que se muestra es una miniatura.

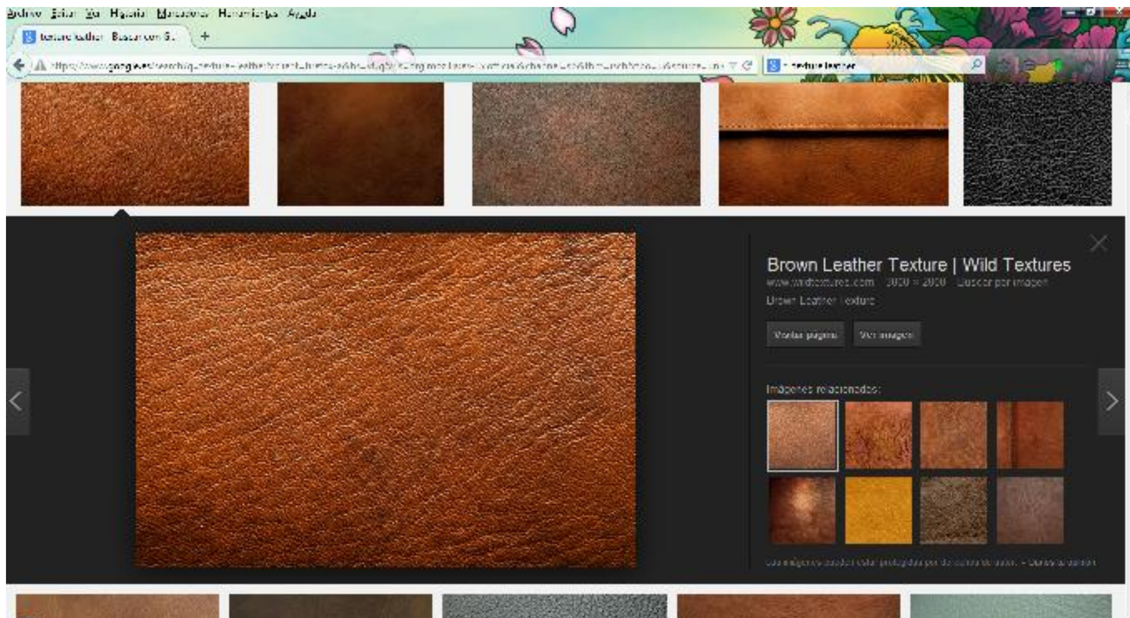


Figura 4.67 Textura cuero

Una vez hecho esto, podremos ver la imagen a tamaño completo en nuestro navegador. Vamos a guardarla haciendo clic derecho sobre ella y seleccionando la opción guardar imagen como…

Vamos a importar la textura como recurso de proyecto antes de utilizarla. En la vista de proyecto, dentro de la carpeta “Texturas” vamos a crear una nueva carpeta con el nombre “Cuero”. Hacemos clic derecho sobre ella y seleccionamos `Import New Asset`, se abrirá una ventana de navegador donde buscaremos la textura que nos hemos descargado, la seleccionaremos y pulsaremos Import para añadirla como recurso. Podemos repetir el mismo proceso cuantas veces queramos para añadir distintos tipos de textura a nuestro proyecto.

Vamos a comenzar a hacer uso de las texturas importadas. Vamos a crear un cubo: `Game Object->Create other->Cube`. Localizamos la textura que acabamos de importar en la vista de proyecto, dentro de la carpeta texturas y la arrastramos directamente sobre el cubo, bien sea en la vista de escena o en la de jerarquía. Observaremos que el cubo aplica automáticamente la textura sobre todas sus caras.

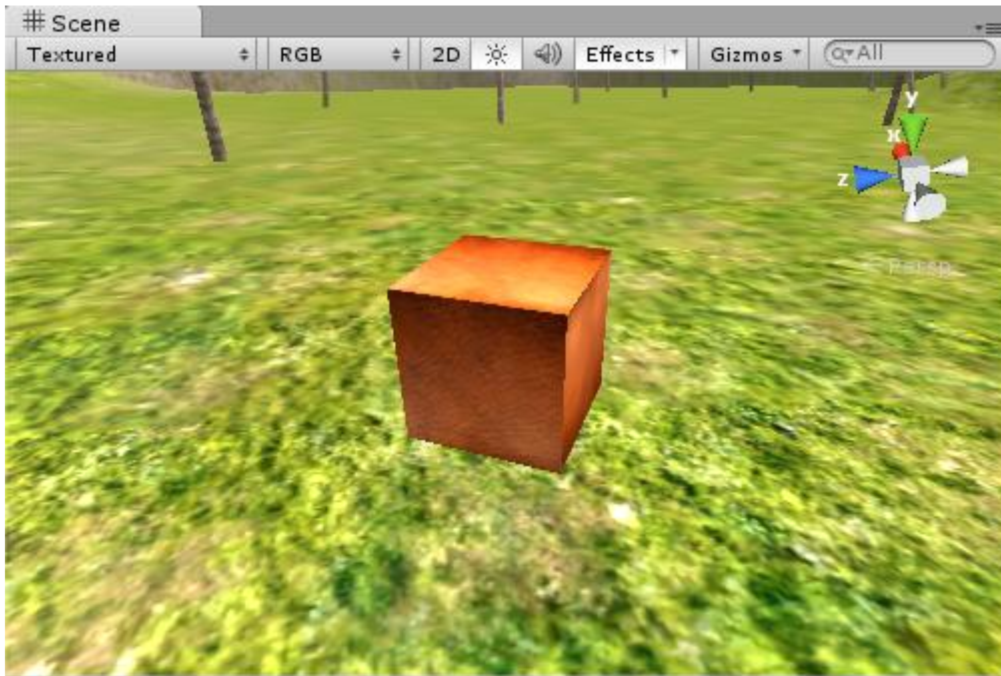


Figura 4.68 Cubo con textura cuero aplicada

Bueno, ahora que ya nos hemos familiarizado con el manejo de texturas y su aplicación sobre objetos vamos a pasar a trabajar con un aspecto visual más avanzado de Unity. Los materiales indican el aspecto visual que nos muestra finalmente el objeto, atendiendo especialmente a su reacción frente a las fuentes de luz y los reflejos del entorno que lo rodea.

En la vista de jerarquía vamos a buscar el cubo texturizado que hemos creado anteriormente y hacemos doble clic sobre su nombre para centrarlo en la vista de escena.

Si observamos ahora en la vista del inspector, abajo del todo, comprobaremos que hay una pequeña previsualización en forma de esfera con el aspecto exterior del cubo, una caja de color y su textura (ver Figura 4.69). Se trata de la ventana del material del objeto y vamos a trabajarla en profundidad.

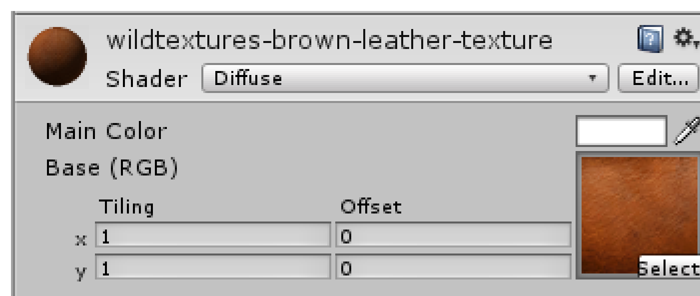


Figura 4.69

La caja de color nos permite aplicar un color de teñido en caso de que queramos ofrecer un aspecto distinto. Podemos probar a modificar el color y observar el resultado.

También son interesantes los valores de repetición (`Tiling`), que indican la cantidad de veces que la textura se proyecta sobre las caras del cubo.

Vamos a modificar los valores de la columna `Tiling` poniendo valores como 2, 5 ó 10 y observaremos el cubo de cerca en la vista de escena. Veremos como la textura ahora se muestra repetidas

veces sobre cada cara del cubo. Esto resulta especialmente útil para por ejemplo componer suelos y paredes con una única textura de baldosa y un plano sobre el que se repite varias veces.

Una vez entendido esto es hora de configurar la reacción que nuestro material tiene con la luz. Los materiales pueden reflejarla o absorberla en distinta medida, los materiales que absorben la luz se denominan difusos (*Diffuse*), mientras que los que la reflejan se denominan especulares (*Specular*).

Observemos que en el apartado `shader` hay una lista desplegable. Vamos a hacer clic en la lista desplegable del `shader` y seleccionamos `Specular` para hacer que el material del cubo pase a reflejar la luz que le llega. Es normal que el menú tarde un poco en responder.

Aunque Unity incluye por defecto aproximadamente unos 50 *shaders* distintos para materiales, la gran mayoría de ellos parten de ser o bien difusos (Absorben la luz) o bien especulares (reflejan la luz).

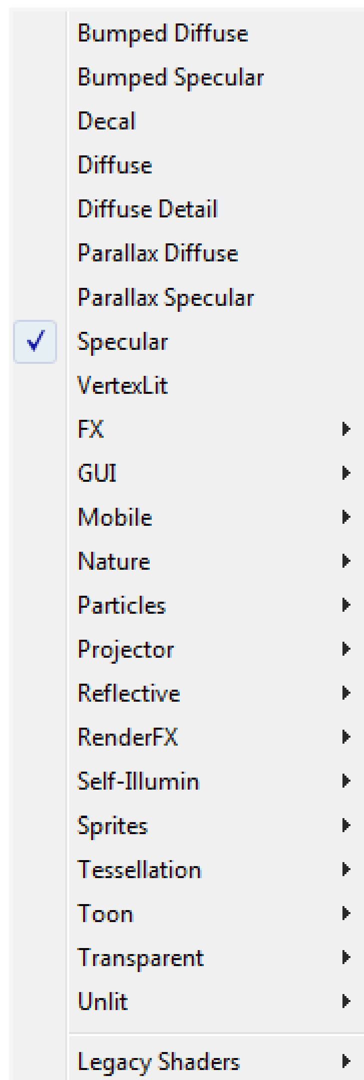


Figura 4.70 Lista desplegable de *shaders*

En la vista del inspector, veremos abajo del todo que la ventana del material ha cambiado ligeramente después de haber seleccionado el *shader* especular. Ahora tenemos una caja de color adicional en la que podremos dar un color de teñido especular para los reflejos de luz (`specular color`) y una barra que podemos arrastrar de izquierda a derecha para indicar el brillo (`Shininess`), que indicará en qué medida el material refleja la luz que recibe.

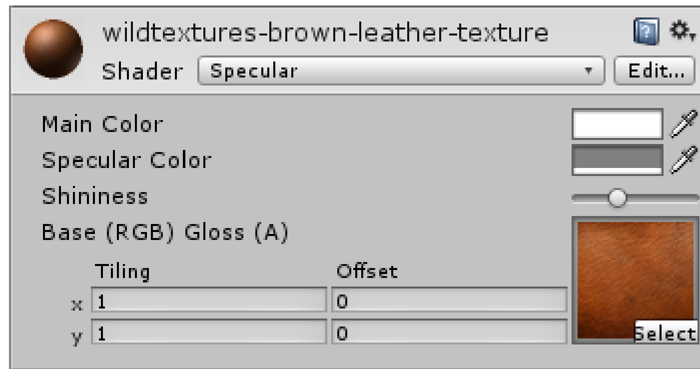


Figura 4.71 Propiedades del *Shader* “Specular”

Normalmente, en el color especular deberíamos asignar algún tipo de gris, consiguiendo así que la luz reflejada pierda un poco de intensidad sin modificar su color original. En cuanto a la barra de brillo, cuanto más a la izquierda la pongamos conseguimos más brillo y cuanto más a la derecha más conseguimos el efecto difuso. A modo de curiosidad, veremos que el *shader* especular resulta más apropiado para texturas metalizadas.

El *shader* especular es el más básico de los que trae Unity y no resulta para nada realista, pero es importante comprender su funcionamiento antes de pasar a utilizar otros tipos de *shader* más avanzados.

En esta captura (ver Figura 4.72) podemos ver el resultado visual de un objeto a medida que modificamos la cantidad de luz (de derecha a izquierda) y la cantidad de brillo (de arriba abajo) de su material.

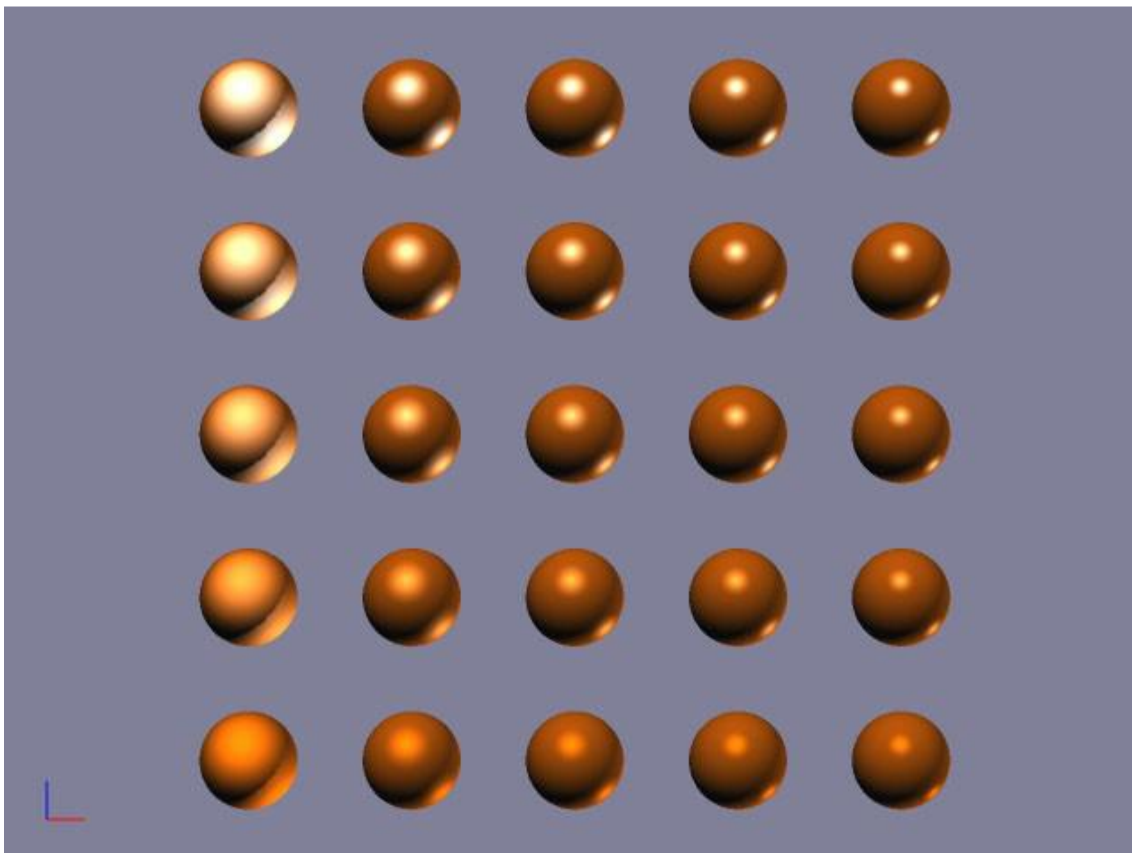


Figura 4.72

Otra conclusión importante que podemos sacar de esto es que con una única textura podemos obtener infinidad de materiales distintos.

Vamos a dar paso a los materiales más avanzados de Unity. Si volvemos a echar un vistazo a la lista completa de *shaders* disponibles en la vista del inspector, observaremos también que gran parte de ellos vienen prefijados con la palabra "Bumped". Este término hace referencia a un efecto de profundidad que se aplicará sobre la superficie del material dando la sensación de que no es totalmente plana, si no que realmente las irregularidades dibujadas en la textura tienen una profundidad real.

La mejor manera de entender esto último es configurando un *shader* de este tipo. Vamos a desplegar la lista de *shaders* y a seleccionar `Bumped Specular`. Podríamos utilizar también `Bumped Diffuse`, pero la única diferencia será si reflejará la luz o no y en este caso queremos ver en todo su esplendor el efecto de profundidad en la superficie.

De nuevo, en la vista del inspector podemos ver que la ventana del material ha vuelto a cambiar.

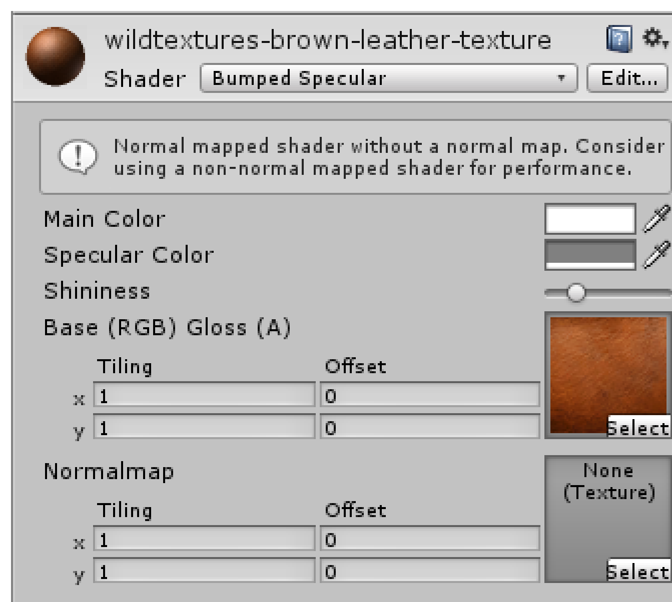


Figura 4.73 Propiedades del *shader* "Bumped Specular"

La única diferencia respecto del *shader* especular anterior es que ahora hay un segundo campo llamado "Normalmap" (Mapa de normales) para una textura.

La textura de mapa de normales es la que generará las irregularidades en la superficie del material. Existen muchas formas de obtener mapas de normales, pero vamos a utilizar una muy sencilla.

En la vista de inspector, abajo del todo, hacemos clic en la textura de base que tiene asignada nuestro objeto.

Veremos que en la vista de proyecto aparece marcada en amarillo. Esta es una forma rápida de encontrar el recurso de proyecto que estamos usando, muy útil cuando comenzamos a tener gran cantidad de texturas.

Vamos a ir a la vista de proyecto, seleccionamos la textura y pulsamos el atajo de teclado Ctrl+D para duplicarla. También podemos ir a la pestaña superior `Edit->Duplicate`.

Hacemos clic sobre la nueva textura duplicada para poder nombrarla. En nuestro caso, si la textura original se llama "Textura cuero" vamos a renombrar la nueva como "Textura Cuero Bump".

Con la nueva textura seleccionada en la vista de proyecto, observamos que en la vista de inspector se mostrarán diversos parámetros con la configuración actual de la textura y abajo del todo una previsualización. La ventana que estamos viendo se denomina Importador de Texturas (ver Figura 4.74).

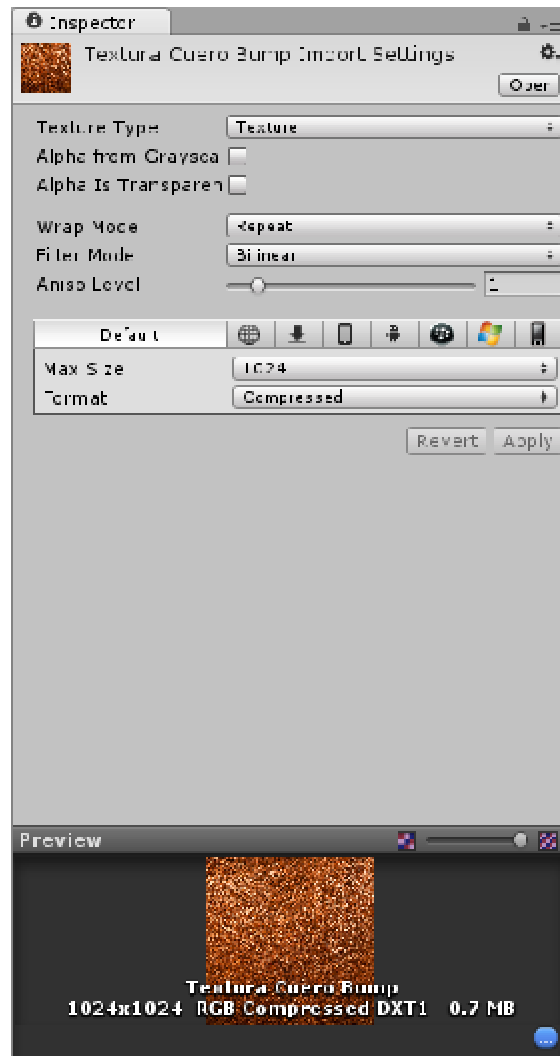


Figura 4.74

Vamos a configurar nuestra nueva textura para que pase a ser un mapa de normales. En el apartado `Texture Type` (Tipo de textura), hacemos clic en la pestaña desplegable y seleccionamos la opción `Normal Map`.

Observaremos que la celda `Alpha from Grayscale` se activará automáticamente, esto servirá para que las irregularidades se generen a partir de la textura en escala de grises, siendo las zonas más oscuras las más profundas.

También aparecen dos nuevos parámetros configurables: La barra desplazable de profundidad (`Bumpiness`) indica cómo de pronunciadas serán las irregularidades en la superficie, mientras que el filtro (`Filtering`) nos permite seleccionar 2 modos distintos para calcularlas.

Vamos a configurar un factor de `Bumpiness` no muy alto, por ejemplo 0,1 será suficiente y dejamos el filtro por defecto ya que generalmente es cuestión de probar el filtro que mejor resultado nos ofrece.

Abajo del todo en la Vista de Inspector también, hacemos clic en Apply para aplicar los cambios y observaremos abajo del todo la nueva textura que hemos generado.

La nueva textura está compuesta por tonos azulados en su mayoría. Todas las desviaciones de azul puro serán las irregularidades de las que hablábamos. Vamos a hacer uso de este mapa de normales en nuestro material. En la vista de jerarquía, volvemos a seleccionar el cubo.

En la vista del inspector, abajo del todo, hacemos clic en el botón select, en la celda correspondiente a la textura de mapa de normales. Esto abrirá una ventana con todas las texturas del proyecto.

Buscamos en ella la textura de mapa de normales que acabamos de crear, la seleccionamos y pulsamos intro para finalizar.

Otra opción para este último paso hubiese sido arrastrar directamente la textura de mapa de normales desde la vista de proyecto hasta la celda en la vista de inspector.

En la vista de jerarquía, hacemos doble clic en el cubo para centrarlo en la vista de escena y observemos el resultado.

En la vista de jerarquía, hacemos doble clic en la luz direccional para centrarla en la vista de escena. Utilizamos el botón de rotar en la parte superior izquierda (Atajo de teclado E) y vamos a probar diferentes orientaciones de la luz para comprobar su efecto sobre el nuevo material (ver Figura 4.75).

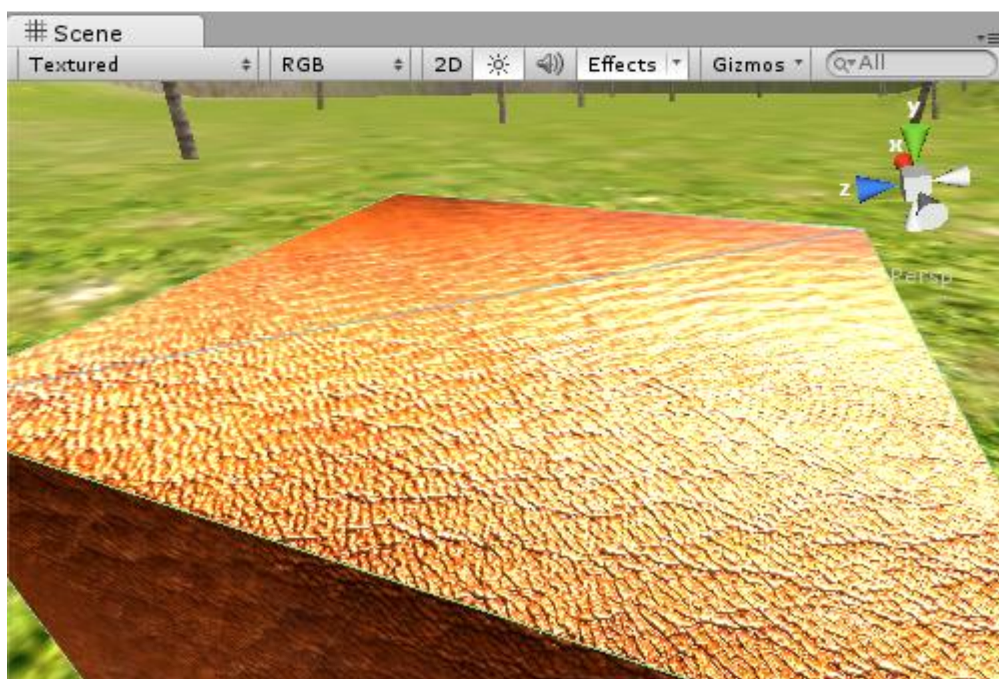


Figura 4.75

Iluminación

Un material con mapa de normales como el que hemos creado anteriormente simula la presencia de irregularidades en una superficie que es realmente plana. Generalmente no apreciaremos con gran detalle el resultado si permanecemos quietos o si no tenemos luces en movimiento.

Vamos a añadir una luz sobre nuestro cubo y moverla sobre su superficie para ver el resultado.

En la vista de jerarquía, hacemos doble clic en el cubo para centrarlo en la vista de escena.

En las pestañas superiores, hacemos clic en `Game Object->Create Other->Point Light` para crear una luz puntual en la misma posición que el cubo.

Utilizamos el botón Mover que se encuentra arriba a la izquierda del interfaz (Atajo de teclado W) y colocamos la luz puntual justo sobre la superficie del cubo (ver Figura 4.76).

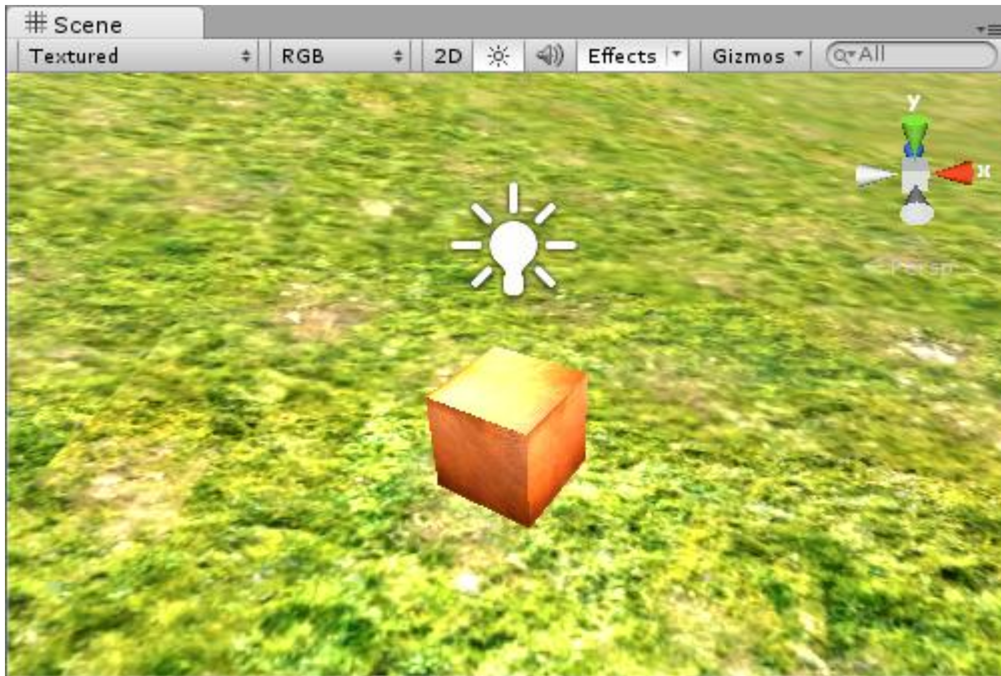


Figura 4.76

Desplazamos la vista de escena hasta la superficie del cubo para ver el efecto (ver Figura 4.77), movemos de nuevo la luz puntual para comprobar cómo las irregularidades ganan realismo con el movimiento de la cámara o de las luces en la escena.

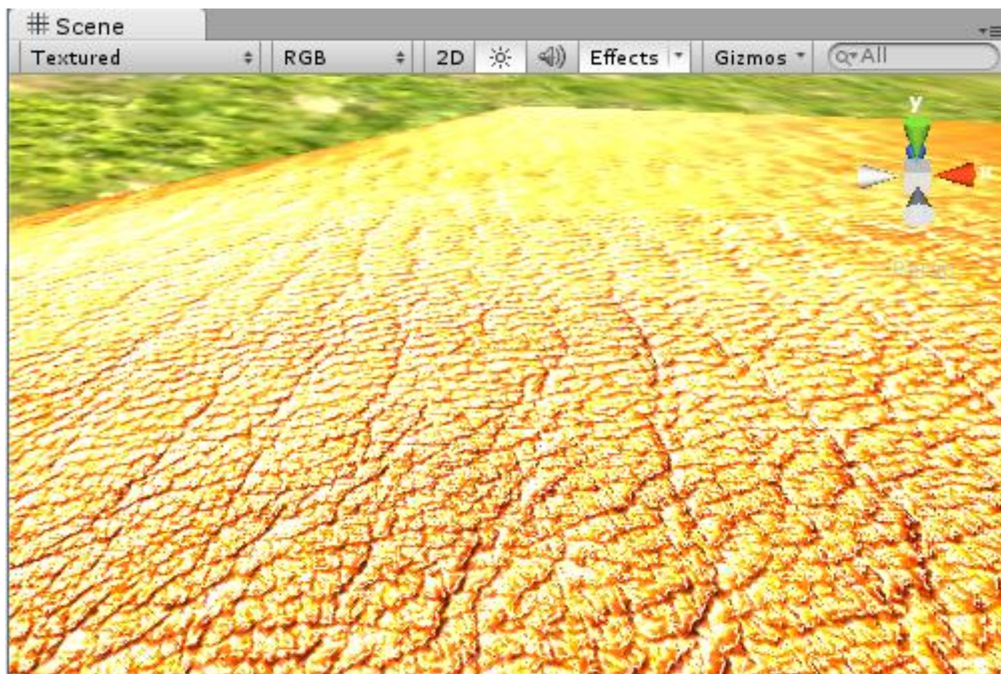


Figura 4.77

4.8.-Animación

4.8.1.-La vista de animación

El proceso para comenzar a animar un objeto es siempre igual. En primer lugar debemos seleccionar el objeto que queremos animar. En nuestro caso vamos a trabajar con un simple cubo en una escena nueva, de forma que no tengamos ningún otro elemento que pueda molestarnos.

Vamos a ir a las pestañas superiores y a seleccionar `File->New Scene` para crear una nueva escena. Podemos aprovechar directamente para guardarla seleccionando `File->Save Scene` y guardándola en la carpeta "Escenas" con el nombre "Prueba Animación".

Crearemos un cubo: `Game Object->Create Other->Cube`. En la vista de jerarquía, hacemos doble clic en su nombre para centrar la vista de escena en él (ver Figura 4.78).

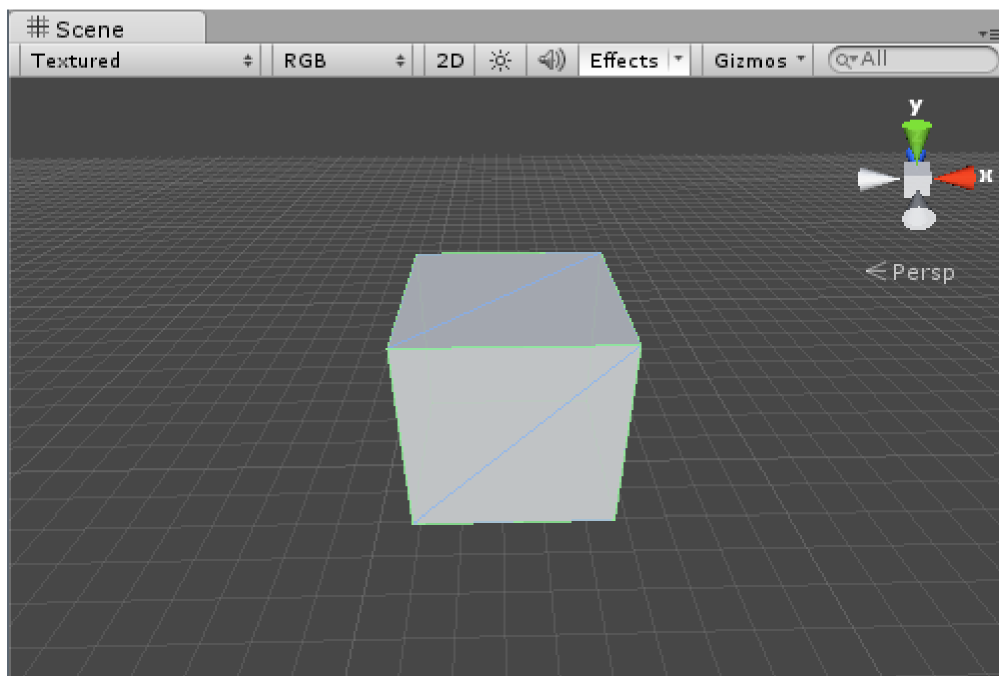


Figura 4.78

En la vista del inspector, en Position, vamos a introducir estos valores: `X`=0, `Y`=5, `Z`=10.

En la vista de jerarquía, seleccionamos la cámara principal y hacemos que mire directamente al cubo. Podemos hacerlo fácilmente seleccionando `Game Object->Align with view`.

Podremos comprobar que tanto la vista de escena como la vista de juego muestran el cubo desde el mismo ángulo.

Vamos a añadir una luz direccional a la escena: `Game Object->Create other->Directional light`.

Nos aseguramos de tener seleccionado el cubo. A continuación vamos a: `Window->Animation` para abrir la vista de animación.

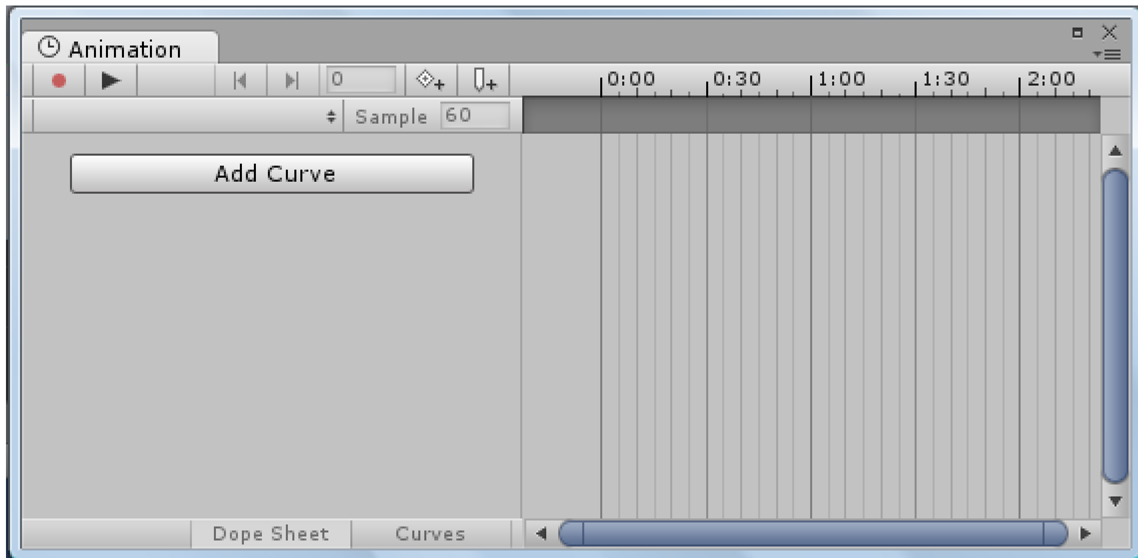


Figura 4.79 Vista de animación

Vamos a evitar hacer clic fuera de la vista de animación y seleccionar otro objeto durante todo el proceso, ya que en este caso el contenido de la ventana cambiará y es bastante probable que perdamos parte del trabajo realizado.

Vamos a crear un nuevo clip de animación para nuestro cubo.

En primer lugar, en la vista de proyecto, crearemos una nueva carpeta llamada “Animaciones” haciendo clic derecho en una zona vacía y seleccionando `Create->Folder`.

Hacemos clic derecho en la carpeta “animaciones” de la vista de proyecto y creamos una carpeta más llamada “pruebas”.

Volvemos a la vista de animación (Podemos volver a sacarla haciendo clic en `window->Animation` si la habíamos cerrado) y hacemos clic en el botón que aparece en la siguiente imagen, seleccionamos la opción `Create New clip` para crear un nuevo clip de animación.

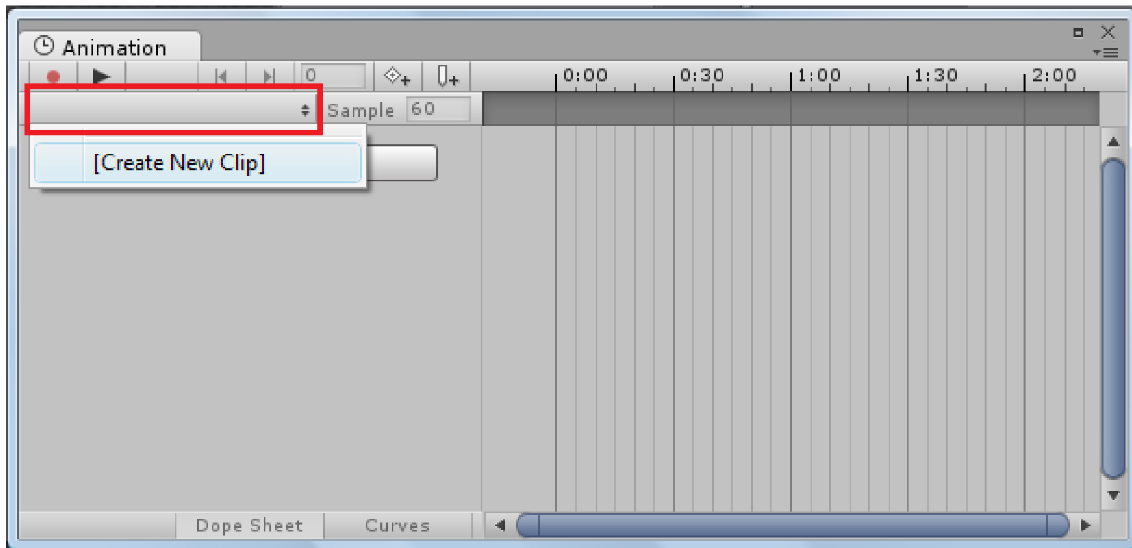


Figura 4.80

En la ventana del navegador que se abrirá, accederemos a la carpeta de animaciones que acabamos de crear, dentro de ella vamos a “Pruebas” y guardaremos ahí el clip de animación con nombre “Prueba Mover”.

Tras haber creado el nuevo clip, la vista de animación debería verse así:

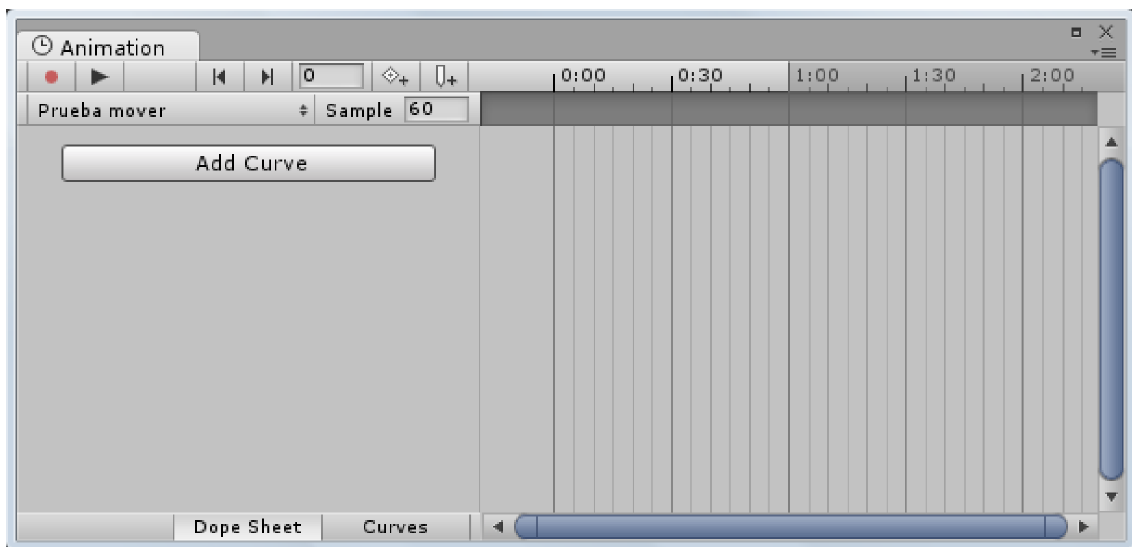


Figura 4.81

El siguiente paso es determinar qué valores del cubo son los que queremos animar. Si observamos la columna de la izquierda de la vista de animación veremos un botón Add Curve (añadir curva), si le pulsamos aparecerá un menú desplegable:

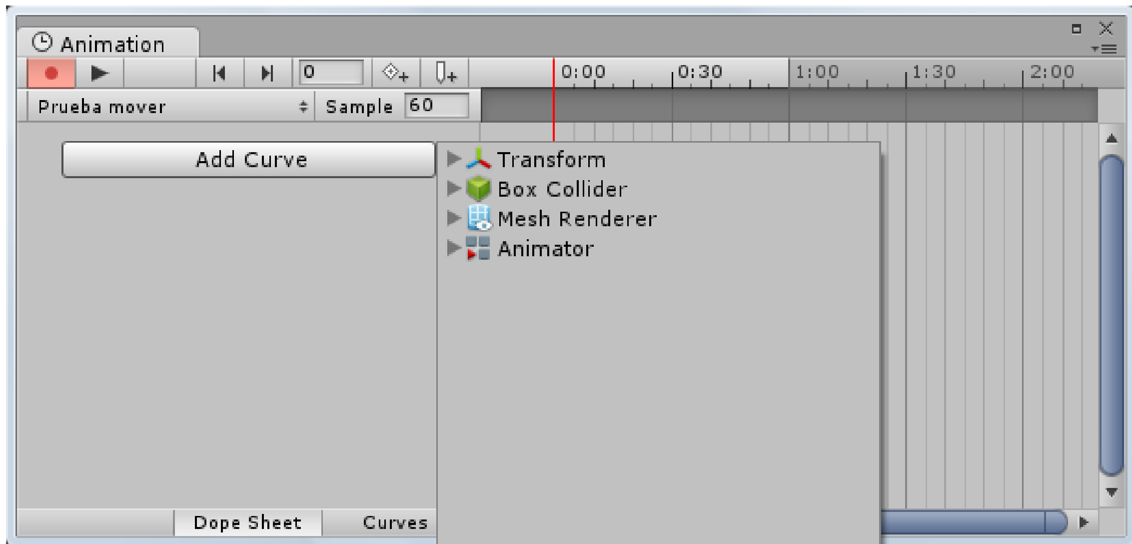
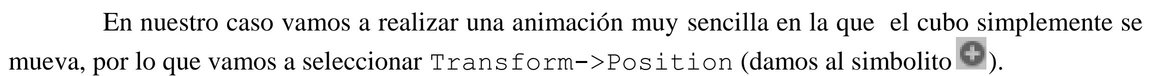


Figura 4.82

En nuestro caso vamos a realizar una animación muy sencilla en la que el cubo simplemente se mueva, por lo que vamos a seleccionar `Transform->Position` (damos al simbolito).



A continuación veremos cómo funciona el gráfico de la parte derecha de la vista de animación. Básicamente se trata de una gráfica en la que podemos controlar la posición de nuestro objeto a lo largo del tiempo.

Con el ratón dentro de la gráfica de la vista de animación, podemos utilizar la rueda del ratón para hacer zoom hacia dentro o hacia afuera.

La Unidad de medida mínima del tiempo es un *frame* que es el tiempo que tarda el videojuego en redibujarse, que por defecto es 60 veces por segundo. De ahí observaremos que el tiempo pasa de 0:30 a 1:00, el 0:30 marca 30 *frames* (Medio segundo) mientras que el 1:00 indica un segundo.

Vamos a usar la rueda del ratón dentro de la gráfica para tener visible unos 5 segundos (5:00) de tiempo. Es importante no confundir los *frames* con los segundos, en la regla horizontal superior no deberíamos ver 0:05 sino 5:00 (ver Figura 4.83).

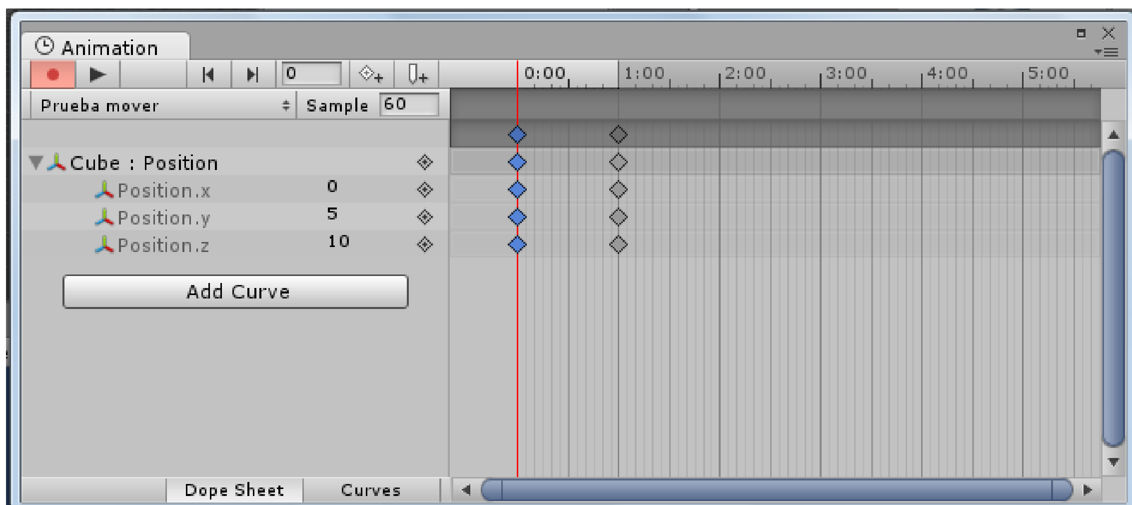


Figura 4.83

A continuación crearemos los puntos clave (*keyframe*) de la animación. Los puntos clave se sitúan a lo largo del tiempo y en cada uno de ellos se determina un estado en el objeto a animar. El clip de animación se encargará de calcular automáticamente el camino necesario para que el objeto pase por todos los puntos clave suavemente.

La regla de puntos clave está debajo de de la regla de tiempo, justo encima de la gráfica. Hay que tener cuidado porque tiene otra regla superior que se ocupa de los eventos de animación. La reconoceremos porque al hacer clic derecho sobre ella nos permite añadir *keyframes*.

En la regla de puntos clave, en el segundo 5:00 aproximadamente, vamos a hacer clic derecho y seleccionar la opción `Add keyframe` para añadir un nuevo punto clave al clip de animación (ver Figura 4.84).

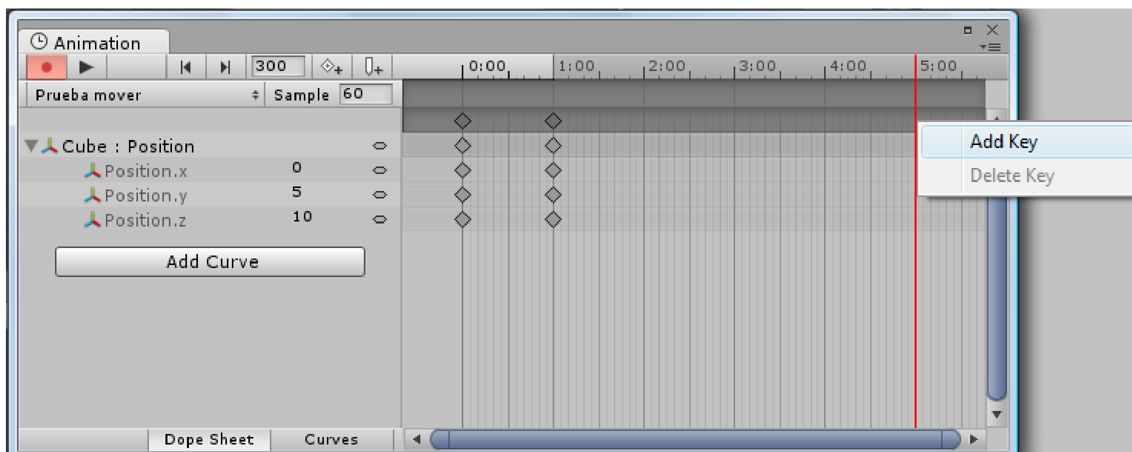


Figura 4.84

Los rombos que representan las coordenadas de posición ahora también aparecerán bajo el nuevo punto clave.

Llegados a este punto sólo necesitaríamos indicar, para el nuevo punto clave, los nuevos valores de posición que queremos que tome nuestro cubo.

Esto puede hacerse de varias formas distintas, como por ejemplo introduciendo los valores numéricamente para cada *keyframe* en la vista de animación. Nosotros lo vamos a hacer de forma visual desde la vista de escena, ya que resulta mucho más sencillo para principiantes.

4.8.2.-Animar de forma gráfica

Vamos a tener el `layout` en modo `2 by 3` y la vista de animación visible.

El *keyframe* generado automáticamente en el instante 0:00 generalmente no requerirá ningún cambio, ya que por defecto guarda los valores iniciales del objeto. Vamos a trabajar directamente con el segundo *keyframe*.

En la regla de *keyframes* de la ventana de animación, vamos a seleccionar el segundo *keyframe* haciendo clic sobre él.

En la vista de escena y sin cerrar la vista de animación, vamos a utilizar la herramienta de posicionar (Botones arriba a la izquierda o atajo del teclado W) para mover ligeramente el cubo hacia arriba (Eje Y). Podemos moverlo metro por metro si mantenemos pulsado Ctrl, con un par de metros sería suficiente.

Vamos a previsualizar el resultado de nuestra animación. Observemos que la vista de animación tiene su propio botón Play en su parte superior izquierda.

Hacemos clic en el botón play de la vista de animación y observamos el cubo en la vista de escena.

Si la vista de escena muestra correctamente el movimiento del cubo, podemos cerrar la vista de animación y pulsar Play en Unity. La animación se reproducirá también en la vista de juego.

Observaremos que la animación se realiza una y otra vez, infinitas veces. Esto sucede porque el modo de animación por defecto reproduce el clip de manera infinita pero también podemos hacer que el clip se reproduzca sólo una vez, seleccionando en la vista de proyecto Assets->Animaciones->Pruebas->Prueba Mover y en el inspector desmarcaríamos la casilla `Loop Time`. Vamos a marcarla de nuevo para realizar lo siguiente.

4.8.3.-Curvas de animación

Una animación de videojuego generalmente se compone de varios puntos clave. Vamos a seguir mejorando nuestra animación hasta darle un aspecto profesional.

Vamos a volver a poner el `layout` en modo `2 by 3` y abriremos la vista de animación haciendo clic en la pestaña `Window-> Animation`.

En la vista de jerarquía, seleccionemos de nuevo el cubo y comprobaremos que en la vista de animación el clip de animación sigue intacto.

En la vista de animación, volvamos a pulsar el botón de grabación (icono redondo y rojo arriba a la izquierda). Colocamos el cursor en la gráfica y utilizando la rueda del ratón dejamos visible más allá del segundo 10:00.

En la barra de puntos clave de animación (*keyframes*), en el segundo 10:00, hacemos clic derecho y seleccionamos la opción `Add keyframe` para añadir un nuevo punto clave.

Llegado este punto observaremos que el nuevo punto clave añadido tiene los mismos valores que el primero, esto sucederá siempre que añadamos un nuevo punto clave.

Si pulsamos en la vista de animación el botón curves y seleccionamos Cube:Position, veremos que además la línea de color verde que nos muestra la posición en el eje Y (Arriba/Abajo) ha tomado ahora la forma de una parábola para unir los tres puntos clave de la manera más suave posible.

Al añadir un nuevo punto clave, este queda por defecto seleccionado, podemos estar seguros de ello debido a la línea vertical roja que lo atraviesa.

Vamos a hacer clic en el botón play de la vista de animación y comprobamos como el cubo sube y baja y en la parte superior realiza un movimiento muy suave.

Sólo deberíamos notar un problema y es que, al llegar abajo, el cubo hace un cambio brusco de dirección. Si observamos la vista de animación, el punto en el que la gráfica termina y vuelve a enlazar por el otro extremo sigue formando un cambio brusco de dirección. Es lo que vamos a solucionar ahora mismo.

En la vista de animación, dentro de la gráfica, cada uno de los rombos señala el valor de la magnitud que representan y pueden ser arrastrados a mano alzada para modificar sus valores. Si lo comprobamos, utilicemos el atajo de teclado Ctrl+Z para como siempre deshacer la acción.

Además de poder modificar su valor, también es posible hacer clic derecho sobre ellos para acceder a un menú (ver Figura 4.85) que entre otras cosas nos permite configurar como es la curva de animación al llegar a ellos.

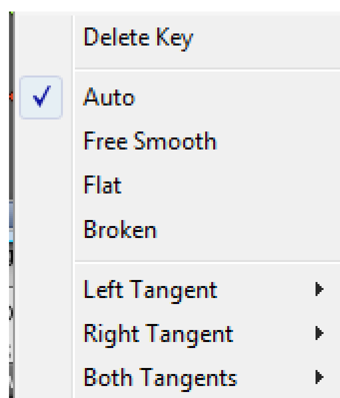


Figura 4.85

En la curva de animación de la posición Y (Curva verde) vamos a hacer clic derecho en el rombo correspondiente al *keyframe* del momento 0:00, debería desplegarse un menú.

Este menú permite realizar diversas modificaciones en la gráfica como generar escalones, realizar rupturas bruscas en las curvas, aplanarlas, etc. En nuestro caso simplemente vamos a utilizar la opción de aplanado para evitar el cambio brusco que se produce al concatenarse la animación.

En este menú, vamos a seleccionar la opción de aplanado (`Flat`) y observaremos como el lateral derecho de la curva de animación se aplana suavizando en gran manera el cambio brusco.

Vamos a repetir la misma operación en el rombo de la curva de posición Y (La verde también), pero esta vez en el último *keyframe*. La curva ahora mismo debería quedar completamente suavizada.

Pulsemos ahora el botón de Play en la vista de animación y observemos como el movimiento de subida y bajada del cubo resulta totalmente suave y sin cambios bruscos.

4.8.4.-Animaciones de cámara

Una utilidad muy interesante de la vista de animación es crear animaciones de cámara que recojan toda nuestra escena a modo de cinemática. Para ello necesitaremos trabajar sobre una escena en la que hayamos añadido elementos gráficos como por ejemplo terreno, objetos 3D, luces, etc.

Para realizar la siguiente tarea, asegurémonos de que hay una cámara principal en la vista de jerarquía y que en la misma vista no hay ningún controlador de personaje ya que este tomaría el control de la cámara y generalmente tendría prioridad sobre la animación que vamos a crear.

Vamos a seguir los mismos pasos del apartado 2 de este mismo capítulo, pero esta vez sobre la cámara principal (*Main camera*) para crear un clip de animación al que podemos poner como nombre "Prueba cámara".

En la vista de animación, en curves, vamos a añadir también las curvas de rotación, ya que en principio la cámara además de moverse por la escena también rotará para seguir su camino.

Los 2 puntos clave de animación que tenemos por ahora conservan los mismos valores. Los dejaremos así para asegurar que la cámara no empieza moviéndose directamente, ya que sería una sensación demasiado brusca.

A partir de este punto, añadiremos nuevos puntos clave a lo largo del tiempo, posicionaremos la cámara como deseemos y repetiremos la operación hasta conseguir una cinemática que recorra automáticamente todos los puntos de interés de nuestra escena. Vamos a seguir los pasos siguientes ya que tendremos que repetirlos de la misma manera para completar la cinemática de la cámara.

En la vista de animación, en la regla de puntos clave, vamos a añadir un nuevo punto clave unos segundos después del último haciendo clic derecho y seleccionando la opción `add keyframe`.

En la vista de escena, movámonos libremente hasta visualizar un punto interesante de la escena por el que queramos que pase nuestra cámara. Vamos a evitar recorridos muy largos ya que el desplazamiento entre puntos será calculado automáticamente y no será capaz de evitar atravesar obstáculos si los hay.

Vamos a hacer clic en la pestaña superior `Game Object->Align with view` para que la cámara pase a enfocar lo que se está mostrando en la vista de escena. Observemos que el punto clave cambia sus valores, tanto de posición como de rotación, para tomar los que tiene ahora mismo la cámara.

Vamos a repetir los pasos anteriores las veces necesarias hasta conseguir que la cámara recorra los puntos de interés de la escena. Las curvas en la vista de animación terminarán teniendo un aspecto similar al siguiente:

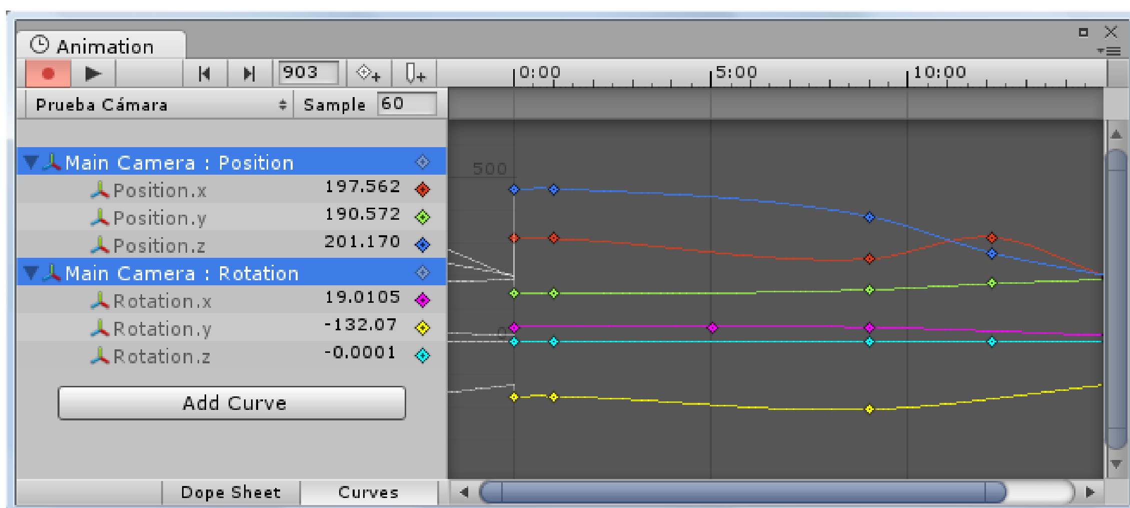


Figura 4.86

Podemos realizar animaciones de cámara de varios minutos. No obstante, en cinemáticas de larga duración comprobaremos que cada nuevo punto clave de animación tarda más tiempo en procesar, esto se debe a que las curvas deben recalcularse cada vez y requerirán más memoria cuantos más puntos clave intervengan.

4.9.-Programación en Unity

4.9.1.-Introducción a la programación orientada a objetos.

Los objetos son entidades de una clase determinada, que define para ellos una serie de variables y métodos asociados.

A continuación definiremos en profundidad todos esos términos y los adaptaremos al campo en el que estamos trabajando: Los videojuegos.

- Clase: Concepto abstracto que tenemos del conjunto de variables y métodos que definen una entidad, es único. Por ejemplo: El concepto de especie animal, aunque no haya animales de esa especie.
- Objeto: Instancia que existe en la realidad de una clase determinada. Cada instancia tendrá sus propios valores para sus variables de clase y contará con los métodos de su clase para modificarlos. Por ejemplo: el perro de Alejandro, que es de la clase `perros`.
- Herencia entre clases: Al heredar de otra, una clase adquiere todas las variables y métodos de la clase padre y permite añadir más. Por ejemplo: la clase `magos` podría heredar de la clase `seres humanos` y añadirles el maná y los hechizos.
- Variable: Celda de memoria que almacena un valor de un determinado tipo o también un objeto de una determinada clase, siempre dentro de un ámbito. Por ejemplo: `posición` almacena un valor de la clase `Vector3` en un ámbito público.
- Tipo: Podemos ver un tipo como una clase muy sencilla, tanto que almacena un único valor y no tiene métodos asociados. Por ejemplo: `int` es el tipo que representa los números enteros, `float` es el tipo que hace referencia a los números reales, etc.
- Ámbito: Hace referencia a la accesibilidad de una variable de una clase, aunque también es aplicable a métodos. El ámbito puede restringir que esa variable sea accesible sólo por los métodos de su propia clase, o bien que también sea accesible por otras clases. En clases creadas por nosotros, el ámbito es público por defecto a no ser que lo definamos nosotros como privado. Por ejemplo: La `transform` es pública (`public`) y por tanto accesible por otras clases.

  Especifica el alcance de la variable, que puede ser accesible tan sólo desde un método, desde una misma clase, o desde todas.

  También especifica si es de carácter único para la clase a la que pertenece o si cada instancia de la clase posee una distinta.

  Finalmente también puede especificar si su valor es constante y no modificable mediante código.

  Tipos:

  <u>Variables de ámbito estático</u>: Únicas en la clase, independientemente de que esta haya sido instanciada. Pueden además ser de los ámbitos siguientes. Generalmente

requeriremos pocas variables estáticas. Se prefija a nivel de código con `static`. Ejemplo: El daño total causado por los enemigos, cada enemigo lo acumularía en una única variable de este tipo.

Variables de ámbito final: Toman un valor inicial que no podrá modificarse durante la ejecución. Al tener el mismo valor para todas las instancias de la clase también son estáticas, aunque conviene especificarlo a nivel de código. Pueden además ser de los ámbitos que vienen a continuación. Se prefija a nivel de código con: `final`. Ejemplo: El valor de la gravedad.

Variables de ámbito privado: Única para cada instancia de la clase y no accesible desde ninguna otra clase. Se prefija a nivel de código con: `private`. Ejemplo: El tiempo que queda hasta que la granada explota, que es distinto para cada instancia de la granada, y sólo lo utiliza la granada.

Variables de ámbito público: Única para cada instancia de la clase y accesible desde las clases que queramos. Se prefija a nivel de código con: `public`. Ejemplo: La vida actual de cada enemigo, que se accederá desde el interfaz para mostrarse en pantalla.

Variables locales: Fuera de todos los ámbitos anteriores, pueden declararse dentro de un método para realizar cálculos intermedios. Estas variables no conservan el valor en diferentes llamadas del método. No requieren prefijo de ámbito. Ejemplo: El contador numérico (1, 2,3,…) que permite lanzar un disparo triple.

- Declaración de variables: Genera una variable nueva. Para ello debemos especificar su ámbito, su tipo, su nombre y (opcionalmente) su valor inicial.

4.9.2.-Introducción a los *scripts* de Unity 3D

Antes que nada, lo primero es conocer con qué clase queremos trabajar. Por ejemplo, si queremos hacer que un planeta orbite, buscaremos en la clase `transform` si hay alguna operación que lo resuelva. En cambio, si queremos aplicar una fuerza de explosión, buscaremos en la clase `Rigidbody`.

Clase `Transform`

Figura 4.87 Diagrama de clases de la clase `Transform`

Almacena todos los valores de posición, rotación y escala de un objeto de juego. Además tiene los métodos necesarios para aplicar los cambios más habituales que queramos hacer sobre esos valores en un videojuego.

Método `Translate()`:

El método `Translate` aplicado sobre la *transform* de un objeto de juego permite trasladar el objeto fácilmente en su espacio de coordenadas o en el espacio de coordenadas del mundo.

Clase `Rigidbody`

Figura 4.88 Diagrama de clases de la clase `Rigidbody`

Almacena todos los valores de masa, resistencias al movimiento y parámetros necesarios para ser procesado por el motor físico. También incluye todos los métodos necesarios para la aplicación de todo tipo de fuerzas.

Método `AddExplosionForce()`:

El método `AddExplosionForce` aplicado sobre el *rigidbody* de un objeto de juego aplica una fuerza según una posición central de la explosión, un radio y una fuerza determinada.

Variables de clase

Aunque casi siempre trabajaremos con métodos, en algunas ocasiones querremos acceder a las variables de clase para conseguir lo que queremos. Por ejemplo, si tenemos un *Game Object* algunas de las variables que podemos obtener a partir de él son su `transform` o su `rigidbody`, que son objetos de las clases `Transform` y `Rigidbody`, respectivamente.

Métodos de clase

Son funciones que procesan cálculos complejos sobre un objeto de la clase o a partir de la clase, según sea o no instanciable. El 99% de las veces trabajaremos con los métodos en lugar de con las variables.

Por ejemplo: `Translate()` es un método de la clase `Transform` que aplicado a la *transform* de un objeto lo desplaza.

Otro ejemplo: `GetKey()` es un método de la clase `Input` (No instanciable) que determina si se ha pulsado alguna tecla.

Valores de retorno

Algunos métodos generan valores al ser procesados. Esos valores pueden ser de algún tipo básico (`int, float`…) o de una clase determinada (`Quaternion, Vector3`…).

Por ejemplo: `transform.Translate` es un método que devuelve `void`, eso significa que realiza sus operaciones, pero no genera ningún valor nuevo.

Otro ejemplo: `Vector3.Distance` devuelve `float`, que es la distancia entre dos posiciones dadas.

Parámetros

Generalmente todos los métodos reciben una serie de parámetros, que deben ser satisfechos entre dos paréntesis.

Vamos a ver algunos ejemplos.

- Número de parámetros

```
Función Translate(x:float, y:float, z:float, relativeTo:
Space=Space.Self):void
```

Si nos fijamos en las comas que los separan, son 4 los parámetros que recibe esta función.

- Tipos de parámetros

```
Función Translate(x:float, y:float, z:float, relativeTo:
Space=Space.Self):void
```

Los tres primeros parámetros son de tipo `float` (números reales), el cuarto parámetro es un objeto de la clase `Space`.

- Parámetros por defecto

```
Función Translate(x:float, y:float, z:float, relativeTo:
Space=Space.Self):void
```

El operador "=" en el cuarto parámetro nos indica que es opcional. Podemos omitirlo y tomará un valor por defecto, en este caso `Space.Self`.

- Valores de retorno

```
Función Translate(x:float, y:float, z:float, relativeTo:
Space=Space.Self):void
```

Al final de la especificación, podemos observar que el tipo de valor de retorno es `void`, que indica que no retorna ningún valor. En otros casos podría ser `int, float`, etc.

- Funciones estáticas

```
Función Translate(x:float, y:float, z:float, relativeTo:
Space=Space.Self):void
```

Al principio de la especificación, no nos indica que se trate de un método `static`. Los métodos `static` se invocan a partir de la Clase, mientras que los que no lo son se invocan a partir de un objeto de la clase.

4.9.3.-Introducción al *scripting* con Javascript

Vamos a crear un proyecto nuevo, no vamos a importar de momento ningún *asset*, así que dejamos desmarcadas todas las casillas y creamos el proyecto.

Lo único que nos aparecerá en la interfaz será la cámara principal. La marcamos y en el inspector, la colocamos en las siguientes coordenadas: `X`=0, `Y`=1, `Z` =-5.

Ahora vamos a introducir un cubo en la escena. Nos vamos a `Gameobject -> Create other -> Cube`. Vamos a ubicar nuestro cubo en las coordenadas 0,0,0.

Vamos a guardar la escena: `File ->Save Scene` y la llamaremos "Ejemplo_1".

Ahora, en la vista Project, en un espacio libre debajo de donde está la escena guardada, hacemos click derecho con el ratón y en el menú emergente seleccionamos `Create -> Folder`. A la carpeta que aparecerá le llamaremos "Mis scripts". Con el puntero del ratón sobre dicha carpeta vamos a hacer de nuevo click derecho y esta vez seleccionamos `Create->Javascript`. Nos aparecerá un icono representando al nuevo *script* con el nombre por defecto "NewBehaviourScript". Lo renombramos, llamándolo "MiPrimerScript".

Como podemos observar, en el inspector al seleccionar nuestro *script* aparece una función por defecto. Sería lo que se conoce como una función sobreescribible, que quiere decir que Unity decide cuándo se va a llamar (la función `Update`, que es la que sale por defecto, es llamada por Unity cada *frame*) y nosotros decidimos qué hará cuando sea llamada, introduciendo código en su interior.

Para poder editar y crear nuestros *scripts*, hemos de acceder al editor que viene con Unity y la mejor manera para hacerlo es hacer doble click sobre el nombre de nuestro *script*. Se nos abrirá esto:

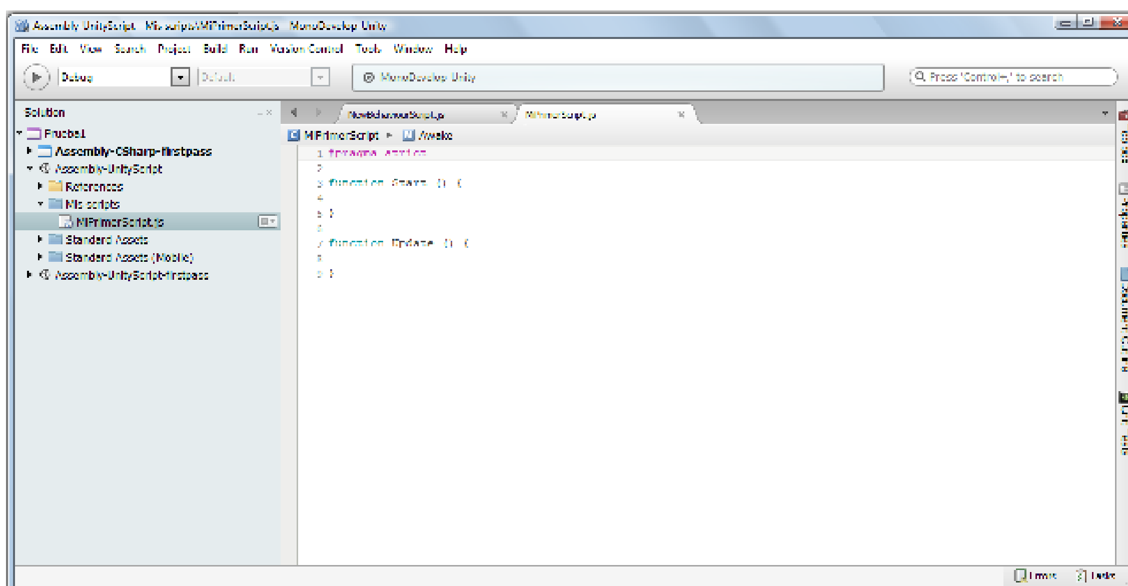


Figura 4.89 Editor de *scripts* de Unity

Vamos a aprovechar que tenemos el editor abierto para explicar otra característica de los *scripts* en Unity. Unity permite asignar/modificar los valores a las variables globales desde la propia interfaz de usuario y concretamente desde el inspector, bien asignando esos valores "a mano" o bien arrastrando un

objeto o componente del tipo de dicha variable global. Para que podamos hacer eso es preciso que la variable tenga un ámbito global y para ello que la misma se declare fuera de cualquier función.

Lo veremos mejor con un ejemplo: en el editor de *scripts*, encima de donde pone `function Update`, escribiremos lo siguiente:

```
var camaraDeSeguridad : Camera;
camaraDeSeguridad.name = "Mi camara";
Debug.Log(camaraDeSeguridad.name);
```

Declaramos una variable global (está fuera de cualquier función) de tipo `Camera` y lo que hacemos a continuación es asignarle un nombre a la variable: `"Mi camara"`. La tercera declaración muestra en pantalla (imprime) el valor de lo que esté entre sus paréntesis.

Guardamos el *script* (si no, no funcionará) en el editor de *scripts*. De vuelta a la interfaz de Unity, si seleccionamos el nombre del *script* vemos que en el inspector se ha actualizado dicho *script* con el código que hemos introducido.

Sin embargo nuestro *script* aún no es funcional, ya que no lo hemos vinculado a ningún objeto de nuestra escena. Pensemos que los *scripts* por sí solos no hacen nada, de la misma forma que cualquier *asset* que está en la carpeta/vista del proyecto (por ejemplo, una textura) no participará de alguna manera en una escena de nuestro juego si no lo arrastramos a esa escena, convirtiéndolo en parte de cualquier *gameobject*.

Por lo tanto, vamos a vincular nuestro *script* a uno de los *game objects* de nuestra escena y en concreto al cubo. Para hacer eso vamos a arrastrar el *script* desde la vista de proyecto hasta el propio objeto cubo, bien sea sobre su nombre en la jerarquía, bien sobre la propia imagen del cubo en la escena.

Tras arrastrar el *script* y con el cubo seleccionado en la jerarquía, en el inspector deberíamos estar viendo lo siguiente:

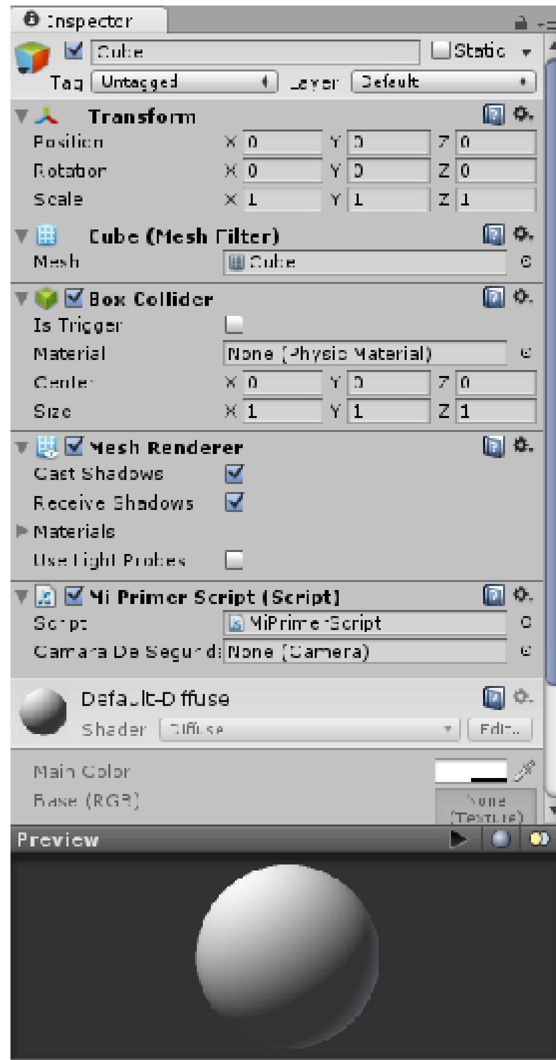


Figura 4.90

Observemos que en el inspector aparece ahora el nombre de nuestro *script* como una parte más del Cubo que tenemos en la escena. Veremos que la variable global queda expuesta en el inspector, de tal forma que no necesitamos ir al editor de *scripts* para asignarle un valor. Si la variable hubiera sido, por ejemplo, de tipo `float`, podríamos haber introducido o cambiado su valor simplemente escribiéndolo en el inspector. Si la variable fuera de tipo `booleano`, nos aparecería en el inspector con un *checkbox* al lado del nombre, bien marcado (true) o sin marcar (false) para que lo cambiáramos a conveniencia.

Pero como en este caso la variable es de tipo `Camera`, lo único que podemos hacer para inicializarla es proveerla de un objeto de ese tipo. Dado que en nuestra escena precisamente tenemos una cámara (la *main camera*), simplemente tendremos que arrastrarla desde la jerarquía hasta el lugar del inspector donde se asigna el valor de la variable y que ahora mismo pone "none".

Ya tenemos nuestro *script* asignado a un *gameobject* (nuestro cubo) y la única variable global inicializada con otro *object* (la cámara principal). Vamos a darle al play y ver qué sucede.

Si nos fijamos, debajo de la ventana game nos aparecerá impreso el nombre que le asignamos a nuestra cámara (ver Figura 4.91).

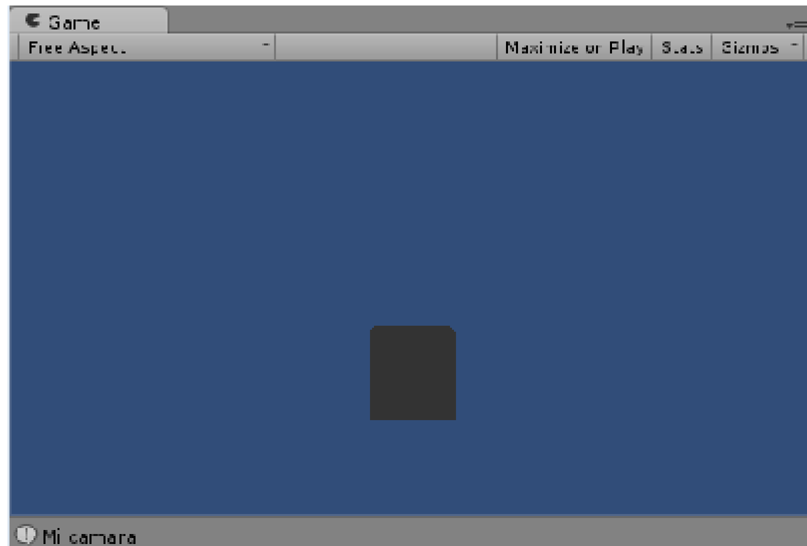


Figura 4.91

Por último, vamos a practicar un pequeño ejercicio que nos permitirá comprender las diferentes maneras (correctas e incorrectas) en que se puede declarar una variable. Borremos todo el contenido de nuestro *script* MiPrimerScript y tecleemos lo siguiente:

```
var sinTipoNiIni;
var conTipoSinIni : int;
var sinTipoConIni = 10;
var conTipoConIni : int = 10;
var aPlazos : int;
aPlazos = 10;
private var miSecreto : int;
var arrastrame : GameObject;
function Update() {
var enLaFuncion : int;
}
```

Salvamos el *script* y con el Cubo seleccionado (recordemos que este *script* está todavía vinculado al cubo que tenemos en la escena) echamos un vistazo al inspector. Podemos llegar a las siguientes conclusiones para cada manera de declarar una variable:

La variable `sinTipoNiIni`, que como su nombre apunta hemos declarado sin indicar ni el tipo de datos que debería contener, ni la hemos inicializado conjuntamente con la declaración, no aparece en el inspector, porque este no tiene forma de saber qué tipo de variable estamos declarando.

La variable `conTipoSinIni` sí es tenida en cuenta por el inspector, ya que aunque no la hemos inicializado, sí que hemos indicado el tipo de dato que queremos almacenar en ella.

La variable `sinTipoConIni` sí aparece en el inspector, ya que aunque no hemos declarado el tipo que contendrá, al estar inicializada permitimos a Unity deducir en base al valor el tipo de variable apropiado. Fijémonos que además de aparecer la variable en el inspector, lo hace con el valor inicial que le hemos dado.

La variable `conTipoConIni` aparece obviamente en el inspector.

La variable `aPlazos` aparece en el inspector, pero sin inicializar. En cambio, si pulsamos play observaremos que automáticamente le es asignado el valor 10 que le dimos en segunda declaración.

La variable `miSecreto` no es accesible desde el inspector porque es privada. De hecho, esa variable no podrá ser accedida desde ningún *script* distinto del que la contiene.

La variable `arrastrame` es recogida en el inspector y al ser de un tipo propio de Unity, nos aparecerá con una flechita diminuta en la derecha que nos indica que podemos importar del juego cualquier componente de ese tipo para asignar un valor a la misma, bien usando el menú que emerge al clickar dicha flecha, bien vía arrastrar el objeto o componente desde las carpetas de proyecto o jerarquía.

Por último, la variable `enLaFunción` no aparece en el inspector, ya que al ser declarada dentro de una función no tiene carácter público.

4.9.4.-Tutorial Transform

Figura 4.92 Diagrama de clases de la clase `Transform`

La clase `Transform`, como vemos en el gráfico, hereda de `Component`, que a su vez deriva de `Object` y por lo tanto esta clase tiene como propias las variables y funciones de las dos clases de las que hereda.

¿Qué es un *transform*?

Si seleccionamos cualquier *gameobject* de la escena, automáticamente el inspector nos muestra en la sección transform los valores de posición, rotación y escala de dicho *gameobject*.

No puede existir un *gameobject* en la escena sin su propio *transform.* En consecuencia, todo *gameobject* en nuestra escena está situado en un lugar determinado de esta (`position`), con una determinada inclinación (`rotation`) y le es aplicada una determinada escala comparativa de tamaño (`scale`).

Nota: Ahora, para nuestros ejemplos vamos a crear un *gameobject* vacío cuya única utilidad será contener nuestro *script* MiPrimerScript, para no andar vinculando *scripts* a *gameobjects* que nada tienen que ver con él. Para ello nos vamos a `GameObject->Create empty`. Al objeto que nos aparecerá en la jerarquía le renombramos como "PortaScripts" y le vinculamos MiPrimerScript, arrastrando MiPrimerScript a PortaScripts.

VARIABLES:

`position:`

`var position : Vector3`

Esta variable nos indica en qué punto del espacio global se ubica nuestro *transform* (y por lo tanto el *gameobject* al que el mismo va vinculado). El espacio global es el que hace referencia a las coordenadas de la escena y no a las de un determinado objeto. Si quisiéramos consultar/indicar el lugar del espacio local de un objeto, utilizaríamos la variable `localPosition`.

La variable `position` es de tipo `Vector3` y nos permite tanto obtener los datos de posición del *transform* como modificarlos para darle una nueva ubicación. Por ejemplo, si quisiéramos ubicar un objeto en el centro del espacio global, haríamos:

```
elObjetoQueSea.transform.position = Vector3(0, 0, 0);
```

Podemos asimismo acceder a uno sólo de los ejes de la posición, tal como sigue:

```
Debug.Log(elObjetoQueSea.transform.position.x);
```

Lo cual nos permitiría imprimir la posición que tiene nuestro *transform* referida al eje horizontal.

`localPosition:`

`var localPosition : Vector3`

`LocalPosition` indica la posición del *transform* al que dicha variable pertenece, calculada en términos relativos al *transform* del que aquél depende.

Parece más difícil de lo que es en realidad, tal como vamos a demostrar con un ejemplo sencillo:

Nos vamos a Unity. Vamos a crear un cubo: `Game Object->Create Other->Cube` y una cápsula: `Game Object->Create Other->Capsule`. Al cubo lo vamos a ubicar desde el inspector en la posición (0,0,0) y la cápsula en la posición (3,4,2).

Ninguno de estos *gameobjects* es hijo a su vez de otro, por lo que están y se mueven en coordenadas globales.

Abrimos nuestro *script* MiPrimerScript, borramos todo y escribimos:

```
transform.position = Vector3(-2,-1,-1);
```

Salvamos y arrastramos el *script* a nuestro cubo. Al darle al play, vemos que este se desplaza respecto la posición que ocupaba en las coordenadas globales (el centro justo) dos unidades a la izquierda, una hacia abajo y una hacia delante. Al pararlo, nuestro cubo vuelve a su lugar original, en 0,0,0.

Vamos ahora a, en la Jerarquía, arrastrar el *gameobject* cubo y soltarlo dentro de *gameobject* cápsula. Vemos que el cubo mantiene su posición en la vista de escena, pero si lo seleccionamos (está ahora contenido en la cápsula), sus coordenadas en el *transform* del inspector son ahora las inversas que las que le dimos al *gameobject* cápsula. Esto es así porque ahora para el cubo sus coordenadas de posición no dependen ya del espacio global, sino del local de la cápsula de la cual depende. Para el cubo la coordenada 0,0,0 no es ya el centro de la escena, sino el centro del *gameobject* capsula, y dado que con respecto a la cápsula el cubo está desplazado tres unidades a la izquierda, cuatro hacia abajo y dos hacia delante, son las que muestra.

El hecho de que el cubo dependa de la cápsula o dicho en terminología Unity, el cubo sea hijo de la cápsula, no implica que no pueda seguir siendo localizado/ubicado en coordenadas globales. De hecho, si le damos al play de nuevo, como estamos utilizando en nuestro *script* la variable `position` (global), el cubo se seguirá desplazando al mismo sitio al que lo hacía cuando aún no dependía de la cápsula.

Pero, obviamente, al tener ahora un padre, le podemos aplicar ahora al cubo la variable `localPosition` y así podemos modificar el *script* como sigue:

```
transform.localPosition = Vector3(-2,-1,-1);
```

Vemos ahora que el movimiento del cubo lo es en relación con la posición que ocupa papá capsula (o para ser más precisos el *transform* de papá cápsula). Probad si no lo veis claro a pasarle un vector (0,0,0) a nuestro *script* en lugar del que hemos escrito antes.

Si le aplicáramos esta variable `localPosition` a un *transform* sin padre, dicho *transform* se movería en el espacio global, tal como si le hubiéramos aplicado `position` y no `localPosition`.

eulerAngles:

var eulerAngles : Vector3

Esta variable indica la rotación del *transform* en grados. Se corresponde plenamente con los valores que aparecen en el apartado rotation de cada *transform* en el inspector. Los valores de esta variable vienen dados, al igual que los de `position`, en formato `Vector3`. Lo que sucede es que un valor de (90,0,0) referido a la posición implica que nuestro *transform* se desplace 90 unidades a la derecha, mientras que los mismos valores relativos a la rotación implicarían que nuestro *transform* gire 90 grados (el equivalente a un cuarto de vuelta) sobre el eje X.

Al igual que con la variable `position`, `eulerAngles` nos permite consultar individualmente la rotación de nuestro *transform* sobre un determinado eje. Pero si lo que queremos no es consultar sino cambiar la rotación, deberemos indicar los tres valores vía un `Vector3` y Unity los convertirá entonces en los valores que él usa internamente.

Unity trabaja internamente con Quaterniones, por motivos de rendimiento y exactitud, mientras que a los humanos nos resulta más sencillo hacerlo con grados Euler, así que en la práctica existen unos "atributos de traducción" que nos permite pasar de quaterniones a grados y viceversa, a fin de que hombres y máquina sigan trabajando con los elementos que mejor se adapten a sus capacidades.

Al igual que con `position`, existe una variable `eulerAngles` para funcionar con valores globales (la que estamos estudiando) y otra `localEulerAngles` que trabaja con valores locales (dependientes del padre del *transform* que rota).

A esta variable sólo podemos asignarle valores exactos, no de incremento, ya que fallaría pasados los 360º. Para incrementar una rotación de esta forma, tenemos que usar la función `Transform.rotate`.

Usemos un ejemplo muy simple. Antes arrastremos nuestro cubo en la jerarquía fuera de la cápsula, para desemparentarlo. Y rehacemos nuestro *script* (que debería continuar vinculado al cubo).

```
transform.eulerAngles = Vector3(60,0,0);
```

Podemos ver que nuestro cubo ha girado 60 grados sobre el eje X, entendido este como un alambre invisible que cruza de manera horizontal la escena (por decirlo de alguna manera).

Unity automáticamente convierte los ángulos Euler que le pasamos al *transform* en los valores usados para la rotación almacenada en `Transform.rotation`.

localEulerAngles:

var localEulerAngles : Vector3

Se refiere a la rotación en grados de un *transform* con relación a la que tiene el *transform* del cual depende en una relación hijo/padre.

Al igual que con la variable anterior, Unity automáticamente convierte los ángulos Euler en los valores usados para la rotación almacenada en `Transform.localRotation`.

rotation:

var rotation : Quaternion

Es la variable que contiene la rotación del *transform* en el mundo global almacenada en quaterniones.

Unity almacena internamente todas las rotaciones en quaterniones, lo cual no quiere decir que nosotros tengamos que trabajar directamente con ellos. Por norma general, o bien trabajaremos con grados Euler que luego se convertirán implícita o explícitamente en quaterniones o bien a lo sumo usaremos alguna función que nos permita encapsular las operaciones con quaterniones.

localRotation:

var localRotation : Quaternion

Esta variable indica la rotación en quaterniones de un *transform* relativa a la rotación de su padre

right, up y forward:

var right : Vector3

var up : Vector3

var forward : Vector3

Estas variables hacen referencia respectivamente al eje X (derecha/izquierda), Y (arriba/abajo) y Z(delante/detrás) de las coordenadas globales del *transform*. No las hemos de confundir con las variables del mismo nombre de la estructura `Vector3`. Así, `Vector3.up` es un atajo que equivale a `Vector3(0,1,0)`, esto es, implica un movimiento de una unidad hacia arriba. En cambio, `transform.up` meramente hace referencia al eje arriba/abajo, pero no implica movimiento. Para mover un *transform* por esta vía tendríamos que hacer algo como esto:

En Unity eliminamos el *script* que tenemos vinculado a nuestro cubo. En Proyecto abrimos nuestro *script* y tecleamos:

```
var unObjeto : GameObject;

unObjeto.transform.position = transform.right*10;
unObjeto.transform.eulerAngles = transform.up*45;
```

Tras salvarlo, con el PortaScripts seleccionado arrastramos el cubo a la variable expuesta Un Objeto en el inspector. Play.

Efectivamente, el cubo se desplaza 10 unidades a la derecha y 45 grados sobre el eje Y.

localScale:

var localScale : Vector3

Como su nombre indica, es una variable de tipo `Vector3` que permite consultar o establecer la escala de un *transform* en relación con la de su padre. Obviamente, un *transform* con esta variable en 0,0,0 tendrá la misma escala que su padre.

parent:

var parent : Transform

Hace referencia al padre del *transform* (que es obviamente otro *transform*) y a través de esta variable se puede consultar y/o cambiar dicho padre.

Veamos primero de qué forma podemos consultar valores relativos al padre de nuestro *transform*: Nos vamos a Unity, arrastramos en la Jerarquía el cubo dentro de la cápsula, borramos MiPrimerScript de PortaScripts para que no nos dé errores y en MiPrimerScript borramos todo y escribimos lo que sigue:

```
Debug.Log(transform.parent.gameObject.name);
```

Arrastramos tras salvar el *script* dentro del cubo (que está dentro de la cápsula). Play.

Aparece el nombre de la cápsula que es lo que pretendíamos. Vamos a fijarnos cómo a través del operador punto le decimos a Unity (leyendo de derecha a izquierda) que busque el nombre del *gameobject* al que pertenece el *transform* que es el padre del *transform* del *gameobject* que hace la consulta (al que va vinculado el *script*) y lo imprima en pantalla.

Vamos ahora a darle un padre nuevo a nuestro cubo. Abrimos el *script* y sin borrar nada, desplazamos la declaración que habíamos escrito un par de renglones hacia abajo y encima escribimos lo siguiente:

```
var nuevoPadre : Transform;

transform.parent = nuevoPadre;
Debug.Log(transform.parent.gameObject.name);//Esto ya lo teníamos
escrito
```

Salvamos, y seleccionamos el cubo para que en el inspector quede expuesta la variable `nuevoPadre`. Arrastramos hasta ella la cámara principal desde la jerarquía y le damos al play. Nuestro cubo ha cambiado de padre.

Pensemos, llegados a este punto, que nuestro *transform* hijo tiene una posición, rotación y escala relativa con respecto a su padre. Si cambiamos su padre, estas obviamente cambiarán.

root:

var root : Transform

Esta variable hace referencia al *transform* más alto de la jerarquía, por lo que guarda muchas similitudes con `parent`, pero referida al "parent de los parent" del que depende nuestro *transform*. Si este tiene un padre sin padre, el resultado equivaldrá a `parent`. Si su padre tiene un padre que tiene un padre que tiene un padre, la variable irá escalando hasta llegar al último padre y devolverá su valor (o nos permitirá modificar dicho valor). Para el caso de que el objeto que invoca esta variable no tenga padre, lo que se devuelve no es null, sino el propio *transform* que la invoca.

childCount:

var childCount : int

Es simplemente una variable de tipo `int` que indica el número de hijos que tiene un *transform*.

lossyScale:

var lossyScale : Vector3

Es esta una variable de tipo `Vector3` que representa la escala global del objeto. Es una variable de sólo lectura, por lo que obviamente no podemos modificar la escala global (la local ya vimos que sí con `localScale`) asignando valores a esta variable.

FUNCIONES:

Translate:

```
function Translate (translation : Vector3, relativeTo : Space =
Space.Self) : void
```

Esta función nos permite mover el *transform* en la dirección y distancia indicados en el parámetro de tipo `Vector3 traslation`. El segundo parámetro nos permite decidir si el *transform* se moverá según sus propias coordenadas (locales) o lo hará en base al espacio global. Por defecto, si no

indicamos el segundo parámetro, Unity entiende que el *transform* se moverá según sus propias coordenadas (`Space.Self`).

Desarrollemos esto con un ejemplo:

Para empezar, desemparentamos nuestro cubo respecto de la cápsula, arrastrándolo a un espacio vacío de la jerarquía. Acto seguido, con el cubo seleccionado, en el inspector le asignamos a la coordenada Y de rotación un valor de 35. Abrimos el *script* que tenemos asignado al cubo y tecleamos:

```
transform.Translate(Vector3.forward * 5);
```

Salvamos y presionamos play. El cubo avanza cinco unidades en su eje Z y recalcamos lo de "su" eje, ya que al no añadirle un segundo parámetro que diga lo contrario, se mueve según sus coordenadas locales.

Para ver la diferencia, añadamos como segundo parámetro a la función la variable `World` de la enumeración `Space`, para que sustituya al parámetro por defecto `Space.Self`:

```
transform.Translate(Vector3.forward * 5, Space.World);
```

```
function Translate (x : float, y : float, z : float, relativeTo : Space = Space.Self) : void
```

Existe un segundo prototipo para la función `Translate`. Como se puede comprobar, se diferencia del primero en que en lugar de pasar como parámetro para establecer la dirección y longitud del movimiento un `Vector3`, se pasa cada eje como un `float` independiente.

```
function Translate (translation : Vector3, relativeTo : Transform) : void

function Translate (x : float, y : float, z : float, relativeTo : Transform) : void
```

Estos dos prototipos varían de los anteriores en el segundo parámetro, que ya no versa sobre si el movimiento se hará tomando en cuenta las coordenadas locales del objeto que se mueve o las globales de la escena. En cambio, aquí el movimiento de nuestro *transform* vendrá fijado por otro *transform*. De esta manera, nos moveremos de acuerdo con las coordenadas locales de ese segundo objeto al que hacemos referencia. Si por ejemplo el parámetro `relativeTo` es una cámara, la derecha del `traslation` no es la derecha local de nuestro *transform*, o la derecha global, sino la derecha de la cámara. Por ejemplo:

Tecleamos lo siguiente en nuestro *script:*

```
transform.Translate(Vector3.right * 5);
```

Esto hará que nuestro cubo se mueva cinco unidades a **su** derecha.

Y ahora modificamos el *script* para que quede así:

```
var miCamara : Transform;

transform.Translate(Vector3.right * 5, miCamara);
```

Arrastramos la cámara principal hasta la variable `miCamara`. De nuevo al play.

Ahora el cubo se mueve cinco unidades a la derecha de la cámara.

Como último apunte hemos de añadir que si `relativeTo` es nulo, las coordenadas del movimiento pasarán a ser las globales.

Rotate:

function Rotate (eulerAngles : Vector3, relativeTo : Space = Space.Self) : void

Esta función permite rotar un *transform*. Acepta como parámetro un `Vector3` con los grados de rotación en ángulos Euler. Por defecto, al igual que sucediera con la función `translate`, el *transform* rota sobre sus coordenadas locales, pudiendo hacerlo según las coordenadas globales si se lo indicamos con `Space.World` como segundo parámetro.

Veamos un ejemplo. Rehagamos nuestro *script* con el siguiente contenido:

```
function Update() {

transform.Rotate(Vector3.right * 25 * Time.deltaTime);
transform.Rotate(Vector3.up * 20 * Time.deltaTime, Space.World);
}
```

Al darle al play, observaremos que nuestro cubo rota sobre su eje X a razón de 25 grados por segundo mientras a la vez gira sobre el eje Y global a razón de 20 grados por minuto.

Es de señalar que ambas instrucciones se encuentran contenidas dentro de la función `Update`. Esta función, es llamada cada *frame* por nuestro ordenador (el *framerate* de cada ordenador varía), de tal manera que el movimiento es continuo, pues los valores de rotación de nuestro cubo son actualizados constantemente.

Precisamente porque el *framerate* de cada ordenador es distinto, y para evitar que en función de cada PC los objetos se movieran más o menos deprisa (lo que tendría indeseables consecuencias, sobre todo en juegos en línea multijugador), se suele utilizar la variable de la clase `Time` `Time.deltaTime`. `Time.deltaTime` lo que consigue es transformar la unidad de frecuencia de cualquier tipo de movimiento de *frames* a segundos. Si por ejemplo en el último *script* no hubiéramos usado estas variables de tiempo, el cubo giraría 25 y 20 grados cada *frame*, quedando al albur de cada ordenador la frecuencia real que eso supondría. Al multiplicarlo por `Time.deltaTime` las rotaciones dependerán del tiempo y en consecuencia se producirán con la misma cadencia en cualquier ordenador.

function Rotate (xAngle : float, yAngle : float, zAngle : float, relativeTo : Space = Space.Self) : void

En esta versión de la función, la única diferencia es que se sustituye el `Vector3` por tres `floats`, en los cuales se consignarán igualmente los grados de rotación.

function Rotate (axis : Vector3, angle : float, relativeTo : Space = Space.Self) : void

La variante que nos ofrece este tercer prototipo es que por un lado se indica en el primer parámetro sobre qué eje queremos que rote el *transform* y en un segundo parámetro de tipo `float` le hemos de indicar el número de grados ha de rotar.

RotateAround:

function RotateAround (point : Vector3, axis : Vector3, angle : float) : void

Esta función nos permite que nuestro *transform* rote alrededor de un punto (u objeto situado en ese punto), como si orbitara.

El parámetro `point` sería el punto, descrito por un `Vector3`, alrededor del cual queremos hacer girar nuestro *transform*. `Axis` nos servirá para indicar sobre qué eje queremos girar y `angle` el número de grados por *frame* (si está dentro de la función `update`) o por segundo (si implicamos la variable de clase `Time.deltaTime`) que queremos que gire. Obviamente, aquí estamos variando tanto la rotación como la posición de nuestro *transform*.

Si por ejemplo quisiéramos que nuestro cubo girara sobre su eje Y alrededor del centro exacto del espacio global, a razón de 20 grados por segundo, escribiríamos este *script*:

```
function Update() {
transform.RotateAround (Vector3.zero, Vector3.up, 20 *
Time.deltaTime);
}
```

Dado que el cubo se hallaba ya en las coordenadas globales 0,0,0 el *script* anterior lo único que consigue es que el cubo parezca girar sobre sí mismo. Vamos a hacer lo siguiente: colocamos en el inspector a nuestro cubo en position.`x` = -1 y volvemos a darle al play.

Y vemos ya más claramente cómo orbita alrededor del punto dado, cambiando simultáneamente de posición y de rotación.

Podemos hacer también con esta función que un objeto orbite alrededor de otro. Escribimos:

```
var centroDeGravedad: Transform;

function Update() {
transform.RotateAround (centroDeGravedad.position, Vector3.up, 20 *
Time.deltaTime);
}
```

Acto seguido, arrastramos el objeto sobre el que queremos que orbite nuestro cubo, en este caso la cápsula. Le damos al play y comprobamos.

Observemos que aunque arrastramos un *gameobject*, lo que está esperando nuestro *script* es un *transform*. Lo que sucede es que todos los componentes de nuestro *gameobject* comparten nombre, de tal

manera que cuando en casos como este Unity detecta que uno de los componentes del *gameobject* que le estamos arrastrando tiene el nombre y el tipo del que está esperando para cumplimentar una variable, automáticamente selecciona -en este caso- el *transform* homónimo y no el *gameobject*.

Ese *transform*, no obstante, no es del tipo `Vector3` que espera nuestra función. Sí que lo es la variable `position` del mismo, que además contiene las coordenadas globales del *transform* sobre el que queremos orbitar.

LookAt:

function LookAt (target : Transform, worldUp : Vector3 = Vector3.up) : void

El nombre de esta función se podría traducir al castellano como "mira a" y es exactamente eso lo que hace: Rotar nuestro *transform* hacia un objetivo -parámetro `target`- que es asimismo de tipo `transform` (y no `Vector3`, como en el caso de la función anterior). Esa rotación se efectúa sobre un eje global y por defecto este será el eje Y. Ello implicará que nuestro *transform*, si no le indicamos lo contrario, girará hacia la derecha y la izquierda "mirando" al *transform* que hayamos designado como objetivo.

Esta función se usa mucho para cámaras, cuando pretendemos que estas sigan en todo momento a un determinado personaje.

Para que el *transform* gire sobre el eje que le indiquemos, ha de estar totalmente perpendicular a dicho eje global.

Vamos a la práctica. En el Proyecto, colocamos el ratón sobre la carpeta Mis Scripts y le damos al botón derecho->`create->javascript`. Al nuevo *script* lo renombramos como “MiSegundoScript” y le damos doble click para se nos abra el editor.

Desplazamos un poco hacia abajo la función `Update` y la dejamos como sigue:

```
var sigueme : Transform;

function Update () {
transform.LookAt(sigueme);
}
```

Salvamos y la arrastramos hasta nuestra cámara en la jerarquía. Seleccionamos entonces la cámara y arrastramos ahora hasta la variable `sigueme` nuestro cubo.

Previamente a darle al play nos aseguramos de que el otro *script* (MiPrimerScript) que tenemos vinculado al cubo, contenga lo siguiente:

```
var centroDeGravedad: Transform;

function Update() {
transform.RotateAround (centroDeGravedad.position, Vector3.up, 20 *
Time.deltaTime);
}
```

Y ahora sí, ya podemos darle al play y perseguir a nuestro cubo.

function LookAt (worldPosition : Vector3, worldUp : Vector3 = Vector3.up) : void

Este segundo prototipo se diferencia del primero en que el objetivo que ha de seguir nuestro *transform* no es otro *transform*, sino un Vector3. Esto tiene sentido, siguiendo con el ejemplo de la cámara, si quisiéramos que esta enfocara un punto concreto de la escena (en coordenadas globales).

Por ejemplo, si queremos que una cámara enfoque al centro de la escena, le vinculamos este *script*:

```
transform.LookAt(Vector3.zero);
```

Otra posibilidad que nos permite esta función es que sea nuestro jugador, desde su teclado, quien se encargue de decidir dónde ha de enfocar la cámara. Para ello tendremos que recurrir a una clase que aún no hemos visto, la clase `input`.

Abrimos el *script* MiSegundoScript, y tecleamos lo que sigue:

```
function Update () {
transform.LookAt(Vector3(Input.GetAxis("Horizontal") * 10.0,0,0));
}
```

Le damos al play y movemos la cámara con las flechas de desplazamiento horizontal. Quizás notemos que el resultado es un poco basto, pero de esta manera nos hacemos una idea de las utilidades de esta función.

TransformDirection:

function TransformDirection (direction : Vector3) : Vector3

Esta función toma como único parámetro la dirección local de un *transform* almacenada en un Vector3 y la convierte en dirección global, devolviéndola en otro Vector3.

La utilidad de esta función puede resultar un poco confusa al principio. Pensemos, para intentar aproximarnos al concepto, que quisiéramos hacer nuestra propia versión del FIFA 2011. Modelamos un campo de futbol y para darle más realismo y emoción al juego, colocamos varias cámaras en los laterales del campo, cámaras que mediante `LookAt` irían siguiendo los lances del juego. El problema que nos encontraríamos es que cuando Messi (por poner un caso) está avanzando hacia delante (en las coordenadas globales), en cambio en nuestra cámara lateral pareciera que lo está haciento -por ejemplo- hacia la izquierda. Y en consecuencia, cuando intentamos que nuestro defensa lo ataje moviéndolo hacia atrás (según la perspectiva que nos da la cámara lateral) veremos consternados que el defensa en lugar de retroceder se desplaza hacia la derecha.

Esto no nos pasaría si, gracias a esta función, convertimos la coordenada "hacia detrás" de nuestra cámara en su equivalente en coordenadas globales. Si esa función se la aplicamos a todas las cámaras, no tendremos que estar pensando "Ojo, que esta cámara está en el gol norte, por lo que si me baso en ella cuando haga un movimiento, he de recordar que arriba es abajo y viceversa y la derecha es la izquierda y bla, bla, bla".

Veámoslo en un ejemplo.

En Unity borramos el *script* MiSegundoScript de la cámara. Colocamos en el inspector a nuestro cubo en las coordenadas 0,0,0 con una rotación igualmente de 0,0,0. Asimismo, la cámara debería estar en las coordenadas de posición 0,1,-5 con los tres ejes de rotación a 0.

Abrimos MiPrimerScript. Tecleamos esto:

```
var miCamara : Transform;
var estaEsMiDerecha : Vector3;

estaEsMiDerecha = miCamara.TransformDirection(Vector3.right);
transform.Translate(estaEsMiDerecha * 3);
```

Arrastramos la cámara a miCamara. Le damos al play. El cubo se moverá 10 unidades a la derecha. Pero, ¿a la derecha de quién? Si observamos, la derecha del cubo es también la derecha de la cámara.

Para averiguarlo, vamos a recolocar la cámara en estas coordenadas:

Position: -10, 1, -0.5

Rotation: 0, 90, 0

Y de nuevo le damos al play. Obviamente, el cubo se mueve a la derecha de la cámara. Visto desde la ventana game, coincidirá ahora "nuestra" derecha (entendiendo como tal la que nos muestra la pantalla) con el sentido del movimiento.

El *script* no necesita mucha explicación. Inicializamos una variable con el *transform* de la cámara que hemos arrastrado. La derecha de esa cámara la almacenamos en una variable de tipo `Vector3`, la cual luego pasamos como parámetro a nuestra función `TransformDirection` para que nos convierta la derecha de nuestra cámara en la derecha de las coordenadas globales. A partir de ahí, todo lo que le suceda a la derecha de nuestra cámara (por así decirlo) le estará pasando a la derecha del mundo.

`function TransformDirection (x : float, y : float, z : float) : Vector3`

Es la misma función, pero aplicando como parámetros 3 floats para cada eje en lugar de un Vector3.

`InverseTransformDirection:`

`function InverseTransformDirection (direction : Vector3) : Vector3`

O bien

`function InverseTransformDirection (x : float, y : float, z : float) : Vector3`

Se trata de la función inversa a la anterior y por consiguiente transforma una dirección global en dirección local.

Veamos un ejemplo:

Devolvemos antes que nada a nuestra cámara a su lugar y rotación originales:

Position: 0,1,-5

Rotation: 0,0,0

La posición y rotación de nuestro cubo, por su parte, está totalmente a 0.

Abrimos MiPrimerScipt, y tecleamos:

```
var estaEsLaDerechaDelMundo : Vector3;

estaEsLaDerechaDelMundo =
transform.InverseTransformDirection(Vector3.right);

transform.Translate(estaEsLaDerechaDelMundo * 2);
```

Como vemos, declaramos una variable de tipo `Vector3` que luego inicializamos de la siguiente forma: le pasamos a la función `InverseTransformDirection` el parámetro `Vector3.right`, que en esta función representa la derecha en coordenadas globales (no la derecha de ningún *transform*). Esa derecha del mundo, global, es "traducida" por la función en una coordenada local susceptible de usar por cualquier *transform* y es asignada, como decíamos, a nuestra variable `estaEsLaDerechaDelMundo`. Dicha variable, por último, es pasada como parámetro de movimiento al *transform* del *gameobject* que tiene vinculado el *script* (en este caso el cubo).

Vamos a probarlo:

El cubo se desplaza a la derecha. Pero, para saber si la derecha es la derecha del cubo o la derecha global, podemos en el inspector darle un valor al rotation.Y del cubo de 45 grados, por ejemplo. Y probamos de nuevo.

Definitivamente, el cubo se mueve ahora siguiendo el eje X global y no el suyo local.

TransformPoint:

function TransformPoint (position : Vector3) : Vector3

function TransformPoint (x : float, y : float, z : float) : Vector3

A diferencia de `TransformDirection`, lo que esta función transforma de local en global es la posición y no la dirección. Esto es, esta función no versa sobre si un *transform* se desplaza a su derecha o a la derecha de las coordenadas globales, sino de si las coordenadas en que se encuentra el *transform* son globales (respecto al mundo) o locales (respecto al padre de dicho *transform*). Con esto entendido, como decimos, esta función acepta como parámetro las coordenadas locales de un *transform* (su ubicación respecto de la de su padre) y devuelve dichas coordenadas traducidas a coordenadas globales.

Lo veremos más claro con un ejemplo. Con el cubo en position 0,0,0 y rotation en 0,0,0 (si no, no funcionará) arrastramos el cubo dentro de la cápsula en la Jerarquía, para convertir a cubo en hijo de la cápsula. Si seleccionamos el cubo, vemos en el inspector que sus coordenadas de posición han pasado a ser locales respecto de su padre. Abrimos MiPrimerScript (que debería seguir siendo parte del cubo) y escribimos el siguiente *script*:

```
var coordenadasLocales : Vector3;
var coordenadasGlobales: Vector3;
```

```
var coordenadasTransformadas: Vector3;

coordenadasLocales = transform.localPosition;
coordenadasGlobales = transform.position;
coordenadasTransformadas = transform.position =
transform.TransformPoint(transform.localPosition);

Debug.Log("El cubo tiene las coordenadas locales " +
coordenadasLocales.ToString() +
" las globales " + coordenadasGlobales.ToString() + " y las
transformadas " +
coordenadasTransformadas.ToString());
```

El ejemplo parece más complicado de lo que es. Declaramos tres variables de tipo `Vector3` para que contengan las tres coordenadas que imprimiremos para nuestro cubo (los nombres de las variables son lo suficientemente explicativos). Acto seguido inicializamos las dos primeras variables con la posición global y local del *transform* y la tercera con las coordenadas que tendrá el *transform* cuando convirtamos sus coordenadas locales (respecto de la cápsula) en globales.

Salvamos y le damos al play. Observamos que por un lado el cubo se desplaza en dirección y distancia opuesta al cilindro, ya que el -3,-4,-2 que constituían sus coordenadas locales ahora ha pasado a ser su posición en coordenadas globales.

InverseTransformPoint:

function InverseTransformPoint (position : Vector3) : Vector3

function InverseTransformPoint (x : float, y : float, z : float) : Vector3

Función inversa a la precedente, que por tanto transforma la posición de un *transform* dado del espacio global al espacio local.

DetachChildren:

function DetachChildren () : void

Es una función muy sencilla que sirve para desemparentar los hijos. Para probarla debemos eliminar el *script* vinculado al cubo y posteriormente hacer doble click en MiPrimerScript en el Proyecto para teclear lo siguiente:

```
transform.DetachChildren();
```

Salvamos y lo arrastramos a la cápsula, que es el *transform* que tiene hijos. Le damos al play y observaremos cómo en la Jerarquía el cubo deja de aparecer como hijo de la cápsula.

Esta función es útil, entre otras cosas, cuando por alguna razón queramos destruir al objeto padre sin destruir a sus hijos:

```
transform.DetachChildren();
Destroy(gameObject);
```

Find:

function Find (name : String) : Transform

Con esta función podemos buscar por su nombre -y en su caso acceder- a un *transform* hijo de nuestro *transform*. La función devuelve dicho *transform* hijo, si lo encuentra. Si no lo encuentra, retorna null.

Si tenemos que buscar a varios niveles, esto es, hijos de los hijos de nuestros hijos, podemos utilizar un *slash* o barra inclinada ("/") para recrear la jerarquía donde queremos buscar (p ej. "Cuerpo/Brazo/Mano/Indice").

Dado que deberíamos tener el cubo aún dentro de la cápsula y MiPrimerScript vinculado a esta, lo aprovecharemos para realizar un ejemplo:

```
var aquiMiHijo : Transform;

function Update() {
aquiMiHijo = transform.Find("Cubo");
aquiMiHijo.Rotate(Time.deltaTime*60, 0, 0);
}
```

Como podemos comprobar al darle al play, la función `Find` encuentra un *transform* hijo llamado "Cubo" (recordemos que, por un lado hemos de suministrarle un *string*, esto es, no nos debemos olvidar de las comillas y por otro lado que todos los componentes de un *gameobject* comparten por defecto el mismo nombre que este, así que el *transform* del Cubo se llama Cubo) y almacena ese *transform* que encuentra dentro de la variable `aquiMiHijo`. A partir de ahí, podemos utilizar esa variable como si fuera un alias del propio *transform*.

IsChildOf:

function IsChildOf (parent : Transform) : boolean

Esta función casi no requiere explicación. Devuelve true si el *transform* que hace la pregunta es hijo, "nieto" o incluso idéntico al *transform parent* que pasamos como parámetro. En otro caso, devuelve false.

4.9.5.-Tutorial Input

Es la interfaz que controla todo el sistema de entradas (*inputs*) de Unity. Esta clase se usa, por ejemplo, para leer la configuración de ejes en el *input Manager*.

Para leer un eje usaríamos `Input.GetAxis` con uno de los ejes por defecto: "Horizontal" y "Vertical" están configurados para *joystick*, así como A,W,S,D, y las teclas de flecha. "MouseX" y "Mouse Y" están configurados para el movimiento del ratón. Por su parte "Fire1", "Fire2" y "Fire3" están configurados para Ctrl, Alt y tres botones de ratón o *joystick*. Pueden añadirse nuevos ejes de entrada en el *Input Manager*.

Si hemos de usar *inputs* para cualquier tipo de comportamiento que entrañe movimiento, es aconsejable usar `Input.GetAxis`. Esto nos dará una entrada más suave y configurable que puede servir para teclado, *joystick* o mouse. En cambio es mejor usar `Input.GetButton` para acciones como eventos, mejor que para movimientos.

Las llamadas a *inputs* se deben hacer dentro de la función `update`.

VARIABLES DE CLASE:

`mousePosition`:

`static var mousePosition : Vector3`

Variable de sólo lectura que indica la posición actual del ratón en coordenadas de píxeles, donde la esquina inferior izquierda de la pantalla o ventana está en (0, 0) y la superior derecha en (`Screen.width, Screen.height`).

Podemos probarlo con un ejemplo sencillo, para el cual eliminamos el *script* vinculado a la cápsula y reeditamos MiPrimerScript como sigue:

```
function Update(){
 var apunten : Vector3 = Input.mousePosition;
 Debug.Log(apunten);
}
```

Salvamos y arrastramos hasta PortaScripts. Meramente se nos imprimirá en pantalla la posición exacta en píxeles de nuestro cursor. Podemos observar que la esquina inferior izquierda se mueve en parámetros del 0,0.

`anyKey`:

`static var anyKey : boolean`

Booleano de sólo lectura que comprueba si hay alguna tecla o botón del ratón apretada en ese momento.

Podemos modificar nuestro *script* anterior para ilustrar esta variable:

```
function Update(){
 if(Input.anyKey){
    Debug.Log("Se ha pulsado una tecla o botón");
}
}
```

No tenemos más que apretar cualquier tecla o botón del ratón para ver el mensaje impreso.

`anyKeyDown`:

`static var anyKeyDown : boolean`

Variable de sólo lectura que devuelve true el primer *frame* en que el usuario golpea cualquier tecla o botón del ratón. Debemos colocar esta variable dentro de la función `update`, ya que el estado de la misma se resetea cada *frame*. No devolverá true de nuevo hasta que el usuario libere todas las teclas/botones y presione alguna tecla/botón otra vez.

Para entender gráficamente la diferencia entre la variable anterior y la siguiente vamos a hacer una cosa: volvemos a darle al play (sigue en vigor el ejemplo anterior) y hacemos un click con un botón del ratón. Automáticamente nos aparece el mensaje impreso bajo la ventana del juego. Hacemos un click ahora sobre dicho mensaje, para que nos aparezca el *popup* de la consola, tal como muestra la captura:

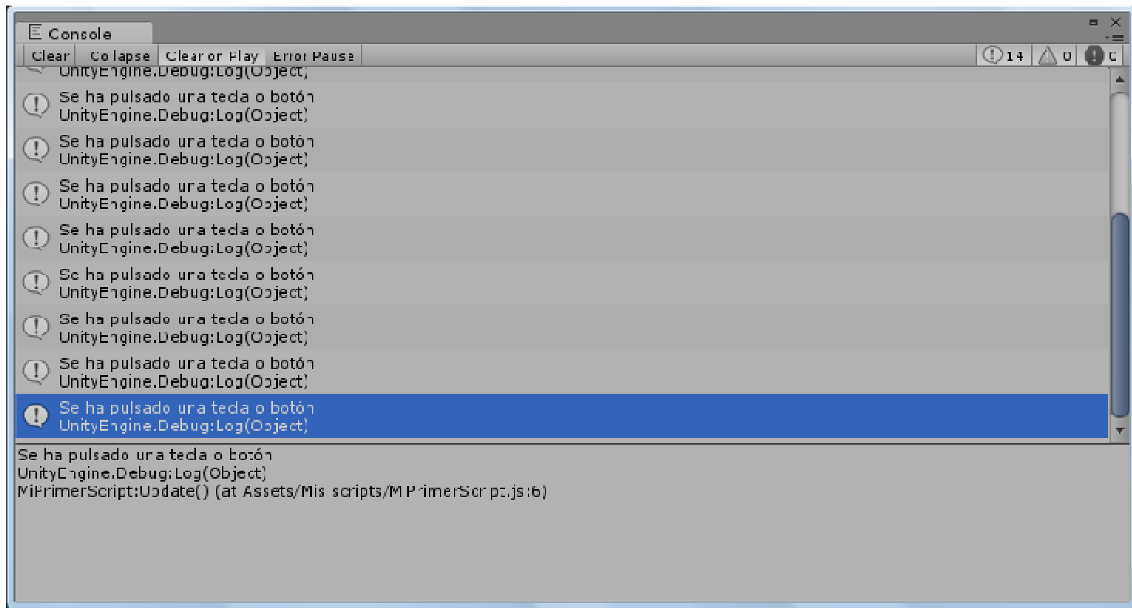


Figura 4.93

Vale. Ahora, sin retirar dicho *popup*, presionamos sin soltar el botón del mouse sobre la ventana Game. Vemos que se imprime de manera continuada el mensaje (una por cada *frame* que mantenemos pulsado el botón del ratón).

Y ahora sustituimos en el *script* la variable `anyKey` por `anyKeyDown` y repetimos los pasos anteriores. Comprobaremos que aunque mantengamos el botón del ratón presionado, el mensaje sólo se imprime el primer *frame* que lo pulsamos y no se vuelve a imprimir hasta que lo soltamos y lo volvemos a apretar.

`inputString:`

`static var inputString : String`

Variable de sólo lectura que devuelve la entrada de teclado introducida este *frame*. La misma sólo puede contener caracteres ASCII o un par de caracteres especiales que deben ser manejados: el carácter “\b” significa retroceso y “\n” representa return o enter.

`acceleration:`

`static var acceleration : Vector3`

Última medición de aceleración lineal de un dispositivo en espacio tridimensional. Sólo lectura

`accelerationEvents:`

`static var accelerationEvents : AccelerationEvent[]`

Variable de sólo lectura que devuelve una lista de mediciones de aceleración ocurridas durante el último *frame*. Dichas medidas son alojadas en variables temporales.

Dichas variables temporales son un array del tipo `AccelerationEvent`, que es una estructura con dos valores:

`acceleration`: De tipo `Vector3`, es el valor de aceleración.

`deltaTime`: De tipo `float`, el tiempo transcurrido desde la última medición de aceleración.

`accelerationEventCount`:

`static var accelerationEventCount : int`

Número de mediciones de aceleración ocurrida durante el último *frame*.

`eatKeyPressOnTextFieldFocus`:

`static var eatKeyPressOnTextFieldFocus : boolean`

Propiedad que indica si las teclas impresas son comidas por la entrada de texto si esta tiene el foco (por defecto true).

FUNCIONES DE CLASE:

Recordemos que las funciones de clase, como su nombre indica, no van vinculadas a objetos o instancias de una clase, sino a la clase en sí, esto es, no se utiliza el operador punto entre el nombre del objeto/instancia y el de la función, como sucedería en una función normal (o método) dentro de una clase.

`GetAxis`:

`static function GetAxis (axisName : String) : float`

Devuelve el valor del eje virtual identificado por `axisName`.

El valor estará en el rango -1...1 para entradas de teclado (tradicionalmente la flechas de desplazamiento) y *joystick*. Si el axis es indicado por el movimiento del ratón, este será multiplicado por el eje de sensibilidad y su rango será distinto a -1…1.

Es una de las funciones con las que más tropezaremos cuando programemos con Unity, así que merece la pena que nos detengamos para ilustrarla con algún ejemplo.

El primer ejemplo lo hemos sacado del manual de referencia. Vamos a crear una esfera: `GameObject->Create Other->Sphere` y le vinculamos MiPrimerScript (Para que funcione bien la esfera no puede tener vinculado un componente *Rigidbody*). Y acto seguido editamos nuestro *script* como sigue:

```
var velocidad : float = 2.0;
var velocidadRotacion : float = 45.0;

function Update () {

 var translacion : float = Input.GetAxis ("Vertical") * velocidad;
 var rotacion : float = Input.GetAxis ("Horizontal") *
velocidadRotacion;

 translacion *= Time.deltaTime;
 rotacion *= Time.deltaTime;

 transform.Translate (0, 0, translacion);
```

```
 transform.Rotate (0, rotacion, 0);
}
```

Tratemos de explicar lo que hemos hecho. A través de `Input.GetAxis` recogemos las entradas de teclado provenientes de las flechas de desplazamiento verticales y horizontales. Así, la flecha de desplazamiento hacia arriba vale 1 y la de desplazamiento hacia abajo vale -1 y lo mismo hacia la derecha (1) y la izquierda (-1). Dichos valores son multiplicados por la velocidad contenida en la variable `velocidad` en el caso del eje arriba/abajo, y por la contenida en la variable `velocidadRotacion` para el eje derecha/izquierda. Ambos valores son almacenados en variables de tipo `Vector3` que luego -tras ser reconvertidos en velocidad por *frame* a velocidad por segundo- se utilizan para participar en el movimiento del *game object* tanto en el eje delante/detrás de translación como en el eje Y de rotación.

En suma, nos queda un *script* -muy básico y torpe, eso sí- para manejar el desplazamiento y giro de nuestra esfera en base a las flechas de desplazamiento. Probémoslo un rato.

Hemos dicho que `Input.GetAxis` también acepta como entrada el provocado por el movimiento vertical y horizontal del ratón, así que ilustrémoslo:

```
var velocidadHorizontal : float = 20.0;
var velocidadVertical : float = 20.0;

function Update () {
 var h : float = velocidadHorizontal * Input.GetAxis ("Mouse X");
 var v : float = velocidadVertical * Input.GetAxis ("Mouse Y");
 transform.Rotate (v, h, 0);
}
```

Es un ejemplo sencillo. Se almacena en sendas variables el fruto de multiplicar el valor de los ejes de desplazamiento horizontal y vertical del ratón (que recordemos que no es -1,1) por el valor de velocidad que le hayamos dado a las variables expuestas del inicio de *script*. En base a ello, nuestra esfera girará sobre su eje X e Y en respuesta a los movimientos vertical y horizontal de nuestro ratón.

GetAxisRaw:

static function GetAxisRaw (axisName : String) : float

Devuelve el valor del eje virtual identificado por el parámetro `axisName` sin ningún filtro de suavizado aplicado. El valor estará en el rango de -1…1 para entrada de teclado y *joystick*. Como a la entrada -al contrario de la función anterior- no se le aplica *smooth* (suavizado), la entrada de teclado será siempre o -1 o cero o 1. Esta función puede sernos útil para el caso de que queramos hacer todo el proceso de suavizado de entrada nosotros mismos manualmente.

GetButton:

static function GetButton (buttonName : String) : boolean

Devuelve true mientras el botón virtual que le pasemos como parámetro en formato `string` esté presionado. Podemos pensar por ejemplo en un disparador automático, que devolvería true mientras el botón estuviera presionado.

Eso sí, así como esta función es óptima para acciones como disparar un arma, para cualquier tipo de movimiento es mejor usar `GetAxis`, que a este le introduce valores de suavizado que en `GetButton` no hallaremos (ni podremos implementar manualmente, al contrario que en `GetAxisRaw`.

Un ejemplo:

```
var proyectil : GameObject;
var frecuenciaDisparo : float = 0.5;
private var proximoDisparo : float = 0.0;

function Update () {
  if (Input.GetButton ("Fire1") && Time.time > proximoDisparo) {
  proximoDisparo = Time.time + frecuenciaDisparo;
  var clon : GameObject =
  Instantiate(proyectil, transform.position+Vector3.forward,
transform.rotation) as
  GameObject;
}
}
```

Salvamos y arrastramos el cubo a la variable expuesta `proyectil`. Pulsamos play y observaremos que estamos clonando/disparando cubos cada vez que -dentro del lapso de tiempo permitido- pulsamos el botón izquierdo del ratón o la tecla Ctrl situada a la izquierda del teclado (que son por defecto los dos elementos que tenemos vinculados al evento Fire1. Por otro lado, si mantenemos pulsado de manera ininterrumpida bien el botón o bien la tecla indicados, observaremos que disparamos un cubo cada medio segundo.

En sí el *script* comprueba si hemos pulsado la tecla o botón que tengamos asignada a Fire1. En caso afirmativo pasa a comprobar si ha transcurrido un lapso de tiempo superior al que le hemos fijado en `frecuenciaDisparo` (que para el primer disparo valdrá cero). Si también es true esta segunda condición se le añade medio segundo de espera al resto de disparos más el tiempo transcurrido en hacerlo, el *script* nos permite clonar/disparar otro proyectil más.

GetButtonDown:

static function GetButtonDown (buttonName : String) : boolean

Devuelve true durante el *frame* en que el jugador aprieta el botón virtual identificado como `buttonName`. Debemos llamar siempre a esta función desde la función `Update`, dado que el estado se resetea cada *frame*. No devolverá true hasta que el usuario libere la tecla y la presione de nuevo, al igual que sucedía con `anyKeyDown`.

GetButtonUp:

static function GetButtonUp (buttonName : String) : boolean

Devuelve true el primer *frame* en que el jugador libera el botón virtual identificado como `buttonName`. Recordemos llamar esta función desde `Update` ya que se resetea su estado cada *frame*. No devolverá true hasta que se libere la tecla y se vuelva a presionar.

GetKey:

static function GetKey (name : String) : boolean

Devuelve true mientras el jugador aprieta la tecla identificada como `name` (pensemos de nuevo en un disparador automático).

Para ver la lista de identificadores de tecla podemos consultar *Input Manager* en el menú->`Edit->Project Settings->Input`.

Un ejemplo sencillo:

```
function Update () {
if (Input.GetKey ("up"))
print ("Has presionado la flecha de desplazamiento superior");

if (Input.GetKey ("down"))
print ("Has presionado la flecha de desplazamiento inferior");
}
```

static function GetKey (key : KeyCode) : boolean

En este segundo prototipo la función devuelve true mientras el jugador presiona la tecla identificada por el parámetro de tipo `KeyCode`.

Así, el ejemplo anterior en esta segunda modalidad se quedaría así:

```
function Update () {
if (Input.GetKey (KeyCode.UpArrow))
print ("Has presionado la flecha de desplazamiento superior");

if (Input.GetKey (KeyCode.DownArrow))
print ("Has presionado la flecha de desplazamiento inferior");
}
```

Paso a relacionar todo el `enum KeyCode`:

None	Not assigned (never is pressed)
Backspace	The backspace key
Delete	The forward delete key
Tab	The tab key
Clear	The Clear key
Return	Return key
Pause	Pause on PC machines
Escape	Escape key
Space	Space key
Keypad0	Numeric keypad 0
Keypad1	Numeric keypad 1
Keypad2	Numeric keypad 2
Keypad3	Numeric keypad 3
Keypad4	Numeric keypad 4
Keypad5	Numeric keypad 5
Keypad6	Numeric keypad 6
Keypad7	Numeric keypad 7
Keypad8	Numeric keypad 8
Keypad9	Numeric keypad 9
KeypadPeriod	Numeric keypad '.'
KeypadDivide	Numeric keypad '/'
KeypadMultiply	Numeric keypad '*'
KeypadMinus	Numeric keypad '-'
KeypadPlus	Numeric keypad '+'
KeypadEnter	Numeric keypad enter
KeypadEquals	Numeric keypad '='
UpArrow	Up arrow key
DownArrow	Down arrow key
RightArrow	Right arrow key

LeftArrow	Left arrow key
Insert	Insert key key
Home	Home key
End	End key
PageUp	Page up
PageDown	Page down
F1	F1 function key
F2	F2 function key
F3	F3 function key
F4	F4 function key
F5	F5 function key
F6	F6 function key
F7	F7 function key
F8	F8 function key
F9	F9 function key
F10	F10 function key
F11	F11 function key
F12	F12 function key
F13	F13 function key
F14	F14 function key
F15	F15 function key
Alpha0	The '0' key on the top of the alphanumeric keyboard.
Alpha1	The '1' key on the top of the alphanumeric keyboard.
Alpha2	The '2' key on the top of the alphanumeric keyboard.
Alpha3	The '3' key on the top of the alphanumeric keyboard.
Alpha4	The '4' key on the top of the alphanumeric keyboard.
Alpha5	The '5' key on the top of the alphanumeric keyboard.
Alpha6	The '6' key on the top of the alphanumeric keyboard.
Alpha7	The '7' key on the top of the alphanumeric keyboard.
Alpha8	The '8' key on the top of the alphanumeric keyboard.
Alpha9	The '9' key on the top of the alphanumeric keyboard.
Exclaim	Exclaim key
DoubleQuote	Double quote key
Hash	Hash key
Dollar	Dollar sign key
Ampersand	Ampersand key
Quote	Quote key
LeftParen	Left Parent key
RightParen	Right Parent key
Asterisk	Asterisk key
Plus	Plus key
Comma	Comma ',' key
Minus	Minus '-' key
Period	Period '.' key
Slash	Slash '/' key
Colon	Colon ',' key
Semicolon	Semicolon ';' key
Less	Less '<' key
Equals	Equals '=' key
Greater	Greater '>' key
Question	Question mark '?' key
At	At key
LeftBracket	Left bracket key
Backslash	Backslash key
RightBracket	Backslash key
Caret	Caret key
Underscore	Underscore '_' key
BackQuote	Back quote key
A	'a' key
B	'b' key

C	'c' key
D	'd' key
E	'e' key
F	'f' key
G	'g' key
H	'h' key
I	'i' key
J	'j' key
K	'k' key
L	'l' key
M	'm' key
N	'n' key
O	'o' key
P	'p' key
Q	'q' key
R	'r' key
S	's' key
T	't' key
U	'u' key
V	'v' key
W	'w' key
X	'x' key
Y	'y' key
Z	'z' key
Numlock	Numlock key
CapsLock	Capslock key
ScrollLock	Scroll lock key
RightShift	Right shift key
LeftShift	Left shift key
RightControl	Right Control key
LeftControl	Left Control key
RightAlt	Right Alt key
LeftAlt	Left Alt key
LeftApple	Left Apple key
LeftWindows	Left Windows key
RightApple	Right Apple key
RightWindows	Right Windows key
AltGr	Alt Gr key
Help	Help key
Print	Print key
SysReq	Sys Req key
Break	Break key
Menu	Menu key
Mouse0	First (primary) mouse button
Mouse1	Second (secondary) mouse button
Mouse2	Third mouse button
Mouse3	Fourth mouse button
Mouse4	Fifth mouse button
Mouse5	Sixth mouse button
Mouse6	Seventh mouse button
JoystickButton0	Button 0 on any joystick
JoystickButton1	Button 1 on any joystick
JoystickButton2	Button 2 on any joystick
JoystickButton3	Button 3 on any joystick
JoystickButton4	Button 4 on any joystick
JoystickButton5	Button 5 on any joystick
JoystickButton6	Button 6 on any joystick
JoystickButton7	Button 7 on any joystick
JoystickButton8	Button 8 on any joystick
JoystickButton9	Button 9 on any joystick

JoystickButton10	Button 10 on any joystick
JoystickButton11	Button 11 on any joystick
JoystickButton12	Button 12 on any joystick
JoystickButton13	Button 13 on any joystick
JoystickButton14	Button 14 on any joystick
JoystickButton15	Button 15 on any joystick
JoystickButton16	Button 16 on any joystick
JoystickButton17	Button 17 on any joystick
JoystickButton18	Button 18 on any joystick
JoystickButton19	Button 19 on any joystick
Joystick1Button0	Button 0 on first joystick
Joystick1Button1	Button 1 on first joystick
Joystick1Button2	Button 2 on first joystick
Joystick1Button3	Button 3 on first joystick
Joystick1Button4	Button 4 on first joystick
Joystick1Button5	Button 5 on first joystick
Joystick1Button6	Button 6 on first joystick
Joystick1Button7	Button 7 on first joystick
Joystick1Button8	Button 8 on first joystick
Joystick1Button9	Button 9 on first joystick
Joystick1Button10	Button 10 on first joystick
Joystick1Button11	Button 11 on first joystick
Joystick1Button12	Button 12 on first joystick
Joystick1Button13	Button 13 on first joystick
Joystick1Button14	Button 14 on first joystick
Joystick1Button15	Button 15 on first joystick
Joystick1Button16	Button 16 on first joystick
Joystick1Button17	Button 17 on first joystick
Joystick1Button18	Button 18 on first joystick
Joystick1Button19	Button 19 on first joystick
Joystick2Button0	Button 0 on second joystick
Joystick2Button1	Button 1 on second joystick
Joystick2Button2	Button 2 on second joystick
Joystick2Button3	Button 3 on second joystick
Joystick2Button4	Button 4 on second joystick
Joystick2Button5	Button 5 on second joystick
Joystick2Button6	Button 6 on second joystick
Joystick2Button7	Button 7 on second joystick
Joystick2Button8	Button 8 on second joystick
Joystick2Button9	Button 9 on second joystick
Joystick2Button10	Button 10 on second joystick
Joystick2Button11	Button 11 on second joystick
Joystick2Button12	Button 12 on second joystick
Joystick2Button13	Button 13 on second joystick
Joystick2Button14	Button 14 on second joystick
Joystick2Button15	Button 15 on second joystick
Joystick2Button16	Button 16 on second joystick
Joystick2Button17	Button 17 on second joystick
Joystick2Button18	Button 18 on second joystick
Joystick2Button19	Button 19 on second joystick
Joystick3Button0	Button 0 on third joystick
Joystick3Button1	Button 1 on third joystick
Joystick3Button2	Button 2 on third joystick
Joystick3Button3	Button 3 on third joystick
Joystick3Button4	Button 4 on third joystick
Joystick3Button5	Button 5 on third joystick
Joystick3Button6	Button 6 on third joystick
Joystick3Button7	Button 7 on third joystick
Joystick3Button8	Button 8 on third joystick
Joystick3Button9	Button 9 on third joystick

Joystick3Button10	Button 10 on third joystick
Joystick3Button11	Button 11 on third joystick
Joystick3Button12	Button 12 on third joystick
Joystick3Button13	Button 13 on third joystick
Joystick3Button14	Button 14 on third joystick
Joystick3Button15	Button 15 on third joystick
Joystick3Button16	Button 16 on third joystick
Joystick3Button17	Button 17 on third joystick
Joystick3Button18	Button 18 on third joystick
Joystick3Button19	Button 19 on third joystick

GetKeyDown:

static function GetKeyDown (name : String) : boolean

Devuelve true durante el *frame* en que el usuario empieza a presionar la tecla identificada como `name`. Recordemos llamarla dentro de la función `Update`, ya que resetea su estado cada *frame*. No devuelve true hasta que el usuario suelta y luego aprieta la tecla de nuevo (tal como hace por ejemplo `GetButtonDown` con respecto a `GetButton`).

static function GetKeyDown (key : KeyCode) : Boolean

Devuelve true durante el *frame* en que el jugador empieza a presionar la tecla identificada por la `key` de tipo enumeración `KeyCode`, que vimos en el capítulo anterior.

GetKeyUp:

static function GetKeyUp (name : String) : boolean

static function GetKeyUp (key : KeyCode) : boolean

Devuelve true durante el *frame* en que el jugador libera la tecla identificada por `name`.

GetJoystickNames:

static function GetJoystickNames () : string[]

Devuelve un array de strings describiendo los *joysticks* conectados. Esto puede ser útil en una configuración de entradas de pantalla de usuario. Así, en lugar de enseñar etiquetas como "joystick 1", puedes mostrar títulos más personalizados.

GetMouseButton:

static function GetMouseButton (button : int) : boolean

Devuelve true si el botón indicado del ratón es apretado. La variable `button` es un int que representa 0 para el botón izquierdo, 1 para el derecho y 2 para el central.

Por poner un ejemplo muy simple:

```
function Update() {
```

```
  if(Input.GetMouseButton(0))
  Debug.Log("presionado botón izquierdo.");

  if(Input.GetMouseButton(1))
  Debug.Log("presionado botón derecho.");

  if(Input.GetMouseButton(2))
  Debug.Log("presionado botón central.");
}
```

GetMouseButtonDown:

static function GetMouseButtonDown (button : int) : boolean

Devuelve true durante el *frame* en que el usuario aprieta el botón del ratón indicado. Debes llamar esta función dentro de `update`, ya que el estado se resetea cada *frame*. No devolverá true hasta que el botón sea liberado y vuelto a pulsar (recordemos de nuevo la diferencia de `GetButtonDown` con respecto a `GetButton`, para aplicarla también aquí).

GetMouseButtonUp:

static function GetMouseButtonUp (button : int) : boolean

Devuelve true durante el *frame* en que el usuario libera el botón del ratón indicado.

ResetInputAxes:

static function ResetInputAxes () : void

Resetea todos los *inputs*, con lo que todos los *axes* y botones retornan a 0. Esto puede ser útil cuando se regenera al jugador y no te interesa conservar ningún *input* que proceda de alguna tecla que pudiera continuar presionando.

GetAccelerationEvent:

static function GetAccelerationEvent (index : int) : AccelerationEvent

Devuelve mediciones de aceleración que ocurrieron durante el último *frame*.

4.9.6.-Tutorial GUI

La clase `GUI`, que representa la interfaz de Unity:

VARIABLES DE CLASE:

skin:

static var skin : GUISkin

La *skin* (que podemos traducir por "piel" o "aspecto") en uso. Cambiando esta variable podemos cambiar el look de nuestra GUI. Si su valor es null, se muestra la *skin* que está por defecto.

Es una instancia de la clase `GUISkin`.

color:

static var color : Color

Color del tintero global de la GUI. Afectará tanto a la parte trasera de los elementos (*background*) como a los colores del texto que escribamos sobre cualquier superficie.

Vamos con el primer ejemplo. Borramos el *script* de la esfera y editamos miPrimerScript:

```
function OnGUI() {
 GUI.color = Color.yellow;
 GUI.Label (Rect (10, 10, 200, 20), "Esto es una etiqueta");
 GUI.Box(Rect(10, 50, 100, 50), "Una caja");
 GUI.Button(Rect(10,110,90,50), "Un botón");
}
```

Salvamos y la arrastramos a PortaScripts. Si pulsamos play observaremos algo como esto:

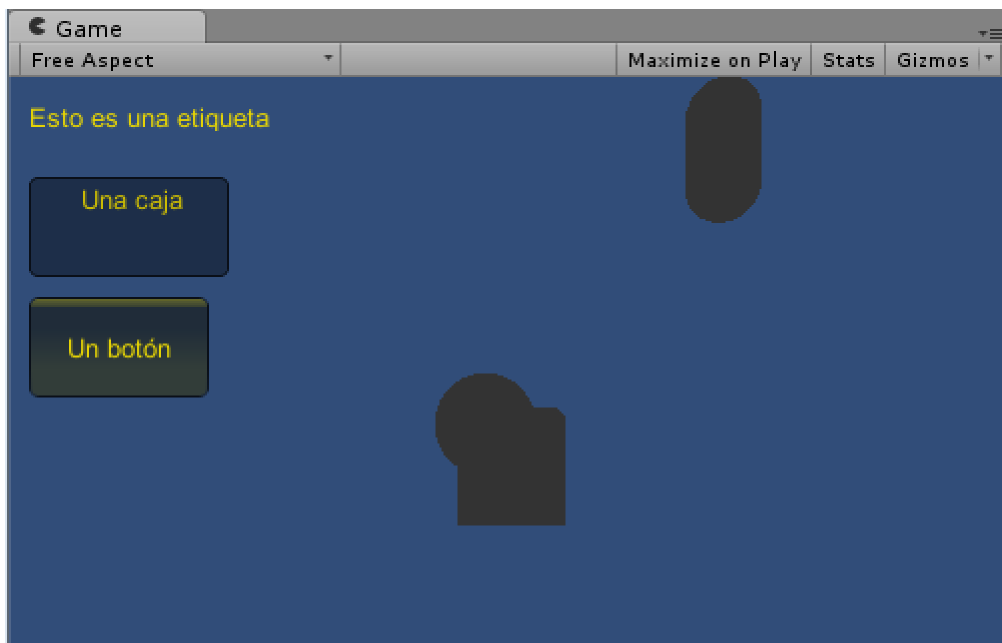


Figura 4.94

Aquí hemos de explicar varias cosas. Empezando por el final, comprobamos en la imagen que el texto de la GUI, sobre las diferentes superficies, es del color que le hemos indicado a la variable `color`. Asimismo, los bordes de elementos como el botón adquieren en su parte posterior (*background*) ese color amarillo.

Esa declaración y las anteriores están contenidas dentro de la función `OnGUI` que renderiza y maneja eventos GUI. Dicho de otra manera, al llamar a esta función activamos un evento GUI, que en este caso asigna un color a la letra y luego crea una etiqueta, una caja y un botón.

backgroundColor:

static var backgroundColor : Color

Color del tintero para todas las partes traseras (*background*) de los elementos renderizados para la GUI.

Dicho de otra manera, esta variable de clase hace parte del trabajo que efectuaba `color`, ya que colorea el *background* pero no el texto.

Reeditamos miPrimerScript como sigue:

```
function OnGUI() {
  GUI.backgroundColor = Color.red;
  GUI.Button(Rect(10,10,70,30), "Mi botón");
}
```

Al darle al play podemos observar que los bordes del botón que hemos creado son de color rojo, color que se acrecienta cuando colocamos el ratón encima. En cambio, el color del texto sigue siendo blanco (el color por defecto).

contentColor:

static var contentColor : Color

Color de tinta para todo el texto renderizado en la GUI. Esta función es la complementaria de la anterior con relación a `color`, ya que no afecta al color del *background*, sino al del texto.

Por ejemplo:

```
function OnGUI() {
  GUI.contentColor = Color.yellow;
  GUI.backgroundColor = Color.red;
  GUI.Button(Rect(10,10,70,30), "Mi botón");
}
```

Vemos aquí claramente las diferencias entre `ContentColor` (amarillo) y `backgroundColor` (rojo).

changed:

static var changed : boolean

Devuelve true si algún control cambia el valor de los datos de entrada de la GUI.

Podemos aprovechar el ejemplo que aparece en el manual de referencia para ilustrarnos. Editamos miPrimerScript como sigue:

```
var miTexto : String = "Cambiame";
```

```
function OnGUI () {
miTexto = GUI.TextField (Rect (10, 10, 200, 20), miTexto, 25);
if (GUI.changed)
Debug.Log("El campo de texto se modificó");
}
```

El contenido del *script* es bastante intuitivo: creamos un campo de texto editable con la función `TextField`, que en breve veremos y si procedemos a cambiar su contenido inicial se nos imprimirá en pantalla un texto avisándonos de dicha modificación.

enabled:

static var enabled : boolean

Habilita/deshabilita la GUI.

Si establecemos esta variable en false se deshabilitan todas las interacciones de la GUI. Todos los controles se dibujarán semitransparentes y no responderán a las entradas del usuario.

tooltip:

static var tooltip : String

Un *tooltip* es esa pequeña nota emergente que aparece a veces con determinada información cuando colocamos un ratón sobre un control o dicho control tiene el foco del teclado.

Vamos con el pertinente ejemplo. Abrimos nuestro *script* y:

```
function OnGUI () {
GUI.Button (Rect (10,10,100,20), GUIContent ("Pulsame", "Este es el
tooltip"));
GUI.Label (Rect (10,40,100,40), GUI.tooltip);
}
```

El *script* funciona de la siguiente manera: Empezamos creando un evento GUI con la función `OnGUI`. Acto seguido creamos un botón con una determinada ubicación y dimensiones y le pasamos como segundo parámetro la función constructora de la clase `GUIContent`, que a su vez admite como parámetros el texto del botón, una imagen (en este caso no) y en su caso el texto del *tooltip* que se deba activar al pasarle el ratón por encima.

Acto seguido hemos de crear propiamente la etiqueta del *tooltip*, indicando su ubicación y dimensiones.

Al darle al play y colocar el ratón sobre el botón, automáticamente nos aparecerá un *tooltip* con el texto indicado, que desaparecerá al retirar el ratón de dicho control.

depth:

static var depth : int

El orden de profundidad que tendrá cada actividad GUI en ejecución. Quiere esto decir que cuando tengamos varios *scripts* ejecutándose simultáneamente, los elementos GUI que tengan valores de profundidad más bajos aparecerán en la pantalla encima de los que lo tengan más altos

FUNCIONES DE CLASE:

label:

static function Label (position : Rect, text : String) : void

static function Label (position : Rect, image : Texture) : void

static function Label (position : Rect, content : GUIContent) : void

static function Label (position : Rect, text : String, style : GUIStyle) : void

static function Label (position : Rect, image : Texture, style : GUIStyle) : void

static function Label (position : Rect, content : GUIContent, style : GUIStyle) : void

Crea una etiqueta (*label*) de texto o de imagen en la pantalla. Como podemos observar, esta función tiene varios prototipos, permitiéndonos pasarle distintos parámetros en base a la necesidad y datos que tengamos en cada momento. Dichos parámetros serían:

position: Rectángulo en la pantalla a usar para la etiqueta.
text: Texto a mostrar en la etiqueta.
image: Textura/imagen a mostrar en la etiqueta.
content: Texto, imagen y *tooltip* para esta etiqueta.
style: El estilo a usar. Si no se indica, se aplicará el estilo para etiquetas que tenga el GUISkin que se esté usando.

En base al significado de estos parámetros, podemos fácilmente deducir la diferencia entre los distintos prototipos de esta función.

Las etiquetas o *labels* en sí no implican ningún tipo de interacción con el usuario, no captan clicks del ratón y son siempre renderizadas en un estilo normal (no por ejemplo como los botones, que aparte del estilo normal tienen otro para cuando se pasa el ratón por encima, otro cuando se presionan, o cuando se activan, etc.).

Pongamos el más sencillo de los ejemplos: mostremos en pantalla el famoso “Hello world”.

```
function OnGUI () {
GUI.Label (Rect (10, 10, 100, 20), "Hello World");
}
```

Como vemos, hemos optado por el primer prototipo de la función de los que mostramos al inicio. Simplemente el rectángulo con la posición (10,10) y tamaño (100,20) del rectángulo donde ubicaremos la etiqueta y como segundo parámetro un *string* con el texto.

Si en lugar de mostrar un texto quisiéramos hacer lo propio con una imagen, modificaríamos la función como sigue:

```
var unaTextura : Texture2D;

function OnGUI () {
GUI.Label (Rect (10, 40, unaTextura.width, unaTextura.height),
unaTextura);
}
```

Salvamos y arrastramos una textura que tengamos a la variable expuesta de este *script* que tenemos vinculado a PortaScripts. En este prototipo de nuevo pasamos como parámetro primero un rectángulo con la posición del rectángulo (10,40) y en lugar de indicar directamente las dimensiones que ha de tener dicho rectángulo, aprovechamos la anchura y altura originales de la imagen arrastrada para no recortarla (aunque por supuesto podemos sentirnos libres de establecer unas dimensiones fijas). El segundo parámetro es donde difiere este prototipo del anterior, ya que en lugar de suministrar un *string* pasamos una textura.

DrawTexture:

static function DrawTexture (position : Rect, image : Texture, scaleMode : ScaleMode = ScaleMode.StretchToFill, alphaBlend : boolean = true, imageAspect : float = 0) : void

Dibuja una textura dentro de un rectángulo. Tiene los siguientes parámetros:

`position`: Rectángulo en la pantalla para dibujar la textura dentro.
`image`: Textura a mostrar.
`scaleMode`: Cómo escalar la imagen cuando la proporción hace que no encaje bien dentro del rectángulo.
`alphaBlend`: Si el canal alfa es mezclado en la imagen (por defecto).
`imageAspect`: Proporción a usar para la imagen fuente. Si vale 0 (por defecto), es usada la proporción de la imagen. Pasad un w/h para la deseada proporción, lo cual permite cambiar la proporción de la imagen original sin cambiar la anchura y altura de píxeles.

El parámetro `scaleMode` es a su vez una enumeración que acepta los siguientes valores:

StretchToFill: Estira la textura para rellenar el rectángulo entero.
ScaleAndCrop: Escala la textura, manteniendo la proporción, hasta cubrir completamente el rectángulo. Si la textura se dibuja en un rectángulo con una proporción diferente, la imagen se recorta.
ScaleToFit: Escala la textura, manteniendo la proporción, hasta encajar completamente dentro del rectángulo.

Vamos ahora a dibujar una textura en la esquina izquierda de la pantalla, textura que se dibujará en una ventana de 60 x 60 píxeles. A la textura original le daremos una proporción de 10 x 1 y la haremos encajar luego en el rectángulo anterior con el `Scalemode.ScaleToFit`, de tal manera que la textura se escalará hasta encajar horizontalmente con el rectángulo, manteniendo la proporción de 10/1.

```
function OnGUI() {
}
GUI.DrawTexture(Rect(10,10,60,60), aTexture, ScaleMode.ScaleToFit,
true, 10.0f);
}
```

Box:

static function Box (position : Rect, text : String) : void
static function Box (position : Rect, image : Texture) : void
static function Box (position : Rect, content : GUIContent) : void
static function Box (position : Rect, text : String, style : GUIStyle) : void
static function Box (position : Rect, image : Texture, style : GUIStyle) : void
static function Box (position : Rect, content : GUIContent, style : GUIStyle) : void

Como su nombre indica, crea un cuadro o caja. Sus diferentes prototipos cuentan con estos parámetros:

`position`: Rectángulo en la pantalla a usar para la caja.
`text`: Texto a mostrar en la caja.
`image`: Textura a mostrar en la caja.
`content`: Texto, imagen y *tooltip* para la caja.
`style`: El estilo a usar. Si no se indica expresamente, el estilo de la caja será el del GUISkin en uso.

Pongamos un ejemplo sencillo:

```
function OnGUI() {
 GUI.Box(Rect(10,20,100,40),"Hola, mundo");
}
```

Button:

static function Button (position : Rect, text : String) : Boolean

static function Button (position : Rect, image : Texture) : Boolean

static function Button (position : Rect, content : GUIContent) : Boolean

static function Button (position : Rect, text : String, style : GUIStyle) : Boolean

static function Button (position : Rect, image : Texture, style : GUIStyle) : Boolean

static function Button (position : Rect, content : GUIContent, style : GUIStyle) : boolean

Función que crea un botón que, al ser pulsado por un usuario, dispara algún tipo de evento.

Una cosa que puede llamar la atención inicialmente y que tiene mucho que ver con la manera un poco atípica de usar esta función, es que retorna un booleano que es establecido en true cuando el usuario hace click sobre el botón. De esta suerte, la función devolverá false hasta que se pulse el botón, lo que implica en la práctica que esta función se suele usar tras una declaración `if`, como enseguida veremos.

Antes detengámonos un momento en los parámetros:

`position`: Rectángulo en la pantalla a usar para el botón.
`text`: Texto a mostrar en el botón.
`image`: Textura/imagen a mostrar en el botón.
`content`: Texto, imagen y *tooltip* para este botón.
`style`: El estilo a usar. Si no se indica, se aplicará el estilo para botones que tenga el GUISkin que se esté usando.

Y ahora es el momento de entender con un ejemplo la peculiar manera de usar la función `button`:

```
var unaTextura : Texture;

function OnGUI() {
 if (!unaTextura) {
 Debug.LogError("Asigne por favor una textura en el inspector");
 return;
 }
 if (GUI.Button(Rect(10,10,50,50),unaTextura))
 Debug.Log("Has hecho click en el botón que tiene una imagen");
 if (GUI.Button(Rect(10,70,50,30),"Click"))
 Debug.Log("Has hecho click en el botón con texto");
}
```

Vamos por partes. Salvamos el *script* y sin arrastrar ninguna textura a la variable expuesta que nos debería aparecer en el inspector al tener PortaScripts seleccionado, pulsamos play. Nos aparecerá un mensaje de error avisándonos de que debemos arrastrar dicha textura. Es esta una variante de `Debug.Log` llamada `Debug.LogError`, que hace que el mensaje aparezca en rojo.

Detenemos el reproductor, arrastramos la textura y volvemos a pulsar play. Observaremos que nos aparecen dos botones, uno con la textura arrastrada por nosotros, otros con el texto indicado y que al pulsarlos aparecen en pantalla sendos textos.

Lo que personalmente me parece más interesante de todo esto es la manera de utilizar la función:

```
if (GUI.Button(Rect(10,70,50,30),"Click"))
```

Pensemos que con esta declaración estamos haciendo dos cosas: primero, al pasarla como condición de `if` nos aseguramos de que una vez el *user* clicke el botón y por ende la función `Button` devuelva true, se ejecutará las declaraciones que se vean afectadas por ese `if`. Pero, además, previo a devolver true o false, Unity ejecuta la función, y por consiguiente se renderiza el botón en la pantalla.

RepeatButton:

static function RepeatButton (position : Rect, text : String) : boolean

```
static function RepeatButton (position : Rect, image : Texture) : Boolean

static function RepeatButton (position : Rect, content : GUIContent) : Boolean

static function RepeatButton (position : Rect, text : String, style : GUIStyle) : Boolean

static function RepeatButton (position : Rect, image : Texture, style : GUIStyle) : Boolean

static function RepeatButton (position : Rect, content : GUIContent, style : GUIStyle) : boolean
```

Función que crea un botón que está activo mientras el usuario lo presiona.

TextField:

```
static function TextField (position : Rect, text : String) : String

static function TextField (position : Rect, text : String, maxLength : int) : String

static function TextField (position : Rect, text : String, style : GUIStyle) : String

static function TextField (position : Rect, text : String, maxLength : int, style : GUIStyle) : String
```

Crea un campo de texto de una línea donde el usuario puede editar un *string*. Devuelve un *string* con el texto editado.

Tiene estos parámetros:

`position`: Rectángulo en la pantalla a usar para el campo de texto.
`text`: Texto a editar. El valor de retorno debe ser reasignado de vuelta al *string* tal como se enseña en el próximo ejemplo.
`maxLength`: La longitud máxima del *string*. Si no se indica, el usuario puede escribir sin ningún tipo de límite.
`style`: El estilo a usar. Si no se indica, se aplicará el estilo para `textField` que tenga el GUISkin en uso.

Esta función es importante porque permite una mayor interactividad con el usuario que meramente pulsar determinamos botones. Por ello es importante que el texto que dicho usuario introduce se almacene debidamente. Tal como indica el manual de referencia al hablar del parámetro `text`, la solución ideal es inicializar una variable *string* con el valor que le damos por defecto y reutilizar dicha variable para posteriormente contener el *string* ya modificado y devuelto por la función. En definitiva:

```
var mensaje : String = "Escribe algo";
```

```
function OnGUI () {
 mensaje = GUI.TextField (Rect (10, 10, 200, 20), mensaje, 25);
 Debug.Log(mensaje);
}
```

Como vemos, la idea es que `mensaje` sirva tanto para contener el *string* inicial como el modificado. Le he añadido a continuación un `Debug.Log` para que comprobemos en tiempo real cómo va cambiando el contenido de la variable.

PasswordField:

static function PasswordField (position : Rect, password : String, maskChar : char) : String

static function PasswordField (position : Rect, password : String, maskChar : char, maxLength : int) : String

static function PasswordField (position : Rect, password : String, maskChar : char, style : GUIStyle) : String

static function PasswordField (position : Rect, password : String, maskChar : char, maxLength : int, style : GUIStyle) : String

Crea un campo de texto donde el usuario puede introducir una contraseña. Devuelve un *string* con la contraseña editada.

Cuenta con los siguientes parámetros:

`position`: Rectángulo en la pantalla a usar para el campo de texto.
`password`: Contraseña a editar. El valor de retorno de esta función debe ser reasignado al *string* que contenía el valor original.
`maskChar`: El carácter con el que queremos enmascarar la contraseña.
`maxLength`: La longitud máxima del *string*. Si no se indica, el usuario no tendrá límite para escribir.
`style`: El estilo a usar. Si no se indica, se usará el estilo para `textfield` que tenga el GUISkin que se esté usando.

Podemos apreciar que en definitiva esta función es como la anterior, con la salvedad de que en la presente le añadimos un carácter (tradicionalmente un asterisco) que queremos que aparezca en pantalla cuando el usuario teclee su contraseña.

```
var miPin : String = "";

function OnGUI () {
 miPin = GUI.PasswordField (Rect (10, 10, 200, 20), miPin, "*"[0],
25);
}
```

TextArea:

static function TextArea (position : Rect, text : String) : String

static function TextArea (position : Rect, text : String, maxLength : int) : String

static function TextArea (position : Rect, text : String, style : GUIStyle) : String

static function TextArea (position : Rect, text : String, maxLength : int, style : GUIStyle) : String

Crea un área de texto de varias líneas donde el usuario puede editar un *string*. Devuelve el *string* editado.

SetNextControlName:

static function SetNextControlName (name : String) : void

Función que establece el nombre del siguiente control. Esto hace que el siguiente control sea registrado con un nombre dado.

GetNameOfFocusedControl:

static function GetNameOfFocusedControl () : String

Obtiene el nombre del control que tiene el foco. El nombre de los controles es asignado usando `SetNextControlName`. Cuando un control tiene el foco, esta función devuelve su nombre. `GetNameOfFocusedControl` funciona especialmente bien cuando tratamos con ventanas para loguearse.

FocusControl:

static function FocusControl (name : String) : void

Mueve el foco del teclado a un control nombrado.

Toggle:

static function Toggle (position : Rect, value : boolean, text : String) : Boolean

static function Toggle (position : Rect, value : boolean, image : Texture) : Boolean

static function Toggle (position : Rect, value : boolean, content : GUIContent) : Boolean

static function Toggle (position : Rect, value : boolean, text : String, style : GUIStyle) : Boolean

static function Toggle (position : Rect, value : boolean, image : Texture, style : GUIStyle) : Boolean

static function Toggle (position : Rect, value : boolean, content : GUIContent, style : GUIStyle) : Boolean

Crea un botón de tipo interruptor (on/off). Devuelve el nuevo valor del botón (true/false).

Toolbar:

static function Toolbar (position : Rect, selected : int, texts : string[]) : int

static function Toolbar (position : Rect, selected : int, images : Texture[]) : int

static function Toolbar (position : Rect, selected : int, content : GUIContent[]) : int

static function Toolbar (position : Rect, selected : int, texts : string[], style : GUIStyle) : int

static function Toolbar (position : Rect, selected : int, images : Texture[], style : GUIStyle) : int

static function Toolbar (position : Rect, selected : int, contents : GUIContent[], style : GUIStyle) : int

Función que crea una barra de herramientas. Devuelve -un int- el índice de la *toolbar* seleccionado.

Tiene estos parámetros:

`position`: Rectángulo en la pantalla a usar para la barra de herramientas.
`selected`: El índice del botón seleccionado.
`texts`: Un array de *strings* a enseñar en los botones de la barra.
`images`: Un array de texturas para los botones de la barra de herramientas.
`contents`: Un array de texto, imágenes y *tooltips* para los botones del *toolbar*.
`style`: El estilo a usar. Si no se indica, se usará el estilo para botones que tenga la GUISkin que se esté usando.

Veámoslo con un ejemplo:

```
var toolbarIndice : int = 0;
var toolbarStrings : String[] = ["Toolbar1", "Toolbar2", "Toolbar3"];

function OnGUI () {
 toolbarIndice = GUI.Toolbar (Rect (25, 25, 250, 30), toolbarIndice,
toolbarStrings);
 Debug.Log("El índice pulsado es " + toolbarIndice);
}
```

Observaremos que nos aparece en pantalla al pulsar play una barra con tres botones, estando por defecto activado el primero, que corresponde con el índice cero que le hemos pasado como parámetro. Le hemos añadido la función `Debug.Log` para acreditar que al presionar los distintos botones se le asigna a la variable el índice de cada botón.

SelectionGrid:

static function SelectionGrid (position : Rect, selected : int, texts : string[], xCount : int) : int

static function SelectionGrid (position : Rect, selected : int, images : Texture[], xCount : int) : int

```
static function SelectionGrid (position : Rect, selected : int, content : GUIContent[], xCount : int) : int

static function SelectionGrid (position : Rect, selected : int, texts : string[], xCount : int, style : GUIStyle) : int

static function SelectionGrid (position : Rect, selected : int, images : Texture[], xCount : int, style : GUIStyle) : int

static function SelectionGrid (position : Rect, selected : int, contents : GUIContent[], xCount : int, style : GUIStyle) : int
```

Hace una cuadrícula (*grid*) de botones. Devuelve un int con el índice del botón seleccionado.

Permite los siguientes parámetros:

`position`: Rectángulo de la pantalla a usar para la cuadrícula.
`selected`: El índice del botón seleccionado de la cuadrícula.
`texts`: Un array de *strings* que mostrar en los botones de la cuadrícula.
`images`: Un array de texturas en los botones de la cuadrícula.
`contents`: Un array de texto, imágenes y *tooltips* para los botones de la cuadrícula
`xCount`: Cuántos elementos caben en la dirección horizontal. Los controles serán escalados para encajar a menos que el estilo elegido para usar sea fixedWidth.
`style`: El estilo a usar. Si no se indica, se usa el estilo de botón marcado por el GUISkin que se esté usando.

Veamos un breve ejemplo:

```
var selGridInt : int = 0;
var selStrings : String[] = ["Grid 1", "Grid 2", "Grid 3", "Grid 4"];

function OnGUI () {
selGridInt = GUI.SelectionGrid (Rect (25, 25, 100, 30), selGridInt,
selStrings, 2);
}
```

HorizontalSlider:

```
static function HorizontalSlider (position : Rect, value : float, leftValue : float, rightValue : float) : float

static function HorizontalSlider (position : Rect, value : float, leftValue : float, rightValue : float, slider : GUIStyle, thumb : GUIStyle) : float
```

Crea una barra de desplazamiento horizontal que el usuario puede arrastrar para cambiar un valor entre un mínimo y un máximo. Devuelve un *float* con el valor que ha sido elegido por el usuario.

Parámetros:

`position`: Rectángulo en la pantalla a usar para la barra.
`value`: El valor que muestra la barra. Esto determina la posición del deslizable móvil.
`leftValue`: El valor del extremo izquierdo de la barra.

`rightValue`: El valor del extremo derecho de la barra.
`slider`: El GUIStyle a usar para mostrar el área de arrastre. Si no se utiliza, se usará el estilo de horizontalSlider que tenga por defecto el GUISkin que se esté usando.
`thumb`: El GUIStyle a usar para mostrar el deslizable móvil. Si no se usa, se usará el estilo de horizontalSliderThumb style que tenga por defecto el GUISkin que se esté usando.

Y por último el ejemplo:

```
var valorDelSlider : float = 0.0;

function OnGUI () {
 valorDelSlider = GUI.HorizontalSlider (Rect (25, 25, 100, 30),
valorDelSlider,
 0.0,  10.0);
}
```

VerticalSlider:

static function VerticalSlider (position : Rect, value : float, topValue : float, bottomValue : float) : float

static function VerticalSlider (position : Rect, value : float, topValue : float, bottomValue : float, slider : GUIStyle, thumb : GUIStyle) : float

Crea una barra de deslizamiento vertical que el usuario puede arrastrar para cambiar un valor entre un mínimo y un máximo. Devuelve un float con el valor que ha sido escogido por el usuario.

Tiene los siguientes parámetros:

`position`: Rectángulo en la pantalla a usar para la barra.
`value`: El valor que muestra la barra. Esto determina la posición del deslizable móvil.
`topValue`: El valor en lo alto de la barra
`bottomValue`: El valor en lo bajo de la barra
`slider`: El GUIStyle a usar para mostrar el área de arrastre. Si no se utiliza, se usará el estilo de verticalSlider que tenga por defecto el GUISkin que se esté usando.
`thumb`: El GUIStyle a usar para mostrar el deslizable móvil. Si no se usa, se usará el estilo de verticalSliderThumb que tenga por defecto el GUISkin que se esté usando.

HorizontalScrollbar:

static function HorizontalScrollbar (position : Rect, value : float, size : float, leftValue : float, rightValue : float) : float

static function HorizontalScrollbar (position : Rect, value : float, size : float, leftValue : float, rightValue : float, style : GUIStyle) : float

Crea una barra de desplazamiento (*scrollbar*) horizontal. Un *scrollbar* es lo que usamos para desplazarnos por un documento. Por norma general en lugar de *scrollbar* usaremos *scrollviews*.

Devuelve un float con el valor modificado. Este puede ser cambiado por el usuario arrastrando el *scrollbar* o clickando en las flechas de los extremos.

Un breve ejemplo:

```
var valorDeBarra : float;

function OnGUI () {
 valorDeBarra = GUI.HorizontalScrollbar (Rect (25, 25, 100, 30),
valorDeBarra,
 1.0, 0.0, 10.0);
}
```

VerticalScrollbar:

static function VerticalScrollbar (position : Rect, value : float, size : float, topValue : float, bottomValue : float, style : GUIStyle) : float

Crea una barra de desplazamiento (*scrollbar*) vertical.

BeginGroup:

static function BeginGroup (position : Rect) : void

static function BeginGroup (position : Rect, text : String) : void

static function BeginGroup (position : Rect, image : Texture) : void

static function BeginGroup (position : Rect, content : GUIContent) : void

static function BeginGroup (position : Rect, style : GUIStyle) : void

static function BeginGroup (position : Rect, text : String, style : GUIStyle) : void

static function BeginGroup (position : Rect, image : Texture, style : GUIStyle) : void

static function BeginGroup (position : Rect, content : GUIContent, style : GUIStyle) : void

Comienza un grupo. Esta función debe emparejarse con una llamada a `EndGroup`.

Cuando comenzamos un grupo, el sistema de coordenadas para los controles GUI será tal que coincidirá la coordenada (0,0) con la esquina superior izquierda del grupo. Los grupos pueden ser anidados, estando lo hijos agrupados respecto de sus padres.

Esto es muy útil cuando movemos un montón de elementos GUI a lo largo de la pantalla. Un caso de uso común es diseñar nuestros menús para que encajen en un específico tamaño de pantalla, centrando la GUI en pantallas más amplias.

Los distintos prototipos de función usan estos parámetros:

`position`: Rectángulo en la pantalla a usar para el grupo.
`text`: Texto a mostrar en el grupo.
`image`: Textura a mostrar en el grupo.
`content`: Texto, imagen y *tooltip* para este grupo. Si el parámetro es proporcionado, cualquier click de ratón es capturado por el grupo y si no se proporciona, no se renderiza ningún *background* y los clicks del ratón son renderizados.
`style`: El estilo a usar para el *background*.

Veremos muy claramente la funcionalidad de este método con un ejemplo:

```
function OnGUI () {

 GUI.BeginGroup (new Rect (Screen.width / 2 -200 , Screen.height / 2 -
150, 400,
 300));

 GUI.Box (new Rect (0,0,400,300), "Este cuadrado está ahora centrado,
y dentro del
 mismo podemos colocar nuestro menú");

 GUI.EndGroup ();
}
```

Expliquemos paso a paso en qué consiste lo que hemos hecho. Para empezar usamos la función `BeginGroup` para crear un grupo en un rectángulo que se iniciará en el centro de la pantalla. Si lo ubicáramos en width/2 el rectángulo quedaría desplazado, pues no se estaría teniendo en cuenta en este caso la anchura del propio rectángulo. De esta manera, le hemos de restar al centro de la pantalla la mitad de la anchura del rectángulo, asegurándonos así que queda justo en el centro. Hacemos lo propio con la altura.

Una vez ya tenemos definido para el grupo un rectángulo centrado con unas dimensiones de 400 X 300, ahora para los controles dentro de dicho grupo la esquina superior izquierda del rectángulo pasa a ser la coordenada 0,0. Así, cuando a continuación invocamos una caja con un texto y la ubicamos en las coordenadas 0,0, esta se nos viene a colocar al inicio del rectángulo del grupo.

No hemos de olvidarnos, por último, que si usamos una función `BeginGroup` hemos de usar cuando acabemos de diseñar el grupo una función `EndGroup` obligatoriamente, para indicarle a Unity que las instrucciones referidas al grupo ya se han acabado.

EndGroup:

static function EndGroup () : void

Finaliza un grupo.

BeginScrollView:

static function BeginScrollView (position : Rect, scrollPosition : Vector2, viewRect : Rect) : Vector2

static function BeginScrollView (position : Rect, scrollPosition : Vector2, viewRect : Rect, alwaysShowHorizontal : boolean, alwaysShowVertical : boolean) : Vector2

static function BeginScrollView (position : Rect, scrollPosition : Vector2, viewRect : Rect, horizontalScrollbar : GUIStyle, verticalScrollbar : GUIStyle) : Vector2

static function BeginScrollView (position : Rect, scrollPosition : Vector2, viewRect : Rect, alwaysShowHorizontal : boolean, alwaysShowVertical : boolean, horizontalScrollbar : GUIStyle, verticalScrollbar : GUIStyle) : Vector2

Inicia una vista de desplazamiento (*scrollview*) dentro de la GUI. Un *scrollview* nos permite poner un área más pequeña en la pantalla dentro de un área mucho mayor, usando barras de desplazamiento (*scrollbars*) a los lados.

Esta función devuelve la posición del *scroll* modificada. Al igual que con otras variables, se recomienda reintroducir en la variable que se le pasa a la función los datos nuevos que esta devuelve.

Tiene estos parámetros:

`position`: Rectángulo en la pantalla a usar para el *ScrollView*.
`scrollPosition`: La distancia en píxeles que la vista es desplazada en las direcciones X e Y.
`viewRect`: El rectángulo usado dentro del *scrollview*.
`alwayShowHorizontal`: Parámetro opcional para enseñar siempre el *scrollbar* horizontal. Si lo establecemos en falso o no lo aportamos a la función, sólo se enseñará cuando `clientRect` sea más ancho que la posición.
`alwayShowVertical`: Lo mismo para el *scrollbar* vertical.
`horizontalScrollbar`: GUIStyle opcional a usar por el *scrollbar* horizontal. Si no se aporta, se usará el estilo horizontalScrollbar del GUISkin que se esté usando.
`verticalScrollbar`: Lo mismo para el *scrollbar* vertical

EndScrollView:

static function EndScrollView () : void

Acaba un *scrollview* iniciado con una llamada a `BeginScrollView`.

ScrollTo:

static function ScrollTo (position : Rect) : void

Desplaza todos los *scrollviews* incluidos para tratar de hacer visible una determinada posición.

Window:

static function Window (id : int, clientRect : Rect, func : WindowFunction, text : String) : Rect

```
static function Window (id : int, clientRect : Rect, func :
WindowFunction, image : Texture) : Rect

static function Window (id : int, clientRect : Rect, func :
WindowFunction, content : GUIContent) : Rect

static function Window (id : int, clientRect : Rect, func :
WindowFunction, text : String, style : GUIStyle) : Rect

static function Window (id : int, clientRect : Rect, func :
WindowFunction, image : Texture, style : GUIStyle) : Rect

static function Window (id : int, clientRect : Rect, func :
WindowFunction, title : GUIContent, style : GUIStyle) : Rect
```

Crea una ventana emergente y devuelve el rectángulo en el que dicha ventana se ubica.

Las ventanas flotan sobre los controles GUI normales, *Windows float above normal GUI controls* y pueden ser opcionalmente arrastrados por los usuarios finales. A diferencia de otros controles, necesitamos pasarles una función separada para los controles GUI para colocarlos dentro de la ventana.

Nota: Si usamos `GUILayout` para colocar nuestros componentes dentro de la ventana, debemos usar `GUILayout.Window`.

Parámetros:

`id`: Una ID única a usar para cada ventana.
`clientRect`: Rectángulo en la pantalla a usar por el grupo.
`func`: La función que crea el GUI dentro de la ventana. Esta función debe tomar un parámetro, que será la ID de la ventana para la que se está creando la GUI.
`text`: Texto a mostrar como título para la ventana.
`image`: Textura que muestra una imagen en la barra del título.
`content`: Texto, imagen y *tooltip* para esta ventana.
`style`: Un estilo opcional a usar por la ventana. Si no se aporta, se usará el estilo de ventana del GUISkin corriente.

Y vamos con un ejemplo:

```
var windowRect : Rect = Rect (20, 20, 120, 50);

function OnGUI () {
  windowRect = GUI.Window (0, windowRect, CreaMiVentana, "Mi
ventana");
}

function CreaMiVentana (windowID : int) {
  if (GUI.Button (Rect (10,20,100,20), "Hola mundo"))
  print ("Recibí un click");
}
```

Creamos una ventana con ID 0, que ubicamos en un rectángulo cuyas coordenadas y dimensiones están almacenadas en una variable, variable en la cual almacenaremos el rectángulo

retornado por la función. Como tercer parámetro le pasamos una función que es la que crea los controles que irán dentro de la ventana y por último el título de dicha ventana.

La función que crea los controles toma como parámetro a su vez el primer parámetro de `GUI.Window`, que es la id de la ventana, y en este caso meramente creamos un botón con un texto, que al ser pulsado imprime un mensaje en pantalla.

DragWindow:

static function DragWindow (position : Rect) : void

Crea una ventana que puede arrastrarse. Si llamamos a esta función dentro del código de la ventana, automáticamente esta podrá arrastrarse.

Le hemos de pasar a la función un parámetro que indica la parte de la ventana que puede ser arrastrada, dando un rectángulo que recorta la ventana original.

Para constatar esto, sólo tenemos que añadir esta línea a la función `CreaMiVentana` del ejemplo anterior:

```
GUI.DragWindow (Rect (0,0, 100, 20));
```

Pensemos que 0,0 viene referido a la ventana emergente, no a las coordenadas generales.

static function DragWindow () : void

Esta función tiene un segundo prototipo que no requiere parámetros. Si queremos que nuestra ventana pueda ser arrastrada desde cualquier parte del *background* de la misma, es preferible utilizar esta forma de la función y colocarla al final de las funciones de la ventana.

Así, si modificamos esta parte del *script*:

```
function CreaMiVentana (windowID : int) {
  if (GUI.Button (Rect (10,20,100,20), "Hola mundo"))
  print ("Recibí un click");
  GUI.DragWindow ();
}
```

podremos arrastrar nuestra ventana emergente desde cualquier punto de esta.

BringWindowToFront:

static function BringWindowToFront (windowID : int) : void

Trae una ventana determinada al frente del resto de ventanas flotantes. Tiene como único parámetro la ID de la ventana que queremos poner en primer plano.

BringWindowToBack:

static function BringWindowToBack (windowID : int) : void

Coloca una ventana determinada al fondo de las ventanas flotantes.

FocusWindow:

static function FocusWindow (windowID : int) : void

Hace que una ventana se convierta en la ventana activa. Se le pasa como parámetro la ID de dicha ventana.

UnFocusWindow:

static function UnfocusWindow () : void

Quita el foco de todas las ventanas.

4.9.7.-Tutorial Object

Es sabido que en POO las clases base de las que heredan las demás suelen tener una funcionalidad bastante limitada, cuando no inexistente, siendo su función más la de punto de partida de las que habrán de heredarla que la de instanciar objetos de dicha clase. La clase `Object` no parece ser una excepción a esta regla.

Figura 4.95 Clase `Object`

La clase `Object` consta de dos variables, dos funciones y nueve funciones de clase (FC). Las funciones de clase se diferencian de las funciones standard en que aquéllas no precisan ser llamadas por una instancia u objeto de la clase, perteneciendo -como su nombre indica- a la clase y no a las instancias de dicha clase.

VARIABLES:

name:

```
var name : String
```

Hace referencia al nombre del objeto y comparten ese nombre tanto el objeto en sí como los componentes de dicho objeto. Esta variable nos sirve tanto para cambiar el nombre a nuestro objeto...

```
name = "Cualquier nombre";
```

...como para obtener el nombre de un objeto.

Probemos esta segunda opción con un ejemplo muy sencillo. Abrimos Unity y en la vista del proyecto, hagamos doble click en el *script* que llamamos miPrimerScript. Una vez se abre el editor, borramos todo el contenido y:

```
function Update(){
Debug.Log(name);
}
```

`Debug.Log` nos permite imprimir en pantalla el valor de lo que se halle entre paréntesis. Salvamos el *script* (pero no cerramos el editor, solamente lo minimizamos).

Nota: Lo que hemos modificado y guardado es el *script* que se halla en la carpeta de proyecto. No obstante, al hacer eso automáticamente se modifican todas las copias de este *script* que tengamos por ahí.

Vamos a borrar el componente MiPrimerScript de PortaScripts y arrastramos MiPrimerScript al cubo.

Probémoslo. Si le damos al play, observaremos que bajo la ventana game aparece el nombre de nuestro cubo.

Todo bien hasta aquí. No obstante, en la definición del manual de referencia dice que todos los componentes del objeto comparten el mismo nombre. Si echamos un vistazo a nuestro cubo en el inspector, observaremos que entre otros componentes tiene uno llamado *transform*, así que vamos a hacer la prueba.

Sustituimos el "`name`" que habíamos escrito en el editor de *scripts* entre paréntesis por `transform.name`. O sea:

```
Debug.Log(transform.name);
```

Guardamos el *script*, le damos al play: el componente *transform* comparte el nombre "cubo".

hideFlags:

```
var hideFlags : HideFlags
```

Esta variable no parece a priori tener una gran utilidad. Se conseguiría con su manipulación hacer que un determinado objeto desaparezca o bien del inspector, o bien de la jerarquía, o impedir que dicho objeto se pueda editar, o no permitir que un objeto sea salvado con la escena y en consecuencia destruido con la nueva escena que se cargue.

Para tal fin esta variable enumera las diferentes opciones de una de las muchas enumeraciones que hay en Unity. Esta enumeración, de nombre `HideFlags`, tiene las siguientes opciones:

HideInHierarchy: El objeto no aparecerá en la jerarquía.

HideInInspector: El objeto no aparecerá en el inspector.

DontSave: El objeto no será salvado en la escena ni destruido en la nueva que se cargue.

NotEditable: El objeto no será editable en el inspector.

HideAndDontSave: Combinación de HideInHierarchy y DontSave.

Así, si por ejemplo quisiéramos que el cubo desapareciera del inspector mientras se desarrolla el juego, le vincularíamos un *script* parecido a este:

```
var meEvaporo: GameObject;meEvaporo.hideFlags =
HideFlags.HideInInspector;
```

Este *script* lo podríamos arrastrar a la cámara principal y con posterioridad arrastrar nuestro cubo en la variable global `meEvaporo`, que está esperando un *gameObject*. Observaremos que cuando le demos al play, el cubo desaparece del inspector.

¿Por qué `meEvaporo` es de tipo `GameObject`? ¿No debería tratarse de un *Object*, que es la clase que estamos tratando? Pues efectivamente, pero sucede que si sustituimos `GameObject` por `Object` en nuestro *script*, comprobaremos que desaparece del inspector la variable `meEvaporo`, posiblemente porque la clase `Object` está pensada como clase base para dotar de funcionalidad a las que la heredan, que para ser instanciada ella misma.

FUNCIONES:

`ToString`:

`function ToString () : String`

Devuelve un *string* con el nombre del *gameobject* que hace la consulta.

`GetInstanceID`:

`function GetInstanceID () : int`

Unity asigna a cada objeto un identificador único. Esta función devuelve ese identificador.

Vamos a utilizar estas dos funciones en un mismo ejemplo. Anteriormente teníamos vinculado nuestro *script*-para-todo a la cámara principal. Abrimos el editor de *scripts*, borramos todo y tecleamos lo siguiente:

```
var todoSobreMi: GameObject;
Debug.Log("El nombre de este objeto es " + todoSobreMi.ToString() +
" y su id única es " + todoSobreMi.GetInstanceID());
```

El *script* no merece mucha explicación. Asegurémonos de arrastrar el cubo a la variable `todoSobreMi` en el inspector. Al pulsar el play debería mostrarse el nombre e id del cubo.

FUNCIONES DE CLASE

operator bool, == y !=

Estas tres funciones simplemente habilitan la posibilidad de establecer comparaciones de igualdad/desigualdad entre dos objetos o componentes, o (en el caso del operador `bool`) si existe dicho objeto componente y tiene un valor distinto de null.

Por ejemplo:

```
if (rigidbody)
Debug.Log("Este gameobject tiene vinculado un Rigidbody");
```

Que sería lo mismo que haber escrito

```
if (rigidbody != null)
Debug.Log("Este gameobject tiene vinculado un Rigidbody");
```

Instantiate:

```
static function Instantiate (original : Object, position : Vector3,
rotation : Quaternion) : Object
```

Esta función lo que hace es clonar el objeto que le pasamos como primer parámetro y devolver ese clon del objeto original, ubicándolo en posición y rotación determinadas.

Observamos que el parámetro "`position`" es de tipo `Vector3`. Un Vector3 es un punto en el espacio tridimensional. Dicho punto viene dado por las coordenadas en el eje X (derecha-izquierda) Y (arriba-abajo) y Z (delante-detrás). Cada unidad se corresponde a una unidad en Unity (valga la redundancia), que a su vez equivale a un metro. Por lo tanto para que un objeto se desplace un metro a la derecha, escribiríamos en nuestro Vector3 (1,0,0).

Notemos que esto es ni más ni menos que lo que hace de manera más visual las tres variables position del *transform* que aparecen en el inspector.

El otro parámetro de la función, `rotation`, es de tipo `Quaternion`. Un cuaternión se compone de tres números reales y uno imaginario, el cálculo de los cuales establece la rotación de un objeto. Es muy raro que se trabaje directamente con cuaterniones, sino que se hace con funciones que simplifican el trabajo.

Nos vamos a Unity y le quitamos a la cámara principal el *script* que le habíamos colocado anteriormente. Para hacer eso, con la cámara seleccionada en el inspector colocamos el ratón sobre el nombre del *script*, botón derecho->`Remove Component`.

Ahora, para no andar vinculando *scripts* a *gameobjects* que nada tienen que ver con él (como hicimos no hace mucho con la cámara), vamos a vincular nuestro *script* a Portascripts.

Vale, ahora hacemos doble clic sobre nuestro *script* en la vista de Proyecto para abrir el editor de *scripts*, y escribimos:

```
var reproducete : GameObject;

Instantiate(reproducete, Vector3(2.0,0,0), Quaternion.identity);
```

Aquí lo que hemos hecho meramente es dejar una variable global "expuesta" (que pueda ser accesible desde el inspector) y llamar a la función `instantiate`, de tal manera que clonará el *Gameobject* que le arrastremos y lo situará dos unidades/metros a la derecha del objeto original. `Quaternion.identity` meramente significa que el objeto clonado tendrá rotación cero, esto es, su transform rotation estará a 0,0,0 (salvo que el objeto clonado dependa a su vez de otro objeto, en cuyo caso tendrá la misma rotación que el objeto padre).

Salvamos el *script* y lo arrastramos hasta nuestro PortaScripts en la jerarquía. Vemos que en el inspector, con PortaScripts seleccionado, aparece el *script* y la variable expuesta que llamamos `reproducete`. Arrastramos el cubo hasta ella y ya podemos darle al play. Debería aparecernos un segundo cubo, tal que así:

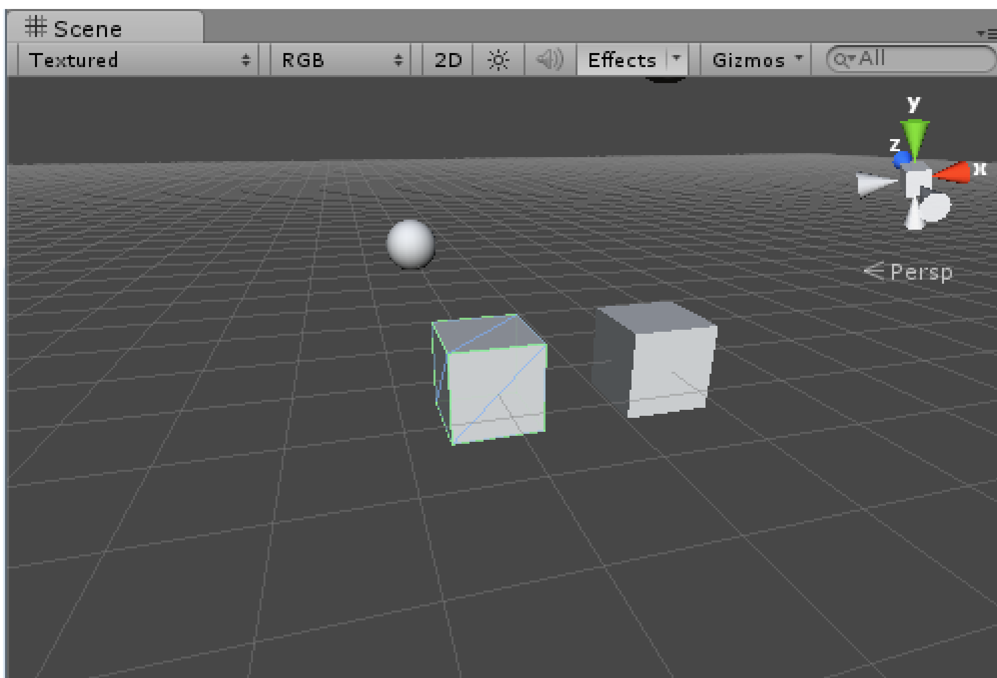


Figura 4.96

Podemos observar alguna cosa más. En la jerarquía aparece -mientras se está reproduciendo la escena- un segundo cubo, el cual Unity nos indica expresamente que es un clon del original. Si lo seleccionamos, podemos ver en su *transform* en el inspector que la variable `x` (el eje derecha/izquierda) no está en el cero, sino en el dos, tal como le habíamos indicado.

Probemos otra cosa. Detenemos la reproducción de la escena. Seleccionamos el cubo original, y le damos a los ejes X e Y de transform rotation los valores 25 y 65 respectivamente. El cubo girará sobre dichos ejes. Démosle al play.

Podemos observar que el cubo clonado se muestra alineado con la cuadrícula global y no con el cubo original. Esto es lo que conseguimos con `Quaternion.identity`.

La función `instantiate` es muy usada para crear proyectiles, partículas en explosiones e incluso AI (inteligencia artificial) para enemigos.

Cabe una segunda forma distinta para la función `Instantiate`, que es esta:

static function Instantiate (original : Object) : Object

Como podemos observar, aquí meramente estamos duplicando el objeto. El objeto clonado se ubicará en el mismo lugar y con la misma rotación que el original. Podemos probarlo en nuestro *script* eliminando el segundo y tercer parámetro. Al darle al play, sabemos que ha aparecido un clon porque así lo indica la jerarquía, pero no podemos distinguirlo porque está justo en el mismo sitio que el original y se están solapando.

Destroy:

static function Destroy (obj : Object, t : float = 0.0F) : void

Como su nombre indica, esta función borra del juego un *gameobject*, un componente o un *asset*. Tiene dos parámetros, siendo el primero el elemento a borrar y el segundo el tiempo en segundos que tardará en borrarlo desde que se llame a la función (si no se indica lo contrario, por defecto el parámetro indica cero segundos, esto es, la destrucción del objeto es automática).

Si en el parámetro `obj` colocamos un Componente, la función sólo eliminará ese componente, haciéndolo desaparecer del *Gameobject* al que pertenezca. Pero si en cambio lo que le pasamos a la función es un *gameobject*, se destruirá tanto el *gameobject* como todos los hijos de ese *gameobject* (esto es, todos los objetos y componentes e inclusive otros *gameobjects* que dependan del eliminado en una relación de parentesco). Para tener un acercamiento intuitivo a esta relación de dependencia, pensemos en un *gameobject* coche que tiene, entre otros, cuatro componentes rueda y un *gameobject* conductor que hemos vinculado al *gameobject* coche para que allá donde se desplace el coche vaya el conductor. Si un misil destruye un componente rueda usando la función `destroy`, sólo se destruirá la rueda. Si en cambio lo que destruye es el *gameobject* coche, se destruirá tanto el vehículo como las ruedas como el conductor.

Vamos a probar la función `Destroy`. Para empezar devolvamos a nuestro cubo original a su rotación original (0,0,0). Vamos a rehacer ahora nuestro sufrido *script*, que tenemos vinculado al cubo. En el editor de *scripts* modificamos nuestro código así:

```
var reproducete : GameObject;
var nasioPaMorir : GameObject;

nasioPaMorir = Instantiate(reproducete, Vector3(2.0,0,0),
Quaternion.identity);
Destroy (nasioPaMorir, 10.0);
```

Lo que hemos hecho es añadir una segunda variable de tipo `GameObject`, que no vamos a inicializar desde el inspector, porque lo que hará será almacenar el cubo clonado que devuelve la función `Instantiate`. Inmediatamente es llamada la función `Destroy`, que borrará el elemento clonado que hemos almacenado en `nasioPaMorir` pasados diez segundos.

Salvamos, le damos al play, contamos diez, y adiós clon.

DestroyImmediate:

static function DestroyImmediate (obj : Object, allowDestroyingAssets : boolean = false) : void

Esta función destruye inmediatamente un objeto, igual que la función anterior. Desde el manual de Unity se nos dice, no obstante, que es preferible usar `Destroy` en lugar de esta función, ya que puede borrar más de lo que deseamos.

FUNCIONES DE CLASE

`FindObjectOfType`

`static function FindObjectOfType (type : Type) : Object`

Esta función devuelve el primer objeto activo que Unity encuentre que sea del tipo que le pasamos como parámetro.

Veámoslo en un ejemplo. Recuperamos una vez más nuestro *script*, borramos todo y tecleamos lo siguiente:

```
var dameAlgo : Object;

dameAlgo = FindObjectOfType(Camera);
Debug.Log("Debería haber encontrado 1 " + dameAlgo);
```

Salvamos, le damos al play y observamos que Unity ha encontrado nuestra cámara principal.

Nos advierte el manual de referencia que esta función puede ser un poco lenta y costosa en términos de rendimiento, por lo que se recomienda no usarla a su vez como parte de una función que se actualice cada *frame* (como por ejemplo la función `Update`).

`FindObjectsOfType`

`static function FindObjectsOfType (type : Type) : Object[]`

Esta función es idéntica a la anterior, con la diferencia de que aquí lo que se devuelve no es el primer objeto activo de tipo `Type`, sino que se devuelven en un array todos los objetos activos cuyo tipo coincide con el que solicitamos.

Veamos la diferencia rehaciendo nuestro *script*:

```
var dameAlgo : Object[];
var chivato : String;

dameAlgo = FindObjectsOfType(GameObject);
chivato = "He encontrado " + dameAlgo.Length + " objetos: ";

for(var contador = 0; contador < dameAlgo.Length; contador++)
chivato += dameAlgo[contador] + "  ";

Debug.Log(chivato);
```

Lo que hemos hecho aquí es lo siguiente: primero hemos reconvertido nuestra variable `dameAlgo` en un array de *Objects*. Declaramos después una variable de tipo `string` para que vaya recopilando toda la información que al final imprimiremos. Inicializamos luego nuestro array con todos los objetos de tipo `GameObject` que Unity encuentre en nuestra escena. A partir de ahí, montamos un bucle `for` para ir añadiendo a nuestro *string* "chivato" el nombre de todos los objetos encontrados. Finalmente, imprimimos. Si todo ha salido bien, el resultado tendría que ser este al darle al play:

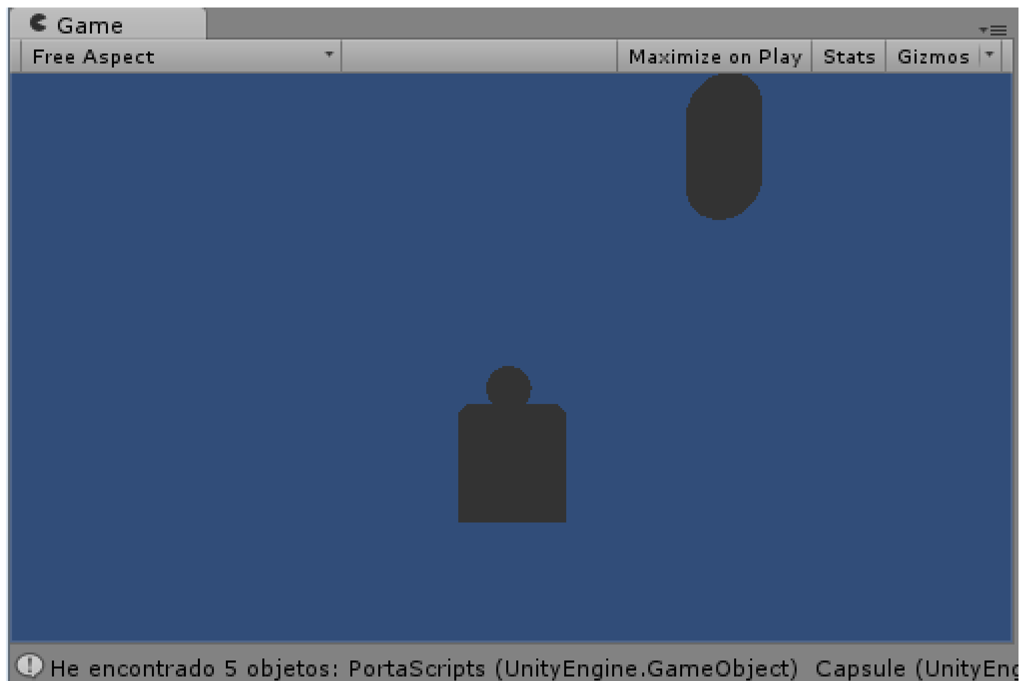


Figura 4.97

DontDestroyOnLoad

```
static function DontDestroyOnLoad (target : Object) : void
```

Con esta función conseguimos que el objeto que colocamos como parámetro no se destruya cuando se cargue una nueva escena. Por defecto, cuando el jugador cambia de escena o nivel, todos los objetos de la escena que abandona se destruyen antes de crear los que corresponden al nuevo nivel. Con esta función, conseguimos que los objetos que queramos pervivan al cambio de escena.

4.10.-*Triggers* y colisiones

4.10.1.-Tutorial *Triggers*

Trigger: Es un disparador o activador de un evento. Esto se produce en el momento en el que se tocan dos cuerpos, en ese momento se activa un *Trigger*.

Para esto necesitamos un objeto con un *collider* y que tenga activada la propiedad `Trigger`. También necesitamos otro objeto que tenga un *rigidbody* y este puede tener o no tener activada la propiedad `trigger`. Estas serían las condiciones mínimas para que se produzcan estos eventos o *triggers*.

Vamos a ver un pequeño ejemplo:

Vamos a crear dos cubos: `Game Object->Create Other->Cube`. Y les nombramos: “Cubo1” y “Cubo2”. El Cubo1 tendrá por defecto un *Box Collider* y le añadiremos un *Rigidbody*: `Component->Physics->Rigidbody`. El Cubo2 simplemente tendría un *Box Collider* con la propiedad `Is trigger` activada.

Vamos a crear un javascript al que llamaremos "Trigger". Los *trigger* manejan tres funciones o eventos: `OnTriggerEnter`, `OnTriggerExit` y `OnTriggerStay`.

`OntriggerEnter`: Se produce en el momento que dos objetos se tocan.

`OnTriggerExit`: Se produce en el momento en que los dos objetos dejan de tener contacto.

`OnTriggerStay`: Se mantiene activo mientras que los objetos estén en contacto.

Vamos a probar la primera función:

```
Function OnTriggerEnter(obj: Collider)

{

       Debug.Log("Enter");

}
```

Con esto lanzamos un pequeño mensaje. Este *script* lo vamos a arrastrar a Cubo2, que es el que tiene la propiedad `Is trigger` activada, aunque se puede también arrastrar a Cubo1, es indiferente. Situamos al Cubo1 sobre Cubo2 a una cierta distancia. Damos al play y veremos como aparece el mensaje "Enter" ya que han entrado en colisión.

Vamos a probar la siguiente función:

```
Function OnTriggerExit (Obj: Collider)

{

       Debug.Log("Exit");

}
```

Damos al play y veremos que aparecen dos mensajes en la consola: el mensaje "Enter" y "Exit", ya que el objeto entra en colisión con el otro y sale por así decirlo de la colisión.

Por último probamos:

```
Function OnTriggerStay(Obj: Collider)

{

       Debug.Log("Stay");

}
```

Damos al play y veremos que en la consola aparecen los mensajes: "Enter", "Stay" que se repite mientras los objetos están en contacto y "Exit".

Estos son los tres eventos que manejan los *Triggers*. Éstos se pueden utilizar, por ejemplo, para activar puertas, botones, etc.

4.10.2.-Tutorial Colisiones

Vamos a ver el funcionamiento de las colisiones. Vamos a crear varios objetos y los haremos hacer colisionar entre ellos. Una vez los objetos choquen detectaremos que objetos colisionan con que objetos.

Vamos a crear un plano que será el suelo: GameObject->Create Other->Plane. Vamos a ponerle medidas a nuestro suelo. En Scale: X=100, Z=100, la Y no la tocamos. En Position: X=0, Y=0, Z=0.

Ahora crearemos una caja: Game Object->Create Other->Cube. En Scale: X=10, Y=10, Z=10. En Position: X=0, Y=25, Z=0.

Por último crearemos un cilindro: Game Object->Create Other->Cylinder. Vamos a Scale: X=5, Y=15, Z=5. En Position: X=0, Y=75, Z=0. En Rotation: X=Y=0, Z=45º.

Vamos a mover la cámara para ver mejor lo que estamos haciendo. La seleccionamos, vamos a Position: X=0, Y=40, Z=-100.

Ahora ponemos una luz: GameObject->CreateOther->PointLight. Vamos a Position: X=0, Y=100, Z=-200. En Range: 300 y subimos la intensidad de la luz a 5.

Ahora cada objeto tiene que tener el control de colisión activado. Seleccionamos el plano y en las propiedades vemos que tiene el *Mesh Collider* activado. Seleccionamos el cubo y vemos que tiene el *Box Collider* activado y para terminar seleccionamos el cilindro y también tiene el *capsule collider* activado. Fijémonos que el *collider* en cada primitiva se llama de una forma.

Ahora necesitamos un controlador para la caja y para el cilindro que se llama *Rigidbody*. Con este controlador pondremos las propiedades físicas como la gravedad o masa entre otros. Seleccionamos el cubo y vamos a Component->Physics->Rigidbody. Seleccionamos el cilindro y vamos a Component->Physics->Rigidbody.

Ejecutamos y vemos qué ocurre. Como vemos, los objetos se caen, chocan entre ellos y del suelo no pasan. Si la reproducción va lenta, vamos al menú edit->Project Settings->Time. En Time Scale, en la vista del inspector, ponemos 3. Al volver a ejecutar veremos que ahora tiene una velocidad bastante aproximada a la realidad.

Ahora vamos a detectar las colisiones. Vamos a Create en la vista Project y añadimos un Javascript y le ponemos de nombre “DetectarColisiones”. Vamos al programa y tecleamos:

```
Function OnCollisionEnter(Colision: Collision)

{

       Debug.Log(Colision.gameobject.name);

}
```

La función OnCollisionEnter sirve única y exclusivamente para que nuestro programa haga algo cuando entramos en una colisión. Vemos que como parámetro tenemos Colision. En esta variable se guarda la información sobre la colisión. Debug.Log nos permite ver mensajes de texto en la consola. Con Colision.gameobject.name imprimimos el nombre del objeto que colisiona con nuestro objeto.

Guardamos el *script*, vamos a Unity, arrastramos el *script* al cilindro. Vamos a menú `Window`, mostramos la consola para ver los mensajes y damos al play. Fijémonos en que cada vez que colisiona el cilindro contra el cubo o el suelo, se imprime el nombre del objeto con que colisiona. Una vez conocemos el nombre del objeto que colisiona con nuestro objeto podemos hacer muchas cosas, por ejemplo: podemos hacer que para destruir una nave enemiga haya que disparar varias veces, podemos hacer desaparecer estos objetos y que aparezca una explosión, etc.

5.-Prototipo videojuego de disparos (*Shooter*)

5.1.-Especificación

El prototipo que he desarrollado para este proyecto es un *Shooter* o videojuego de disparos en primera persona. Consiste simplemente en un jugador que porta un arma (en este caso una metralleta), que se puede desplazar por un escenario en el que se distribuyen unos pilares metálicos a modo de diana a los que este puede disparar y derribar.

Este prototipo consta de un nivel o escena en el que el jugador como he dicho antes puede ir disparando y derribando pilares metálicos y no tiene un fin, es decir, no hay ningún evento (por ejemplo derribar todos los pilares, que se acabe la vida del jugador, etc.) que haga que finalice el nivel, simplemente, si el jugador se sale de los límites del escenario se reinicia el juego.

Decidí decantarme por un *shooter* (en este caso muy básico), ya que me permitía realizar algo sencillo y que a la vez cumpliera perfectamente la función de ilustrar la utilización del motor.

5.2.-Pasos seguidos en el desarrollo del juego

5.2.1.-Definir la escena (elementos pasivos)

Primero, lo que hice fue definir los elementos pasivos, que en este caso son: el escenario, los pilares metálicos que el jugador puede derribar disparándolos y la música de fondo.

El escenario es simple, consiste en un terreno con montañas y unos cuantos árboles.

A continuación paso a describir como creé los "enemigos simples", es decir, los pilares metálicos y cómo añadí música de fondo a la escena.

Crear un enemigo simple

Para hacer un enemigo muy básico al que podamos "matar", usaremos un cubo: `GameObject->Create Other->Cube`, lo escalaremos un poco para que parezca más bien un pilar y le añadiremos una textura, en mi caso he añadido una textura tipo metal que he encontrado en internet. Nos quedará algo parecido a esto:

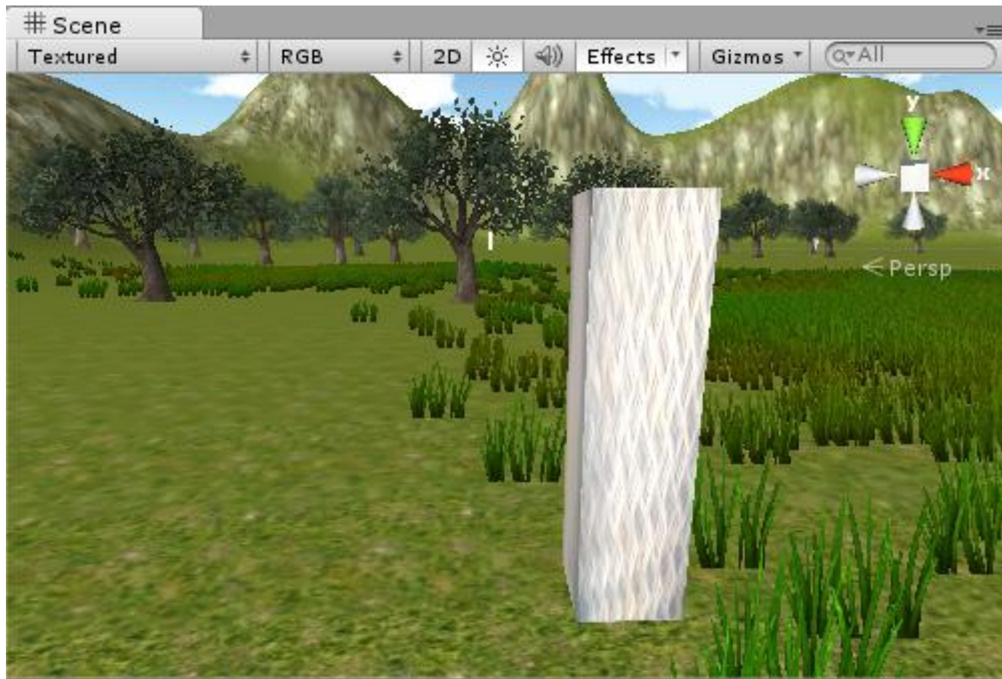


Figura 5.1 Pilar metálico

A nuestro cubo le llamaremos “Enemigo”. Con el enemigo seleccionado nos iremos a `Component->Physics->Rigidbody`.

Para que interactúen nuestras balas con el cubo, en el *prefab* de la bala, en su *sphere collider* tiene que estar marcada la opción `Is Trigger`. Si esta opción estuviera desmarcada, en el momento en que disparásemos al cubo, este saldría despedido hacia adelante porque nuestras balas le empujarían hacia el fondo. Por lo tanto, si está marcada esta opción, las balas traspasan el cubo pero interactúan con él, haciendo que en el momento en que lo traspasan se dispare un evento, es decir pase algo.

A nuestro enemigo que tiene un *box collider* y un *rigidbody* le vamos a añadir un javascript al que llamaremos “enemigo muerte básica”:

```
1 var choke : Transform;
2 function OnTriggerEnter(other : Collider){
3 if (other.tag == "bala"){
4 var clone : Transform;
5 clone = Instantiate(choke, transform.position, transform.rotation);
6 Destroy(gameObject);
7     }
8 }
```

Figura 5.2 Javascript “enemigo muerte básica”

Este *script* hará que todo lo que tenga el *tag* “bala” al chocar con un enemigo, este se convierta en algo, lo que queramos (variable `choke`). En este caso, para probarlo, vamos a elegir que se convierta en una cápsula por ejemplo, haciendo clic en el símbolo con forma de circulito que hay al lado de la variable `choke`, y seleccionando en la ventana que nos aparecerá `Graphics`.

Para que esto funcione, nuestra bala tiene que tener el *tag* “bala”, para ello iremos a nuestro *prefab* de bala y en el inspector desplegamos *tag* que por defecto estará en `Untagged` y seleccionamos

`Add Tag`. Desplegamos `Tags` y en `size` pondremos 4 por ejemplo y en `element 0` pondremos "bala" en minúsculas para que de esta manera nuestro *script* lo reconozca.

Nos iremos de nuevo al *prefab* de la bala y donde pone `tag` lo cambiamos de `Untagged` a `bala`. Ahora al darle al play y disparar a nuestro enemigo cubo, este se convertirá en una cápsula.

Vamos a imaginar que queremos que nuestro enemigo "muera", es decir, que se caiga al suelo cuando nuestras balas colisionen con él. Para ello, vamos a duplicar nuestro cubo y al duplicado le llamaremos "enemigo muerto". A este cubo le vamos a escalar de manera que parezca que esta tumbado, es importante que lo escalemos, no que lo rotemos, ya que si lo rotamos para que esté tumbado lo que pasará es que cuando disparemos a nuestro enemigo y este cambie a enemigo muerto, al ser idéntico este (aunque tumbado) a enemigo, lo dará la vuelta dejándolo derecho y no caerá pareciendo que no ha pasado nada.

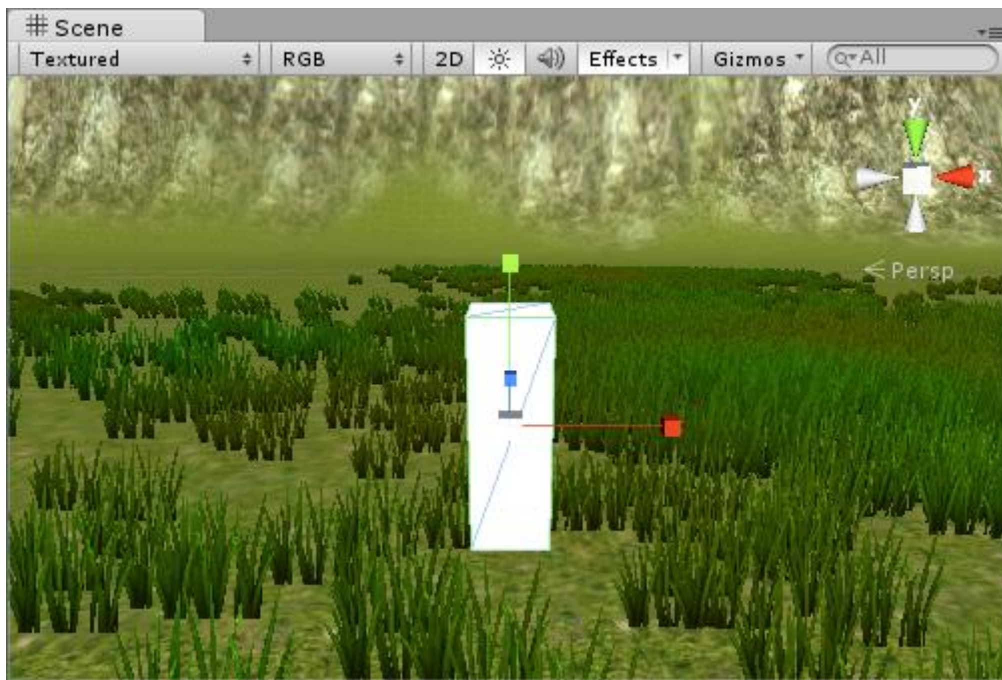


Figura 5.3 "Enemigo muerto" antes de escalarlo

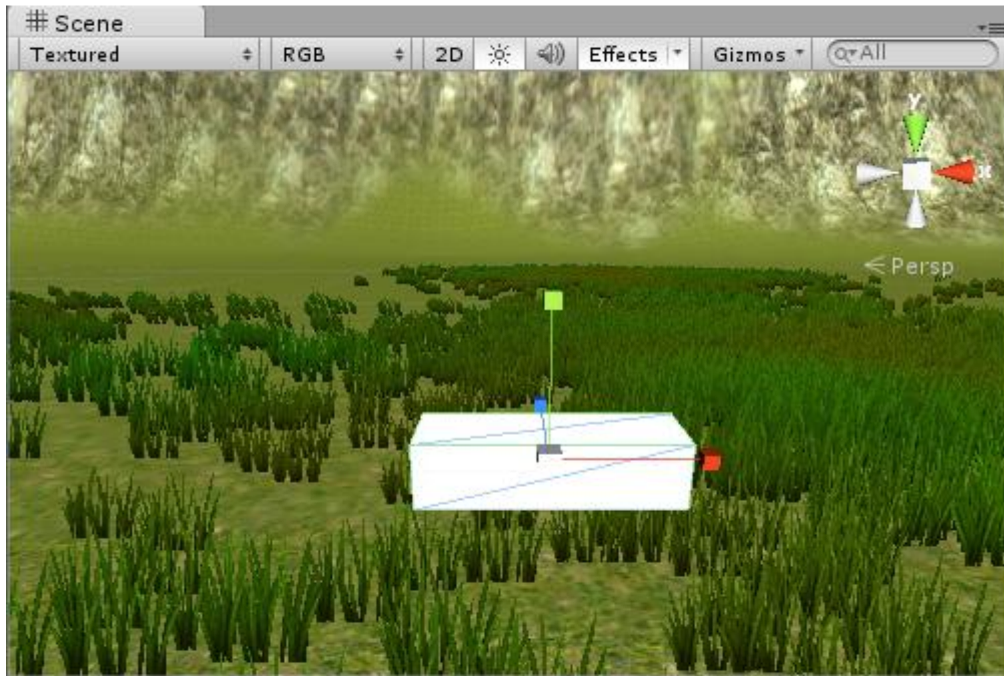


Figura 5.4 “Enemigo muerto” después de escalarlo

Después, crearemos un *prefab* al que llamaremos también “enemigo muerto” y arrastraremos nuestro cubo enemigo muerto de la vista de jerarquía a nuestro *prefab* con el mismo nombre.

Como enemigo muerto es un clon de enemigo, seguirá teniendo el *script* enemigo muerte básica, este le quitamos de nuestro *prefab* enemigo muerto. Enemigo muerto ya no tendrá dicho *script*, pero enemigo si y nos pide un *transform* (`Choke`), con lo cual arrastraremos nuestro *prefab* enemigo muerto a la variable `choke`. Ahora, al darle al play y disparar a nuestro enemigo, este caerá. Podemos eliminar enemigo muerto de la vista de jerarquía que ya no nos sirve para nada.

Poner música de fondo

Primero buscamos un audio en internet. En el *Asset Store* de Unity tenemos muchos audios que podemos descargarnos gratis.

Para que no nos dé un fallo típico que es el de que haya dos o más *audio listener*, nos aseguramos de que sólo tenemos el *audio listener* de la cámara que lleva incorporado nuestro *first person controler*. El *audio listener* sirve para avisar a nuestro nivel de que nuestro mapa tiene audio y cuando ponemos más de uno en nuestra escena, da fallos.

Vamos a añadir a la cámara del *first person controller* un componente *audio source*, para ello, en el inspector de la cámara del *first person controller*, hacemos clic en `Add Component->Audio->Audio Source`. Una vez añadido, nos vamos a este componente en el inspector y donde pone `Audio clip` arrastramos el sonido o melodía que nos hemos descargado. Además marcaremos la casilla `Loop`. Si no la marcásemos, simplemente lo que ocurriría es que la melodía se escucharía una vez y al finalizar no se volvería a repetir. Como lo que queremos tener es una música de fondo para el nivel, lo que nos interesa es que se repita constantemente y por lo tanto marcaremos `Loop`. Por último bajaremos el volumen de la melodía ya que si no se oirá muy alto.

Bien, ya tenemos nuestra música de fondo puesta en nuestro videojuego.

5.2.2.-Definir los elementos activos de la escena

En este caso como elementos activos, es decir, elementos con los que podemos interactuar dentro del juego tenemos el arma. A continuación paso a explicar cómo añadí esta al juego e hice que disparara.

Añadir un arma y que esta dispare

Partimos de tener un terreno creado con montañas, árboles, césped, etc., y muy importante: un *first person controller*. Este será nuestro jugador en primera persona.

Una vez hecho esto, pondremos una metralleta 3D, que hemos descargado e importado previamente (por ejemplo en el *asset store* tenemos modelos gratis), en la escena.

Ahora, para colocar nuestra metralleta de manera que parezca que nuestro personaje la sostiene, nos vamos a nuestro *first person controller* y seleccionamos la metralleta con un solo clic y nos vamos a menú `Game Object->Move to view` para que la metralleta se vaya al centro de nuestra vista.

Si no vemos nuestra metralleta, es posible que sea tan pequeña que no la podamos ver, en ese caso con la herramienta escalar haremos la metralleta más grande hasta que sea de un tamaño más o menos proporcionado.

Moveremos la metralleta hasta situarla en el medio más o menos de nuestra vista, de este modo:

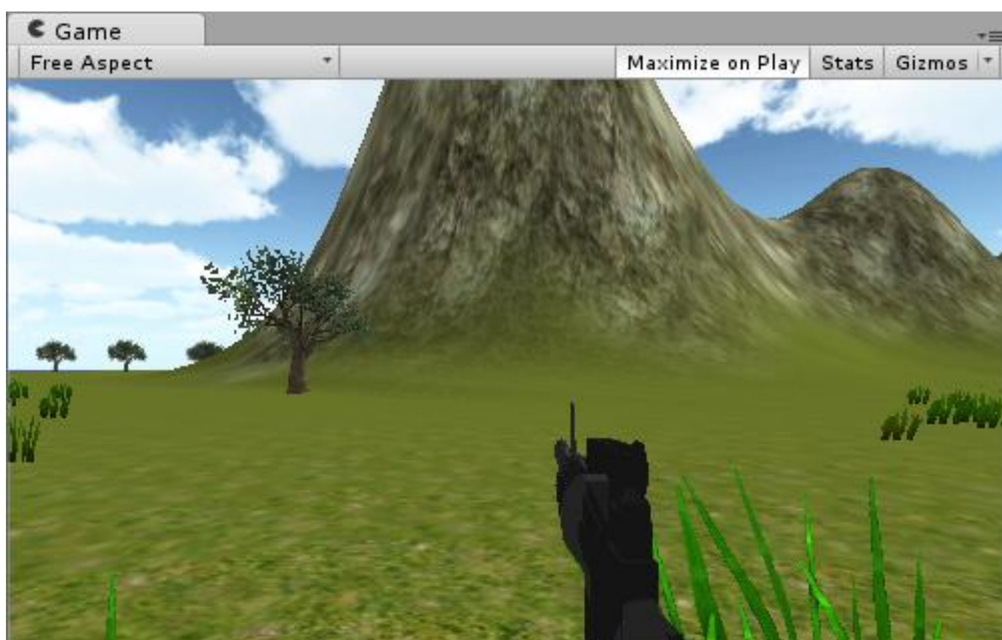


Figura 5.5 Metralleta situada

La metralleta que me descargué está compuesta por un desplegable de varias carpetas hasta llegar a un *Mesh*, que es la malla del modelo 3D. Para trabajar con ello más cómodamente es mejor simplemente quedarse con el *mesh* que es lo que importa, por lo que arrastré el *mesh* fuera de las carpetas y después borré las carpetas.

La metralleta, ya que va a formar parte de nuestro personaje, la arrastramos dentro del *first person controller* en la vista de jerarquía. Desplegamos *first person controller* y nuevamente arrastraremos nuestra metralleta a la *main camera*. La metralleta se quedará "pegada" a la *main camera*.

Ahora crearemos un objeto vacío en el menú `GameObject->Create Empty` y lo arrastramos justo al cañón de la metralleta pero sin estar pegado. De este *game object* vacío saldrán las balas.

Este *game object* vacío lo arrastramos dentro de metralleta en la vista de jerarquía. Al *game object* vacío le arrastraremos un javascript al que llamaremos "disparador":

```
var projectile : Rigidbody;
var speed = 20;
function Update()
{
 if( Input.GetButtonDown( "Fire1" ) )
 {
  var instantiatedProjectile : Rigidbody = Instantiate(
   projectile, transform.position, transform.rotation );
  instantiatedProjectile.velocity =
   transform.TransformDirection( Vector3( 0, 0, speed ) );
  Physics.IgnoreCollision( instantiatedProjectile. collider,
   transform.root.collider );
 }
}
```

Figura 5.6 Javascript "disparador"

Teniendo seleccionado el *game object* vacío, si vemos el inspector, en el *script* que le hemos añadido veremos que hay una variable expuesta: `projectile`. Este proyectil sería la bala. Al arrastrar la bala a la variable expuesta, lo que hará este objeto vacío es generar balas que saldrán disparadas.

Lo que haremos ahora es crear una bala, para ello nos iremos a `gameobject->create other->sphere`. Esta esfera la haremos pequeñita y para que parezca más una bala la deformaremos de manera que quede ovalada y alargada.

Es importante que añadamos a nuestra bala un *rigidbody*, para ello, con la bala seleccionada nos vamos a `Component->Physics->Rigidbody`. De esta manera dotaremos a nuestra bala de una física real. Al darle a play, nuestra bala caería al suelo, porque tiene una masa y unas propiedades físicas. Yo le di una masa de 300.

Ahora crearemos un *prefab* en la vista de proyecto y le pondremos como nombre "bala". A la esfera que creamos con anterioridad, también la llamaremos bala. A continuación arrastraremos nuestra bala de la vista de jerarquía (la esfera que creamos) al *prefab* bala en la vista de proyecto y borraremos la bala de la vista de jerarquía ya que no nos hará falta. El *prefab* de la bala, ahora tiene un *rigidbody*, un *mesh renderer*, un *sphere collider*, etc.

En este momento, si arrastramos nuestra bala (*prefab*) a `projectile`, que es la variable expuesta del *script* disparador, veremos que al darle al play y disparar, nuestra bala se clona cada vez que

disparamos, por lo que se crearán en la vista de jerarquía infinitas balas. Esto ocupará un espacio importante en nuestro proyecto y en el juego y hará que nos vaya mucho más lento.

Para que no nos pase esto, a nuestra bala le añadiremos un javascript al que llamaremos "temporizador balas":

```
1 var nTime : float = 10;
2 public var Aceleracion : float = 1;
3 var DisparoChoque : Transform;
4
5 function Start () {
6 Destroy(this.gameObject, nTime);
7 }
8
9 function OnCollisionEnter(){
10 var clone : Transform;
11 clone = Instantiate(DisparoChoque, transform.position, transform.rotation);
12 Destroy(gameObject);
13 }
```

Figura 5.7 Javascript "temporizador balas"

Arrastraremos este *script* a nuestro *prefab* bala. Veremos en el inspector del *prefab* bala que el *script* temporizador balas tiene una variable llamada `nTime` que es el tiempo que tardará la bala en desaparecer. Le pondremos un valor de 5. También veremos que hay una variable llamada `disparo choque`, que es en lo que queremos que se transforme la bala al chocar con algo (por ejemplo un agujero negro). Para este ejemplo, esta variable no se ha usado.

Ahora si damos al play, vemos que al disparar, las balas se clonan en la vista de jerarquía pero a los cinco segundos van desapareciendo.

Al *prefab* de la bala le vamos a añadir un componente nuevo, haciendo clic en `add component` del inspector de la bala. Acto y seguido iremos a `effects->Trail Renderer` y en este componente que hemos añadido introduciremos estos parámetros: `Time`=1, `Start Width`=0.1 y `End Width`=0.1, esto es lo que tardará el efecto del fuego en aparecer y desaparecer. Desplegaremos la pestaña de materiales y en `element 0` le añadiremos Sparkles1. Al darle al play y disparar veremos que las balas son más reales, pero estas caen al suelo porque no salen con mucha fuerza.

Vamos a corregir esto y para ello nos iremos al *script* del objeto vacío que está dentro de nuestra metralleta y en `speed` o velocidad de la bala pondremos 100 por ejemplo. Ahora las balas no caerán y saldrán disparadas con más velocidad.

Lo siguiente que haremos es que nuestra metralleta emita el sonido del disparo para que nos quede más realista. Para ello nos iremos a nuestro *prefab* de la bala (en el inspector) y le añadiremos un sonido a esta haciendo clic en `add component->Audio->Audio Source`. En el Audio source bajaremos un poco el volumen y añadiremos un sonido de un disparo, que podemos encontrar en internet, a `audio clip`. Ahora al disparar sonará el sonido que le hemos puesto a la bala.

5.2.3.-Control global de la escena

El control global de la escena en este caso consiste en que cuando el jugador llega a uno de los límites del escenario se reinicia el nivel o escena.

En este ejemplo de juego, he controlado la escena de este modo, pero para un videojuego en el que queramos que al pasar por un lugar del escenario (por ejemplo, una puerta) se acceda a otro nivel, se haría exactamente igual, pero cambiando simplemente ese matiz.

Cambiar de nivel al pasar por un sitio

En nuestro caso lo que haremos es que al salirnos de los límites de nuestro mapa, se reinicie el nivel, pero se hace de la misma forma que si tuviéramos dos niveles y quisiéramos cambiar de nivel al pasar por un determinado sitio.

Lo primero que haremos es irnos a `File->Build Settings`. En la ventana que nos aparecerá haremos clic en Add current (Añadir la escena actual) y se añadirá nuestra escena en la compilación de nuestro juego, para que el juego la reconozca. Si tuviéramos más escenas y quisiéramos pasar de una a otra, añadiríamos en esta ventana todas las escenas y las pondríamos un orden para que se determine qué escena va primero y cual va después.

En nuestra escena vamos a crear un cubo: `Game Object->Create Other->Cube` y lo vamos a situar en un extremo de nuestro terreno para que haga de frontera entre el terreno y el vacío y lo escalaremos de tal forma que quede más o menos así:

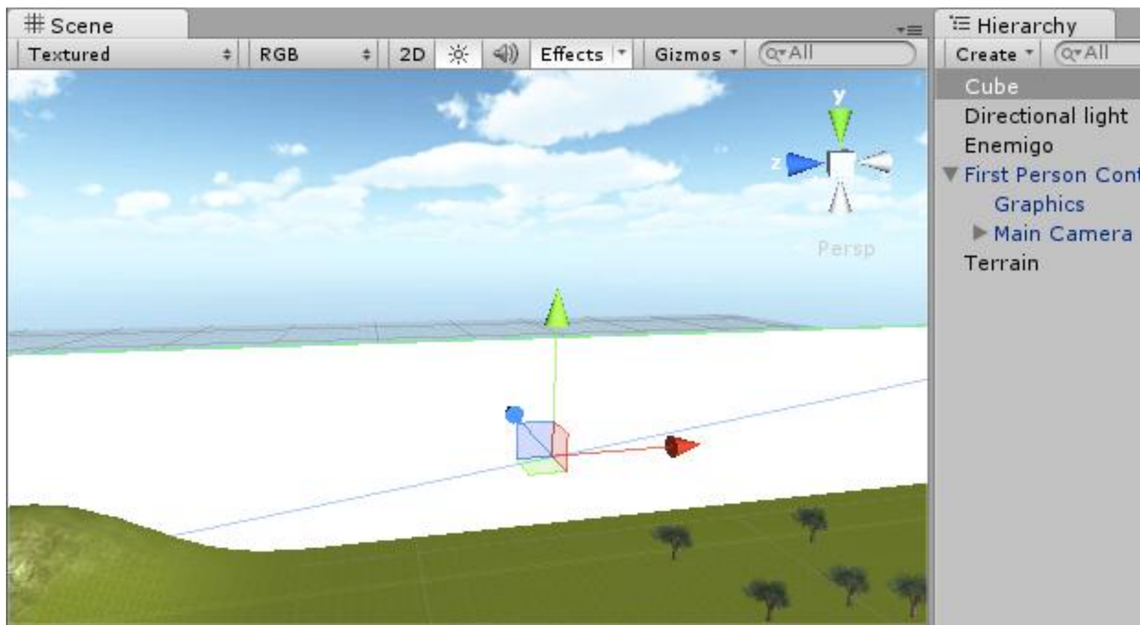


Figura 5.8

Este cubo va a hacer que cuando nosotros colisionemos con él, se reinicie el nivel, pero no va a tener esta apariencia, será invisible. Para que sea invisible simplemente desmarcaremos la casilla `mesh renderer` en el inspector del cubo. Para que podamos atravesar el cubo tendremos que activar la casilla `Is Trigger`. Ahora podemos atravesarle y se ha convertido en un gatillo, por decirlo así, que dispara un evento y le podemos asignar una función a este *trigger*.

El *script* que asignaríamos al cubo sería este y le nombraríamos “Fin” (por ejemplo):

```
1 function OnTriggerEnter (other : Collider) {
2
3     if (other.gameObject.tag == "Player")
4
5         Application.LoadLevel("Escena1");
6
7         }
```

Figura 5.9 Script “Fin”

Vamos a explicar este *script*. Tenemos una función llamada `OnTriggerEnter` y la pasamos un parámetro al que llamamos `Other` de tipo `Collider`. Esta función lo que indica es que en el momento en que entre en el *trigger* un *collider*, pase algo. Dentro de la función tenemos `IF (other.gameObject.tag==”Player”)` que quiere decir que si `other` que es el *collider* que le pasamos a nuestra función y el que va a entrar en el *trigger*, tiene un *tag* que sea “Player” (para que de esta manera otro objeto que no sea nuestro jugador no pueda entrar en el *trigger* y hacer que se reinicie el nivel, de esta forma solo puede hacer esto el jugador), entonces se nos enviará a Escena1 (`Application.LoadLevel(“Escena1”);`) que es la misma escena que en la que estamos, por lo tanto se reiniciaría el nivel. Guardamos nuestro *script*.

Vamos a arrastrar nuestro *script* al cubo. Ahora para que funcione, vamos a poner el *tag* Player a nuestro *first person controller* (El *tag* player viene por defecto en las opciones que hay para el *tag*). Ahora al darle al play y dirigirnos a ese extremo del terreno donde tenemos el cubo invisible haciendo frontera, al chocar con él se reiniciará nuestro nivel y volveremos a aparecer donde estábamos al principio.

5.3.-Controlador del juego

En este ejemplo, no hay un controlador de juego en sí, si no que hay un menú principal muy sencillo que tiene un botón start que al ser pulsado se inicia la única escena del videojuego y un menú pausa también muy simple que congela el juego y además posee un botón que al ser pulsado nos permite salir del juego.

En este prototipo el menú pausa está integrado dentro de la escena, pero lo suyo es que el menú principal y el menú pausa formaran parte de un controlador de juego y el juego propiamente dicho que sería un conjunto de escenas o niveles estuviera dentro del controlador pero en partes independientes para que las diferentes estructuras fueran reutilizables y así poderlas usar en distintos videojuegos.

Este sería el esquema de la estructura:

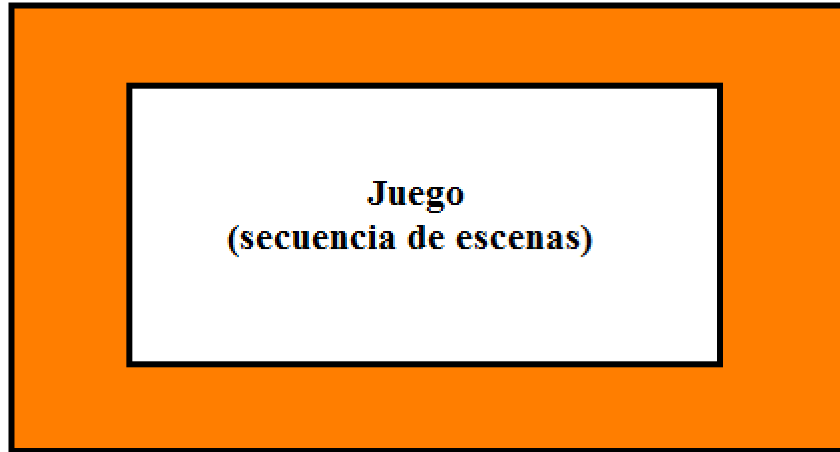


Figura 5.10

Haciendo una analogía con sistemas operativos, un juego o escena equivaldría al programa y el controlador del juego al proceso o hilo donde se ejecuta el programa.

A continuación paso a explicar cómo hice el menú principal y el menú pausa.

Crear un menú principal

Para crear nuestro menú principal, abriremos una nueva escena. En nuestra nueva escena aparece por defecto una *main camera* o cámara principal y nada más. Ajustaremos la vista de escena para que se vea lo que capta la cámara (ver Figura 5.11).

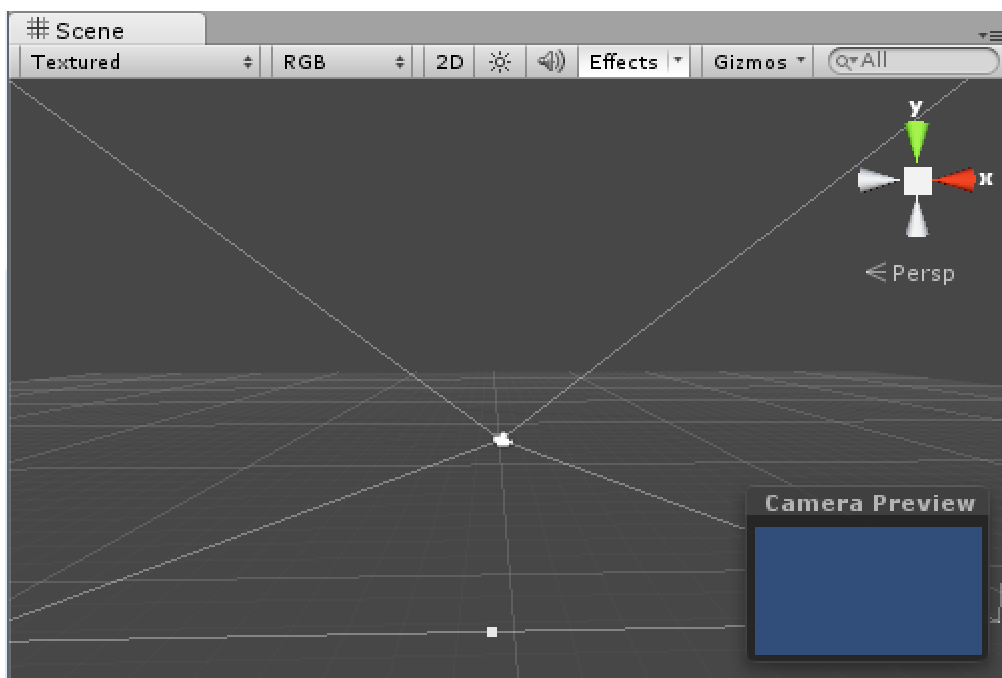


Figura 5.11

Ahora nos vamos a `GameObject->Create Other->Cube` y lo moveremos hacia delante de la cámara para que nos aparezca en la vista de juego (ver Figura 5.12).

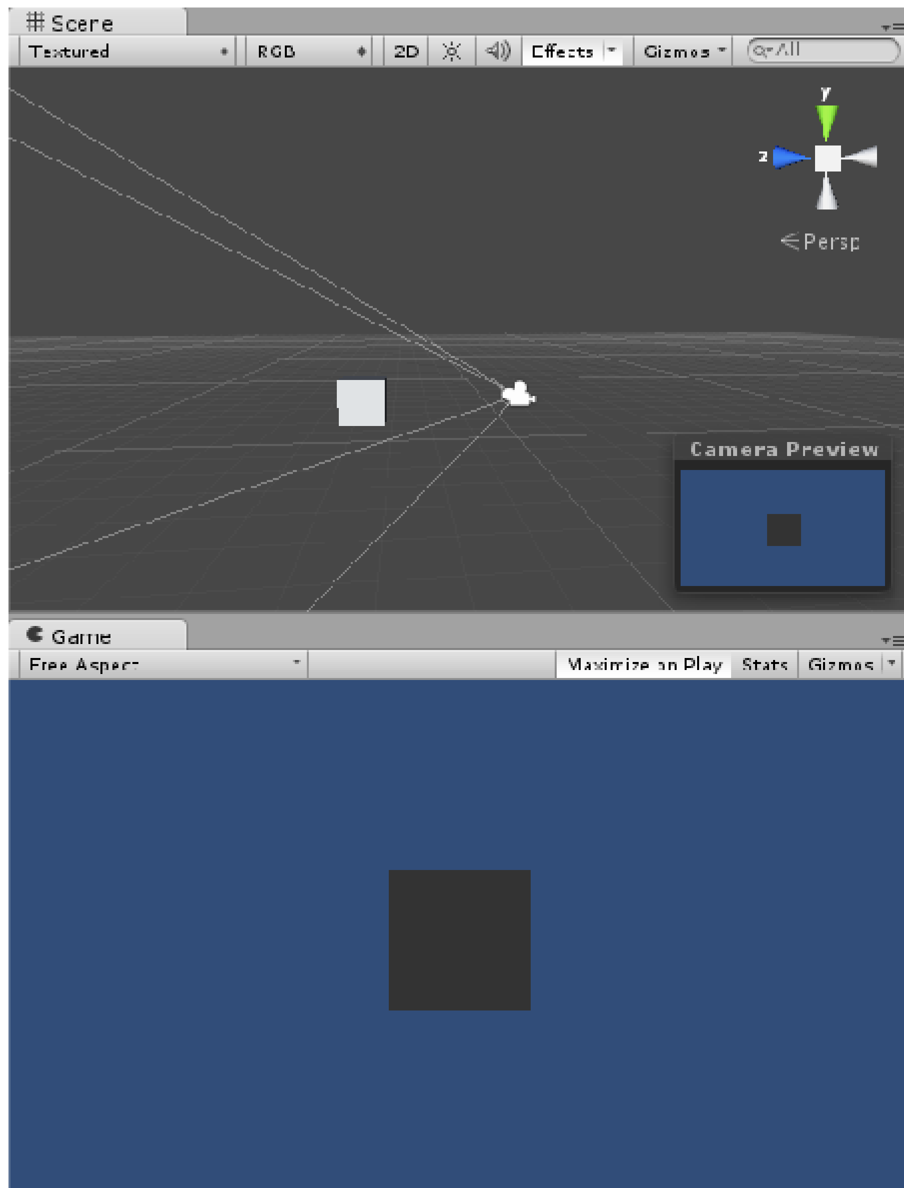


Figura 5.12

A este cubo lo vamos a escalar para que ocupe toda la vista de juego. Nos vamos a `Edit->Render Settings` y pondremos `ambient light` al máximo de blanco porque si no, se verá oscuro.

Ahora buscaremos un fondo para nuestra pantalla inicial (yo escogí una imagen de internet) y la arrastraremos a nuestro proyecto en Unity. De ahí la arrastraremos a nuestro cubo. Si nos fijamos nuestra imagen aparece al revés por lo que tendremos que voltear el cubo para que se vea del derecho. Si damos al play veremos cómo queda y si tenemos que hacerle ajustes para que se vea centrado y no haya huecos en los laterales, de manera que nos quede completamente encajado en la pantalla.

Ahora vamos a añadir un botón, por lo que iremos a `Game Object->Create Other->GUI Texture` y nos aparecerá un botón por defecto de unity. Este botón le podemos mover a la posición que queramos dentro de la que va a ser nuestra pantalla principal.

En nuestra GUI Texture, en el inspector veremos que aparece `Texture` y por defecto tiene asignado una imagen de Unity: Unity Watermark. Aquí podremos agregarle lo que será nuestro botón, que puede ser una imagen con la palabra "Start" por ejemplo. Si hacemos esto nos aparecerá nuestro botón personalizado con la imagen que queramos.

Para que este botón sea funcional debemos crear un JavaScript al que llamaremos "BotonStart":

```
1
2 function OnMouseDown () {
3
4  Application.LoadLevel("Escena1");
5
6  }
```

Figura 5.13 Javascript "BotonStart"

Este *script* es muy simple y lo que hace es que cuando pulsemos con el ratón nuestro botón, vayamos a la Escena1. Guardamos nuestro *Script*.

Una vez hecho esto, arrastraremos este *script* a "Unity watermark" (que es nuestro botón) en la vista de jerarquía.

Si damos al play y pulsamos sobre nuestro botón, no pasará nada porque no tenemos cargadas las escenas en el juego. Para cargarlas iremos a `File->Build Settings` y daremos en la ventana que nos aparece a Add Current y se añadirá nuestra escena de menú principal y lo mismo haremos con nuestra otra u otras escenas y las ponemos en el orden que corresponda, como en primer lugar queremos que nos aparezca la escena del menú principal, la pondremos la primera.

Ahora cuando le demos al play en nuestro juego sí que funcionará y cuando hagamos clic en el botón, apareceremos en la Escena1.

Menú pausa

Para crear un menú de pausa, yo lo que hice fue crear un objeto vacío: `Game Object->Create Empty` y asignarle el javascript que veremos a continuación:

```
1 #pragma strict
2 var pausado : boolean = false;
3
4 function Update(){
5
6     if (Input.GetKeyDown (KeyCode.Escape)) {
7     pausado = !pausado; //Cambia el valor de la varia
8
9         if (pausado == false) {
10        Time.timeScale = 1;
11        }
12        else {
13        Time.timeScale = 0;
14        Screen.showCursor = false;
15        }
16
17    }
18
19 }
```

Figura 5.14 Primera parte del *script*

```
function OnGUI(){

    if(pausado){
    Screen.showCursor = true;
    GUI.Box(new Rect(0,0,Screen.width,Screen.height),"\n\nPAUSA");
    GUI.backgroundColor = Color.black;


        if(GUI.Button(new Rect(Screen.width/2-150,(Screen.height/2)-50,300,50),"Salir del juego")){
        Application.Quit();
        }

    }

}
```

Figura 5.15 Segunda parte del *script*

Vamos a explicar este *script*:

`var pausado : boolean = false;` ->Aquí definimos una variable booleana llamada `pausado` que la damos un valor inicial de false.

Vamos con la función `update`:

```
function Update(){

    if (Input.GetKeyDown (KeyCode.Escape)) {
    pausado = !pausado; //Cambia el valor de l

        if (pausado == false) {
        Time.timeScale = 1;
        }
        else {
        Time.timeScale = 0;
        Screen.showCursor = false;
        }

    }

}
```

Figura 5.16 Función `Update`

La función `update` es llamada cada *frame*. Dentro de esta función tenemos una condición: `if (Input.GetKeyDown (KeyCode.Escape))` que significa que si en algún momento es pulsada la tecla Esc. Dentro de la condición tenemos lo siguiente: `pausado = !pausado;` que significa que si la condición anterior se cumple entonces la variable `pausado` cambiará su valor a lo contrario, es decir si su valor es true cambiará a false y si su valor es false cambiará a true.

Ahora tenemos dos condiciones:

```
if (pausado == false) {
Time.timeScale = 1;
}
else {
Time.timeScale = 0;
Screen.showCursor = false;
}
```

Figura 5.17 Condiciones de la función `Update`

Esto quiere decir que si la variable `pausado` es igual a false entonces el juego no estará pausado por lo tanto su `timeScale` será igual a 1 y por el contrario, si esto no es así entonces el juego será pausado, es decir su `timeScale` será igual a 0 y no se mostrará el cursor.

Por otra parte tenemos:

```
function OnGUI(){

    if(pausado){
    Screen.showCursor = true;
    GUI.Box(new Rect(0,0,Screen.width,Screen.height),"\n\nPAUSA");
    GUI.backgroundColor = Color.black;

        if(GUI.Button(new Rect(Screen.width/2-150,(Screen.height/2)-50,300,50),"Salir del juego")){
        Application.Quit();
        }

    }

}
```

Figura 5.18 Función `OnGUI`

Tenemos la función `OnGUI` que al igual que `update`, se ejecuta durante todo el juego, pero esta solo sirve para *scripts* que utilicen comandos del GUI. Todo lo que sea GUI debe incluirse dentro de la función `OnGUI`.

Dentro de esta función hay una condición: `if (pausado)` que quiere decir que si `pausado` es igual a true se ejecutarán las siguientes sentencias. Estas son:

`Screen.showCursor = true;` ->Aparecerá de nuevo en la pantalla el cursor.

`GUI.Box(new Rect(0,0,Screen.width,Screen.height),"\n\nPAUSA");` -

> `Box` es una función de clase de la clase `GUI` y crea un cuadro o caja. Esta variante de la función `box` es:

`static function Box (position : Rect, text : String) : void` que tiene dos parámetros: la posición que es el rectángulo en la pantalla a usar para la caja, en este caso este rectángulo tiene su posición en la coordenada 0,0 y ocupa toda la pantalla y `text` que es el texto a mostrar en la caja que en este caso es "PAUSA".

`GUI.backgroundColor = Color.black;` -> Esta sentencia determina el color de fondo de la GUI que en este caso es de color negro.

```
if(GUI.Button(new Rect(Screen.width/2-150,(Screen.height/2)-50,300,50),"Salir del juego")){
Application.Quit();
```

Por último tenemos otra condición, en la que además de que se crea un botón con la función de la clase `GUI`, `Button`, (en este caso sería esta variante de la función: *`static function Button (position : Rect, text : String) : boolean)`*, esta función retorna un booleano que es establecido en true cuando el usuario hace clic sobre el botón y cuando retorne true se ejecutará lo que hay dentro del condicional. Este botón tiene una posición determinada en la pantalla y un texto que en este caso es "Salir del juego".

Dentro del condicional hay sólo una sentencia: `Application.Quit();` que hace que se cierre la aplicación.

En resumen, esta última parte significa que si pulsamos el botón Salir del juego, se cerrará la aplicación y por lo tanto el juego.

5.4.-Compilar nuestro juego para PC

Para poder crear el ejecutable de nuestro juego primero nos tenemos que asegurar que no hay errores en los *scripts* ya que si no a la hora de compilar nuestro juego nos van a dar problemas.

Una vez hayamos comprobado esto, nos iremos a `File->Build Settings` y nos aparecerá esta ventana:

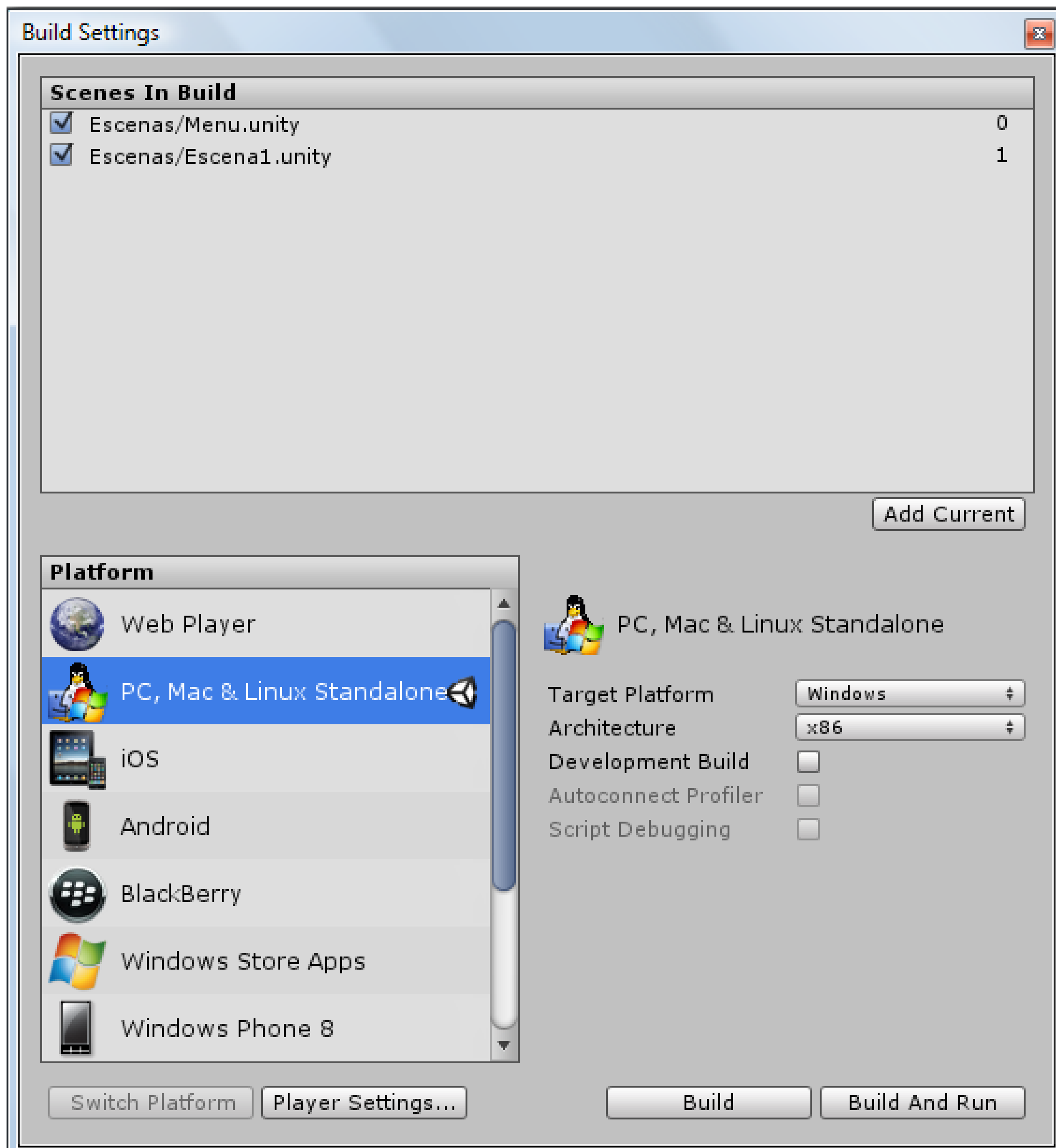


Figura 5.19

En este caso ya tenemos introducidas nuestras escenas del videojuego, si no fuera así, pues las introduciríamos haciendo clic en Add Current desde cada escena.

Por lo tanto en la parte Scenes In Build tienen que ir todas las escenas que van a formar parte de nuestro videojuego.

Por otro lado tenemos Platform donde vienen todas las plataformas para las que se puede compilar nuestro juego. En este caso como nuestro videojuego va a ser para PC, elegiremos esa opción que es la que está marcada en la imagen anterior.

Otra cosa importante de esta ventana es Architecture, que tiene dos opciones, la primera es que nuestro juego funcione en arquitecturas de 32 bits y la segunda que funcione en arquitecturas tanto de 32 bits como de 64. Yo elegí la primera ya que con la segunda opción al ejecutar el videojuego me daba error.

Una vez hecho todo esto, para compilar nuestro juego haremos clic en el botón Build y nos saldrá una ventana para poner un nombre a nuestro videojuego y guardarlo en la ubicación que queramos. Cuando se haya compilado, veremos que en la ubicación que hemos indicado para guardar nuestro videojuego nos aparece el ejecutable y una carpeta con toda la información de nuestro juego.

Cuando ejecutemos el juego simplemente, en la ventana que nos aparecerá daremos al play y listo. Es mejor que en esta ventana dejemos todo como está.

6.-Posibles ampliaciones

Este estudio acerca del motor Unity 3D admite muchas posibles ampliaciones. En este punto voy a enumerar unas cuantas:

- Inteligencia artificial de enemigos
- Cómo animar a un personaje
- Más efectos que se pueden crear con los sistemas de partículas
- Cómo hacer un videojuego en tercera persona
- Cómo conseguir que la cámara siga al personaje principal
- Ampliar la parte del estudio de la programación en Unity
- Cómo crear terrenos realistas mediante *heightmaps* o mapas de altura. Los *heightmaps* son mapas en escala de grises donde las partes más oscuras son las zonas más profundas del mapa y las más claras son las más altas.
- Cómo crear un videojuego *multiplayer*
- Crear efectos meteorológicos, por ejemplo, lluvia.

7.-Conclusiones

Una vez finalizado el proyecto, se puede concluir que la creación de un videojuego es una tarea ardua y que los motores de videojuegos o *engines* son herramientas que hacen del desarrollo de videojuegos algo más ágil y sencillo.

Unity 3D es una herramienta que nos ayuda a desarrollar videojuegos para diversas plataformas mediante un editor y *scripting* para crear videojuegos con un acabado profesional. Esta herramienta está accesible en diferentes versiones, gratuita (que es la que se ha utilizado en este proyecto) y profesional, cada cual con sus ventajas y limitaciones, evidentemente la más completa es la profesional pero es necesario hacer un desembolso que no todo el mundo puede permitirse y sobre todo si estamos comenzando a utilizar dicha herramienta.

Unity 3D nos provee de un editor visual muy útil y completo. Además incluye la herramienta de desarrollo MonoDevelop con la que podremos crear *scripts* en JavaScript, C# y un dialecto de Python llamado Boo con los que extender la funcionalidad del editor.

Además si no somos muy manitas con el 3D o necesitamos recursos para nuestros juego, en la propia aplicación podemos acceder a una tienda como si de la *App Store* se tratase, donde encontraremos multitud de recursos gratuitos y de pago. Incluso podremos extender la herramienta mediante *plugins* que obtendremos en dicha tienda.

Por último decir que Unity 3D está en evolución constante y cada cierto tiempo se presenta una nueva versión que incluye mejoras y avances.

Bibliografía

Manual Unity (Original en inglés):

«Unity - Manual: Unity Manual». http://docs.unity3d.com/Manual/index.html.

Tutoriales Unity:

«FpsETeeski_Español_Cotolonco_Parte1». http://es.scribd.com/doc/101962034/FpsETeeski-Espanol-Cotolonco-Parte1.

«Tutorial de scripts para Unity 3d». http://unityscripts.blogspot.com.ar/.

«UnitySpain». *UnitySpain*. http://unityspain.com/.

«Index of /unity/tutoriales». http://trinit.es/unity/tutoriales/.

«Unity 3D». *Noticias de Redes Sociales y tutoriales Online*. http://thenacoosweb.com/category/software/desarrollo-de-videojuegos/unity/.

«Unity3D – Partículas de Fuego». *Game Development*. http://www.gamedev.es/?p=11872.

Alvaro Cortes Tellez,. «Unity next gen_&_scripting». 22:49:47 UTC. http://www.slideshare.net/4monsters/unity-next-genscripting.

Recursos Unity:

«Index of /unity/packages». http://trinit.es/unity/packages/.

Proyectos fin de carrera:

«UVaDOC: Inicio». https://uvadoc.uva.es/.

«Proyectos Fin de Carrera». http://e-archivo.uc3m.es/handle/10016/4904.

Información sobre Unity:

«Unity (software)». *Wikipedia, la enciclopedia libre*, 30 de junio de 2014. http://es.wikipedia.org/w/index.php?title=Unity_(software)&oldid=74692109.

«Unity, el motor de desarrollo capaz de partir la historia de los videojuegos en dos». http://www.vidaextra.com/industria/unity-el-motor-de-desarrollo-capaz-de-partir-la-historia-de-los-videojuegos-en-dos.

Motor de videojuego (*game engine*):

«Motor de videojuego». *Wikipedia, la enciclopedia libre*, 16 de mayo de 2014. http://es.wikipedia.org/w/index.php?title=Motor_de_videojuego&oldid=73763086.

CFCA1912. «[Megapost] ■ Motores Gráficos - El corazón de los Juegos», 3 de mayo de 2012. http://www.taringa.net/posts/info/14711340/Megapost-Motores-Graficos---El-corazon-de-los-Juegos.html.

«¿Qué es un Engine para videojuegos?» *niubie*. http://www.niubie.com/2011/01/que-es-un-engine-para-videojuegos/.

«¿Qué es un motor gráfico 3D? | A fondo». *Softonic*. http://articulos.softonic.com/que-es-un-motor-grafico-o-motor-3d.

Videotutoriales:

«Arturo Nereu». *YouTube*. http://www.youtube.com/user/hdgam3r.

«Fenix Discom». *YouTube*. http://www.youtube.com/user/FenixDiscom.

«game3Dover». *YouTube*. http://www.youtube.com/user/game3Dover.

«German Medina». *YouTube*. http://www.youtube.com/user/AntoValls.

«MrJocyf». *YouTube*. http://www.youtube.com/user/MrJocyf.

«nodician». *YouTube*. http://www.youtube.com/user/nodician.

«Rey3D Oficial». *YouTube*. http://www.youtube.com/user/edrey3d.

«SrBit». *YouTube*. http://www.youtube.com/user/ElSrBit.

Instalación de Unity:

«Instalar Unity 3D | Profesor Antonio». http://profesor.antonio.com.mx/?p=31.

Glosario de términos

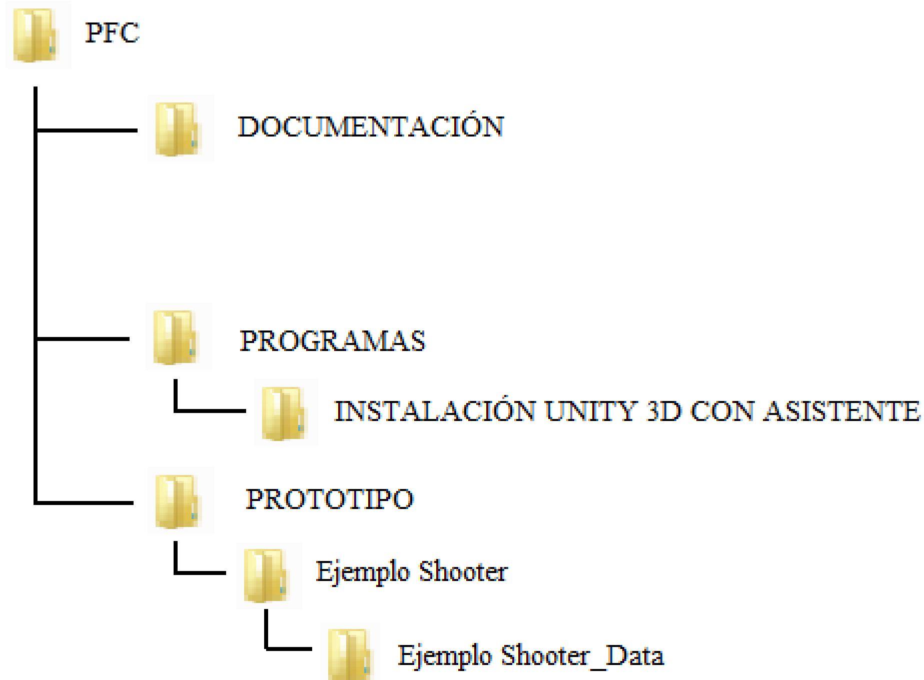
2.5D	Modo tridimensional que se limita a un plano de dos dimensiones.
2D	Dos dimensiones.
3D	Tres dimensiones.
3D Construction Kit	También conocido como *3D Virtual Studio*, es una utilidad para crear mundos 3D en Freescape. Desarrollado por Incentive Software.
3dfx	**3dfx Interactive** fue una compañía especializada en la manufactura de procesadores gráficos 3D y, más tarde, tarjetas gráficas para uso en PC. Fue la gran dominante en la época inicial de las tarjetas aceleradoras 3D, a finales de los años 1990, especialmente en los años 1997 y 1998.
ASCII	**ASCII** (acrónimo inglés de **A**merican **S**tandard **C**ode for **I**nformation **I**nterchange — *Código Estándar Estadounidense para el Intercambio de Información*), pronunciado generalmente [áski] o [ásci], es un código de caracteres basado en el alfabeto latino, tal como se usa en inglés moderno y en otras lenguas occidentales.
Assets	Recursos tales como: modelos 3D, texturas, archivos de sonido, código fuente, etc.
Build Engine	**Build** es un motor de juego para videojuegos de disparos en primera persona creado por Ken Silverman.
Cloth	Objeto en 3D que se comporta como un tejido tanto a nivel visual como de interacciones físicas (Deformaciones, rupturas, etc.).
Collider	Estructura que hace sólidos a los objetos.
Character Controller	El **Character Controller** es un componente que viene incluido en cualquier versión de Unity 3D. Funciona para añadir los comportamientos básicos de un personaje: caminar, brincar y sus animaciones de cada estado. Hay dos tipos: En **Tercera** persona y **Primera** persona.
Checkbox	Casilla de verificación.
Engine	Motor de videojuego.
Frame	Imagen particular dentro de una sucesión de imágenes que componen una animación.
Framerate	Fotogramas por segundo. Medida de la frecuencia a la cual un reproductor de imágenes genera distintos fotogramas (*frames*).
Game Object	Objeto de juego. Son todos aquellos objetos que usamos en nuestro escenario: luces, cámaras, modelos, texturas, etc.
GoldSrc	Versión modificada del código del motor QuakeWorld, que a su vez es un desarrollo posterior del motor Quake engine.
GUI	Interfaz gráfica de usuario.
Heightmap	Mapa de alturas que sirve para crear un terreno 3D en un motor gráfico. El negro marcaría la altura mínima y el blanco la máxima.
Id Software	Empresa estadounidense de desarrollo de videojuegos. Su sede está en Mesquite (Texas, EE. UU.).
Id Tech 1	Motor gráfico que id Software usó para sus videojuegos *Doom* y

	Doom II. Este motor gráfico también es usado por Hexen, Heretic, Strife y HacX, y otros juegos producidos por licenciatarios.
Incentive Software	Fue una compañía británica de software creada en 1983 por Ian Andrew y Ian Morgan.
Input manager	Gestor de entradas.
Jedi Engine	Motor gráfico en 3D que es muy poco conocido debido a que evolucionó con gran rapidez. Este fue empleado para el videojuego Star Wars Dark Forces.
Joystick	Palanca de mando. Es un dispositivo con una palanca especial para ser tomado de manera ergonómica con 1 mano, y una serie de botones integrados en la palanca que controlan en la pantalla los movimientos y acciones de los objetos en los videojuegos.
Lightmap	Mapa de luz. Simplemente consiste en añadir una segunda textura a todas y cada una de las caras existentes en una escena 3D. Esta textura contendría la luz que recibe cada cara.
Pixel	La menor unidad homogénea en color que forma parte de una imagen digital, ya sea esta una fotografía, un fotograma de vídeo o un gráfico.
POO	La **programación orientada a objetos** o **POO** (**OOP** según sus siglas en inglés) es un paradigma de programación que usa los objetos en sus interacciones, para diseñar aplicaciones y programas informáticos.
Popup	Ventana emergente.
Prefab	Tipo de *asset* que nos permite definir las propiedades de un objeto y además poder instanciar una o más veces el mismo.
Quake Engine	Motor de videojuego que fue desarrollado por id Software en 1995 para su videojuego *Quake*, lanzado en Junio de 1996.
Quaternion	Son una extensión de los números reales, similar a la de los números complejos.
Render to texture/Texture baking	Renderizar a textura. Permite crear mapas de textura basados en la apariencia del objeto al ser renderizado. Las texturas son “horneadas” (baked) junto con el modelo, y de esta manera, todos los reflejos, sombras, luces, etc., quedan fusionados en un solo mapa de texturas, aplicado al objeto.
Renderizar	Proceso de generar una imagen o vídeo mediante el cálculo de iluminación GI partiendo de un modelo en 3D.
Rigidbody	Cuerpo rígido. Al añadir a un objeto este componente se le dota de propiedades físicas.
Scroll	Se denomina **scroll**, **desplazamiento**, **rollo** o **voluta** al movimiento en 2D de los contenidos que conforman el escenario de un videojuego o la ventana que se muestra en una aplicación informática (por ejemplo, una página web visualizada en un navegador web).
Scrollbar	La **barra de desplazamiento** es un objeto de la interfaz gráfica de usuario mediante el cual una página de internet, una imagen, un texto, etc., pueden ser deslizados hacia abajo o arriba. También hay barras deslizantes horizontales, las cuales suelen aparecer a pie de página, cuando el contenido es demasiado ancho para la pantalla.
Scrollview	Es el contenedor que ofrece una barra de desplazamiento para el contenido que pongamos dentro de él.
Shader	Efecto gráfico asociado a una o varias texturas que ofrece el resultado visual final de un material renderizado.

Shooter	Videojuego de disparos.
Shuriken	Sistema de partículas muy poderoso que permite generar efectos gráficos vistosos, en su gran mayoría animados.
Streaming	Es la distribución de multimedia a través de una red de computadoras de manera que el usuario consume el producto, generalmente archivo de video o audio, en paralelo mientras se descarga. La palabra *streaming* se refiere a: una corriente continua (que fluye sin interrupción).
Tag	Palabra clave asignada a un dato almacenado en un repositorio.
Tasharen Entertainment	Es una pequeña empresa creada recientemente e independiente con sede en Toronto, Ontario.
Tile	Parte gráfica de cada videojuego que puede ser utilizada para completar partes de un fondo por medio de un tileset (conjunto de tiles).
Toolbar	Barra de herramientas. Es un componente de una interfaz gráfica de usuario mostrada usualmente en pantalla a modo de fila, columna, o bloque que contiene iconos o botones que, al ser presionados, activan ciertas funciones de una aplicación.
Tooltip	Descripción emergente. Herramienta de ayuda visual, que funciona al situar el cursor sobre algún elemento gráfico, mostrando una ayuda adicional para informar al usuario de la finalidad del elemento sobre el que se encuentra.
Transform	Componente que dota a un *gameobject* de una posición determinada (position), de una determinada inclinación (rotation) y de una determinada escala comparativa de tamaño (scale).
Trigger	Desencadenante o disparador.
Unity Technologies	Proveedor de las herramientas de desarrollo multiplataforma Unity.
Vector3	Vector en un espacio tridimensional (3D).

Estructura del CD

Detallo a continuación la estructura del CD que acompaña a esta memoria:



En la carpeta "Documentación" está el texto íntegro de la memoria en formato PDF y un resumen del PFC también en formato PDF.

En la carpeta "Programas" hay una carpeta: en la carpeta "Instalación Unity 3D con asistente" se guarda el ejecutable para instalar el motor Unity 3D mediante asistente, tal y como se describe en el apartado 4.1.

En la carpeta "Prototipo" hay una carpeta: en la carpeta "Ejemplo Shooter" se guarda el ejecutable del videojuego y una carpeta "Ejemplo Shooter_Data" donde se incluye toda la información del prototipo de videojuego.