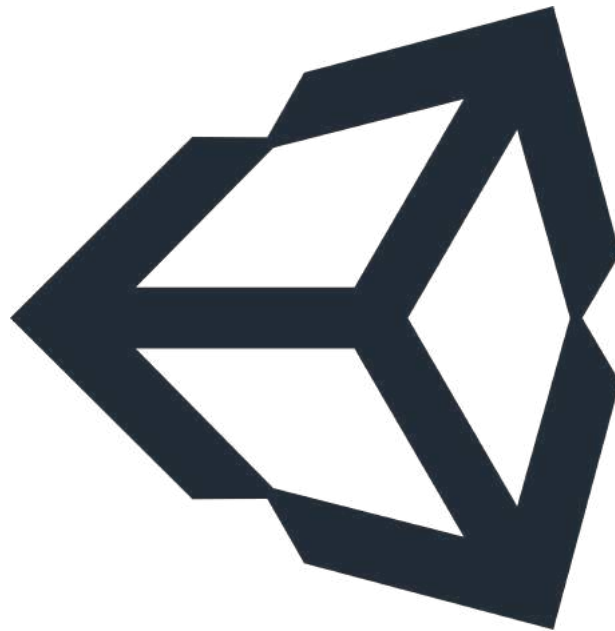




# DESARROLLO DE VIDEOJUEGOS EN UNITY



**Asignatura:** Trabajo Final.

**Autor:** Dn Miguel Ángel Vidal de Blanca.

**Tutor:** Dn Rafael Reina Ramírez.





# ÍNDICE:

<b>1.- INTRODUCCIÓN</b>	<b>5</b>
<b>2.- OBJETIVO</b>	<b>6</b>
<b>3.- INTRODUCCIÓN AL DESARROLLO DE VIDEOJUEGOS Y LA INDUSTRIA</b>	<b>7</b>
3.1.- Breve evolución histórica del desarrollo del videojuego	7
3.2.- Perspectiva actual del desarrollo de videojuegos	20
3.3.- Ciclo básico del desarrollo de un videojuego	22
<b>4.- TECNOLOGÍAS</b>	<b>28</b>
4.1.- C#	28
4.2.- Visual Studio Code	29
4.3.- Krita	29
4.4.- Linux MultiMedia Studio (LMMS)	30
<b>5.- MOTORES DE VIDEOJUEGOS</b>	<b>31</b>
<b>5.1.- ¿Qué es un motor gráfico o motor de videojuegos?</b>	<b>31</b>
5.2.- Breve evolución histórica de los motores gráficos	31
5.3.- Principales motores de videojuegos actuales	37
5.3.1.- UNREAL ENGINE 5	37
5.3.2.- GEMEMAKER STUDIO 2	39
5.3.3.- GODOT	40
5.3.4.- UNITY	41
5.3.- ¿Por qué escoger Unity?	43
<b>6.- UNITY</b>	<b>44</b>
6.1.- Características de Unity	44
6.2.- Entendiendo la interfaz	48
6.3.- GameObject y Components	50
6.4.- Scripting con C#	53
6.5.- Comportamiento espacio-temporal en un plano 2D	56
6.6.- Rigidbody y BoxCollider	58
<b>7.- ASSETS</b>	<b>62</b>
7.1.- Creación de elementos gráficos en 2D con KRITA	63
7.2.- Creación de recursos de sonido con LMMS	66
7.3.- Creando nuestros propios Assets en UNITY a partir de nuestros recursos	76
7.3.1.- Definiendo un Asset gráfico	77
7.3.2.- Definiendo un Asset de sonido	80
7.3.3.- Definiendo animaciones	82
<b>8.- DESARROLLO DE UN NIVEL DE VIDEOJUEGO DE PLATAFORMAS</b>	<b>93</b>



8.1.- Definiendo nuestros objetivos del nivel	94
8.2.- Gestión de Assets y animaciones	94
8.3.- Desarrollo del escenario	97
8.4.- Implementando nuestro Asset Player	105
8.5.- Implementando Assets de recolección	112
8.6.- Implementando comportamientos	119
8.7.- Definiendo condicionantes del nivel (Muerte - Fin)	124
<b>9.- GENERACIÓN DEL EJECUTABLE</b>	<b>127</b>
<b>10.- CONCLUSIONES</b>	<b>129</b>
<b>ANEXO.- BIBLIOGRAFÍA</b>	<b>130</b>



## 1.- INTRODUCCIÓN

A lo largo de este proyecto analizaremos la evolución del desarrollo de videojuegos y de la industria centrándonos en sus avances tecnológicos principales. Posteriormente comprenderemos mejor el funcionamiento y evolución de los motores de videojuegos más conocidos en la actualidad para centrarnos en Unity y en las tecnologías y herramientas externas que podremos utilizar para comenzar a crear nuestros propios recursos de manera totalmente gratuita. Finalmente utilizaremos todo lo aprendido para desarrollar un nivel de videojuegos de plataformas totalmente funcional.



## 2.- OBJETIVO

El objetivo principal de este proyecto es conocer un poco mejor cómo funciona el desarrollo del videojuego actual, su evolución y su puesta en marcha dentro del entorno de trabajo que ofrece la herramienta Unity, un motor para el desarrollo del videojuego muy usado en la actualidad tanto por empresas independientes (**Team Cherry, StudioMDHR o Moon Studios** ) como empresas de renombre en la industria (**SquareEnix, Activision Blizzard King, Game Freak o Nintendo**). Una vez definido el funcionamiento y las bases de **Unity**, haremos una **pequeña demostración** de como crear un nivel de un videojuego acción plataformas en PC para Windows y como compilarlo para crear un ejecutable distribuable.



### 3.- INTRODUCCIÓN AL DESARROLLO DE VIDEOJUEGOS Y LA INDUSTRIA

La industria del desarrollo de videojuegos actual dista bastante de sus inicios, tanto en el **público** al que se dirige como en el **proceso de desarrollo** y perspectiva de **negocio**. Durante este apartado echaremos un pequeño vistazo al **nacimiento de la industria** hasta el momento actual, así como los **principales cambios** que se han ido produciendo en el planteamiento y desarrollo de los videojuegos, el cual comprobaremos que va directamente unido a la percepción del usuario relativa a la **evolución gráfica** del medio y, del mismo modo, de los motores gráficos.

#### 3.1.- Breve evolución histórica del desarrollo del videojuego

Si nos vamos a las raíces originales del medio encontraremos que el primer software que puede considerarse un **videojuego** como tal (Esto es, un software de ocio y expresión que aúna la visualización gráfica sobre pantallas digitales y la interacción sobre éste gracias a mecánicas preprogramadas), encontraríamos el **OXO**, desarrollado por **Alexander S. Douglas** en **1952** en ordenador computacional **EDSAC** (*Electronic Delay Storage Automatic Calculator*), pero al ser un desarrollo personal y no puesto a la venta no podemos considerarlo realmente como parte de la industria.

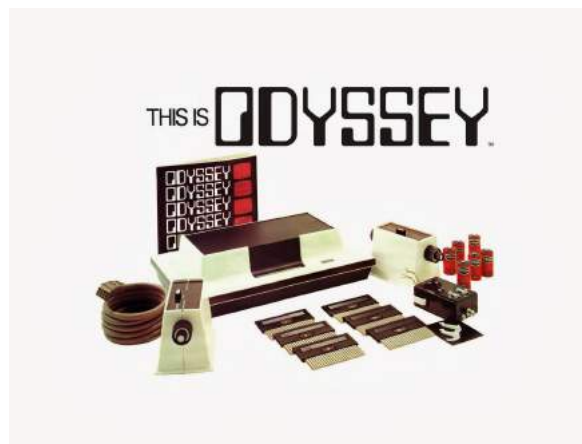
Un paso adelante en este planteamiento llegaría entre el año **1961** y **1962**, cuando **Steve Russell** y su grupo de compañeros desarrolladores presentaron un proyecto basado en los avances documentados de **S. Douglas** sobre el uso y potencial de **gráficos vectoriales** para la creación y dibujo de elementos gráficos mediante vectores y líneas, los cuales eran generados en base a **algoritmos reactivos** que se mantenían a la espera de la interacción del jugador.



Este proyecto fué denominado **Spacewar!** y saltó a la fama en revistas científicas británicas y de software como abanderado de un nuevo tipo de planteamiento de desarrollo del software, naciendo así el primer género reconocible en la industria: El **Shooter** o **Videojuego de disparos**. Pero aún con estos primeros pasos en el nacimiento de la industria

El reconocimiento de **Russell** y su equipo vino no solo de parte de revistas y medios de comunicación, sino de empresas de nueva creación, empresas de renombre y la industria armamentística. A destacar entre las empresas de renombre tendríamos a **Magnavox**, la cual se interesó por el proyecto de **Russell** y comenzó a investigar un modo de crear a un nicho de mercado para este nuevo tipo de productos, pero al no obtener

buenos frutos acabó contratando en el año **1966** a **Ralph Baer**, un reconocido inventor que daría pie al propio concepto comercial de **videojuego** creando la **primera consola doméstica**, conocida en sus primeros prototipos como La **Caja Marrón** (The Brown Box) y finalmente, en su lanzamiento durante el año **1972** como la **Odyssey** de Magnavox.



*Magnavox Odyssey, Magnavox (1972)*

Al ser un producto nuevo ciertamente hubo cierto público interesado, pero al requerir un exceso de **cualificación** para su instalación y comprensión de su ficha técnica de solución de errores comunes no llegó a mantenerse mucho tiempo en mercado, pero el propio interés de **Magnavox** llevó a empresarios menores a investigar por su cuenta ese nicho de mercado. Entre esos empresarios destacaron **Nolan Bushnell y Ted Dabney**, los cuales utilizaron también como base la documentación vectorial de **Spacewars!** (Y el aprovechamiento del propio **Nolan Bushnell** de una visita a las oficinas de Magnavox) para crear en el mismo año de lanzamiento de la **Odyssey** el primer **videojuego recreativo comercial** (también denominado **arcade**) propiamente reconocido: **Pong**.

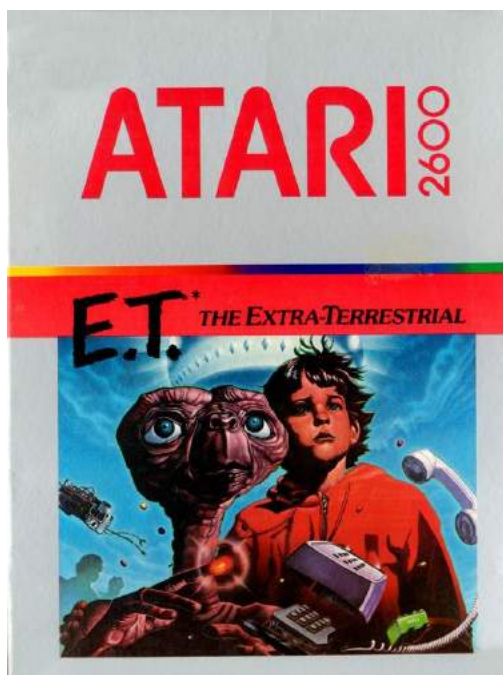


*Máquina de Pong, Ted Dabney y Nolan Bushnell*





El éxito de **Pong** es aún reconocido a día de hoy y supuso el inicio de la primera empresa únicamente orientada al desarrollo de videojuegos: **Atari**, a finales de **1972**. La fiebre por los productos de Atari se palpaba en las calles y muchas empresas empezaron a interesarse por hacerse un hueco en la industria del videojuego. El intento de muchas de estas empresas vino precisamente por informantes de dentro del círculo de confianza del propio **Bushnell**, el cual al enterarse de esta fuga informativa empezó a desconfiar de varios de sus compañeros y a denominarlos “**chacales**” por aprovecharse de sus avances en el medio e industria. Como principal gancho contractual, **Bushnell** ofrecía a las empresas con marcas reconocidas (Principalmente productoras de televisión, cine y literatura) buenos **intereses** por videojuego vendido así como un **rápido desarrollo y puesta en mercado**, de los cuales Atari también saldría muy bien beneficiada al publicarlos en sus propias consolas domésticas, las cuales se basaron en la tecnología ofrecida por **Magnavox** pero potenciando la jugabilidad con un controlador de juego de **2 botones y un Stick rotatorio**. De entre estas consolas la que más destacó fue la **Atari 2600**, la cual borró del mapa a cualquier otro intento de consola hogareña hasta la fecha y se sentó como el camino a seguir.



Los problemas de **Bushnell** se acumulaban como el polvo en el desierto al comprobar que con el paso de los años empezaban a llegar quejas y devoluciones masivas de sus productos. Esto se debía principalmente a que los contratos ofrecidos por **Bushnell** eran normalmente de muy corta duración, por lo que en muchos casos nunca llegaban a ser completamente probados antes de ser empaquetados y enviados a las tiendas. El fin de este auge dorado de la industria llegó durante las navidades de **1982**, fecha en la que se estrenó la película **E.T El Extraterrestre**, de **Steven Spielberg**. Nolan se aseguró de anunciar a bombo y platillo el lanzamiento del videojuego basado en dicha película, el cual fue desarrollado por una única persona (**Howard Scott Warshaw**), sin ningún tipo de comprobación de código y diseño y en un periodo inferior a 3 meses.

El exceso de producción de videojuegos para la **Atari 2600** supuso una reducción enorme en la calidad de estos, promoviendo la cantidad sobre la calidad, pero con la llegada del videojuego de E.T la mecha en la pólvora social no se hizo esperar. Los teléfonos no pararon de sonar durante las primeras semanas de **1983** en las oficinas de Atari y los periódicos sentenciaban su caída a corto-medio plazo.



Es precisamente por eso que a este año se le conoció como el “**crack**” del **83**, cuando **Atari** dejó de lado la industria de las videoconsolas y pasó a ser una empresa de videojuegos principalmente orientada a videojuegos **arcades** y de **recreativas**.

Como podemos ver, durante este inicio de la industria del videojuego el desarrollo tiene las siguientes características:

Plazos muy <b>cortos de desarrollo</b> , los cuales prácticamente nunca superaban los 6 meses.
Si el videojuego se programaba para placas arcades, normalmente se utilizan lenguajes de bajo nivel de forma directa. En lo que respecta al desarrollo para consolas caseras como la <b>Atari 2600</b> se utilizaban lenguajes de programación personalizados para interactuar con el hardware de la máquina, como por ejemplo el <b>Atari BASIC</b> , el cual era un lenguaje de programación desarrollado por <b>Atari</b> y basado en <b>BASIC</b> .
<b>No se realizaban pruebas concienzudamente</b> de código antes de la producción física del videojuego, sobre todo en el desarrollo para consolas caseras.
Se utiliza <b>poca mano de obra</b> y, en muchos casos, <b>poco cualificada</b> en el desarrollo del videojuego al ser una industria de poco trayecto.
Para el desarrollo de software en consolas caseras se utilizaba una consola personalizada y preparada para el desarrollo del software en cuestión. Estas consolas eran conocidas como <b>Proto-Kits</b> o <b>Kits de desarrollo</b> .
Los <b>productos eran de corta duración</b> (normalmente inferior a los 10 minutos), pero solían beneficiarse de un enfoque cíclico de juego, donde al terminarse se empezaba desde el principio pero aumentando la dificultad.
El público objetivo era amplio, dirigido a jóvenes y adolescentes.

Durante los años 70, la fiebre de **Atari** y sus máquinas arcade de recreativas alcanzó el otro extremo del mundo dejando una huella comercial en **Japón**, donde muchas empresas empezaron a interesarse en el desarrollo y producción de videojuegos. Encontramos ejemplos como **Nintendo**, **Services Games** (posteriormente conocida como **Sega Corporation**), **HudsonSoft** o **Enix**, las cuales desarrollaron la industria en el territorio nipón y, tras su merecido éxito, empezaron a expandir sus negocios fuera del país. Entre las empresas comentadas destacan **Nintendo y Sega**, las cuales establecieron una **filial** en **EEUU**, ya que al ser la cuna del videojuego pensaron que era un buen lugar donde aprender y crecer. Centrándonos en **Nintendo**, ésta estableció su filial en **Nueva York** a mediados de **1980**, donde planteaba un modelo de negocio principalmente enfocado a **juguetes interactivos con pantallas LCD** baratos de producir (Conocidas como **GAME & WATCH**), los cuales tuvieron mucho éxito en **Japón** y, mientras tanto, aprendía de **Atari** para crear su propia consola casera, la **Nintendo Family Computer System**, la cual se desarrolló en Japón y únicamente orientada al mercado japonés.



El éxito de esta consola fué rápidamente confirmado en todo el territorio japonés y supuso un gran avance en el desarrollo del videojuego, ya que permitía la **creación de recursos** de manera **externa** de modo que fuese posible su inserción posterior en el disco de juego, porque si, la **Family Computer System** de nintendo no utilizaba cartuchos de juego, sino **disquetes**.

El éxito de Nintendo fuera de su país al principio fué relativamente moderado, pero tras la caída de **Atari** y el **crack del 83** Nintendo aprovechó la herida de muerte del titán para plantear el negocio de modo que se orientase principalmente al público infantil, al cual ya tenía convencido con las **Game&Watchs**. Tras el éxito rotundo de su consola doméstica japonesa, la compañía decidió llevarla al resto del mundo dando nacimiento a la **Nintendo Entertainment System (NES)**, la cual utilizaba cartuchos similares a los que ya se utilizaban en la **Atari2600**.



La **NES** supuso un absoluto **renacer** para el desarrollo de videojuegos, ya que Nintendo ofrecía un kit de desarrollo sencillo de utilizar y que permitía trabajar directamente con **BASIC nativo** e incluía un **chip de sonido** independiente al **gráfico** (requiriendo en ambos casos el uso del lenguaje **ensamblador** para la gestión de la memoria y los recursos de ejecución, debido a las **limitaciones del motor de 8-bits de la máquina**), lo cual supuso un enorme avance en el desarrollo, ya que se simplifican enormemente para el equipo de desarrollo.

Otro de los enormes pasos dados por nintendo y la principal semilla de su éxito fué su **sello de calidad**, por el cual se comprometían a probar todo juego desarrollado para su consola, aliviando el descontento social provocado por **Atari** en el pasado y dando lugar a una nueva edad para la industria, donde ya no solo bastaba con gente poco cualificada sino que se empezaron a **incluir otras figuras en el desarrollo del videojuego**, como los **Diseñadores de niveles** (Como Shigeru Miyamoto, padre de Super Mario y Donkey Kong), **compositores** (Como Nobuo Uematsu, compositor principal de la saga Final Fantasy) y **artistas conceptuales de diseño** (Como Yoshitaka Amano). El éxito de **Nintendo** sacudió el mundo junto con el de **Sega**, dominando esta última parte del mercado americano y principalmente el europeo con su competencia a la **NES**, la Master System.



*Ilustración conceptual de Yoshitaka Amano para Final Fantasy I*

Si tuviésemos que analizar el desarrollo del videojuego durante ésta etapa podríamos destacar lo siguiente:

Siguen existiendo **plazos cortos de desarrollo**, pero rara vez supera el año.

Las **placas arcade** empiezan a manejar mejores recursos, pero siguen haciendo uso de **lenguaje ensamblador de 64K**. En lo que respecta al desarrollo de máquinas caseras como la **NES** y la **MASTER SYSTEM** utilizan lenguaje ensamblador, por ejemplo en el caso de la **NES utilizaba ensamblador del 6502**, un microprocesador de **8 bits** con un bus de dirección de 16 bits.

Las empresas de desarrollo y distribución, como **Nintendo**, **Sega** o **SNK**, comienzan a garantizar la comprobación de calidad de sus productos, mediante la estandarización de **sellos de calidad propios**.

Como las principales empresas de desarrollo no eran americanas no se vieron directamente afectadas por el crack del 83 en términos de relaciones laborales, por lo que aprovecharon el descenso de Atari para aumentar las **incorporaciones de nuevos trabajadores de diversas áreas, tanto digitales como artísticas**.

Con el aumento de los recursos en las nuevas consolas y placas arcades, los desarrollos comienzan a utilizar un **esquema narrativo lineal**, con un inicio y un fin marcado.

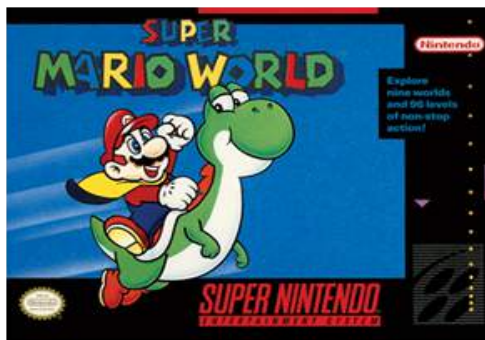
El desarrollo se vio fuertemente beneficiado del enfoque japonés, el cual aportaba enfoques más narrativos y donde se **apoyaba el desarrollo en otras áreas artísticas**, como el cine (nacen las primeras cinemáticas, aunque pretéritas) o la **música** (Nacen las primeras bandas sonoras reconocibles del medio) y surgen las primeras **plasmaciones culturales diferenciadoras**.

**Se siguen haciendo uso de Kits de desarrollo** para videojuegos de consolas. En estos casos los Kits se empiezan a apoyar en programas externos para la incorporación de nuevos recursos, como por ejemplo herramientas de creación de entornos y música.

**El público principal eran jóvenes y niños** en el caso del mercado de videoconsolas, dejando el mercado arcade enfocado a jóvenes y adolescentes.

Es precisamente a partir de este momento cuando la industria del videojuego comienza a tomar un nuevo rumbo. Este puede ser enfocado fácilmente desde 3 perspectivas:

- El **desarrollo de videojuegos domésticos para consola** seguirá el mismo camino marcado anteriormente, aumentando el rango de edad a finales de los 80, principios de los 90, hasta los adolescentes. El enfoque de desarrollo seguirá enfocándose en **mejoras visuales** y en calidad de los recursos de apoyo anteriormente mencionados, llegando a su culmen en el desarrollo **2D** con los lanzamientos de la **SNES** por parte de **Nintendo** y de la **Sega Megadrive** por parte de **Sega**.



*Super Mario World, SNES (1990)*



*Sonic the Hedgehog 2, Megadrive (1992)*

- El **desarrollo de aplicaciones informáticas para el ocio** se comienza a extender por todo el globo, llegando a **tierras europeas** donde surge la era dorada del desarrollo de videojuegos para microordenadores, donde **España e Inglaterra** son sus principales baluartes, destacando obras como **La Abadía del Crimen** (Desarrollado por Paco Menéndez y Juan Delcán en **1987** para la **ZX Spectrum** de Sinclair), **Jetpac**, **Sabre Wolf** o **Knight Lore** (Siendo estos últimos desarrollados por **Ultimate Play de Game**, empresa británica creada por los hermanos **Stamper** y que posteriormente se convertiría en la talentosa **Rare**).





*La Abadía del Crimen,  
ZX Spectrum (1987)*



*Sabre Wolf,  
ZX Spectrum (1984)*



*Knight Lore,  
ZX Spectrum (1984)*

- El **desarrollo de videojuegos arcades** avanzaría como una fuerza imparable sin atisbo de resistencia, mejorando sus ingresos y llamado a su desarrollo a muchas empresas de nueva creación y a directores talentosos (Los cuales generalmente trabajaban para **Sega** o **SNK**) que demostraron un enorme talento en el medio, como sería el caso de **Yu Suzuki**, el cual fue el principal artífice de varios de los mayores éxitos del momento en las recreativas, como sería el caso de **After Burner**, **Out Run** o **Space Harrier**.



*After Burner, arcades (1987)*



*Out Run, arcades (1986)*



*Space Harrier, arcades(1985)*

El siguiente paso al frente dado en la industria del videojuego y que realmente supusiera un cambio en el desarrollo de estos surgiría con la llegada de los **discos compactos** para el almacenamiento de datos en formato digital y con una capacidad de **700MB**, los cuales fueron la evolución natural de los anteriores **Laser Disc** desarrollados conjuntamente por **Philips** y **Sony**.



Es precisamente a **finales de los 80, principio de los 90** cuando **Sony**, que saltó a la fama en el mercado por su hardware de sonido y fotografía principalmente, realizó un **compromiso contractual** con **Nintendo** para el desarrollo de una nueva consola casera que dejase de lado los cartuchos por el uso de CDs. El acuerdo se confirmó por ambas partes y se comenzó el desarrollo de la **Nintendo PlayStation**, liderado por **Ken Kutaragi** (Ingeniero de Sony). Fue entonces cuando en **1991**, en el mismo momento en que se iba a hacer el anuncio de la consola y su unión con **Sony** a los medios, cuando **Hiroshi Yamauchi** (Presidente de Nintendo) anunció a traición la unión con **Phillips** para la creación de un nuevo standard de hardware interactivo, el CD-i.

Este hecho quebró la confianza entre Sony y Nintendo y la empresa conocida por su hardware de sonido se propuso derrocar a la **Gran N** lanzando 3 años más tarde la conocida como **Sony PlayStation**, la cual destacó con un amplio catálogo apoyado por contratos primarios, secundarios y terciarios. Es por entonces que en la industria fueron surgiendo las conocidas **Parties** de Desarrollo:



1. **First Party:** Empresa de desarrollo que pertenece a la propia empresa de Hardware. Un ejemplo de esa época sería el nacimiento de la saga **Gran Turismo**, desarrollado por **Polyphony Digital**, empresa perteneciente a la propia **Sony**, o **Banjo Kazooie**, desarrollado por **Rare**, que en ese momento pertenecía a **Nintendo**.



*Gran Turismo, Sony PlayStation (1997)*



*Banjo Kazooie, Nintendo 64 (1997)*



2. Second Party: Empresa de desarrollo que una empresa distribuidora contrata para el desarrollo de software concreto y enfocado al hardware de la empresa contratante en exclusiva. En este campo encontramos muchos ejemplos, como el desarrollo de **Silent Hill** por parte de **Konami** y en exclusiva para la **Sony PlayStation**, o el caso de **Castlevania: Legacy of Darkness**, también desarrollado por **Konami** pero esta vez en exclusiva para la **Nintendo 64** (Consola desarrollada por Nintendo tras el fracaso comercial de la **CD-i** y que volvió al uso de **cartuchos** para videojuegos).



*Silent Hill, Sony PlayStation (1999)*



*Castlevania: Legacy Of Darkness, Nintendo 64 (1999)*

3. Third Party: Empresa de desarrollo contratada por empresas distribuidoras pero con las que **no se firman acuerdos de exclusividad**. Un buen ejemplo sería **Ubisoft**, empresa francesa que durante los años 90 y 2000 destacó por su **Rayman 2: The Great Scape**.



*Rayman 2: The Great Escape, Sony PlayStation (1999)*



*Rayman 2: The Great Escape, Nintendo 64 (1999)*

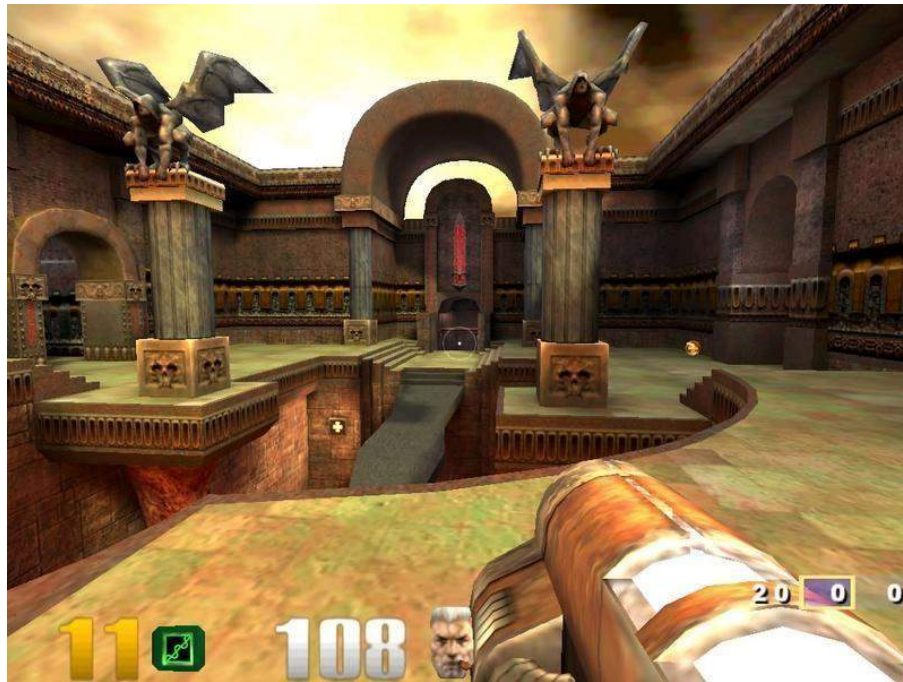




La incorporación del CD permitió enormes avances en el desarrollo del videojuego, permitiendo la incorporación de **bandas sonoras de alta calidad**, **doblaje** de personajes y almacenamiento de **cinemáticas** y **entornos pre-renderizados**, elementos que aportan una enorme calidad y empaque al producto y que fué la seña de identidad que aún mantiene **Sony PlayStation** a día de hoy, promoviendo desarrollos de alto presupuesto y con **enfoque cinematográfico**. Podemos decir por tanto que el desarrollo de videojuegos se destacaba en este momento por:

Los plazos de desarrollo pasan a tener un <b>periodo mínimo de un año a nivel general</b> .
Búsqueda de productos diferenciadores que supusieron una <b>victoria de la calidad sobre la cantidad</b> .
<b>Aumento de los costes de producción</b> , debido a la necesidad de contratación de profesionales de otros medios para recursos complementarios al producto jugable, como las cinemáticas, pre-renderización de escenarios, doblaje de escenas y personajes.
<b>Se refuerza el uso de kits de desarrollo para videojuegos caseros</b> , implementando en ellos compatibilidades directas con herramientas complementarias (normalmente de licencia propietaria). La industria del desarrollo en ordenadores empieza a aplicar este sistema de "kits de desarrollo" y comienzan a aparecer los <b>primeros motores de desarrollo de videojuegos multiplataformas</b> , como sería el caso de <b>Freescape Engine</b> (Incentive Software, 1986) o el <b>Wolfenstein 3D Engine</b> (Creado por <b>John Carmack</b> en 1991 para el desarrollo de su homónimo juego).
El auge del desarrollo del videojuego en PC dio pie a un renacimiento de la programación orientada a este sector, por lo que muchos interesados comienzan a estudiar lenguajes de alto nivel, como <b>C</b> y lenguajes <b>multiparadigmas</b> (LISP) para el desarrollo, lo cual comienza a estandarizar el uso de este tipo de lenguajes en el medio.
<b>Se busca preferiblemente el desarrollo de videojuegos en 3D</b> , utilizando como herramienta para la captación de nuevos e interesados clientes. Es precisamente este cambio uno de los que acabarán propiciando un alza en la industria y promoviendo un incremento de empresas de desarrollo, al existir pocos productos con esta tecnología. Para conseguir este desarrollo en 3D se empiezan a utilizar <b>procesadores más potentes</b> , como el <b>CPU R3000 MIPS de 32 bits</b> , utilizado por la <b>Sony PlayStation</b> o el <b>RISC CPU R4300i de 64 bits</b> utilizada por la <b>Nintendo 64</b> .

El desarrollo de la industria siguió su curso sin contratiempos hasta finales del año **2000**, el cual supuso un enorme salto técnico en el desarrollo de videojuegos con la llegada de tecnologías de **renderización zonal** en **entornos 3D**, permitiendo la creación de **mundos más amplios**, el manejo de **luces volumétricas**, efectos de **partículas** y la implementación de **físicas** más realistas gracias a la llegada de **librerías** que ofrecían la posibilidad de establecer y modificar la aplicación de **leyes físicas** a **componentes** del entorno. Estos avances son posibles gracias a la evolución de los motores de desarrollo, donde destaca el **Unreal Engine 1**, creado en **1998** por **EPIC** en respuesta a los motores 3D desarrollados por **John Carmack**, donde destacaba también su motor **Id Tech 3** (Desarrollado principalmente para **Quake 3: Arena**).



*Quake III: Arena, idSoftware (1999)*

Por su lado en el mercado de consolas domésticas como la **PlayStation 2** de Sony, la **GameCube** de **Nintendo** y la **Xbox** de **Microsoft** destaca el uso de las novedades comentadas previamente pero agregando una implementación interesante: **La posibilidad del juego en línea** desde dicho hardware, aunque en este momento es aún una implementación poco fiable para el usuario y, por ello, de baja tasa de uso. Durante ésta generación de consolas destacó además el uso del **DVD** para el almacenamiento de software, el cual supuso un enorme avance para la lectura y almacenamiento de recursos requeridos para las nuevas tecnologías. Este dato es de enorme interés, ya que por entonces un lector de **DVD** nuevo podría llegar a costarle al público incluso más que la propia consola, lo cual supuso un plus de ventas enorme tanto para la **PlayStation 2** como para la **Xbox**. Nintendo por su parte utilizó **Micro DVDs** para el almacenamiento de juegos de **GameCube** en pos de evitar la **piratería** a etapas tempranas de la llegada al mercado de su consola, por lo que **Sony** y **Microsoft** le comieron terreno.



*Xbox, Microsoft(2001)*



*PlayStation 2, Sony (1999)*



*GameCube, Nintendo (2001)*



Con la llegada del año **2006** llegó una nueva generación de consolas, cuyos baluartes más destacados eran la **PlayStation 3** de **Sony** y la **Xbox360** de **Microsoft**. En este momento la industria de las consolas decidió centrar esfuerzos en **potenciar el juego en línea** como plus de interés para el software producido, de modo que la gran mayoría de videojuegos producidos por las principales empresas tenían componente multijugador en línea.



La única **excepción** a esta afirmación viene de la mano de **Nintendo**, la cual decidió apostar por un público más general sacando al mercado la sucesora de la Gamecube, la **Nintendo Wii**.

Ésta se centraba en ofrecer un **hardware barato**, **menos potente** que la competencia y en el uso de un controlador especial de juego que hacía uso de **captación de movimientos** mediante el seguimiento de **luces infrarrojas**. El motivo de ambas decisiones se hizo notar desde el principio:

- Con un hardware más barato se aseguraban de mayores ventas, tanto por parte del público habitual de Nintendo como de nuevos compradores. Estamos hablando de una diferencia de precio muy destacable ya que, en su salida al mercado español, la **Nintendo Wii** costaba **249€** mientras que la **PlayStation 3** salió al mercado con un precio inicial de **599€**.
- Con la llegada de su **nuevo control** por captación de movimientos (**Wiimote**) se buscaba ofrecer un estilo de juego que **no requiriese de una compleja asimilación** de botones en pantalla, sino que la mayoría de los juegos populares se podrían jugar agitando el mando, apuntando al televisor y pulsando poco más de 3 botones. En esencia se buscaba una simpleza en el manejo, que diera pie a aumentar el público potencial y llegar a clientes menos experimentados en el medio.
- Al ofrecer un **hardware menos** potente pero conocido por los desarrolladores de la época se conseguía un flujo de producción de videojuegos mayor que la competencia y a menores costes de producción y tiempo de desarrollo.

La industria durante esta etapa siguió el flujo de desarrollo esperable hasta el año **2013**, aunque como se ha comentado antes se potenció el **juego en línea** para el software producido. Esta decisión vino acompañada de un apoyo a nuevos **desarrolladores independientes** y el enfoque de comercio del software en PC: La posibilidad de compra de videojuegos por **claves digitales**. El conjunto de estos 3 elementos fué el pistoletazo de salida para fomentar la publicidad indirecta de las compañías, gracias al auge de nuevas plataformas de expresión como **Youtube**, **Vimeo** o los **Videoblogs**.



### 3.2.- Perspectiva actual del desarrollo de videojuegos

Desde el año **2013** hasta la actualidad el desarrollo de videojuegos se ha visto afectado por los cambios sociales y el cambio del ocio digital. Con la llegada en ese año de consolas como la **PlayStation 4**, **Xbox One** y la **Nintendo Wii U** y el auge de los videojuegos para dispositivos móviles, la industria se ha centrado en los siguientes puntos:

**Fomentar la venta del producto digital sobre el físico**, de modo que los ingresos por venta se incrementan enormemente al eliminar los costes de distribución.

Para potenciar las ventas digitales se publicita mucho la creación de contenido post-lanzamiento por descarga (**DLC o DownLoadable Content**), los cuales eran de **precios reducidos** pero aprovechaban los recursos del videojuego ya lanzado para poder refrescar ventas y reducir huecos de novedades de cara al público.

El nuevo hardware disponible ha dado pie a la creación de **productos de altos valores de producción** y que requieren del manejo de diversas **herramientas de modelado, iluminación, rasterizado gráfico y manejo de memoria y recursos de gran peso**.

El **tiempo de producción** de nuevos videojuegos de las principales empresas de desarrollo importantes **ha aumentado enormemente llegando desde los 3 a los 6 años**. Teniendo presente que cada generación de videojuegos consta de entre 6 a 7 años, el lanzamientos de grandes producciones se ha reducido enormemente.

Para paliar el problema del tiempo de producción, desde las principales empresas de hardware en videojuegos se fomenta la **captación de desarrolladores independientes**, los cuales ofrecen productos con menores tiempos de desarrollo y reciben *feedback* directo del público.

El auge del videojuego para dispositivos móviles vino de la mano con los **teléfonos inteligentes o Smartphones**, los cuales contaban con recursos cada vez más idóneos para el desarrollo del medio. El desarrollo de **videojuegos móviles** se ha visto reforzado por las semillas plantadas por Nintendo durante la vida útil de la **Nintendo Wii**, donde se fomentaba la creación de productos para todo el público y que no requiriese de experiencia previa al juego. A este tipo de público se le conoce comúnmente como público **Casual**.

El aprovechamiento de las nuevas plataformas de expresión digital ha fomentado la creación de un nuevo tipo de producto en la industria: El videojuego **Freemium o Free to Play**, que consiste en ofrecer acceso gratuito al videojuego para todo público, pero limitando ciertos aspectos del contenido a partir de **micropagos internos**. Actualmente la mayoría de empresas desarrolladoras independientes tienen el foco comercial en la creación de este tipo de productos.





A finales del año 2020 llegó la nueva y actual generación de videojuegos con el nacimiento de la **Xbox Series X/S** de **Microsoft**, la **PlayStation 5** de **Sony** y la **Nintendo Switch** de **Nintendo** (excepción especial a mencionar, ya que salió al mercado 3 años antes, pero mantiene el enfoque de la industria del 2013). Ésta generación se ha visto principalmente marcada por el **uso estandarizado del almacenamiento SSD**, el cual ofrece altas velocidades de lectura y escritura, tiempos de producción un poco más amplios y el enfoque directo en el uso de **tecnología en la nube** para el **Streaming** de servicios, videojuegos (destacándose en la actualidad mucho el uso de servidores **Azure** de Microsoft para el alojamiento de estos servicios, en las 3 compañías principales) y otros productos digitales como películas, música y aplicaciones externas al videojuego.

Destaca por tanto la potenciación del servicio actual de **PlayStation** y **Xbox**, el servicio **PlayStation Now** y **Xbox Game Pass** respectivamente, los cuales ofrecen mediante suscripción mensual un amplio catálogo de juegos jugables desde la nube. Es decir, sin la necesidad de adquirir el hardware de la compañía. Esto supone una **clara dirección en el desarrollo de videojuegos digitales frente a los físicos**, mismo camino que está comenzando a recorrer la sociedad actual dejando de lado el comercio local y acudiendo cada vez más al comercio online y al uso del **Cloud Computing** frente a la instalación del software.



*PlayStation Now, Sony (2014)*



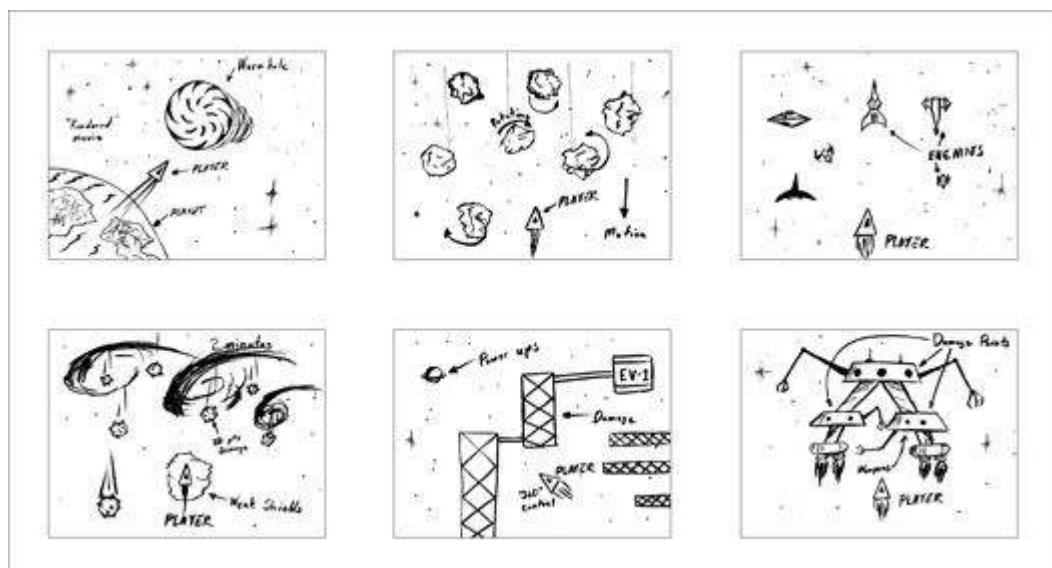
*PlayStation Now, Sony (2017)*

### 3.3.- Ciclo básico del desarrollo de un videojuego

Un videojuego no deja de ser un software, por lo que al igual que todo software cuenta con un ciclo de desarrollo concreto. En este caso el ciclo de vida es un tanto diferente al englobar tantos matices y campos a tener en cuenta, pero comúnmente suele dividirse en 7 fases o etapas principales:

#### 1. Fase de Concepción:

Durante esta primera fase nos dedicamos única y exclusivamente a tratar y proponer ideas que den pie a poder plasmar un guión gráfico concreto (Comúnmente conocido como **Storyboard**). No solo hablamos de ideas gráficas como ideación de personajes o ambientes, sino también de las mecánicas jugables principales que forman el esqueleto reconocible del producto. Con esta proposición y plasmación de ideas buscamos por ejemplo: Determinar el **género** del juego (Plataformas, Acción, Disparos, Simulación...), su **dimensionalidad** (2D o 3D), su **estructura jugable** (Por niveles separados, niveles interconectados o mundo abierto) y sus **mecánicas principales de juego** y de **diseño** (Una mecánica de diseño muy común en el desarrollo de videojuegos móviles, por poner un ejemplo concreto, sería la implantación de una **tienda virtual** que permita al jugador, mediante **micro-transacciones**, avanzar más rápido u obtener ciertos beneficios).

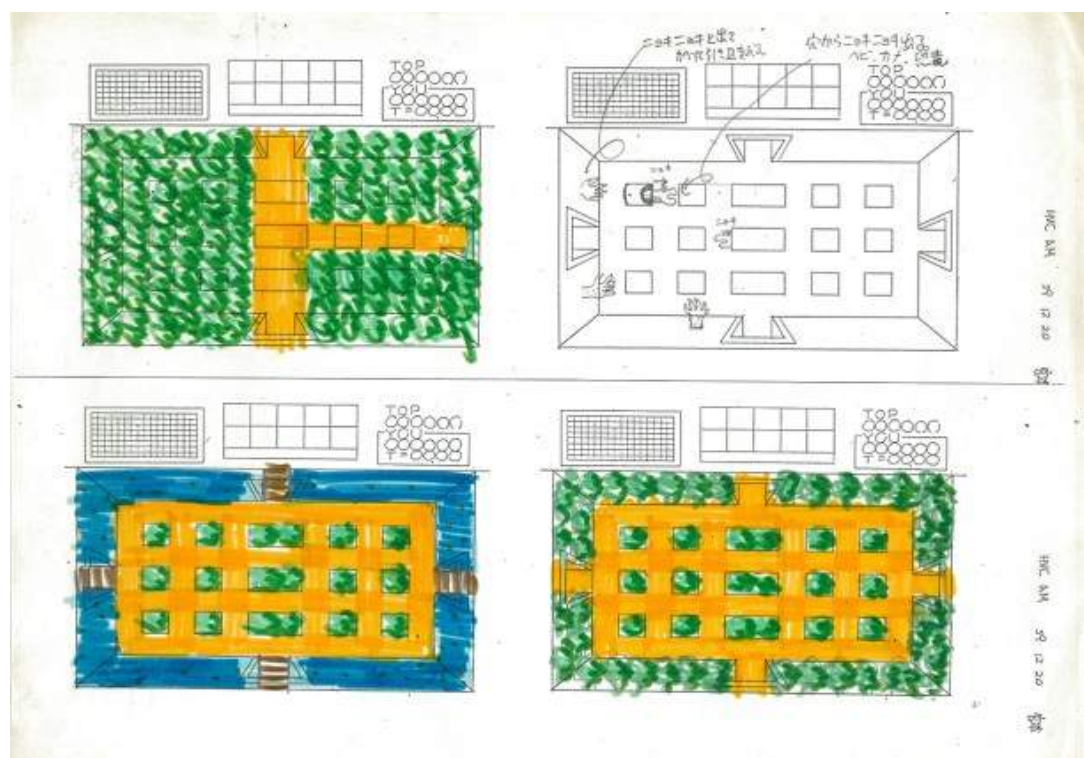


Storyboard para videojuego de naves

## 2. Fase de Diseño:

Al inicio de esta fase se escoge el **Storyboard** y los elementos más definitorios del videojuego planteado que se hayan logrado durante la fase de concepción y comienzan a concretarse en múltiples variantes. Un ejemplo muy común de estas variantes sería la **creación de los recursos principales** como los **modelos** de los personajes, enemigos, objetos del videojuego o los propios recursos **musicales**, donde normalmente se acaban realizando diversas versiones elegibles del mismo elemento para poder ir descartando ideas hasta lograr la orientación correcta.

Lo mismo ocurrirá con las **mecánicas** jugables, no jugables, la **interfaz** de usuarios, los **eventos**. Durante esta etapa se escoge también el **lenguaje de programación a utilizar**, el **motor** de videojuegos y **herramientas principales**, las **plataformas** de destino (Consolas, Ordenadores, Móviles o Tablets) y sus capacidades **online**. Al acabar esta fase se debe lograr producir un **Documento de Diseño** que recoja las conclusiones obtenidas. En el caso de escoger como plataforma de destino el ordenador, móviles o tableta es muy importante tener en cuenta el **hardware** a utilizar para la producción del videojuego, de modo que puedan concretarse los **requisitos mínimos y altos** para la correcta ejecución del producto.



*Diseños de escenarios utilizadas en The Legend Of Zelda, NES (1986)*

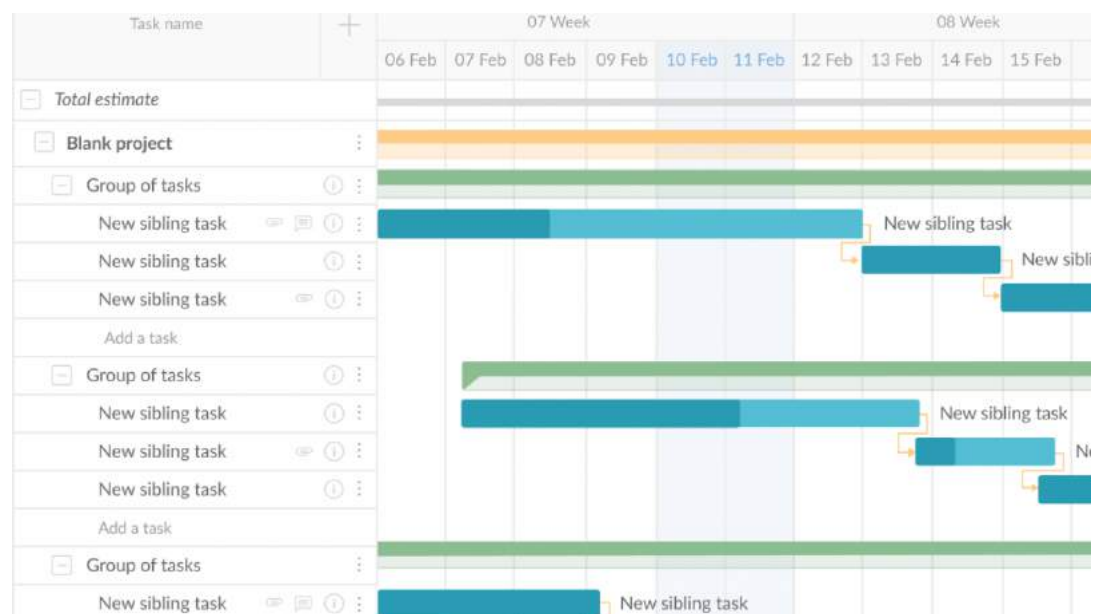


### 3. Fase de Planificación:

Esta es sin duda una de las etapas más importantes de todo el ciclo de desarrollo. El objetivo principal es identificar las tareas a desarrollar de un modo que el trabajo quede repartido entre los diferentes miembros del equipo de desarrollo y se establezcan unos plazos concretos de entregas y reuniones para seguimiento todo el proceso.

Para la planificación se deberá tener en cuenta no solo el **Documento de Diseño** obtenido en la fase anterior, sino además el **presupuesto** disponible, los posibles cambios de diseño producidos por una falta de concreción durante la fase anterior. También es muy común durante esta fase **externalizar** diversas partes del proyecto, ya sea para **ahorro de costes** en formación o por la imposibilidad de contar con los equipos o profesionales adecuados.

Normalmente se utilizan hoy en día **Tablas de Gantt** para la plasmación de las diversas tareas comentadas durante esta fase, que a su vez suele dividirse en diversas subfases de planificación en el caso de proyectos más complejos (Sobre todo cuando van a utilizarse **kits de desarrollo** para consolas en una empresa o grupo de desarrollo que nunca lo ha utilizado, o cuando van a aplicarse técnicas más modernas de renderizado, iluminación o en general de métodos de uso del hardware poco conocidos.)



*Ejemplo de Tabla de Gantt*





#### 4. Fase de Producción:

Durante esta fase se llevan a cabo todas las tareas planificadas y buscando respetar siempre los tiempos marcados. Para la elaboración de las tareas se debe tener siempre presente el documento de diseño para aquellas tareas que no hayan sido claramente especificadas o definidas en la fase de planificación y el principal fin es obtener una versión inicial o prototípica de todo el producto. A esta versión se la conoce como **Bronce Game**, la cual se define como una fase en la que las bases del videojuego y sus mecánicas ya se encuentran ajustadas a los objetivos, pero pueden existir errores de ejecución no resueltos.

Ciertamente en aquellos casos en los que se quiere exponer un componente multijugador del producto a un público concreto para recoger el feedback de mejoras, suele crearse un prototipo previo más concreto conocido como **Pre-Alpha**. El objetivo principal de este tipo de prototipos es la exposición a pruebas de tensión de red y se sobrecargas de peticiones de usuarios. Una vez obtenidos los resultados se aplican los cambios promovidos por el feedback hasta obtener el prototipo **Bronce Game**.

#### 5. Fase de Pruebas:

Ahora que contamos con el **Bronce Game** es necesario acudir a un equipo de pruebas y calidad que se encarguen de probar el prototipo y de apuntar las mejoras y errores que estimen. Es un periodo muy importante y que debe respetarse para evitar mayores problemas a futuro. Una vez se hayan recogido suficientes datos se procede a retroceder a las fases de **Planificación y Producción**. En el caso de requerirse nuevas adhesiones al producto que no hubiesen sido tenidas en consideración hasta esta fase de pruebas se deberá volver a la fase de diseño.

El objetivo final de todo el proceso de pruebas inicial es obtener una versión perfectamente finalizable de inicio a fin, aunque normalmente sin el contenido cinemático aplicado, ni el código refinado. A esta versión se la conoce como **Silver Game o Beta Game**.



## 6. Fase de Distribución y Marketing:

Esta fase se divide en dos partes bien diferenciadas:

### a. **Fase Interna:**

Durante la fase interna de la distribución el equipo o personal de calidad se pone en contacto con un grupo concreto de usuarios para que prueben el estado actual del prototipo **Silver Game**. A este grupo de usuarios se les conoce normalmente como **Focus Testing Group** y se componen de un grupo arquetípico del público de destino. Para la selección de este **Focus Testing Group** se debe respetar la categoría de edad fijada en el Documento de Diseño, el cual debe respetar la legislación sobre información de productos digitales interactivos (En Europa se conoce como la **Pan European Game Information** o **PEGI**).



*Etiquetas de clasificación por edades según el PEGI*

Una vez se haya obtenido el feedback del **Focus Testing Group** se procede a su estudio por parte del equipo de calidad y a la solventación de errores no resueltos. Una vez el producto pase el filtro de calidad establecido por el equipo, se procede a la postproducción audiovisual, donde se refinan los recursos gráficos, musicales y audiovisuales hasta obtener lo que se conoce como **First Version** o **Gold Game**, lo cual identifica al producto como finalizado para el uso del público y listo para ser distribuido.

### b. **Fase Externa:**

Esta fase se realiza a la par que la interna y consiste en entablar relaciones con distribuidoras de videojuegos y grandes comercios así como empresas de impresión, empaquetado, ensamblaje y otros servicios requeridos como el transporte o almacenamiento. Asimismo se deben entablar relaciones y actuaciones que favorezcan una ventana de ventas y una fecha de salida del producto al mercado concreta, junto con su precio y disponibilidad. El objetivo de esta fase es recoger el **Gold Game** y distribuirlo al mercado de destino.



## 7. **Fase de Mantenimiento:**

Durante esta fase el videojuego ya se encuentra a disposición de los usuarios finales y se establece un periodo serio de tiempo durante el cual se efectúa el **mantenimiento del software en venta**, así como los servicios online y de multijugador. Esta es sin duda una de las **fases más largas de todo el proceso** y en la actualidad es muy utilizada por los equipos de desarrollo para implementar todas aquellas funcionalidades prometidas por el equipo de marketing y que no han sido posible incorporarlas por limitaciones de tiempo y contratos de lanzamiento.



## 4.- TECNOLOGÍAS

El desarrollo del videojuego requiere de la aplicación conjunta de una serie de tecnologías que van a permitir la definición de comportamientos y recursos necesarios para alcanzar los objetivos previstos. A la hora de abordar **UNITY** las tecnologías que deberemos conocer previamente serán:

### 4.1.- C#

**C#** es un **lenguaje de programación orientado a objetos** desarrollado inicialmente en el año **1999** por un equipo interno de desarrollo e investigación de **Microsoft** dirigido por **Anders Hejlsberg**, con el objetivo de desarrollar un lenguaje principal para **.NET** (Framework para el desarrollo de aplicaciones creado también por Microsoft) y que surgió como una **evolución natural de C y C++**, ofreciendo un acercamiento más accesible a nuevos programadores.



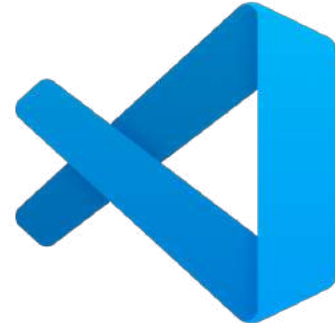
El apoyo de la compañía y la confianza de sus creadores promovieron el enfoque **multiplataforma** del lenguaje, aumentando el interés de la comunidad y uso en otros ámbitos gracias a la aprobación como estándar por la **ECMA-334** en diciembre del **2002**.

La principal utilidad de **C#** en nuestro proyecto será la definición de comportamientos de nuestros recursos dentro del nivel desarrollado. Esta definición de comportamientos recibe el nombre de **Script**, un documento donde declaramos los recursos que vamos a utilizar y el comportamiento que deberán seguir. Un ejemplo sería un documento que definiese el movimiento del recurso Player dentro del nivel o escenario del videojuego.



## 4.2.- Visual Studio Code

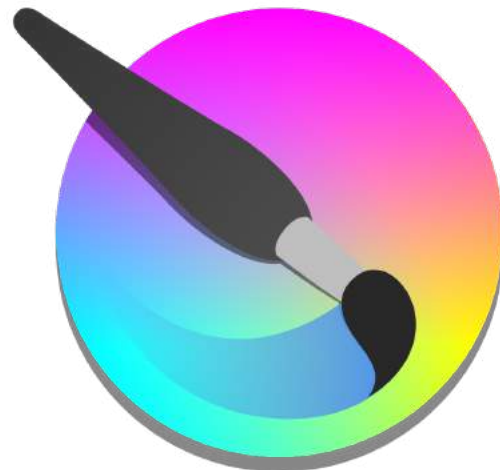
**VSC** es un editor de código desarrollado por **Microsoft** en **TypeScript/JavaScript** y con licencia **MIT** freeware (De software libre permisiva para el usuario). Lo utilizaremos principalmente como el **editor principal** para la creación de **Scripts** en **C#** debido a que ofrece una integración nativa con **Unity**, lo cual nos va a facilitar enormemente la gestión de nuestros recursos.



## 4.3.- Krita

**Krita** es un editor de imagen y color para generar recursos gráficos rasterizados (autodefinidos, sin compromiso de renderizado material), de licencia libre **GNU GPL**. Fue creado a mediados de **2005** por el equipo de desarrollo sueco de **Kimage Shop**, un software para la manipulación de imágenes, como evolución natural de éste.

En esos momento el mercado de la edición gráfica estaba principalmente dominado por **Adobe** con su conocida herramienta **Photoshop**, pero el equipo de **Kimage Shop** buscó un enfoque más liviano y legible para el público general, definiendo las bases de su herramienta y su posterior cambio de nombre a **Krita**.



Con el uso de esta herramienta conseguiremos crear recursos gráficos en 2D, tanto de diseño **conceptual** como de diseño **práctico** (Comúnmente conocidos como **Sprites Sheets**), los cuales suelen dividirse en cuadrículas equidistantes.



Ejemplo de **Sprite Sheet** utilizado para **The Legend of Zelda Link's Awakening**



#### 4.4.- Linux MultiMedia Studio (LMMS)

Estación digital de trabajo de sonido con licencia libre GPL multiplataforma, aunque inicialmente desarrollada para **Linux**. Es un Software creado en el año **2004** con pretensiones de mejorar las prestaciones ofrecidas por **FL Studio**, no solo a nivel de edición sino de accesibilidad a nuevos usuarios. Lo utilizaremos principalmente para generar proyectos de **sonido** fácilmente administrables que nos permitan crear **recursos de audio** para nuestro videojuego. **LMMS** tiene un enorme soporte de la comunidad y además es fácilmente integrable con **Unity**, por lo que es la opción ideal a tomar.





## 5.- MOTORES DE VIDEOJUEGOS

A lo largo de este apartado veremos lo que se considera en la actualidad un motor de videojuegos/motor gráfico, su evolución y haremos un breve análisis de los principales motores usados en la actualidad.

### 5.1.- ¿Qué es un motor gráfico o motor de videojuegos?

Actualmente un **motor de videojuegos** es en esencia un motor de ejecución y renderizado gráfico que viene acompañado de un conjunto de herramientas integradas, como pueden ser **editores de códigos**, software para la **gestión y organización de recursos**, **middlewares** y **scripts pre-programados**. Estas herramientas buscan satisfacer dos necesidades principales durante el proceso del desarrollo:

- **Centralizar el flujo de trabajo**, de modo que se integran la mayor parte de herramientas principales dentro del propio motor.
- **Ofrecer sistemas integrado para el manejo de entornos gráficos** preparados para:
  - **Estructuración, iluminación, diseño** y desarrollo de **entornos** en 3D y 2D.
  - Creación de animaciones basadas en **Assets** 2D y 3D.
  - Manejo de **cámara, posicionamiento** y **scripts de comportamiento focal**.
  - Motor de **físicas** y comportamiento de **flujos**, principalmente orientado al desarrollo 3D.
  - Sistemas para la **renderización** zonal y la **carga/descarga de recursos** controlada.

Como podemos ver la mayor parte de las prestaciones que ofrece un motor de videojuegos busca ofrecer una abstracción al desarrollador principalmente orientada a satisfacer sus necesidades de manejo y procesamiento gráfico de recursos (Es precisamente por ello que a los **motores de videojuegos** se les conoce también como **motores gráficos**), ya que como veremos en el siguiente punto, este apartado fué un verdadero quiebro de cabeza para las principales desarrolladoras en su momento.

### 5.2.- Breve evolución histórica de los motores gráficos

Como hemos podido ver en un inicio los videojuegos no se desarrollaban en base a un motor concreto, es decir un sistema que permita la manipulación simplificada y enlazada de los recursos necesarios para la creación del videojuego y de unas leyes físicas concretas. En los inicios de la industria no existía una separación directa entre el desarrollo de recursos gráficos y físico, sino que **el desarrollo era mucho más pretérito** en comparación con el actual, hasta tal punto en que se desarrollaba en muchos casos en **código binario o a lo sumo en lenguaje nemotécnico**, siendo en ambos casos prácticas ancladas al hardware concreto sobre el que se desarrollaba.





Para entender un poco como fué posible la existencia de los motores de videojuegos actuales tenemos que retrotraernos un poco en el tiempo, hacia el año **1989**, fecha en la cual una pequeña empresa independiente de **Kansas** conocida como **Softdisk** decidió contratar a un grupo de programadores para el **desarrollo de videojuegos para microordenadores**. El planteamiento de la empresa era simple: Buscaban desarrollar un videojuego de plataformas que tuviera por protagonista una **mascota reconocible** y que supusiera el baluarte de entrada al mercado del videojuego de microordenadores. Esta idea tiene su meditado sentido, ya que estamos en una época en la cual la industria del desarrollo de videojuegos era dirigida por empresas potentes con **iconos** tan identificables popularmente como **Super Mario** o **Sonic The Hedgehog**.

Entre ese pequeño grupo de programadores novicios en la industria destacó **John Carmack**, el cual lideró a su grupo de compañeros para sacar al mercado en un periodo de apenas un año **Comander Keen**:



*Commander Keen, idSoftware (1990)*

El videojuego podría definirse como un “clon de *Super Mario*”, es decir un videojuego de plataformas en **perspectiva 2D** basado en mecánicas de salto y recolección de elementos. La clave de este desarrollo es que permitió que el equipo de desarrollo ganase la **confianza** suficiente para montar su propio estudio de videojuegos, ya que habías conseguido un desarrollo sólido y con potencial demostrable.

El mismo año del lanzamiento de **Comander Keen (1990)** **Jhon Carmack** llamó por teléfono a su amigo **Jhon Romero** y le planteó la idea de montar una empresa en conjunto y liderar al equipo de compañeros de **Carmack**. **Romero** aceptó de buen grado con perspectiva de poder llevar a buen puerto todas las ideas creativas que tenía en mente, por lo que nació **IdSoftware**. El enfoque que tenían ambos quedó claro desde el inicio del trato:





**Jhon Carmack** quería total libertad para el desarrollo de nuevas experiencias jugables, así como innovar a la hora de crear entornos de juego sostenibles y que redujese el tiempo de desarrollo de los productos planteados.



**Jhon Romero** quería dar rienda suelta como director de juego y productor, de modo que se encargaría no sólo de liderar el desarrollo artístico y conceptual de cada uno, sino de todo lo relacionado a la puesta a la venta y publicidad.



Durante el primer año la empresa intentó mantener relaciones contractuales con Nintendo, mostrándoles el potencial que tenían como empresa independientes gracias a **Comander Keen** y su **secuela**, creada el mismo año gracias a la reutilización de código de Carmack y la alta capacidad de gestión que tenían en conjunto. Desgraciadamente el plan no salió bien, por lo que tras el **rechazo de Nintendo** decidieron que su siguiente paso debía ser el detonante de un cambio. Un cambio que **pusiese el punto de mira en el mercado de videojuegos de PC** y se alejara de las posibilidades que ofrecían las consolas domésticas.

**Comander Keen**, su marca reconocible actual, era un videojuego de plataformas, lo cual era perfectamente desarrollable para consolas. El enfoque que tenían estas era principalmente orientado a adolescentes y niños. La limitación de memoria era un problema inherente del videojuego en consola, ya que era **hardware no personalizable**. Con estos factores en mente decidieron iniciar un desarrollo totalmente opuesto al de las grandes empresas:

- El personaje principal no debía ser visible, sino que debía desarrollarse en **primera persona** para lograr mayor inmersión.
- Las posibilidades de movimiento deberían no tener límites dentro del plano jugable, por lo que era **necesaria la implementación de profundidad en los escenarios**.
- La **violencia** debía ser marca de la casa, para dejar claro el público objetivo.
- Las **bases del desarrollo** debían ser **sostenibles y escalables**, permitiendo la creación de niveles con relativa facilidad una vez creada la base jugable.



Estos cuatro elementos dieron lugar a uno de los pasos más revolucionarios dados por la industria del videojuego: El desarrollo del primer motor de videojuegos 3D totalmente independiente del hardware, el conocido como **Wolfenstein 3D engine**, el cual estaba escrito en C y lenguaje ensamblador x86 y contaba con gráficos en pseudo-3D y sonido de doble lectura (PCM y FMI, que permitían evitar el solapamiento de sonido provocado por las salidas Mono de audio).



*Wolfenstein 3D, idSoftware (1992)*

El éxito de este motor fue notorio, tanto que en pocos meses tras el lanzamiento de **Wolfenstein 3D** el propio **Jhon Carmack** desarrollo una evolución que realmente permitirse la elaboración de espacios en 3D, espacios multiniveles (permitiendo por ejemplo crear escaleras de suelo) y los primeros proyectiles reales con recorrido origen-destino (Previamente en Wolfenstein 3D todas las armas eran **hit-scan**, lo cual quiere decir que siempre que la punta del arma apuntase a un enemigo y el jugador pulsase el botón de disparo, se efectuaría blanco en el tiro. Es decir no existían proyectiles reales). A esta **evolución** del motor **Wolfenstein 3D engine** se la conoció como **Doom Engine o idTech1**, el cual fue usado para el desarrollo de **DOOM** y **DOOM II**.

El éxito comenzaba a cubrir cada videojuego desarrollado por la compañía, por lo que las expectativas que generaban con cada anuncio de su nuevo proyecto no eran precisamente bajas. **Jhon Carmack** dedicó dos años de trabajo a mejorar y pulir el **idTech1** para llegar a conseguir verdaderos entornos con completo poligonaje, tanto del entorno y elementos de este como de enemigos, armas, munición, efectos y el propio jugador.



Su esfuerzo dió los frutos esperados, naciendo el **idTech2** o también conocido **Quake Engine** en el año **1995**, nombre tomado del videojuego que actuaría de demostración técnica de las posibilidades del motor: **Quake**, lanzado en el año **1996**.

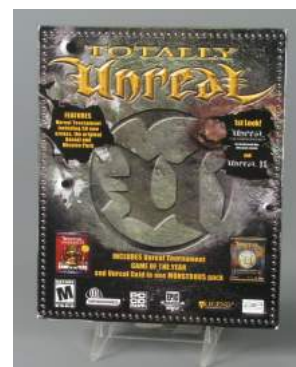


*Quake, idSoftware (1996)*

El lanzamiento de **Quake** supuso la guinda en el pastel del éxito en el mercado del videojuego para PC, pero fué entonces cuando la semilla de la discordia empezó a hacer mella entre ambos creadores. Por su parte **Jhon Romero** quería dedicarse a elaborar otro tipo de videojuegos más centrados en guión, personajes y mecánicas distintas al disparo arquetípico del estudio. Por otro lado **Jhon Carmack** prefería seguir actuando sobre seguro lanzando videojuegos de disparos y puliendo más y más el motor de videojuegos, ya que según su forma de pensar, el futuro del medio debía enfocarse en lograr grandes herramientas para grandes mentes que carecen de ellas.

Esta disparidad dió lugar a la marcha de **Romero**, el cual fundó su propio estudio aparte, quedando **idSoftware** bajo el mando directo de **Carmack**, el cual liberó el **idTech2** al público para que pudiese usarlo para creaciones independientes con total libertad comercial mientras él se dedicaba al desarrollo del **idTech3**.

Es entonces cuando **Epic MegaGames** (actualmente simplemente conocida como **Epic**) aprovechó la ventaja ofrecida por **Carmack** y contrató a los mejores ingenieros de software que pudo encontrar para sacar su propia y mejorada versión del **idTech2** en el año **1998**, el conocido como **Unreal Engine**, el cual se puso en la mira del mercado con el lanzamiento mundial de **Unreal** ese mismo año.





**Unreal Engine** supuso llevar la idea de **Carmack** mucho más allá, ya que ofrecía un uso libre para toda la comunidad que quisiese aprender a utilizarlo. Este motor ofreció **carga de zonas y escenario** mucho más **rápidas**, un manejo muy complejo y bien pulido de gestión de juegos en **redes** y creación de **partidas privadas** con diversos tipos de conexiones en red, posibilidades de creación de **partículas** y **fluidos**, herramientas de poligonaje y modelaje e incluso la posibilidad de crear cinemáticas y manejo de cámaras en entornos.

La llegada del **Unreal Engine** fue un duro golpe para **idSoftware**, pero dió lugar al nacimiento de muchos nuevos motores de videojuegos enfocados al uso de lenguajes y herramientas más propias de otros entornos de trabajo. Un ejemplo de estos nuevos motores sería **GameMaker**, para el desarrollo de **videojuegos** en **2D** en **1999**, o el **jME (JMonkey Engine)** lanzado en el año **2003** y que se orientaba también al diseño 2D pero mediante **Script** codificados en **Java**. Este auge de nuevos motores solo tenía una debilidad palpable, y es que estaban orientados únicamente al desarrollo de videojuegos para PC.

De nuevo **Epic** aprovechó este hueco en el enfoque del desarrollo de motores de videojuegos y tras el éxito de el **Unreal Engine 2** en PC decidió poner toda la carne en el asado para lanzar uno de los motores de mayor éxito en toda la historia del videojuego moderno, el **Unreal Engine 3** (2006), el cual fué el **único** motor multiplataforma con posibilidad de realizar traslaciones de juegos desarrollados en PC directamente a consolas (Con los típicos cambios menores propios de la conversión, como la recolocación de las acciones para adecuarlas al mando de cada consola, o re-renderizado de texturas para reducir el peso del proyecto). El **Unreal Engine 3** ofrecía cargas más rápidas, entornos más vivos, destrucción de escenarios mediante interacción con el entornos con respuesta procedural, mejoró el desarrollo de IAs enemigas y aliadas, supuso un salto enorme en el uso de la iluminación y sonido 3D y, lo más importante: Era un motor que, aunque su plantilla base (**Core**) sea 3D, permitía todo tipo de desarrollos deseables. De este motor destacaron obras magnas de su época, como **Bioshock**, **Batman: Arkham Asylum** o **Dishonored**.



*Batman: Arkham Asylum. RockSteady (2009)*





El éxito alcanzado por **Epic** con **Unreal Engine 3** fué envidiado por muchos, pero una pequeña empresa de **Dinamarca**, conocida como **Over the Edge Entertainment** se propuso destronar al gigante con la creación de un motor de videojuegos que fuese accesible para todos, fácil de aprender, cómodo de usar y con la posibilidad de elegir desde el inicio la plantilla o core a usar. En el año **2005** la compañía sacó su primera versión de su nuevo motor, **Unity**. La comunidad abrazó **Unity** y comenzó a desarrollar **plugins** y **contenido** para el propio motor, por lo que la popularidad de la empresa subió como la espuma. Tal fue el éxito que la compañía decidió mudarse a **San Francisco** e incluso cambió el propio nombre de la empresa por **Unity Software Inc** en el año **2007**, volcando todos sus esfuerzos y recursos al éxito y mejora de **Unity**.

En la actualidad **Unity** es uno de los motores de videojuegos más importantes y más exitosos tanto a nivel del desarrollo en **PC**, **consolas** como **dispositivos móviles**, de tal modo que desde el año **2013** se ha convertido en la herramienta predilecta tanto de desarrolladores primerizos como de veteranos gracias a la enorme evolución que ha ido teniendo versión tras versión como a su gran comunidad.

### 5.3.- Principales motores de videojuegos actuales

En la actualidad contamos con muchos motores de videojuegos disponibles para usar, tanto dedicados para la creación de videojuegos en plataformas móviles, PC, consolas o multiplataformas. Los principales y más usados son precisamente **multiplataformas**, ya que ofrecen mayor posibilidad de distribución, por lo que el usuario aprovechará mejor su tiempo de aprendizaje de la herramienta. Entre los que podemos encontrar en la actualidad destacan sobre todo los **gratuitos**, ya que suelen tener una mayor comunidad detrás dispuesta a aprender y a enseñar a otros así como una enorme cantidad de proyectos creados por terceros en los que podemos basarnos para aprender a manejar la herramienta. Actualmente destacan los siguientes:

#### 5.3.1.- UNREAL ENGINE 5

La compañía **Epic** lanzó en **1998** la primera versión de este motor para poder competir contra el motor de **Quake** ideado por **idSoftware**. En su momento fué un paso muy importante, ya que ofrecía herramientas para el manejo de renderizado de imágenes en **3D**, sistema de detección de **colisiones** sensibles a profundidad de campo, creación y codificación de **IAs** en objetos 3D y en **UIs** así como opciones para la manipulación y configuración de **redes** para el uso del videojuego, lo cual supuso el pistoletazo de salida de la creación de mapas y escenarios por parte de la comunidad. De hecho esta decisión fué tan importante que el resto de la industria empezó a imitar esta posibilidad y agregarlo en sus videojuegos.



En la actualidad la última versión disponible de **Unreal Engine** es la **5**, desde el 13 de mayo de **2020**. Este motor se encuentra disponible para la compilación de proyectos en plataformas de actual generación, así como para las consolas **PlayStation 4** y **Xbox One**.

Tanto esta versión del motor como la anterior cuentan con las mismas ventajas y desventajas con respecto a sus competidores:

Ventajas	Desventajas
<b>Descarga y uso gratuito hasta conseguir ganancias.</b>	<b>Curva de aprendizaje alta.</b> Es de hecho el motor multiplataforma gratuito más complejo de dominar actualmente.
En caso de ganancias el coste de uso actual equivale al <b>5% de la ganancia</b> .	Los <b>proyectos</b> creados con este motor tienden a <b>pesar mucho</b> en comparación con el resto.
Utiliza <b>C++</b> , por lo que tiene una comunidad detrás con mucho soporte con respecto al código.	C++ es un <b>lenguaje</b> un poco más <b>complejo</b> que los usados en el resto de motores.
Ofrece <b>grandes posibilidades</b> de <b>creación</b> y configuración, así como la <b>instalación</b> de plugins y modificaciones de la comunidad para agregar funcionalidades <b>extras</b> .	Varios de los <b>plugins</b> relevantes y <b>modificaciones</b> a agregar son <b>de pago</b> .
Ofrece <b>herramientas</b> muy <b>interesantes</b> para manejo de iluminación, renderizado zonal, generación procedural de objetos y manejo de escenarios y cámaras. Además cuenta con un <b>motor</b> de <b>físicas</b> muy <b>sofisticado</b> .	La <b>comunidad</b> de la herramienta es <b>menor que</b> la que encontraríamos en <b>Unity</b> o <b>Gamemaker</b> , debido a su curva de aprendizaje y a las facilidades que ofrecen las demás.



### 5.3.2.- GAMEMAKER STUDIO 2

Inicialmente conocido como **AMINO** y creado por **Mark Overmars** (Profesor de la Universidad de Utrecht ) el **15 de noviembre de 1999** como herramienta de animación 2D para soporte de sus alumnos. Debido al éxito de la herramienta comenzó a añadirle mejoras y en menos de medio año pasó a ser útil como motor propio del desarrollo de videojuegos, permitiendo cubrir todas las necesidades básicas de un desarrollador actual en entornos 2D.



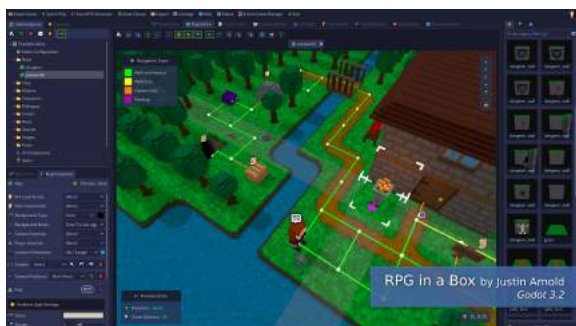
Actualmente su versión más moderna es **GameMaker Studio 2**, la cual se encuentra programada en **C++** (Al igual que sus anteriores versiones) y ha recibido soporte de la **YoYoGames**, empresa de creación y distribución de videojuegos.

Ventajas	Desventajas
Tiene una <b>curva</b> de <b>aprendizaje</b> muy <b>sencilla</b> para el público general.	Es un software de pago, aunque cuenta con una <b>prueba gratuita de 30 días</b> .
Cuenta con una enorme comunidad detrás para la creación de videojuegos en <b>2D</b> .	No es una buena herramienta para la creación de videojuegos en <b>3D</b> . Ciertamente es posible pero poco recomendable.
Permite la creación de videojuegos siguiendo un sistema de <b>programación visual</b> , en el que la mayor parte del tiempo estarás utilizando el ratón para manejar funcionalidades y componentes.	El hecho de que casi toda la manipulación relevante del proyecto se desarrolló con <b>programación visual</b> hace que no sea todo lo personalizable posible.
Puedes programar los <b>Scripts</b> tanto en <b>C++</b> como en <b>C#</b> . Además cuenta con un lenguaje de programación propio conocido como <b>GML (Game Maker Language)</b> .	Las nuevas funcionalidades suelen estar unidas al uso de <b>GML</b> , por lo que es muy complicado completar proyectos sin llegar a aprender el lenguaje.



### 5.3.3.- GODOT

**Godot** es un motor de videojuegos argentino creado inicialmente por **Juan Linietsky y Ariel Manzur** y publicado en el año **2014** por la empresa **OKAM Studios** como **software libre** y **Open Source** y orientado al desarrollo de videojuegos para PC (Windows, OS X y Linux), dispositivos móviles inteligentes y navegadores (Interpretados en **HTML5** y **Javascript**).



Desde su salida ha ido ganando muchos adeptos y es en la actualidad uno de los motores gratuitos multiplataformas más recomendados para aprender, ya que aún se encuentra en proceso de refinamiento y la mayor parte del soporte es ofrecido por la comunidad **Open Source** que la acompaña. Uno de los elementos que definen **Godot** es que permite la creación de videojuegos en entorno 3D y 2D mediante el uso obligatorio de su propio lenguaje de programación, conocido como **GScript**.

Como hemos comentado anteriormente, **Godot** es un software **Open Source**, por lo que podremos acceder a su código más actual desde su propio enlace público **GitHub**, o simplemente descargar su cliente de instalación desde su **página principal**:

- **Enlace al código de GodotEngine de GitHub:**
  - <https://github.com/godotengine/godot>
- **Página principal:**
  - <https://godotengine.org/>

**Godot** cuenta con las siguientes **ventajas** y **desventajas**:

Ventajas	Desventajas
Software gratuito, Open Source y sin regalías.	La mayor parte del soporte viene de parte de la comunidad así como de los creadores de herramientas, plugins y addons.
Permite la anexión de herramientas, plugins y mods totalmente gratuitos.	Por defecto viene con pocos Assets prefabricados, por lo que el desarrollador dependerá de su propio potencial creativo, otros desarrolladores o de la comunidad.
Permite el uso de <b>C++</b> y <b>C#</b> para la creación de Scripts, pero requiere conversión a <b>GScript</b> posterior.	El uso de <b>GScript</b> es prácticamente obligatorio en todo proyecto. Esto ha dado lugar a que en la actualidad haya pocos





	proyectos exitosos creados en Godot.
Trae <b>Cores</b> o <b>Plantillas núcleo</b> básicas tanto para entornos 2D como 3D, mientras que el resto de motores tienden a forzar el entorno 3D y a manejar cámaras fijas en profundidad para aparentar visionado en 2D.	Para la creación de entornos en 3D solo cuenta con soporte OpenGL, por lo que las tarjetas gráficas <b>Nvidia</b> pueden dar problemas de renderizado y manejo al no tener compatibilidad con <b>DirectX</b> .
La comunidad de habla hispana cuenta con la ventaja de que al ser un software argentino, mucha de su comunidad más arraigada tiene el castellano como idioma principal.	La comunidad se ha centrado más en el uso del motor en 2D que el 3D, por lo que el desarrollo de videojuegos en 3D en <b>Godot</b> es mucho más complicado de aprender.
Es un software liviano y ofrece varios tutoriales para un rápido y cómodo acceso al sistema.	Al fomentar el uso del lenguaje propio ( <b>GScript</b> ) requiere inversión extra de tiempo y esfuerzo en un conocimiento que solo servirá para ésta herramienta concreta.

#### 5.3.4.- UNITY

Unity es un motor para videojuegos creado por **Unity Technologies Inc** en el año **2005**. Se caracteriza por ser un motor enteramente **multiplataformas** y por facilitar la entrada de nuevos desarrolladores gracias a una extensa comunidad y un conjunto de detallados **manuales** y **videotutoriales** de enseñanza tanto de la propia herramienta en sí como de herramientas externas de soporte o buenas prácticas para todas las partes del desarrollo de videojuegos.

El motor se encuentra desarrollado en **C++** pero permite el uso de **C#** para sus **Scripts**. Algo que caracteriza a **Unity** frente al resto de motores de desarrollo es que permite escoger desde el inicio la plantilla o core de funcionamiento, permitiendo partir de un **entorno 2D o 3D** (aunque al motor se le conoce comúnmente como **Unity3D**, ya que es el core o plantilla más usado por la comunidad), mientras que la competencia suele tener una de las bases dadas por defecto. Incluye un motor de físicas muy elaborado y que a día de hoy es uno de los mejor valorados de entre los motores de acceso gratuito.



Con respecto a **Unity** encontramos las siguientes **ventajas** y **desventajas**:

Ventajas	Desventajas
Es una <b>excelente opción</b> para el desarrollo de videojuegos <b>multiplataformas</b> .	La <b>herramienta</b> se encuentra en <b>constante cambio</b> , por lo que es probable que proyectos antiguos no funcionen en versiones más actuales del motor.
Cuenta con una <b>enorme comunidad detrás</b> , la cual ha generado gran cantidad de contenido educativo.	La <b>documentación oficial</b> de varias de las funciones de Unity es posible que estén <b>desactualizadas</b> por la incorporación de novedades que suplan necesidades previas.
Tiene un <b>plan educativo</b> que permite el acceso a la <b>versión Pro de Unity</b> , junto con herramientas cloud computing y acceso a assets exclusivos.	<b>El acceso al plan educativo está sujeto a acuerdos difíciles de conseguir</b> para el usuario a nivel individual.
<u>Permite la obtención de un certificado profesional de Unity</u> , muy valorado en el mercado actual de trabajo.	En la actualidad se estima más el certificado de manejo de <b>Unreal Engine 4 o 5</b> para grandes empresas.
Utiliza <b>C#</b> como <b>lenguaje principal</b> , el cual tiene mucha documentación a sus espaldas.	El motor aún no cuenta con mucha capacidad para elaborar cinemáticas profesionales, debiéndose utilizar otras herramientas para dicho fin.
Cuenta con un <b>excelente motor de físicas</b> , pero para el <b>desarrollo 2D</b> permite la modificación de posicionamiento, típica del enfoque clásico de desarrollo.	En caso de obtención de cierta cantidad de ganancias, <b>Unity</b> se queda un porcentaje de los beneficios.
Tiene vinculación directa con herramientas externas, como <b>Visual Studio Code</b> o <b>Audacity</b> .	
Ofrece acceso a una tienda o “market” para recursos y assets muy estimada por la comunidad y el público general que usa la herramienta.	



### 5.3.- ¿Por qué escoger Unity?

**Unity** es sin duda uno de los principales motores gráficos de la actualidad. Como hemos podido ver anteriormente la entrada es asequible para nuevos desarrolladores y ofrece un gran soporte por parte de la comunidad. Además gracias al acceso a la **tienda de recursos** y a la posibilidad de descargar proyectos de terceros aportados por la comunidad es muy sencillo aprender del trabajo ajeno o simplemente hacer uso de estos recursos, permitiendo que no sea un requisito indispensable saber dibujar, componer o diseñar recursos para poder comenzar en el mundillo del desarrollo de videojuegos.



## 6.- UNITY

A lo largo de este apartado veremos las características y requisitos a cumplir para el uso de Unity en la actualidad, instalaremos el motor y aprenderemos un poco la disposición de sus principales elementos en la interfaz principal y de su forma de trabajar (Objetos de juego y componentes, **Scripts**, **Posición**, **Colisiones** y **Cuerpos Rígidos**).

### 6.1.- Características de Unity

Actualmente **Unity** cuenta con los siguientes requisitos mínimos a cubrir para su instalación:

<b>SO</b>	Windows 7 SP1 o posterior, 8, 10, únicamente en las versiones de 64 bits; Mac OS 10.12 o posterior; Ubuntu 16.04, 18.04 y CentOS 7.
<b>GPU</b>	Tarjeta gráfica con capacidades DX10 (shader modelo 4.0).

Para la descarga del programa podremos utilizar el siguiente enlace. Es importante apuntar que para el uso de la herramienta se requiere la creación de una cuenta de **Unity** para obtener un **Unity ID**. Estas cuentas pueden ser de varios tipos, pero nos interesa la cuenta de tipo Personal, la cual nos permitirá usar la herramienta gratuitamente siempre que vayamos a obtener un **ingreso menor anual de 100 mil dólares**.

- <https://unity3d.com/es/get-unity/download>

<h2>Personal</h2> <p>Comienza a crear con la versión gratuita de Unity</p> <p><b>Gratis</b></p>	<div style="display: flex; justify-content: space-between;"> <div>Comencemos</div> <div>Conoce más</div> </div> <p><b>Requisitos:</b> Ingresos o fondos inferiores a USD 100 mil en los últimos 12 meses</p>
---	--

En todas las versiones de **Unity** tendremos acceso a **plantillas listas para ser personalizadas** y que nos van a permitir aprender el funcionamiento de la herramienta y del código aplicado en la plantilla a nuestro propio ritmo. Del mismo modo tendremos acceso a un apartado de **assets** o **recursos prefabricados** tanto por la propia empresa como por la comunidad, de modo que incluso un interesado por la herramienta pueda utilizarla **sin necesidad de saber utilizar herramientas de dibujo 2D, 3D o sonido**.

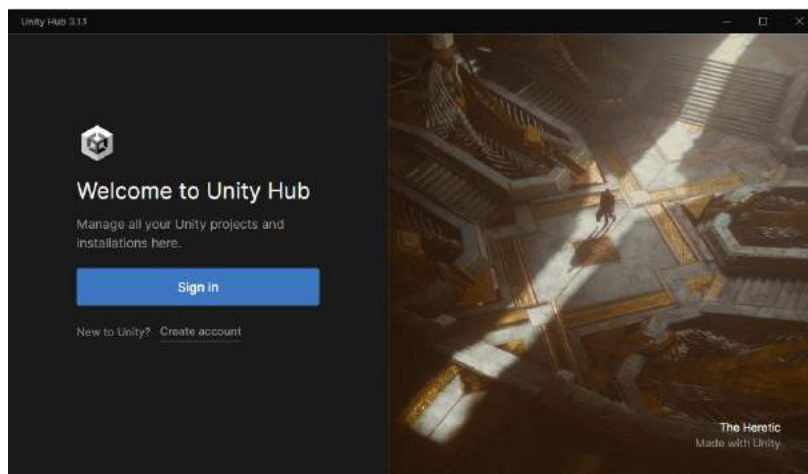


Aunque podemos descargar Unity desde el enlace anterior, la empresa recomienda trabajar con **Unity Hub**, que es una aplicación preparada para permitirnos administrar **múltiples instalaciones del editor de Unity**, de modo que podamos trabajar sobre varias versiones del mismo programa y que podamos ir adaptándonos a la evolución del software a nuestro ritmo. Es muy importante utilizar esta herramienta para **evitar posibles errores de compatibilidad** en actualizaciones futuras de nuestros proyectos.

Para descargar **Unity Hub** desde su web oficial iremos al siguiente enlace, donde tendremos disponibles las versiones de Windows, MAC y Linux:

- <https://unity.com/es/download>

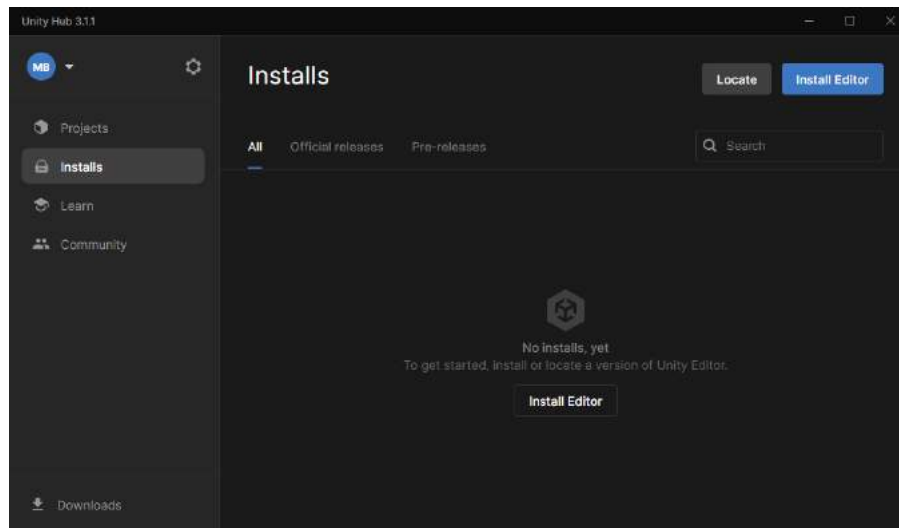
La instalación no merece ningún tipo de explicación, ya que se manipula del mismo modo que cualquier instalación de programa con acuerdo de licencia. Una vez lo tengamos instalado se nos abrirá automáticamente, dejándonos ver la siguiente ventana:



Como podemos ver se nos pide iniciar sesión con nuestra cuenta de **Unity**, la cual crearemos pulsando en el apartado “**Create Account**” que podemos ver en la imagen o desde el siguiente enlace:

- [Unity ID](#)

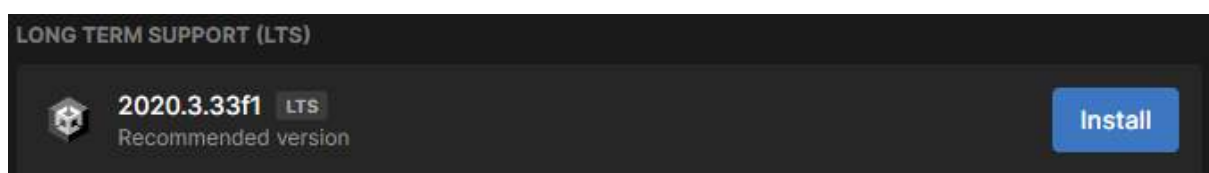
En ambos casos la cuenta nos asignará un **ID de Unity**, el cual es el que define nuestras posibilidades dentro del entorno. Recordemos que estas posibilidades se definen por la versión de licencia que vamos a usar, siendo en nuestro caso la **Personal**. Una vez tengamos hecha la cuenta simplemente iniciaremos sesión en la aplicación **Unity Hub** y veremos como se nos muestra el siguiente contenido:



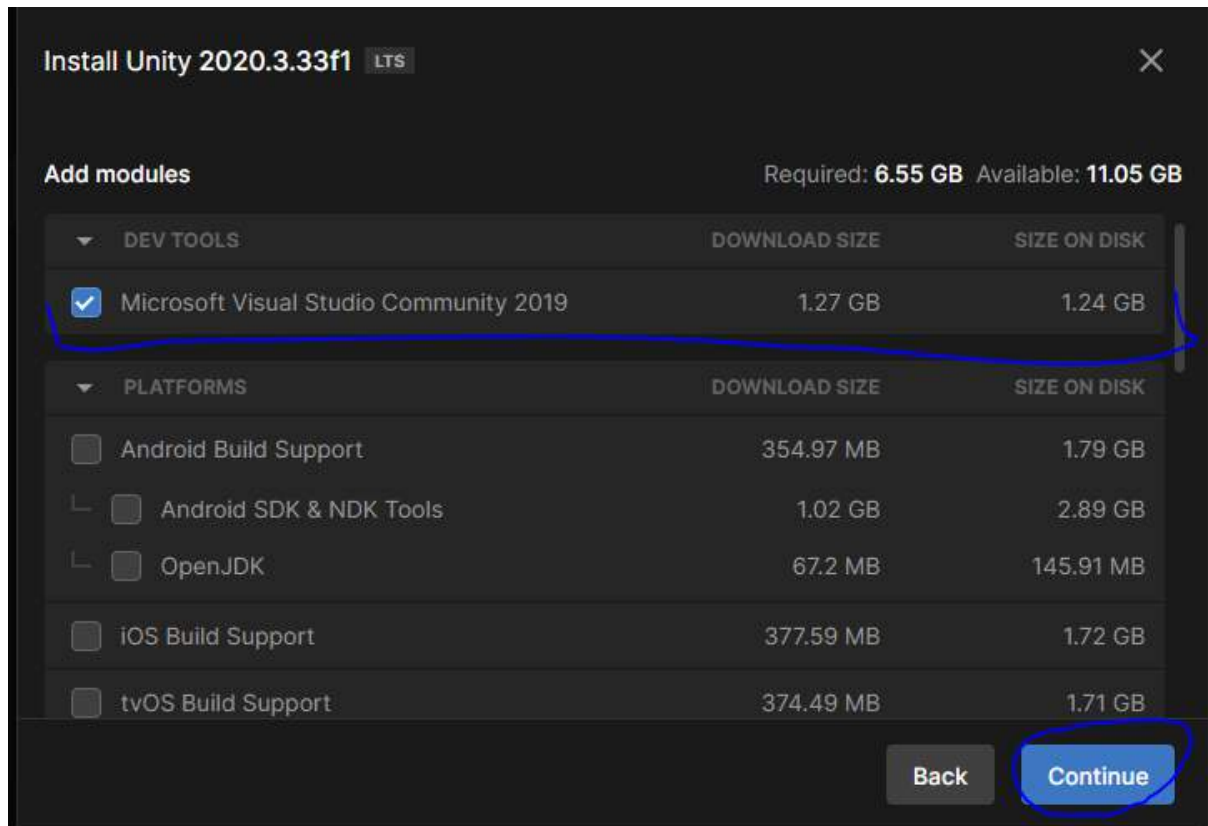
Como podemos ver la herramienta contiene un menú lateral dividido en 4 apartados, de los cuales de momento nos interesan solo los 2 primeros:

- **Projects:** Muestra los **proyectos** que tenemos **creados** y definidos por nuestra cuenta, así como la versión del editor de Unity que utilizamos.
- **Installs:** Muestra los **editores** de Unity **instalados** así como su versión.

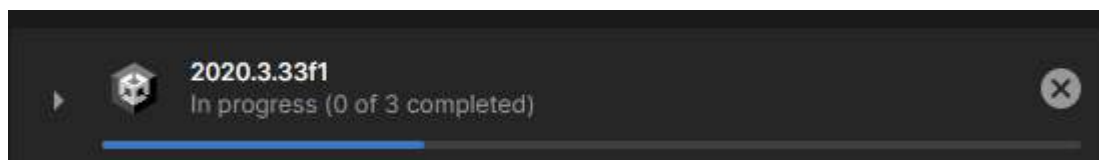
Si nos fijamos atentamente podemos ver como por defecto al instalar Unity Hub nos ha llevado automáticamente al apartado de instalación, ya que no podremos crear proyectos de Unity hasta que no descarguemos e instalemos un editor. En nuestro caso instalaremos el más actual disponible, por lo que pulsadores en “**Install Editor**” e instalaremos la versión **LTS** más actual disponible, que a fecha actual es la **2020.3.33f1**:



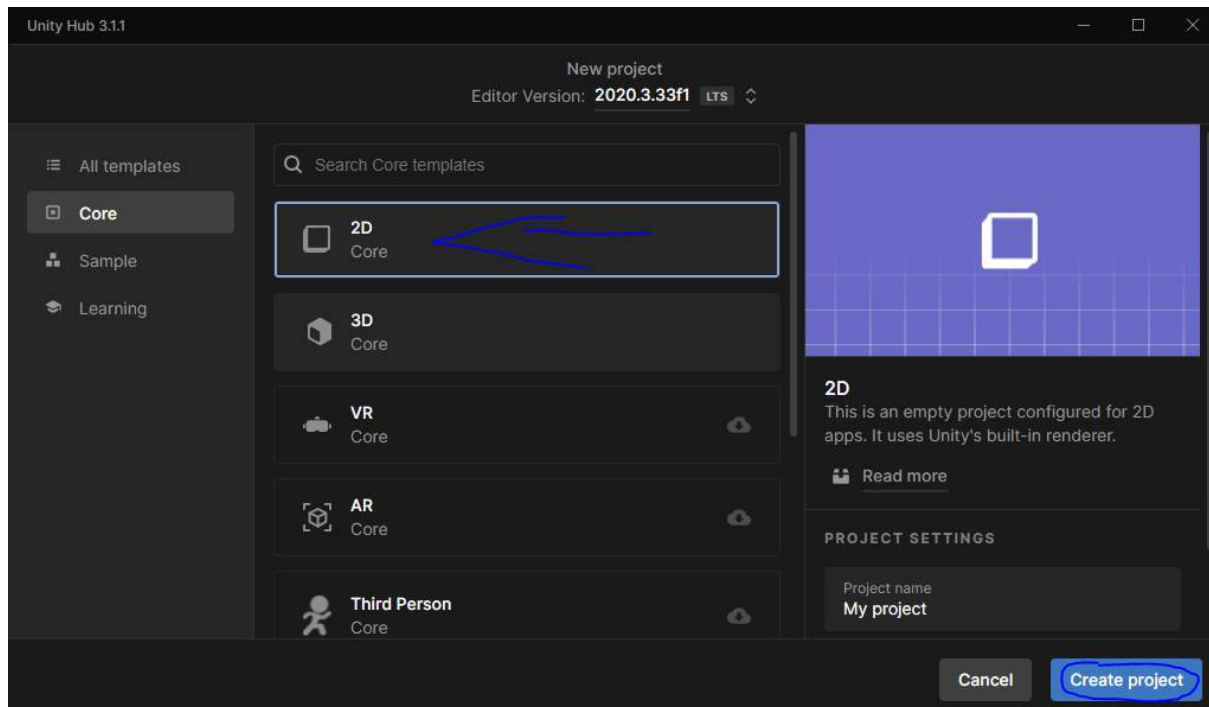
Existen versiones más **modernas**, pero no se encuentran en **versión LTS**, es decir que tienen funcionalidades en estado de prueba y que, por lo tanto, no son editores estables. Una vez pulsamos en el botón de instalación verso que nos aparece un listado de los modelos disponibles para ser instalados en Unity. El más interesante es precisamente el que nos aparece marcado por defecto: **Visual Studio Code**:



Tras esto comenzará la instalación:



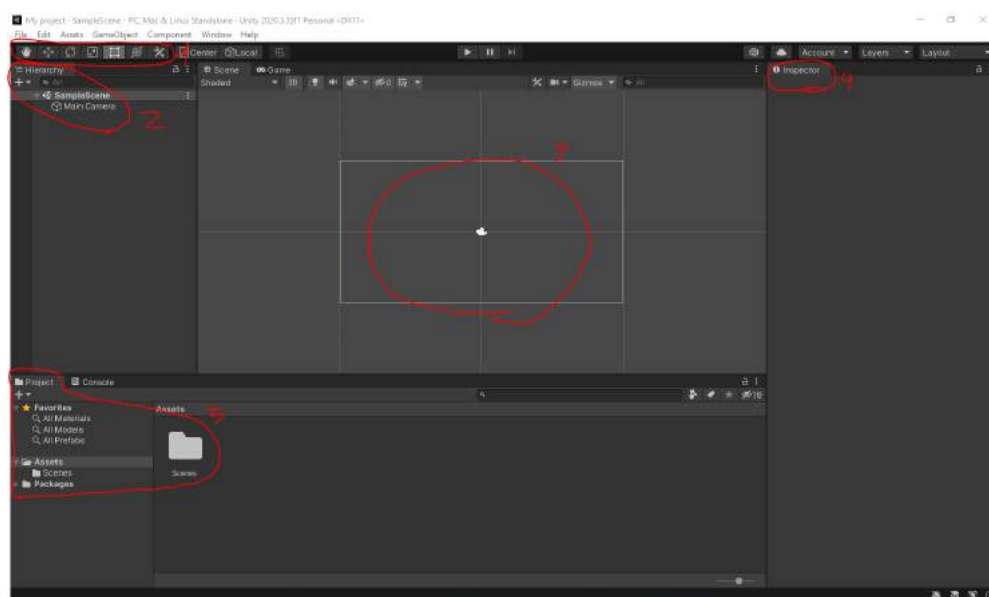
Una vez haya finalizado podremos acudir al apartado de proyectos y podremos crear el nuestro. Para ello pulsamos en “**New Project**” y veremos como se nos va a desplegar un listado de **núcleos o cores** disponibles como plantillas base del proyecto. Las principales son la plantilla **2D** y la **3D**, que establecen las configuraciones básicas para proyectos **Unity** con estas dimensiones, sin agregar código extra. El resto son plantillas que vienen con código predefinido y normalmente con **escenarios** y **assets** de **ejemplo**, pero en nuestro caso elegiremos la **plantilla básica de 2D** y pulsaremos en **crear proyecto**:



Tras crearse el proyecto veremos como se nos abre directamente bajo el nombre por defecto de **My Project**.

## 6.2.- Entendiendo la interfaz

Una vez iniciemos nuestro proyecto de Unity nos encontraremos con la vista o ventana principal, la cual se compone de varios elementos a comentar:







1. **Herramientas principales de la escena:** Conjunto de herramientas que sirven para manipular **objetos de juego** dentro de una **escena**, las principales son:
  - a. Mover a mano alzada.
  - b. Mover en direcciones concretas con mayor precisión.
  - c. Rotar.
  - d. Escalar.
  - e. Transformar componente mediante la alteración de las disposiciones de sus vértices (También se le conoce como **Transformación libre** y engloba las 4 anteriores.
  - f. Mover, rotar y escalar objetos seleccionados. realiza las mismas funciones que indica su nombre. es decir, sin permitir transformación libre. Tampoco permite el movimiento a mano alzada.
  - g. Herramienta de edición: Es una herramienta extra que nos permite acceder a otras herramientas de edición de escenas que hayamos instalado. En nuestro caso carece de funcionalidad.
2. **Jerarquía:** Listado ordenado de las **escenas** del proyecto y sus correspondientes **objetos de juego** y que se encarga de definir la preferencia que van a tener tanto entre ellos como con respecto a la cámara principal del proyecto. Por defecto al crear un proyecto se genera el componente "**SampleScene**", que es una escena básica que contiene un **objeto de juego (GameObject)** de tipo **Cámara** llamado "**Main Camera**".
3. **Escena actual:** Nos muestra la escena actualmente fijada, que en la imagen superior sería la escena "**SampleScene**". Asimismo todo proyecto debe tener una cámara principal, que es la que recoge la información visual captada por el propio videojuego. Al haber únicamente una escena, la cámara principal se dispone sobre ella por defecto.
4. **Inspector:** Nos muestra la **información del objeto de juego seleccionado**, así como de cada uno de sus **componentes**.
5. **Arquitectura del proyecto:** Define el **contenido** de nuestro **proyecto**, es decir sus paquetes, carpetas y recursos.

Como podemos ver la interfaz constantemente nos recuerda que la herramienta está orientada a trabajar sobre escenas, objetos de juego y componentes. Las escenas básicamente son los entornos donde disponemos los objetos de juego, por lo que vamos a centrarnos en desgranar un poco mejor los dos conceptos respants.

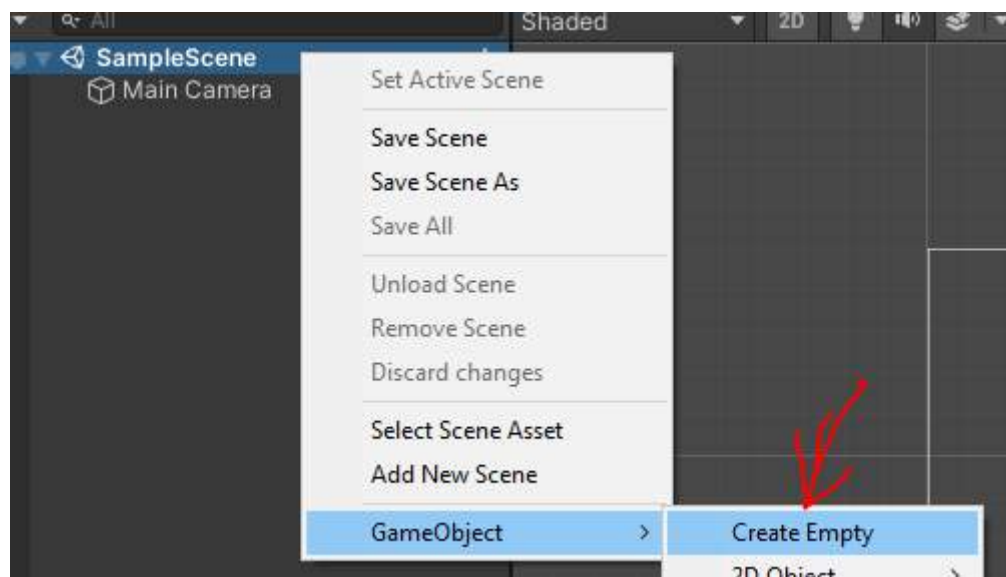


### 6.3.- GameObject y Components

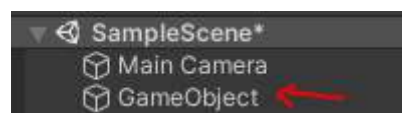
Un videojuego creado en Unity se compone mínimo de una escena, la cual debe contener mínimo un **objeto de juego** o **GameObject** que es la cámara principal, ya que sin ella no se mostraría nada en la pantalla de juego. Los **GameObject**, por lo tanto, son los elementos más básicos que componen un todo mayor.

Por ejemplo en un videojuego de plataformas como **Super Mario Bros**, cada una de las **pantallas** o **fases** sería una **escena**, cada elemento de la escena sería un **GameObject** (El personaje, los enemigos, las monedas, los sonidos que produce cada uno...) y a su vez cada **GameObject** se define por una serie de componentes (Su disposición en la escena, las animaciones que tiene definida, los eventos o triggers a los que responde, los documentos de código ejecutables por este (Denominados **Scripts**)).

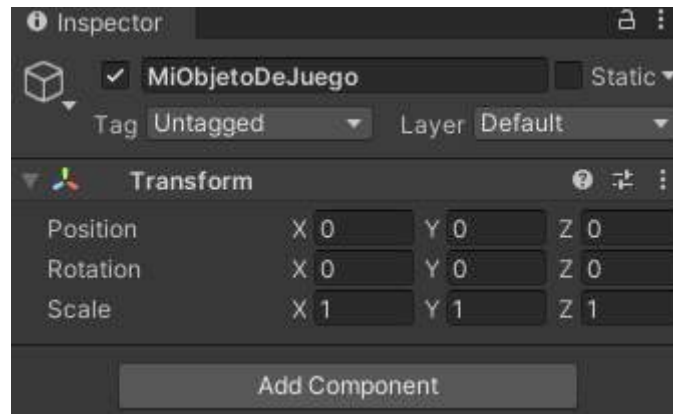
Crear un **GameObject** en **Unity** es muy sencillo, simplemente nos vamos a la escena concreta sobre la que queremos crearlo y hacemos **click derecho -> GameObject -> Create Empty**. Con esto crearemos un **GameObject vacío** sin definición, es decir que ni va a tener nombre ni va a tener atributos de definición, únicamente de posicionamiento sobre la escena (componente **Transform**, el cual se establece por defecto en todo objeto de juego y define su posicionamiento en la escena):



Como el **GameObject** está **indefinido**, **Unity** le pondrá por defecto el nombre **GameObject**

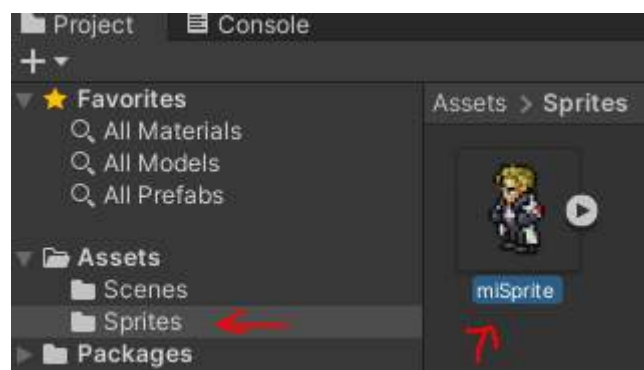


Si pulsamos sobre el nuevo componente veremos como en el **Inspector** (**Menú derecho de Unity**) se nos despliega la **información** de nuestro objeto de juego. Desde ahí podremos cambiar el nombre de este. Por ejemplo lo llamaremos **“MiObjetoDeJuego”**:

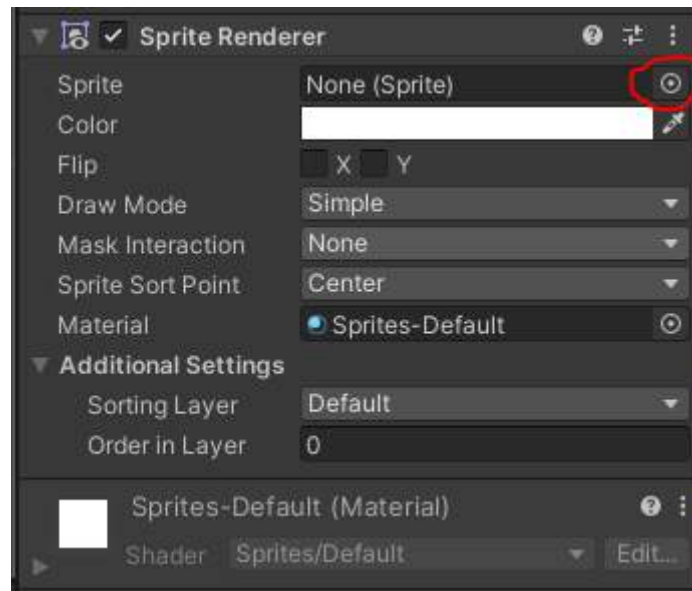


Como podemos comprobar por defecto el **GameObject** dispone del componente **Transform**, que es el que define la posición del objeto de juego dentro de la escena, la rotación y escala dentro de esta. Como podemos ver estos atributos de transformación son los mismos a los que podemos acceder con las herramientas de escena que ya vimos anteriormente (Por defecto todo **GameObject** nuevo se establece en la posición 0,0,0 y sin rotaciones. Es decir, se situará justo en el centro de la escena).

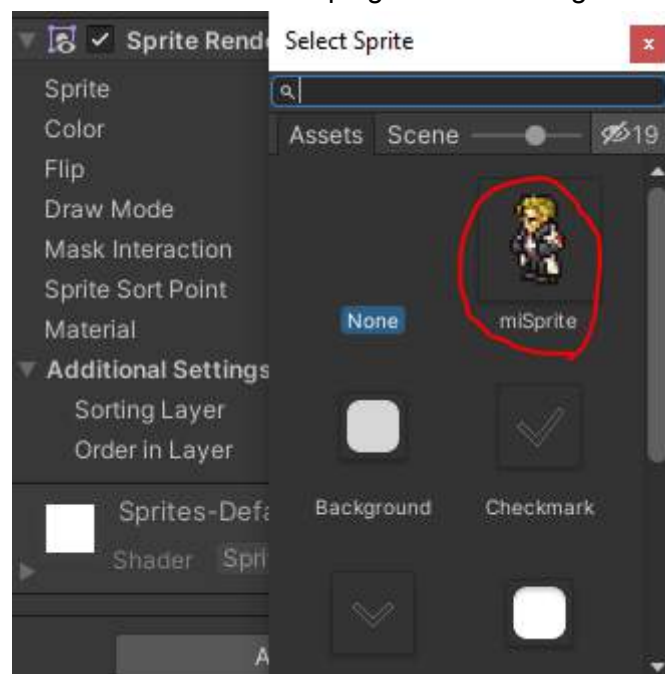
Asimismo podemos agregar nuevos componentes a nuestro **GameObject**. En nuestro caso queremos agregar un componente que permita que nuestro objeto muestre en la escena un Sprite o Imagen concreta. Antes de agregar dicho componente nos vamos a la arquitectura de nuestro proyecto y dentro de la carpeta **Assets** creamos una carpeta a la que llamaremos **Sprites** y que va a contener todos los sprites de nuestro proyecto. En dicha carpeta agregaremos una imagen:



Para hacer que nuestro **GameObject** muestre nuestra imagen o **Sprite** existe el componente **Sprite Renderer**, por lo que pulsamos en **Add Component** y lo agregamos. Una vez lo hayamos hecho simplemente pulsamos en el botón indicado a continuación:



Se nos desplegará una lista de imágenes disponibles en nuestro proyecto, buscando internamente una carpeta denominada **Sprite** en primer lugar. Es precisamente por eso que la hemos creado anteriormente. Una vez desplegada la lista elegimos la nuestra:





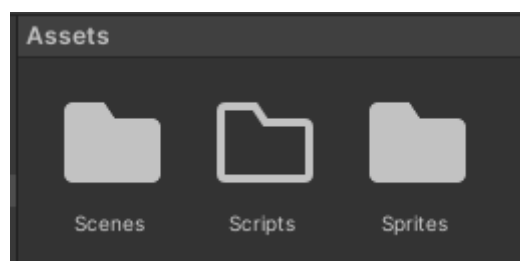
De modo que ya nuestro **GameObject** muestre dicha imagen en la escena **SampleScene**. De este modo nuestro objeto de juego pasaría a ser **visible** en la **escena**:



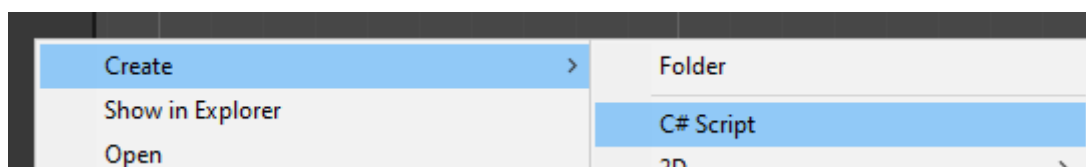
Y con esto ya hemos visto cómo crear objetos de juego y como agregarlos componentes nuevos, pero por supuesto con esto no hemos conseguido gran cosa, ya que **a los objetos de juego hay que definirles un comportamiento a seguir**. Es decir, tenemos que definir unas pautas concretas a respetar dentro de la escena, lo cual conseguiremos en el siguiente punto.

## 6.4.- Scripting con C#

Para mantener un orden correcto para nuestros objetos de juego y los recursos vamos a crear una nueva carpeta para almacenar nuestros documentos de código. A esta carpeta la llamaremos **Scripts** y la crearemos dentro de la carpeta **Assets** principal:



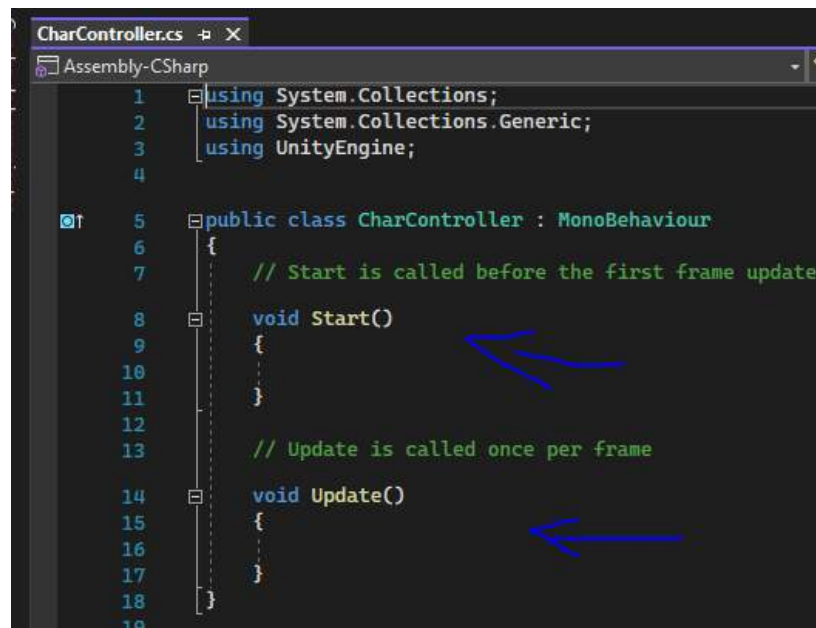
Haremos click derecho sobre esa carpeta y luego haremos **click sobre Create -> C# Script**:



Nuestro **Script** (Al que llamaremos **CharController**) es en esencia un documento de código preparado para leer lenguaje **C#** y en el que vamos a definir el comportamiento de nuestro objeto dentro de la escena en la que se encuentre. Como lo que hemos definido en nuestro objeto es un **personaje** vamos a crear un **Script** en **C#** que nos permite hacer que se mueva hacia **arriba** al **iniciarse la escena**.



Tras crear el documento simplemente lo abrimos con doble click y **Unity** se encargará de abrir **Visual Studio Code** automáticamente con nuestro documento recién creado. Una vez se nos haya abierto el documento veremos que por defecto Unity le ha establecido algunas líneas al documento:



```
CharController.cs
Assembly-CSharp

1 using System.Collections;
2 using System.Collections.Generic;
3 using UnityEngine;
4
5 public class CharController : MonoBehaviour
6 {
7     // Start is called before the first frame update
8     void Start()
9     {
10
11     }
12
13     // Update is called once per frame
14     void Update()
15     {
16
17     }
18 }
19
```

Vamos a explicar paso a paso cada una de ellas:

1. **Importaciones y dependencias:** Se declaran con la palabra reservada “**using**” y en este caso destacamos 2:
  - a. **System.Collections:** Clase del sistema Unity para el uso de funciones que permitan el manejo de colecciones de eventos ejecutables o activadores (Comúnmente llamados **triggers**). En Unity el evento activado por defecto es el FRAME o refresco de la imagen, por lo que cada vez que la imagen se refresque en pantalla (Controlador por el monitor o pantalla donde se visualiza) se ejecutará un método u otro de los establecidos. En este caso destacaremos dos métodos: Start y Update, que luego analizaremos.
  - b. **UnityEngine:** Clase que hace referencia al motor de Unity, encargado de las peticiones para la correcta gestión de recursos internos del equipo. Por ejemplo es el que se va a encargar de administrar la cantidad de memoria RAM límite a utilizar, los flujos de sonido o canales, el trabajo con red, etc. También es la clase que nos va a permitir trabajar con el motor de físicas de Unity, que veremos más adelante.
2. **Uso del tipo MonoBehaviour:** La clase MonoBehaviour es una clase que define el tipo de comportamiento de todos los scripts de nuestros objetos de juego, ya que es



una clase de **Unity** que se encarga de asignar comportamientos de modo singular al componente, en vez de comportamientos para toda la escena en general. Es precisamente por eso que es el tipo de clase más utilizado en **Scripts**, ya que se busca la creación de objetos concretos con fines concretos, a modo de ser reutilizables.

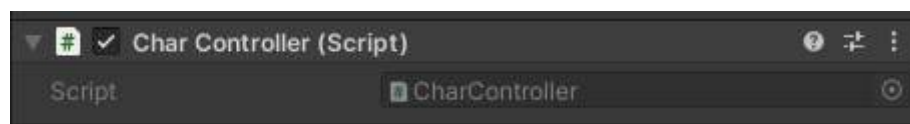
3. **Método Start():** Es un método que no devuelve nada (es void) y que se encarga de ejecutar su bloque de código una única vez, concretamente al nacer el objeto de juego (justo antes del refresco del primer frame). Lo utilizaremos para cargar configuraciones concretas iniciales de nuestro objeto, si es que lo necesita.
4. **Método Update():** Este es el método que más se utilizará. Sirve para ejecutar su bloque de código en cada actualización de frames en pantalla, de modo que, por ejemplo, podamos averiguar qué tecla del teclado se está pulsando en cada refresco de pantalla, de modo que sepamos cómo proceder dentro del juego.

Como lo que buscamos es que justo al iniciarse la escena nuestro objeto de juego se desplace hacia arriba, lo que haremos será, dentro del método **Start**, alterar su componente **Transform** para cambiar su posición (Las **posiciones** en **Unity** son objetos **Vector3** que contiene datos de tipo **float**) de 0,0,0 a 0,20,0, donde la primera posición es la X, la segunda la Y y la última la Z.

Para referirnos a un objeto de juego se utiliza la palabra reservada **gameObject** y para hacer referencia a un componente específico que tenga se utiliza el método **GetComponent**. Algo que debemos saber es que la posición del componente transform requiere de un objeto de tipo Vector3, que es un objeto que incluye tanto la X,Y y Z:

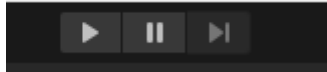
```
Mensaje de Unity | 0 referencias
void Start()
{
    gameObject.GetComponent<Transform>().position = new Vector3(0, 20, 0);
}
```

A continuación deberemos vincular nuestro Script a nuestro objeto de juego. Para ello simplemente arrastraremos el script al inspector de nuestro objeto de juego:

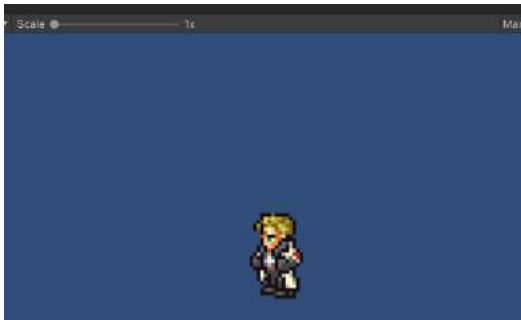




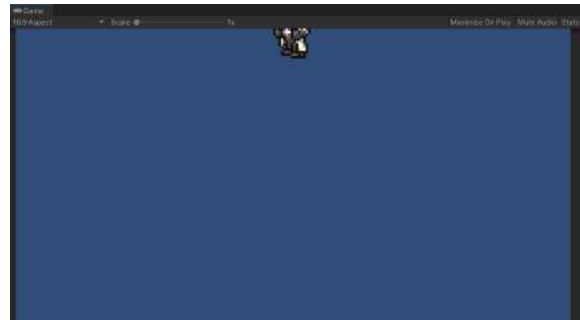
Para ejecutar la escena simplemente pulsaremos en el botón **“Play”** de Unity. Podremos pausar y avanzar periodos de tiempo concreto con estas tres herramientas:



Antes de pulsar en **“Play”** veremos cómo el **objeto de juego** se mantiene en la posición **0,0,0** pero tras pulsar veremos como se desplaza hacia arriba en 20 puntos.



*Antes de pulsar PLAY*



*Tras pulsar PLAY*

Con esto ya habríamos aprendido a crear un **Script** y anexarlo a nuestro objeto de juego.

## 6.5.- Comportamiento espacio-temporal en un plano 2D

Con el **Script** anterior hemos definido un comportamiento concreto a ser ejecutado nada más iniciar la escena del juego, pero recordemos que en un videojuego la clave está en la posibilidad de interactuar con lo que vemos. Por ello en este apartado vamos a ver cómo aplicar un **comportamiento concreto** en movimiento dentro del plano 2D actual. Antes de explicar este proceso debemos conocer un pequeño detalle del funcionamiento interno de Unity.

Como sabemos, la ejecución de los dos métodos básicos de **Unity (Start y Update)** se realizan mediante refresco de pantalla (**Frames**). Esto no siempre es ideal, ya que no todos los monitores cuentan con la misma tasa de refresco en la imagen. Por ejemplo un monitor ordinario de hoy en día suele ser de **55 o 60 hz**, por lo que cada segundo se **refrescará su imagen 55 o 60 veces**, respectivamente. Esto quiere decir que si dejamos la ejecución en manos de los frames, el comportamiento variará con respecto a la pantalla o monitor (A veces incluso muy drásticamente, ya que por ejemplo las pantallas actuales 4K suelen rondar los 144hz).



Para evitar este comportamiento **Unity** cuenta con el método **deltaTime** de la clase **Time**, la cual se encarga de obtener el tiempo actual, por lo que a la hora de ejecutar acciones que repercutan directamente el visionado nos aseguraremos de utilizarla para que las velocidades de ejecución se calculen en base a unidades de tiempo, y no de refresco.

Una vez comentado esto, lo siguiente que tenemos que entender es que para aplicar movimiento desde teclado a la ventana de juego deberemos capturar **activadores de entrada**, los cuales en **Unity** se denominan **Inputs**. Existen muchos **tipos de inputs**, pero en este caso nos interesan entradas de botones flecha de nuestro teclado, por lo que haremos uso del método **GetKey** de la clase **Input** y le pasaremos por parámetro la tecla pulsada. Para este ejemplo vamos a utilizar las flechas de dirección (**left**, **right**). Como nos estamos moviendo en un espacio **bidimensional** orientado a un videojuego de plataformas no tiene mucho sentido buscar movimientos verticales, por lo que nuestro código quedaría tal que así:

```
void Update()
{
    if(Input.GetKey("left")){
        gameObject.transform.Translate(-15f * Time.deltaTime, 0, 0);
    }
    if (Input.GetKey("right"))
    {
        gameObject.transform.Translate(15f * Time.deltaTime, 0, 0);
    }
}
```

Como podemos ver en este caso **no hemos utilizado el método GetComponent ni le hemos establecido un nuevo Vector3**. Esto se debe a que con el método **transform.Translate** hacemos exactamente lo mismo pero reduciendo enormemente el código escrito. En este caso el nuevo objeto **Vector3** se genera directamente con los **parámetros X, Y y Z** que le pasamos a **Translate**. Si nos fijamos en el eje X (Desplazamiento horizontal) le estamos pasando **15f** (este valor va a definir la velocidad de desplazamiento. A mayor número, mayor velocidad) multiplicado por el **cálculo interno de unidades de tiempo que usa Unity**, el cual debe ser multiplicado por la unidad de tiempo que nos ofrece Unity para **mantener la misma velocidad en cualquier tipo o velocidad de refresco**, ya que al estarse ejecutando este bloque de código dentro de la función **Update**, se ejecutará el bloque siempre que se refresque la pantalla (Por ejemplo, si nuestro monitor es de 55hz el método se estaría ejecutando 55 veces por segundo. Con **Time.deltaTime** nos aseguramos de que aunque se llame mas o menos veces al método, **la velocidad va a ser siempre la misma en cualquier monitor**).



Si pulsamos en el botón de **Play** de la escena, comprobaremos que si pulsamos en la tecla de flecha izquierda, nuestro objeto de juego se desplazará en esa dirección:



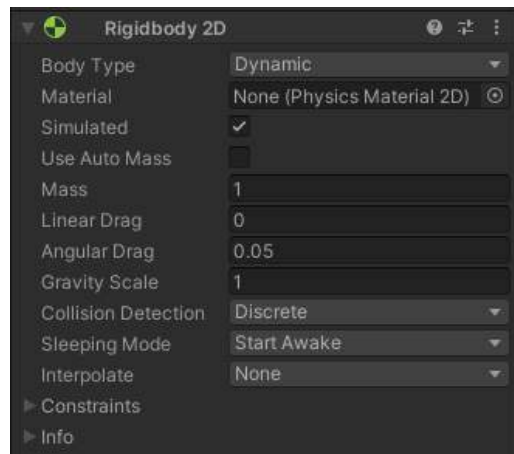
Y con esto ya tendríamos un movimiento **espacio-temporal** aplicado a nuestro **objeto de juego**. Ahora debemos ver como hacer que reaccione a otros elementos del escenario, de modo que podamos definir un comportamiento al objeto para cuando colisione con otro.

## 6.6.- Rigidbody y BoxCollider

Unity incorpora un **motor de físicas**, como la mayoría de motores de videojuegos actuales. Para hacer que un **objeto de juego** o **GameObject** utilice dicho motor de físicas debemos agregarle un componente a nuestro objeto denominado **Rigidbody**, para hacer que pase a ser considerado en la escena como un cuerpo rígido y, por tanto, no traspasable por otro salvo que tenga prioridad sobre este.

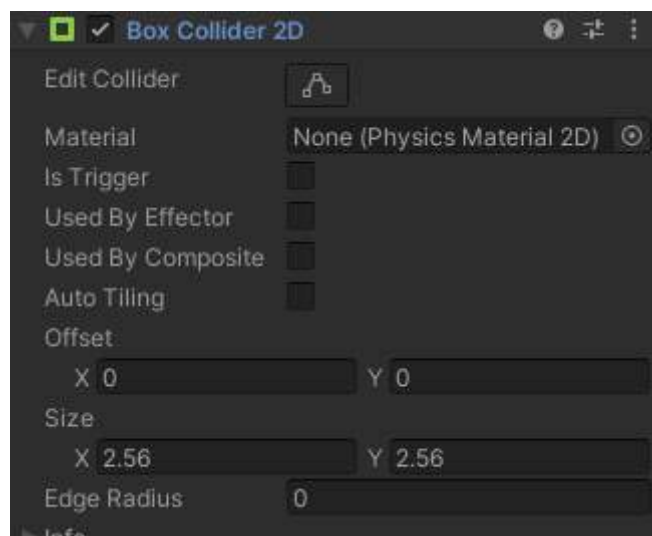
Para establecer este nuevo componente simplemente haremos click sobre nuestro componente y en el apartado de Inspector pulsaremos en **Add Component** y buscaremos **Rigidbody 2D**, ya que estamos trabajando sobre un plano 2D:





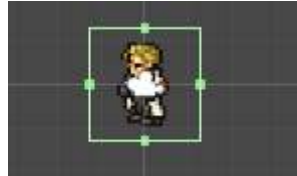
Si le diésemos al botón de **Play** nuestro personaje caería infinitamente, ya que al tener un **RigidBody** se encuentra sometido a las leyes físicas de **Unity** y, por tanto, a su **gravedad**. Ahora necesitamos que el objeto sea capaz de interactuar con otros (Por ejemplo que sea capaz de interactuar con un objeto **SUELO** que impida la caída de nuestro objeto y que crearemos más adelante). Para ello primero debemos dotar a nuestro objeto de juego de una **mallá de colisiones**.

Una **mallá de colisiones** no es más que una **zona delimitada en un plano dimensional concreto** (En este caso X e Y, al estar trabajando sobre 2D) y que se encarga de activar un evento de colisión en el preciso instante en que otro objeto con mallá de colisión entre en contacto con otra. Las mallás de colisiones pueden ser de múltiples vértices, pudiéndose adaptar lo máximo posible al objeto de juego, pero cuantos más vértices tenga más recursos consume y, por lo tanto mejor hardware hará falta para ejecutar el juego. Es por eso que en la mayoría de casos es mejor **sustituir una completa mallá de colisiones** por una **caja de colisiones**, de modo que reduzcamos los vértices a los mínimos posibles. Para nuestro objeto simplemente agregaremos un componente llamado **Box Collider 2D**:





Si a continuación nos vamos a la ventana de la escena veremos como a nuestro objeto lo recubre ahora una **caja de color verde**:



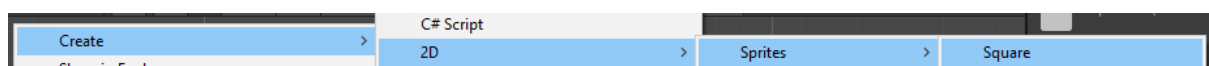
Esa es precisamente la caja de colisiones, que está ocupando todo el objeto. Pero lo que buscamos es adecuarla lo máximo posible al personaje. Para ello simplemente nos iremos al componente **Box Collider 2D** de nuestro objeto y le cambiaremos el tamaño por uno más adecuado:



De esta forma se ajustará lo mejor posible a lo que tenemos en escena:



Ahora tenemos preparado nuestro objeto de juego principal **pero sigue sin existir un suelo sobre el que posarlo**, por lo que si pulsamos **Play** seguirá cayendo infinitamente. Para solucionar eso vamos a crear un nuevo **GameObject** al que llamaremos **Suelo** y al que aplicaremos un componente **Box Collider 2D** así como un **Sprite** que vamos a autogenerar con Unity: **Un cuadrado en 2D**. Para obtener este cuadrado simplemente haremos click derecho sobre nuestra carpeta de Sprite y pulsamos en **Create -> 2D -> Sprites -> Square**



A este objeto de juego Suelo **no le pondremos un componente de Rigid Body**, ya que no queremos que le afecte la gravedad, sino que únicamente **"flote"** en nuestro escenario e impide que caiga nuestro personaje, **pero si le pondremos un componente Box Collider 2D, como hicimos con el personaje**. A continuación desde la propia visualización de la escena veremos como ha aparecido nuestro nuevo objeto de juego:



Lo arrastramos debajo de nuestro personaje y lo agrandamos como queramos mediante los **puntos de transformación azules** que le aparece en las esquinas:



Con este suelo agregado, si pulsamos en el botón **Play** de **Unity** veremos como el personaje principal ya no se sale de la escena por debajo, sino que al colisionar con la caja de colisión del suelo respeta las leyes físicas que su componente **RigidBody** obliga a respetar:



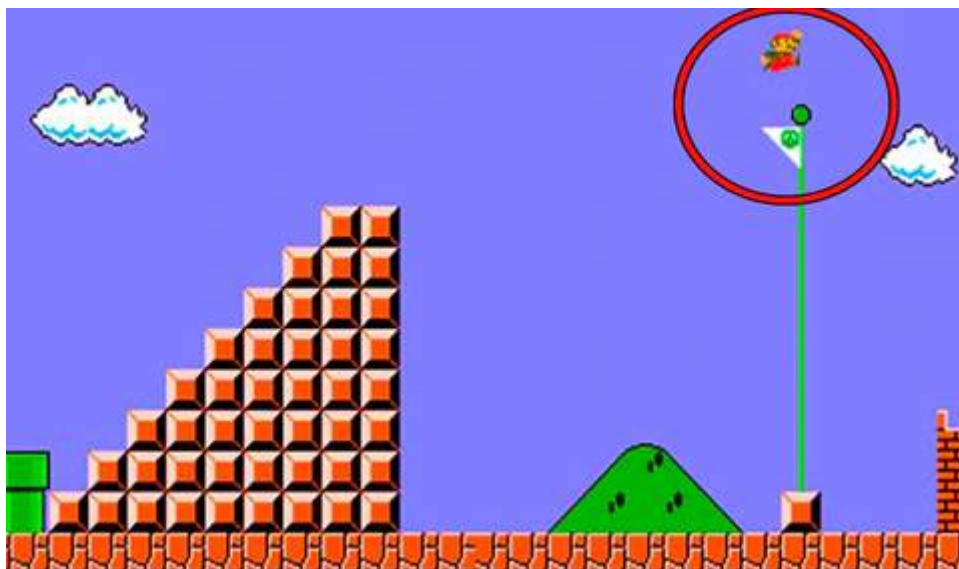
Con esto hemos visto el funcionamiento básico de los **objetos de juego** y su **posicionamiento** sobre las **escenas**, manejo de **Scripts** y aplicación del **motor de físicas**. En los siguientes apartados vamos a ver como crear nuestros propios **Scripts de mecánicas** jugables (Salto sobre plataformas, recolección de objetos, ...) y **Assets** gráficos y de sonido que utilizamos en el nivel de videojuego de plataformas que desarrollaremos más adelante.



## 7.- ASSETS

Un **Asset** o “**activo**” es un recurso o conjunto de recursos que nos van a permitir la creación del videojuego o de parte de éste. Principalmente encontramos assets **gráficos** (que pueden ser en 2D o 3D, dependiendo del enfoque jugable que deseemos cubrir), **multimedia** (tanto de vídeo prerenderizado, imágenes de texturas para materiales, sonidos o melodías enteras) y **Scripts** (que pueden ser de interacción directa como los **Scripts** que definen mecánicas jugables y con las que podremos interactuar directamente desde el punto de vista del jugador, así como **Scripts** que definen **eventos** de activación o del ciclo de vida del videojuego de una subdivisión de éste).

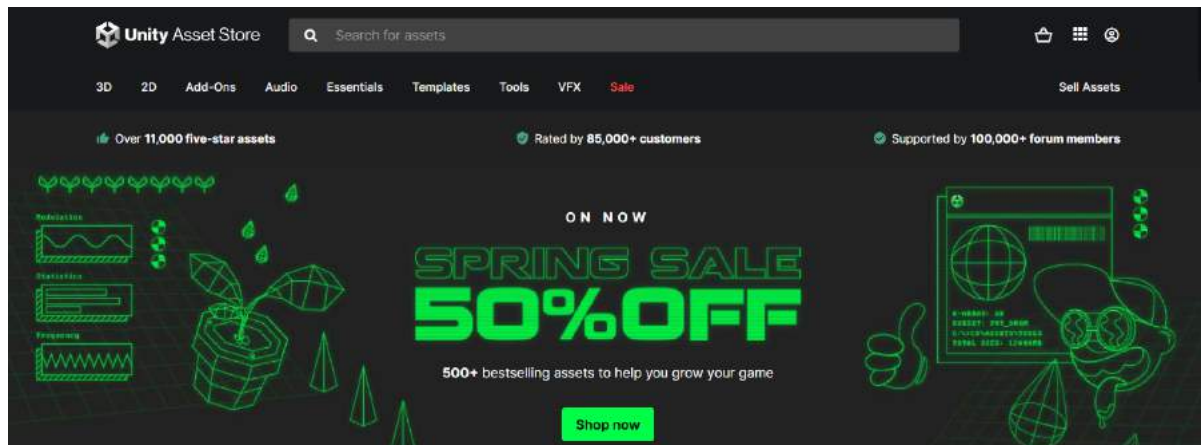
Un ejemplo de esto último encontraríamos en la siguiente imagen:



Mario sería un objeto de juego que llevaría incorporado un **Script que define sus mecánicas jugables** (Saltar, romper bloques, recoger monedas, crecer al comer champiñones...), pero la bandera de final de nivel llevaría un **Script de definición de eventos** (Calcular puntuación final de nivel, Activar cinemática de Mario entrando en el castillo y Cargar la siguiente fase del videojuego).

Algo que también debemos recordar es que, como dijimos en el punto anterior, **no es necesario que diseñemos nuestros assets**, ya que **Unity** dispone de una enorme comunidad de desarrollo y diseño de assets de todo tipo y que podemos encontrar en su tienda dedicada:

- <https://assetstore.unity.com/>



Si accedemos veremos que tenemos un extenso catálogo en el que veremos tanto **recursos de pago como gratuitos**, pero en nuestro caso vamos a optar por crear nuestros propios recursos mediante herramientas externas. Es por ello que a continuación veremos diversas herramientas de creación y cómo exportarlas a **Unity** y, de hecho si nos apasiona, podemos dedicarnos simplemente a crear recursos para venderlos en la **Store de Assets de Unity**.

## 7.1.- Creación de elementos gráficos en 2D con KRITA

Como ya vimos en puntos anteriores, **Krita** es una herramienta **gratuita** de edición de imágenes. Podríamos usar realmente cualquier herramienta de edición de imágenes a la hora de crear nuestros **Sprites** y **texturas**, pero buscamos conocer nuevas herramientas y que sean lo más livianas posible. Una de las principales **virtudes** de **Krita** es que tiene mucho potencial gráfico y su cliente pesa tan solo **120MB**.

Lo primero que necesitamos es instalar el cliente de **Krita**, el cual podremos encontrar de forma gratuita a través del siguiente enlace, tanto para **Windows**, **MAC** o **Linux**:

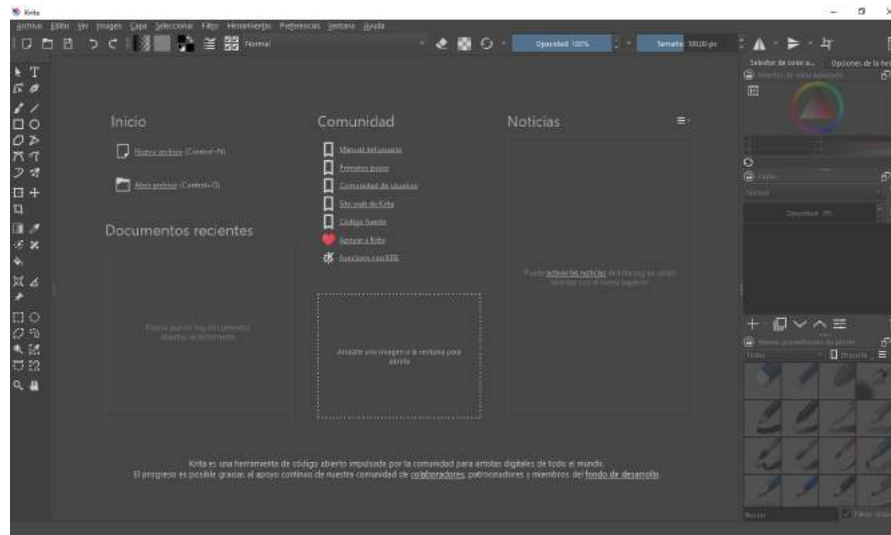
- <https://krita.org/es/>







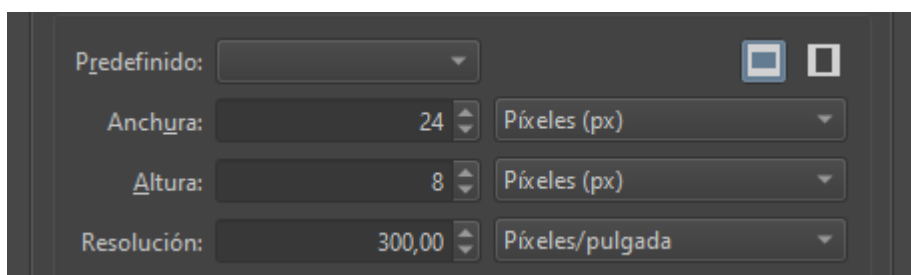
Una vez lo tengamos descargado procedemos a instalarlo y se nos abrirá de manera automática, mostrándonos su interfaz principal:



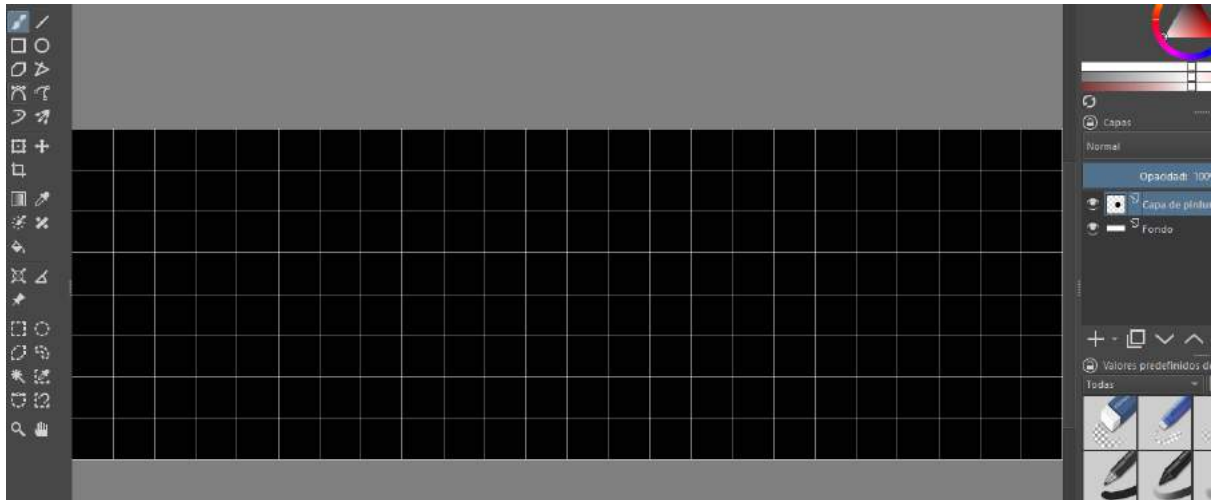
Como podemos comprobar, no dista mucho de cualquier herramienta actual de edición de imágenes como **GIMP** o **Photoshop**, situando las herramientas principales a la **izquierda**, la gestión de capas a la **derecha** y las herramientas globales en la zona **superior**.

Lo primero que debemos hacer es **crear** una nueva **capa de fondo**, buscando que ocupe lo mínimo posible y que cumpla con nuestras necesidades. Por ejemplo, si queremos simplemente un **Sprite de 8x8 píxeles** bastaría con crear un **nuevo archivo** en **Krita** con esas **dimensiones**, pero en nuestro caso lo vamos a crear de **24x8**, de modo que en el mismo documento podamos implementar **3 Sprites**. En dicho documento tendremos una separación perfecta entre los 3 Sprites y podemos crear una **pequeña animación** desde **Unity**, optimizando espacio y dando **vistosidad** al **Asset final**.

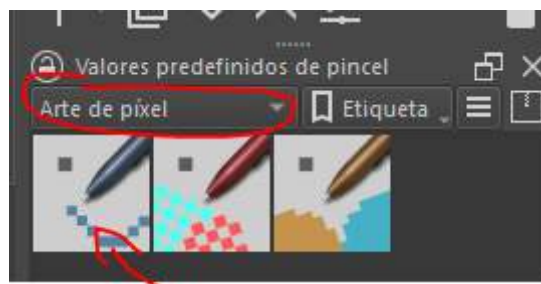
Para crear este documento simplemente nos iremos a **Archivo -> Nuevo** y estableceremos esta configuración:



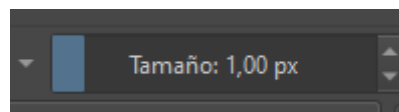
Se nos abrirá el documento automáticamente, con esta disposición:



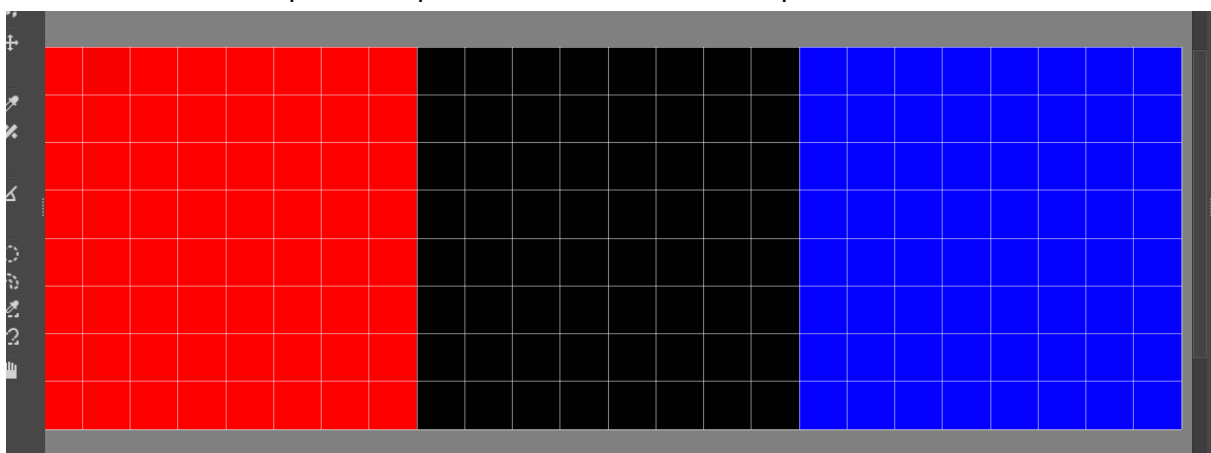
Como bien podemos ver tenemos una subdivisión perfecta entre cada pixel. Ahora vamos a dividir cada **bloque de 8x8** con un color diferente, para no pintar erróneamente sobre otro al crear los sprites. Para ello elegiremos la herramienta **pincel de pixel art**, en el selector de tipos (Situado en la zona inferior derecha), el cual siempre pintará el tono seleccionado en su máximo porcentaje de opacidad:



Del mismo modo estableceremos el tamaño de la punta al tamaño de un único píxel:



De este modo podemos proceder con la subdivisión por colores:





A continuación procederemos a ir dibujando nuestro **Sprite**, que será nuestro personaje jugable dentro de nuestro futuro nivel. En el bloque central dibujaremos el estado por defecto, que será mirando al frente. Los otros dos lados mirarán para el mismo lado pero modificando un tanto la imagen. Éstos los usaremos más adelante para crear nuestra **animación de movimiento en Unity**. Cuando hayamos terminado borramos los fondos, de modo que quede sin color base, sino con **alpha**:



Con esto ya tendremos terminado nuestro **Sprite** básico de **personaje en 2D**. Para futuras posibles modificaciones se recomienda que se almacene tanto el documento en formato **“.krita”** como su exportación en **PNG**, siendo ésta última la que utilizaremos en **Unity**, como ya vimos en nuestro ejemplo anterior de uso de **Sprites**.

## 7.2.- Creación de recursos de sonido con LMMS

Durante este paso nos encargaremos de descargar **LMMS** y configurar los elementos necesarios para crear una melodía base para nuestro nivel de plataformas. Buscaremos un estilo **retro chiptune**, el cual podremos lograr mediante las herramientas básica que nos ofrece el editor. Lo primero que necesitaremos es descargar la herramienta **LMMS**. Para ello podremos acudir al siguiente enlace web:

- <https://lmms.io/download>

Como podemos comprobar que la aplicación se encuentra disponible tanto para **Linux**, **Windows** y **MAC**. Se recomienda la **descarga** de una **versión estable** para evitar malos funcionamientos posibles. En mi caso personal cuento con un SO windows de 64bits, por lo que descargaré su versión estable correspondiente:

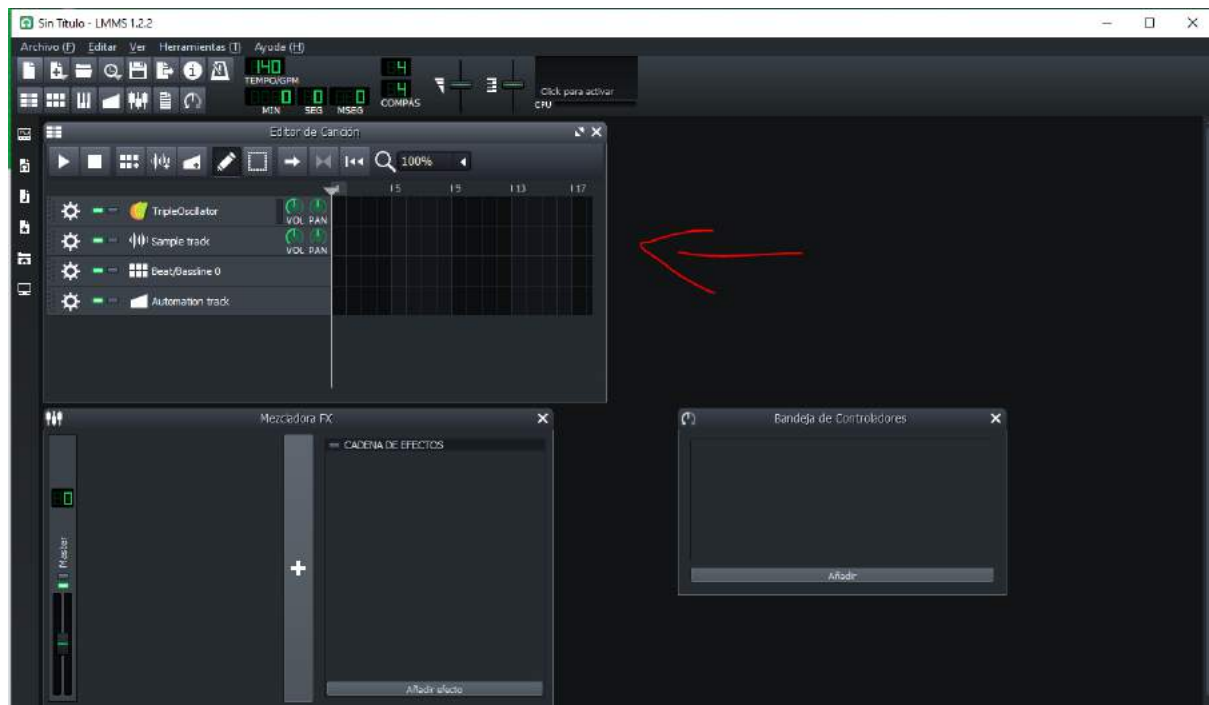


## Versiones Estables

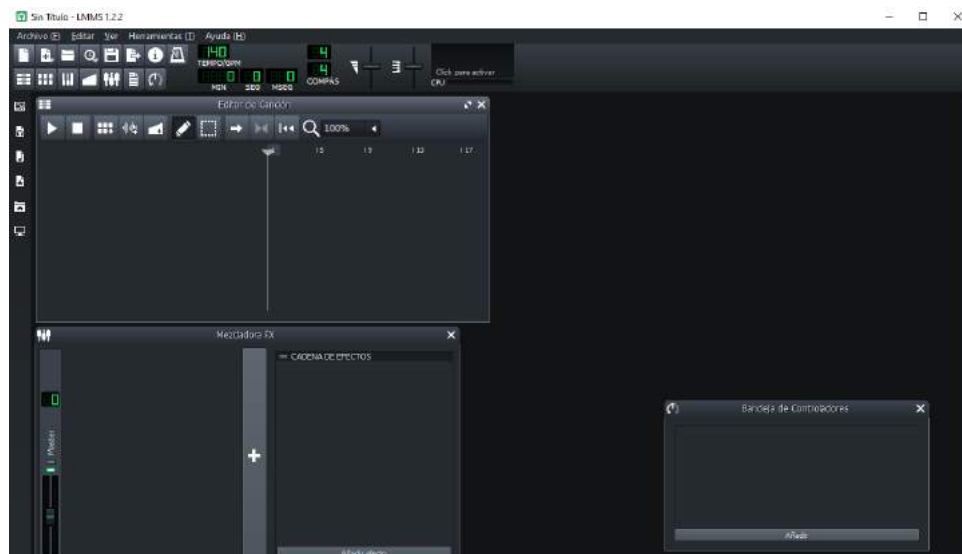


[Mostrar las notas del lanzamiento](#)

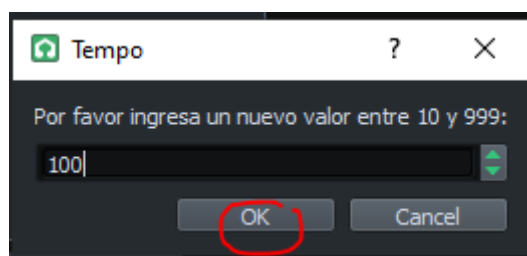
Una vez desagrada procederemos con su instalación, la cual no requiere de ningún paso destacable a mostrar. Tras la instalación se nos abrirá automáticamente el programa y nos mostrará la siguiente interfaz:



La principal ventana que usaremos es la señalada con la flecha roja en la imagen de arriba, la cual se denomina **Editor de canción**. Por defecto se compone de algunos elementos básicos listos para empezar a crear una melodía, pero como queremos no depender de una configuración o plantilla concreta lo primero que haremos pues será crear un nuevo proyecto vacío. Para ello pulsaremos en **Archivo -> Nuevo desde plantilla -> Empty**, para crear un proyecto sin configuración concreta. Al no tener ningún tipo de configuración nos aparecerá prácticamente vacío:



Si nos fijamos en la parte superior de la herramienta veremos que se nos indica que por defecto el **tempo** (pulsaciones por minuto que definen la velocidad) de nuestra melodía, el cual podemos ver que se encuentra establecido en **140**. Lo primero que haremos es **reducirla**, de modo que se quede en **100** para crear una melodía más pausada. Para ello simplemente pulsamos sobre el número y se nos desplegará un selector numérico para efectuar el cambio:



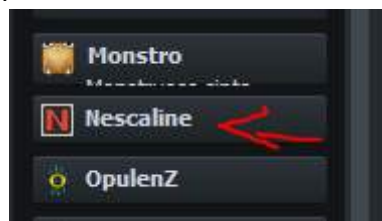
Lo siguiente que haremos es elegir el **instrumento** que vamos a utilizar. Para ello pulsaremos en la primer herramienta que se encuentra en la barra lateral izquierda de LMMS:



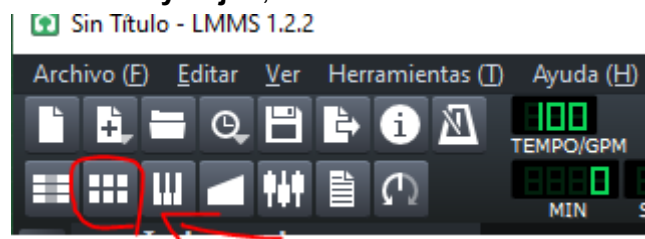
Se nos desplegará un listado de los **Plugins de instrumentos** disponibles para usar y que ya vienen por defecto con la instalación de LMMS:



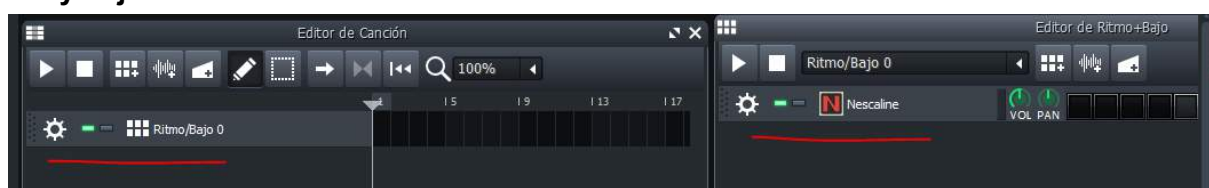
Cómo estamos desarrollando un videojuego de plataformas en 2D y con mecánicas simples buscamos también crear una melodía que encaje con ese enfoque. Como pudimos ver cuando analizamos la evolución de la industria, este género era predominante en la etapa de la **NES** y la **Master System**, las cuales eran consolas de 8 bits, por lo que utilizaremos un **instrumento** que **emule chips de sonido de esa época**. LMMS nos ofrece para ello diversos instrumentos digitales, entre los cuales seleccionaremos de momento **Nescaline**, el cual emula el chip de sonido de la **NES**:



Lo primero que suele hacerse a la hora de componer es encontrar una **base** o **ritmo** marcado sobre la que componer la melodía. Por ello de momento nos vamos a centrar en la herramienta de edición de **ritmo y bajos**, la cual se encuentra en esta pestaña:



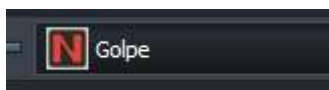
Arrastraremos el instrumento **Nescaline** a nuestro editor de ritmos y bajos. De este modo aplicaremos nuestra primera herramienta, quedando marcado **tanto en el editor de ritmo y bajo como en el editor de canción**:







Ahora centrándonos en la herramienta de **ritmos y bajos**, vamos a cambiarle el nombre al elemento para que quede mejor identificado. En este caso queremos simular un tono constante que marque el ritmo, por lo que lo llamaremos **“Golpe”** pulsando con **doble click sobre el nombre** para renombrarlo:



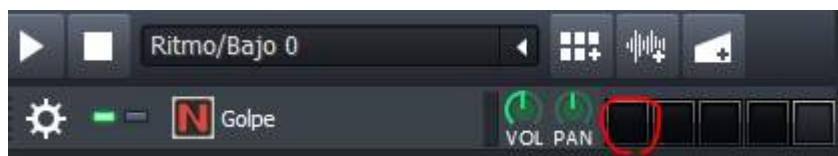
Al pulsar sobre el elemento se nos abrirá una **caja de mezcla de sonido** que nos ofrece todas las posibilidades de **Nescaline**. El objetivo ahora es tocar las configuraciones hasta encontrar un sonido que nos agrade:

Si nos fijamos en el teclado virtual inferior veremos cómo se compone de varias notas. Si con nuestro teclado pulsamos en las teclas **“QWERTYUI”** en ese orden estaremos tocando las notas **“DO-RE-MI-FA-SOL-LA-SI-DO”** desde **C4** hasta **C5**.

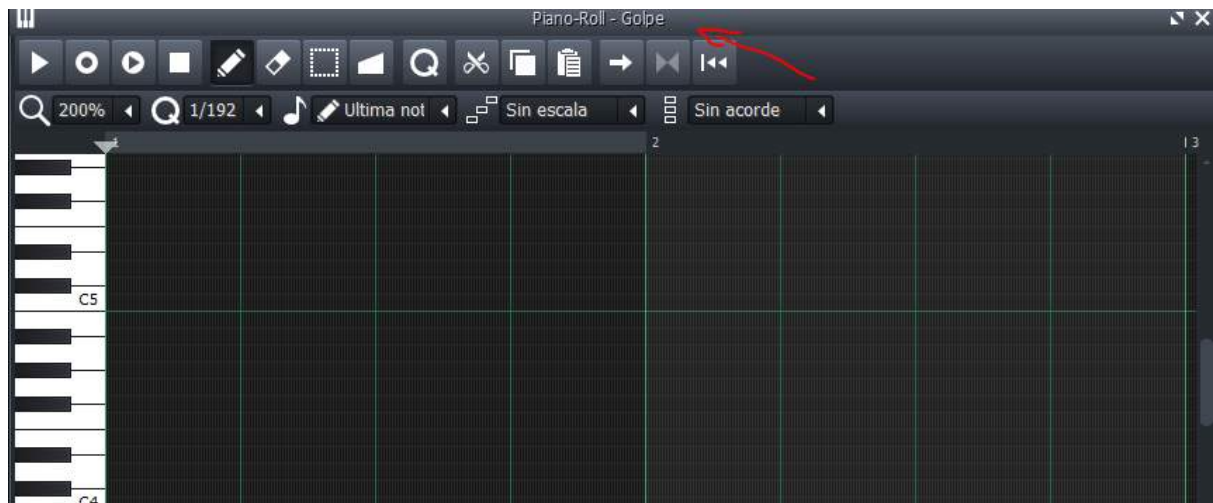
Como en composición musical **C** es el equivalente a **DO**, nos centraremos en la tecla **Q** o en la tecla **I** para ir buscando nuestro tono principal de ritmo modificando la mesa de mezcla que podemos ver en la imagen. Algo que deberemos tener en cuenta es que disponemos de **4 canales de sonido con esta herramienta**. Por defecto viene marcado el canal 1,2 y 3, pero como nosotros **únicamente queremos marcar el ritmo dejaremos marcado el canal 2 únicamente**.



Una vez hayamos conseguido una configuración que nos dé un sonido agradable para el teclado simplemente cerraremos el mezclador y volveremos al editor de **ritmo**:



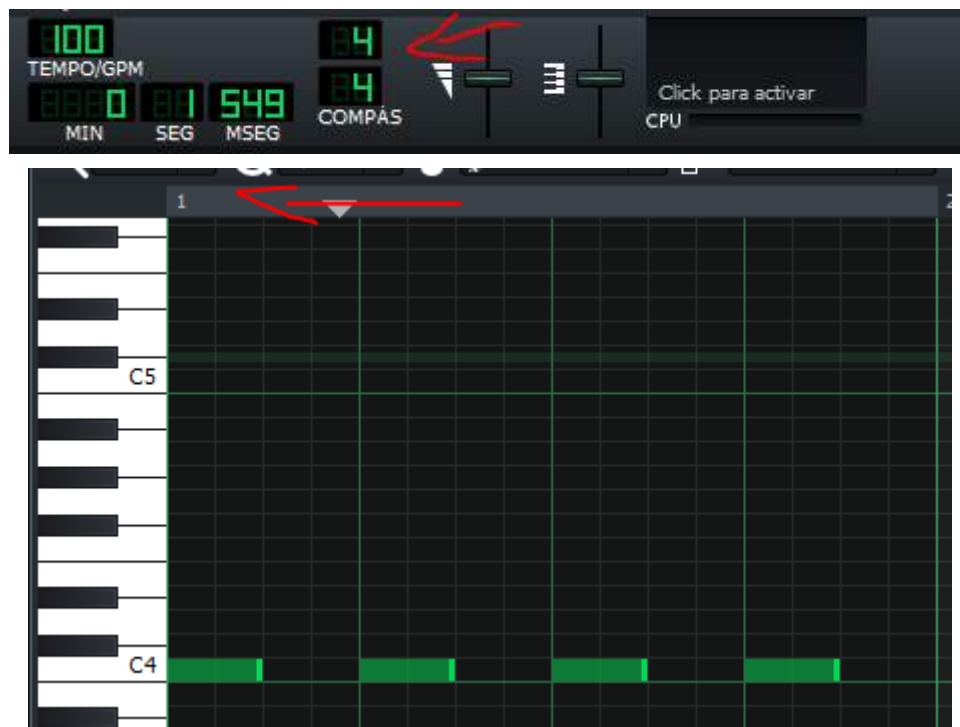
En el apartado señalado pulsaremos **click derecho** y luego en **Piano Roll**, para que se nos abra la ventana de composición, la cual se va a encontrar adaptada a la configuración que acabamos de realizar en la mesa de mezcla. La ventana de **Piano Roll** sería tal que así:



Lo primero que haremos es modificar el **valor de cuadrantes** para reducir la cantidad de celdas de la composición. Escogeremos el valor **1/16**:



Definiremos un ritmo de **4 golpes en la misma nota**, pero de modo que exista una separación de silencio entre cada pulsación. En mi caso estoy definiendo un ritmo de 4 golpes porque el ritmo que marquemos siempre debe ajustarse al **compás** que estemos definiendo, y en nuestro caso es por defecto un **compás** de **4/4**, que es el más estandarizado:

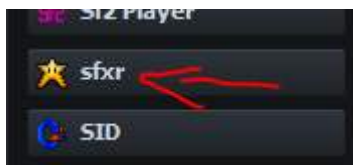




De hecho el propio compás es identificable en el propio **Piano Roll**, donde cada set o compás de audio se está componiendo de **4 golpes de sonido** (Columnas) y cada uno de estos es **divisible en 4 sub golpes**. Una vez lo hayamos logrado cerraremos la ventana de **Piano Roll** y nuestro audio aparecerá reflejado en la **ventana de ritmo y bajos**:



Pulsaremos en el botón de **Play** para comprobar el resultado, el cual siempre se mostrará como un **Loop infinito por defecto**. Si no estamos satisfechos con el resultado volvemos a abrirlo con **Piano Roll** y haremos los ajustes necesarios. En mi caso estoy satisfecho con el resultado, **pero no hemos terminado con el ritmo**, ya que solo hemos establecido el golpe de marcado, pero éste debe ser acompañado por una **cobertura de batería** para darle cierto empaque. Lo normal sería agregar otro elemento **Nescaíne** a la mesa de edición de ritmos e ir jugando con los otros 3 canales, pero **LMMS** nos ofrece una herramienta mejor para nuestro objetivo: **El instrumento sfxr**, que tiene forma de la **estrella** de invencibilidad de **Super Mario Bros**:



Esta herramienta nos va a ofrecer mucha maniobrabilidad a la hora de realizar melodías **chiptune retro**. Le cambiamos el nombre por defecto a **Bateria1**:

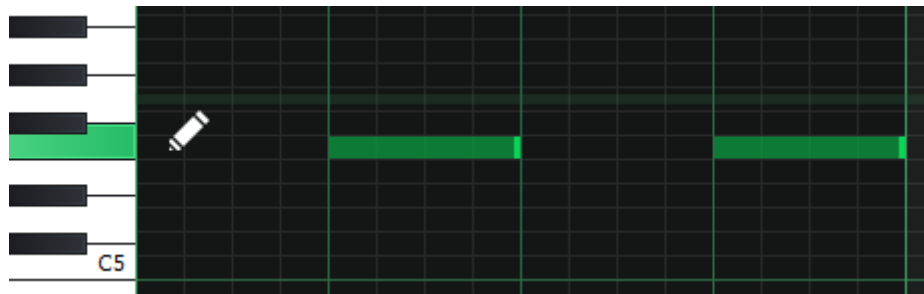




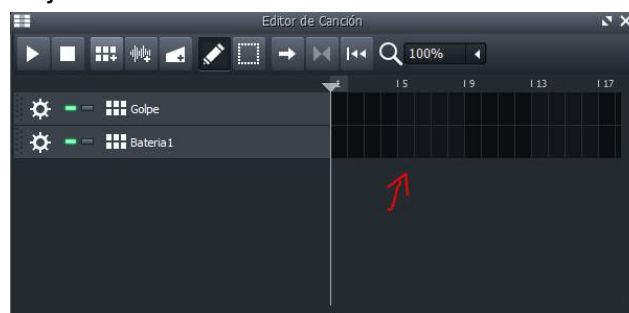
Si hacemos doble click sobre el nombre se nos abrirá su mesa de mezcla, que es un poco diferente a la de **Nescline**:



Buscaremos una configuración que nos agrade, del mismo modo que explicamos con **Nescline** y una vez lo hayamos configurado correctamente lo abrimos con **Piano Roll** y definimos el ritmo:

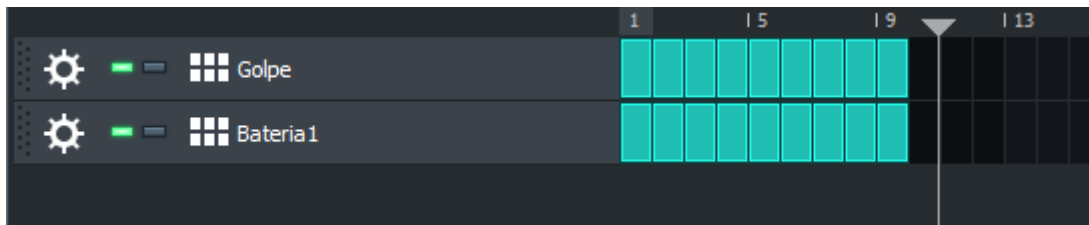


Cuando lo tengamos acabado cerraremos la ventana y acudiremos a la ventana de editor de canción y nos fijamos un momento en la **zona de la derecha**:





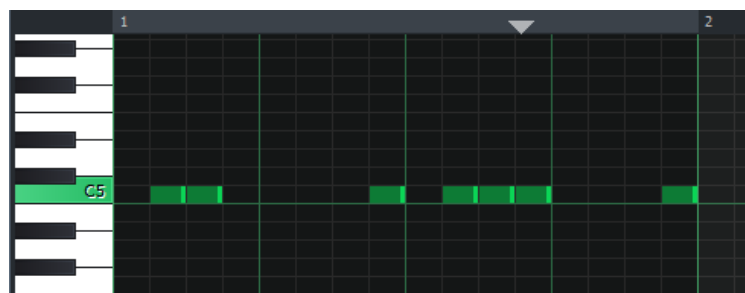
A medida que vayamos pulsando en los **huecos** de la derecha se va a ir insertando el ritmo de cada fila en su posición. Buscaremos la configuración deseada. Como en nuestro caso estamos definiendo ambos elementos como ritmos, por lo que nos interesa que se ejecuten a la vez para comprobar su armonía:



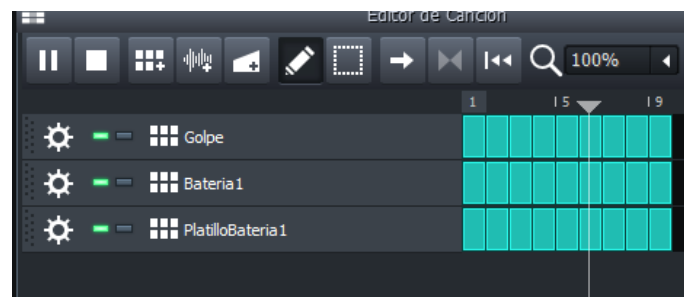
Finalmente para terminar de definir nuestro ritmo o base de la melodía es conveniente agregar un **platillo** de acompañamiento, aunque no sea 100% necesario. Para ello agregaremos otro elemento **sfxr** a nuestra mesa de edición de ritmos y bajo y lo llamaremos **PlatilloBateria1**:



Una vez que lo hayamos configurado a nuestro gusto simplemente lo abriremos como **Piano Roll** y definiremos la musicalidad del platillo:

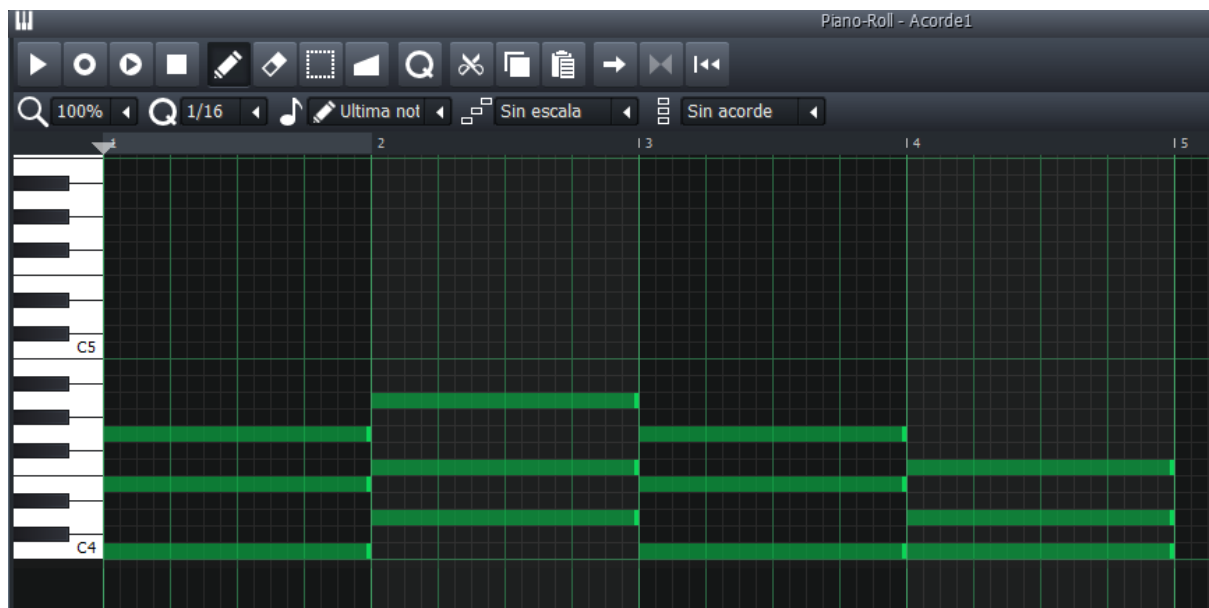


Una vez lo tengamos terminado cerramos la ventana y en la mesa de edición de canción comprobamos el resultado:

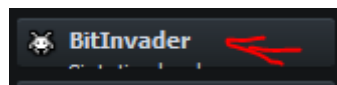




A continuación toca declarar un acorde de nuestra melodía. Para ello agregamos a la mesa de ritmo y bajo un elemento **Nescaliner** y lo configuramos de modo que use únicamente los canales 1 y 2. Buscamos la configuración que nos satisfaga y acudimos a **Piano Roll** para crear la melodía. En este caso estamos buscando **acordes de acompañamiento simple**, los cuales deben respetar también el **compás de 4/4**, por lo que a la hora de crearlos debemos fijarnos en su duración de forma constante basándonos en dicho compás. Debería quedarnos algo similar a esto:



A continuación vamos a definir el tema principal de toda la melodía, el cual se va a ver acompañado de todo lo que tenemos ya creado. Para ello utilizamos otra herramienta instrumental chiptune que nos ofrece **LMMS**, la herramienta **Bit Invader**:



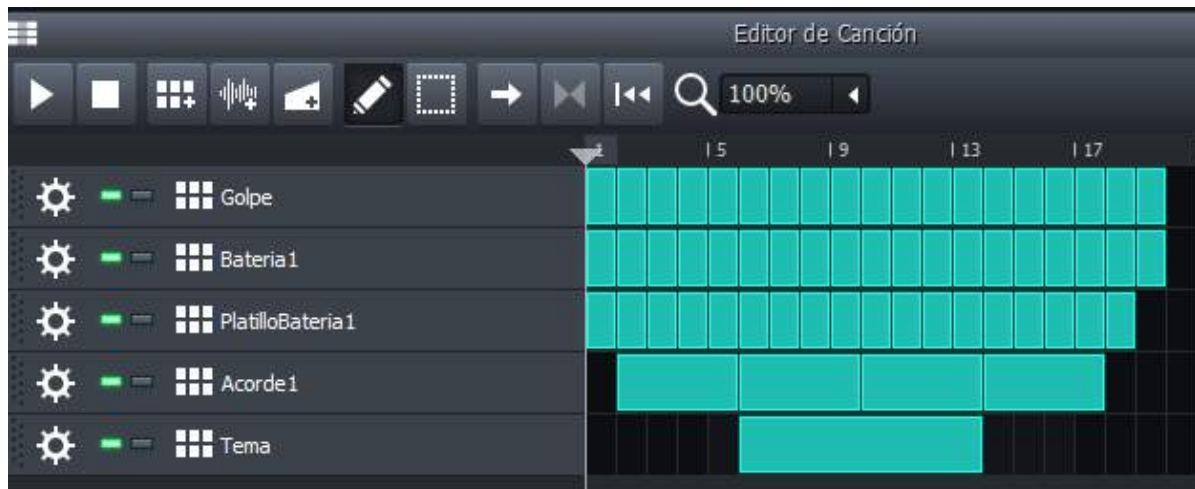
La arrastramos a nuestra ventana de editor de ritmo y bajo, le establecemos como nombre **Tema** y lo configuramos hasta obtener el resultado deseado. Una vez configurado acudimos a su piano roll para la composición. En nuestro caso ha quedado la siguiente composición:



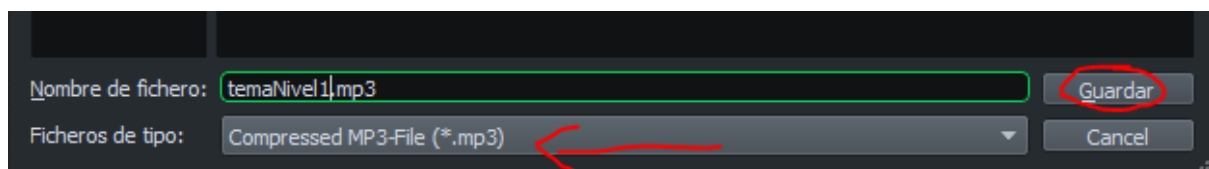




Una vez hayamos implementado el tema nos iremos al editor de canciones y ajustaremos las entradas de cada parte de la canción. Estableceremos que **Golpe** y **Bateria1** entren primero, posteriormente comenzarán los **acordes** y tras acabar el primer recorrido de acordes comenzará la **melodía**. Puede que nos quede un poco corta pero no nos importa, ya que solo estamos desarrollando un único nivel y vamos a usarla a modo de **loop musical**.

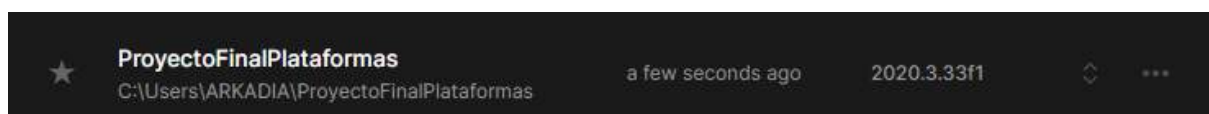


Cuando tengamos ya el resultado deseado guardamos el fichero con extensión **MP3**.



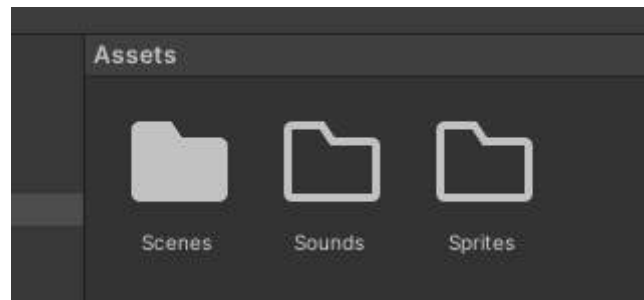
### 7.3.- Creando nuestros propios Assets en UNITY a partir de nuestros recursos

Una vez tengamos listos nuestros recursos podremos importarlos directamente en **Unity**. Para ello vamos a crear un nuevo proyecto en **Unity**, donde almacenaremos nuestros recursos finales y posteriormente desarrollaremos el correspondiente nivel de plataformas. A este proyecto lo denominaremos **ProyectoFinalPlataformas**, el cual será también un proyecto con **núcleo** o **core 2D**.



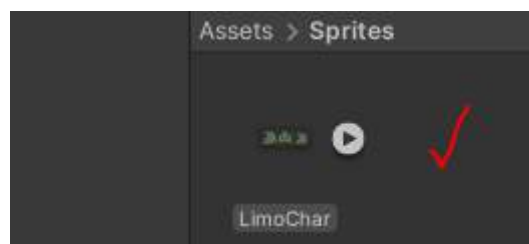


Una vez abierto nos iremos a la arquitectura del proyecto, en la zona inferior, buscaremos paquete de **Assets** y crearemos dos nuevas carpetas con **click derecho -> Create -> New Folder**. Les pondremos de nombre **Sprites** y **Sounds**:

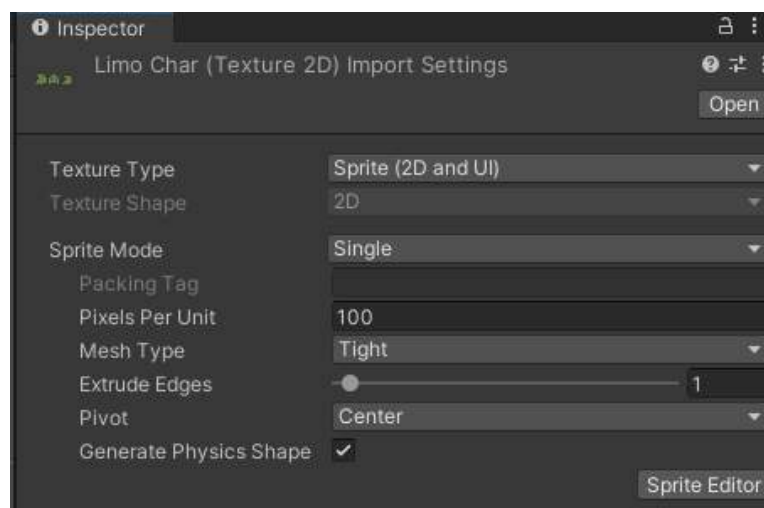


### 7.3.1.- Definiendo un Asset gráfico

Para definir un **Asset gráfico**, en nuestro caso en **2D**, iremos a nuestra **carpeta de Sprites** que acabamos de crear dentro de nuestra **carpeta Asset** y hacemos click derecho -> **Import new Asset** y buscamos el **PNG** de nuestro documento de personaje creado anteriormente:



Si pulsamos sobre el recurso veremos que sus opciones por defecto son las siguientes, destacando que su modo de actuación es singularizado:

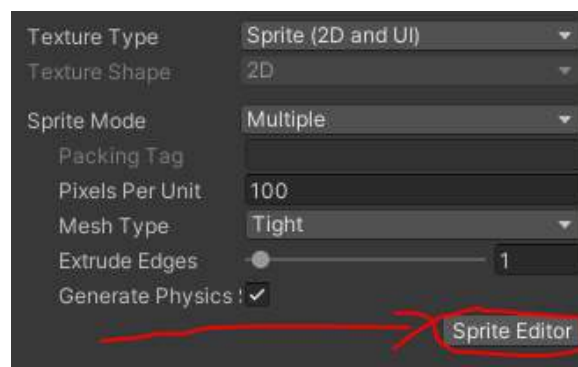




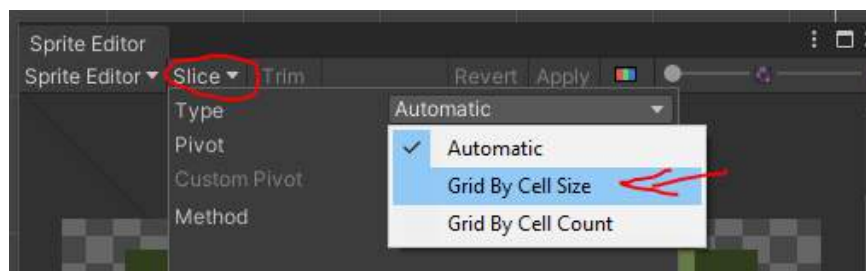
Esto quiere decir que **Unity** está entendiendo que todo el recurso conforma el **Asset**, pero nosotros no queremos eso, sino que el Asset se conforme de 2 estados distintos: **Quieto** y **En Movimiento**. Por ello debemos establecer el modo del **Sprite** en **Múltiple**:

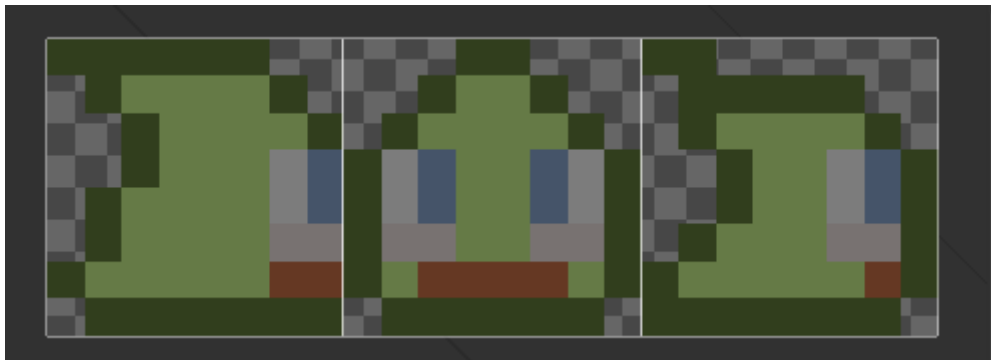
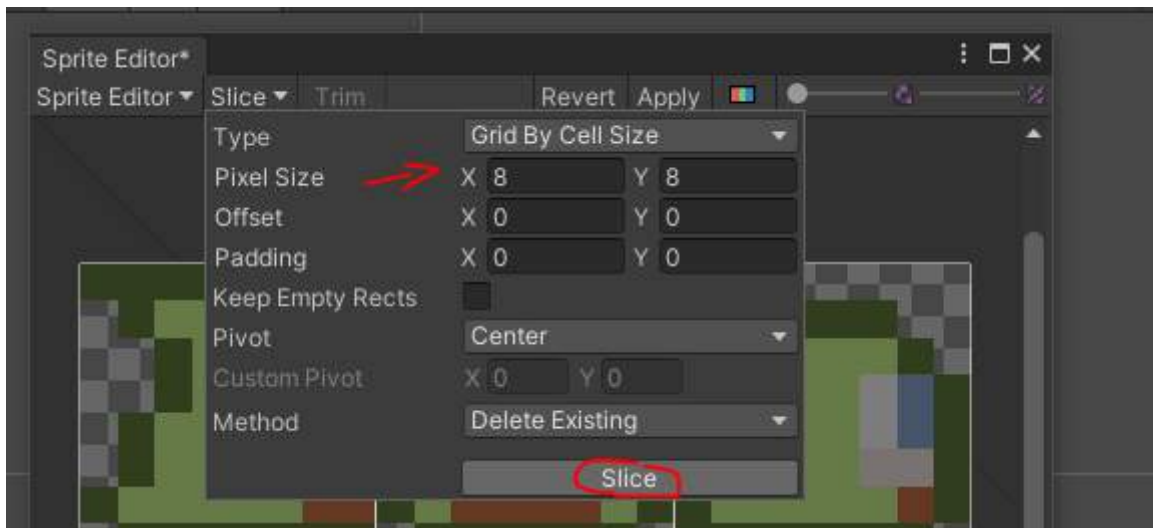


Pero con esto no basta, ya que aunque ahora **Unity** entienda que el recurso contiene múltiples instancias o estados, tenemos que especificarle a partir de cuándo comienza una y otra. Para esto pulsaremos en **Sprite Editor**:

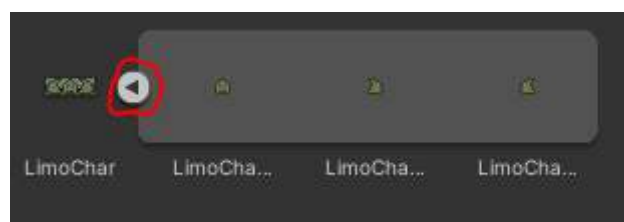


Nos abrirá una nueva ventana en la que deberemos pulsar en **Slice** para separar la imagen en bloques o grids en base a un número fijo de celdas, que en este caso serán bloques de **8x8**:





Como podemos ver ya tenemos nuestro Asset conformado por 1 Sprite subdividido en 3 grids, por lo que hemos optimizado mucho el recurso. Si aplicamos cambios veremos cómo al pulsar en el explorador de recursos de nuestro **PNG**, el **Asset** ahora se compone de 3 subGrids:

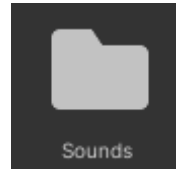


Con esto ya tendríamos nuestro Asset definido y listo para ser aplicado a un objeto de juego, lo cual haremos cuando en breves expliquemos cómo crear animaciones en Unity.



### 7.3.2.- Definiendo un Asset de sonido

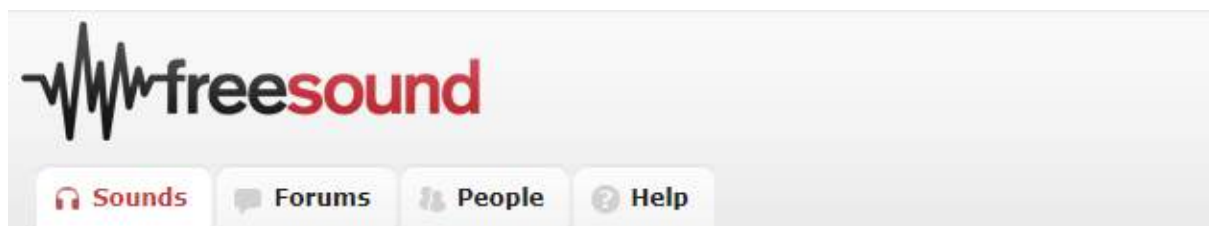
A continuación vamos a implementar el tema que compusimos previamente con **LMMS** en **Unity**. Para ello lo primero que deberemos hacer es acudir a nuestra carpeta de **Assets** y crear dentro una carpeta a la que llamaremos **Sounds**.



Accederemos dentro de nuestra nueva carpeta y haremos click derecho para importar un nuevo **Asset**. Se nos desplegará una ventana de exploración de archivos y buscaremos nuestro archivo de audio. Una vez agregado nos aparecerá dentro de nuestra carpeta **Sounds**:



El siguiente paso será agregar el resto de assets de audio. Necesitaremos un sonido para la **recolección de objetos** (Que en este caso serán manzanas), para el **salto del personaje principal** u para la **muerte** del personaje principal y de los enemigos principales del nivel (**arañas**). Para conseguir estos archivos tenemos 2 opciones: Crearlos nosotros con **LMMS** u otro programa de sonido o encontrar los recursos por internet. En mi caso los he encontrado a través de la web **FreeSounds**, de forma totalmente gratuita:



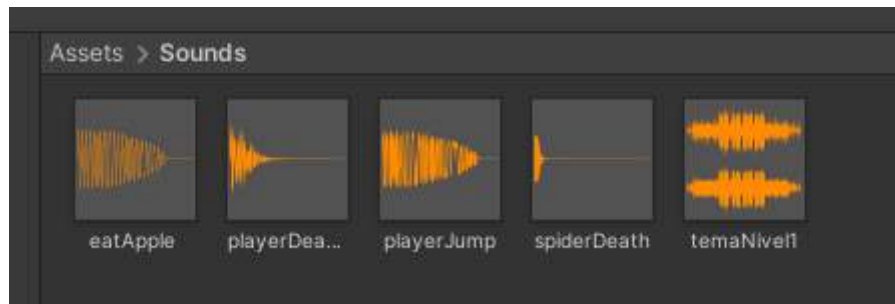
## Pack: 8-Bit Sound Effects Library by LittleRobotSoundFactory

Adjunto además enlace para rápida consulta:

- <https://freesound.org/people/LittleRobotSoundFactory/packs/16681/>



Para agregarlos a **Unity** lo haremos del mismo modo que ya hicimos con el tema principal, por lo que en el momento actual deberíamos contar con estos recursos importados:

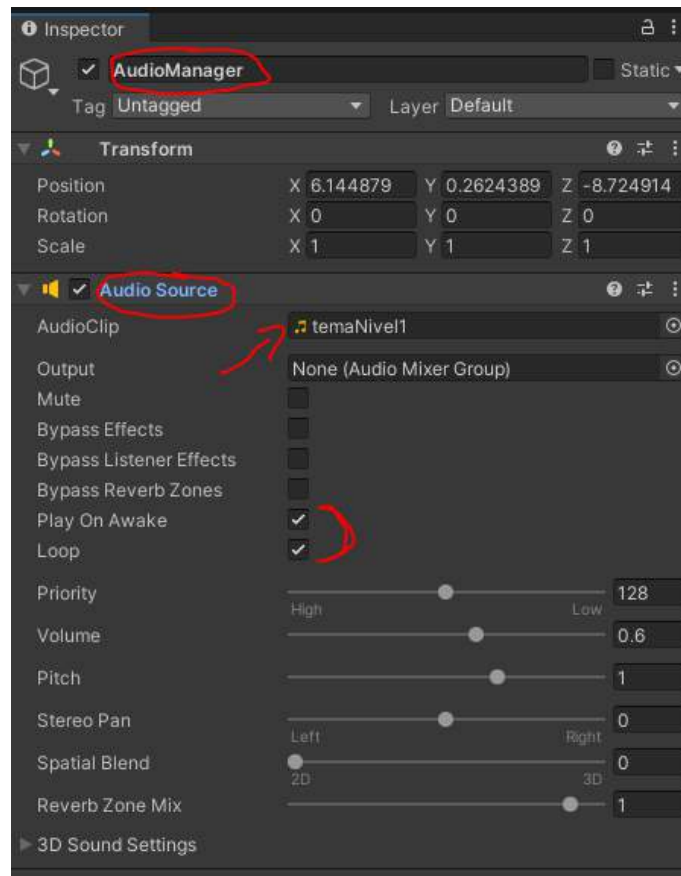


Ahora tenemos que aprender como poder utilizar estos recursos desde el código. En **Unity** existen 2 formas principales de integrar sonidos:

- **Integración por inicialización:** Consiste en aplicar un **motor de audio** (Componente **Audio Source** de **Unity**) de audio a un objeto de juego de modo que se reproduzca el sonido desde el mismo momento en que dicho objeto entre en escena.
- **Integración por activación:** Consiste en aplicar un motor de audio a un objeto de juego de modo que se reproduzca cada vez que el objeto de juego sea activado y se haga un llamamiento directo al motor de sonido.

Para implementar el tema principal del nivel que compusimos previamente en **LMMS** utilizaremos el primer tiempo de integración. Para ello deberemos buscar un objeto de juego que nos sirva para nuestro propósito. Como buscamos implementar un sonido que dure durante todo el nivel necesitaremos un **objeto** que **siempre** esté **disponible** en éste. Por eso la mejor opción y la más obvia sería crear un objeto de juego que se encargue de manejar dicha melodía de una forma constante y sin dependencias de otros objetos de juego. Para ello iremos a nuestra jerarquía y con click derecho crearemos un nuevo objeto de juego al que llamaremos **AudioManager**. A dicho objeto le agregaremos un componente de motor de audio, es decir **Audio Source**:



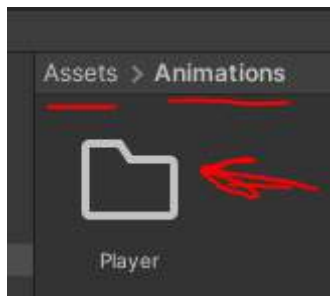


En la opción de **AudioClip** buscaremos el audio que se desea reproducir. Es muy importante marcar la opción **Play On Awake** y **Loop** para que el tema se reproduzca desde que se inicie la escena y para que al finalizarse la melodía ésta vuelva a empezar.

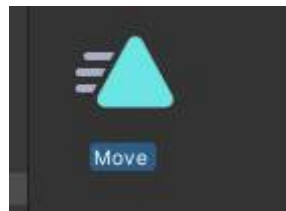
Con esto ya tendríamos nuestra primera implementación de sonido. Más adelante, cuando comencemos con la creación del nivel de plataformas veremos como realizar el otro tipo de integración de sonidos, que es el que más utilizaremos.

### 7.3.3.- Definiendo animaciones

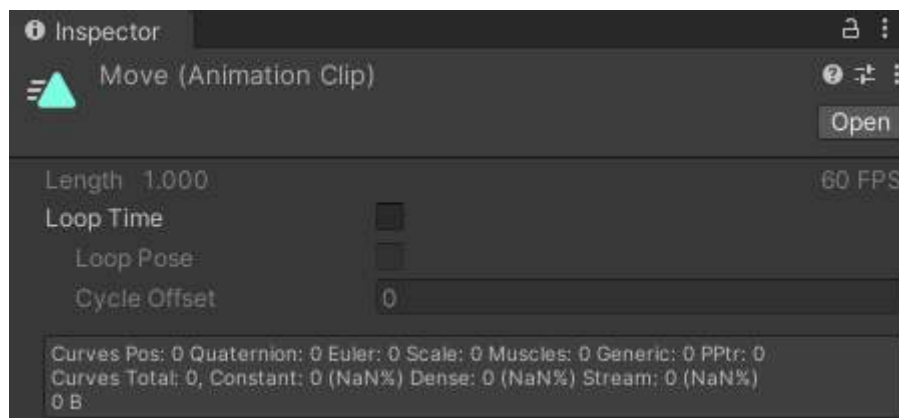
Una vez que tenemos definida una hoja de Sprites (**Sprite Sheet**) para la definición de una animación, podemos empezar a crearla. Para ello en el paquete **Assets** de nuestro proyecto vamos a crear una nueva carpeta llamada **Animations** y dentro de ésta vamos a crear otra por cada objeto de juego que queramos animar. Como de momento solo tenemos a nuestro personaje principal crearemos una única carpeta a la que llamaremos **Player**:



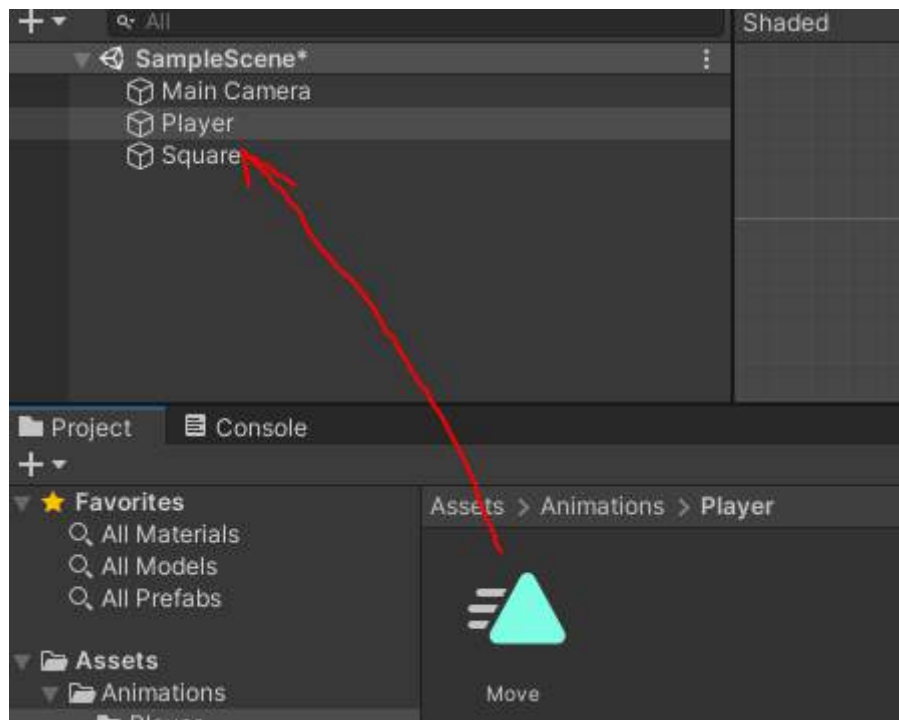
A continuación creamos un tipo de **Asset** nuevo dentro de esa carpeta llamado **Animation**. Para ello simplemente haremos click derecho dentro de la carpeta y pulsamos en **Create -> Animation**. A esa animación la llamaremos **Move**, ya que va a definir el movimiento del personaje:



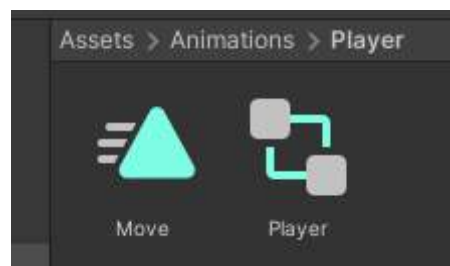
Si pulsamos sobre nuestra animación veremos que en su inspector nos aparece una breve definición y posibilidad de ajuste de funcionamiento:



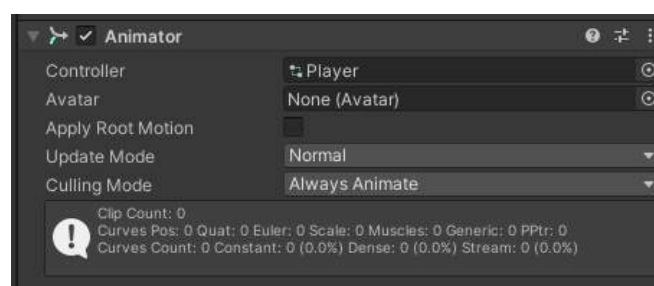
De base debemos marcar **Loop Time**, ya que es la funcionalidad que permite que una animación se repita en el tiempo infinitamente. De lo contrario solo se ejecutaría una única vez. Ahora debemos arrastrar con el ratón nuestra animación hacia el **GameObject Player**:



Esta es una funcionalidad de **Unity** denominada “**Drop'NLinked**”, sirve para crear componentes preconfigurados directamente arrastrando un recurso a un **GameObject**, lo cual nos ahorra el proceso de crear los componentes de uno en uno para cada **GameObject**. Si todo ha ido bien, se ha debido crear un controlador de animación dentro de nuestra carpeta de animaciones, la cual debe tener el mismo nombre que el **GameObject**:



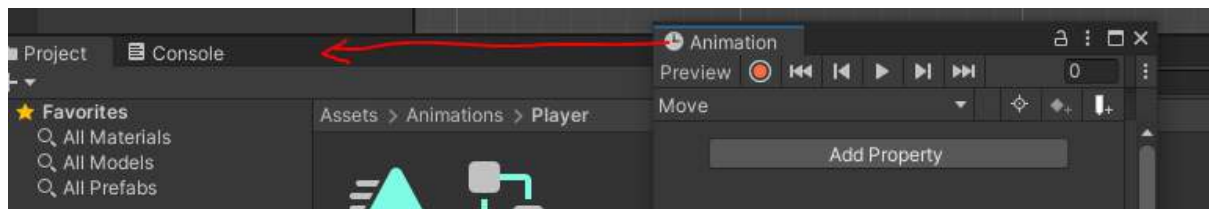
Y por supuesto si ahora comprobamos el inspector de nuestro **GameObject** debería tener un nuevo componente agregado. En este caso es un componente **Animator**:



A continuación tenemos que manipular la animación. Para ello **Unity** cuenta con una herramienta interna llamada **Animation** y que es accesible desde **Window -> Animator -> Animation**:



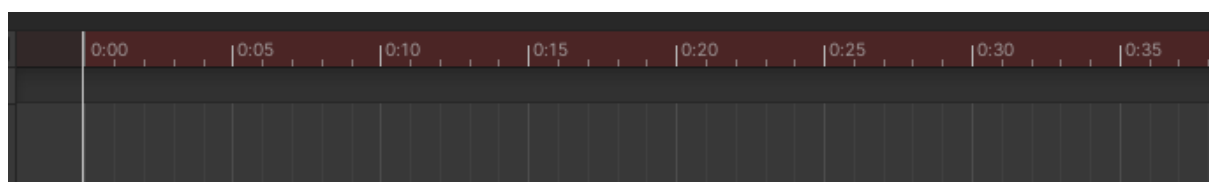
Si nos desplegará en pantalla pero para mayor comodidad vamos a disponerla en nuestra barra de pestañas principal (la derecha de la consola). Para ello simplemente arrastramos la pestaña señalada a la posición marcada:



Una vez dentro de la herramienta comprobaremos que contiene un menú desplegable con todas las animaciones que contiene nuestro proyecto. De momento únicamente debería encontrar la animación **Move**, la cual de hecho se marcará por defecto:



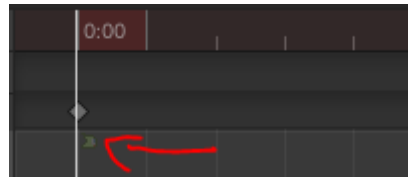
Para activar la modificación manual de la animación **Move** simplemente la elegimos (en este caso no haría falta ya que se ha marcado por defecto) y pulsamos en el icono de **REC** (circunferencia roja). Comprobamos que el **modo edición** está **activado** si la barra de tiempo se vuelve de tonalidad **rojiza**:



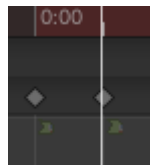
Ahora nos vamos a nuestro objeto de juego **Player** y en su **Sprite Renderer** establecemos el primer **Sprite** en movimiento, en nuestro caso se llama **LimoChar\_2**:



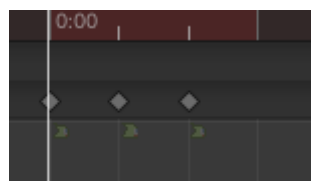
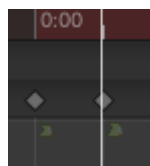
Veremos como en el frame actual se ha marcado dicho sprite en la barra de tiempo:



Avanzamos la siguiente frase y ahora escogemos el segundo **Sprite** del **Player** corriendo, que en nuestro caso se llama **LimoChar\_0**:



Finalmente avanzamos al frame 3 y volvemos a escoger **LimoChar\_2**. La idea es que la animación siga un **bucle infinito** del personaje corriendo o desplazándose:

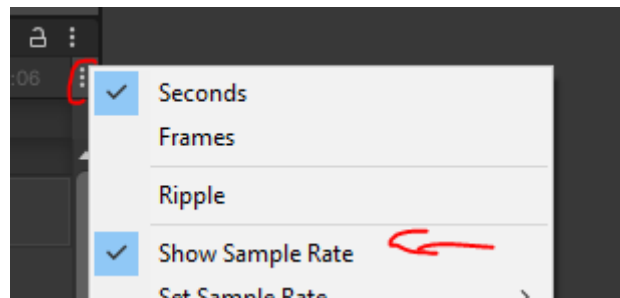


Con esto ya tendríamos la animación más o menos lista. Podemos probarla dándole al botón **Play** de la ventana **animator**:

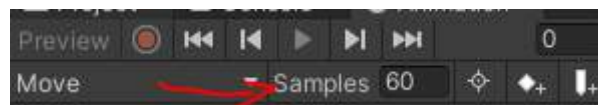




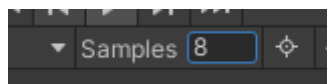
El problema principal es que por defecto **Animator** trabaja con animaciones de **60** fotogramas por segundo y nosotros solo hemos marcado 3 fotogramas, por lo que al momento de darle a Play la animación irá muy rápido. Para evitar esto deberemos modificar los frames de lectura. Para ello primero tenemos que poder modificarlos desde la herramienta, lo cual se hace pulsando en el **botón de tres puntos blancos** que se encuentra en la zona derecha de la herramienta **animator**:



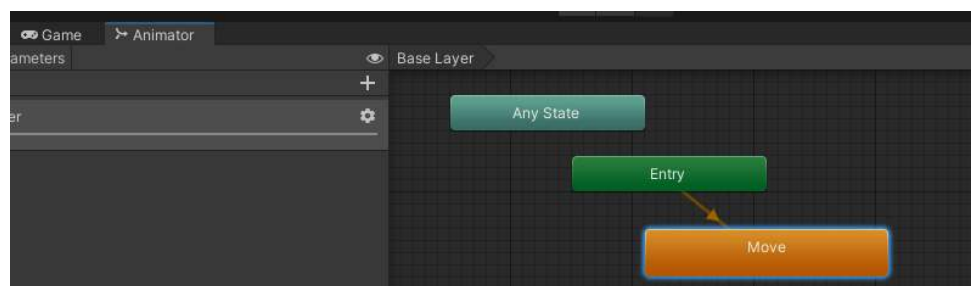
Veremos que ahora nos aparece el **calculador de Frames** en escena, por defecto con valor **60**:



Si lo ponemos **entre 6 y 8 frames** debería reproducirse correctamente. En mi caso lo dejaré en **8**:



Ya tenemos creada nuestra animación, pero no hemos terminado aún, ya que tenemos que aprender ahora a definir los **estados** de nuestro objeto de juego, de modo que entre sus diferentes estados se encuentra la posibilidad de la animación. Para ello vamos a abrir una ventana nueva que se llama **Animator** y la vamos a anclar en nuestra ventana principal, a la derecha de **Game**. La ventana **Animator** se encuentra en **Window -> Animation->Animator**:



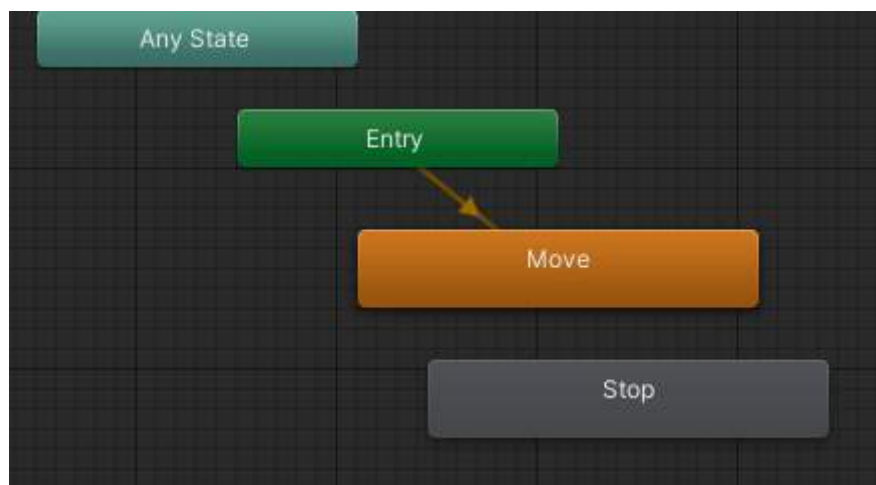




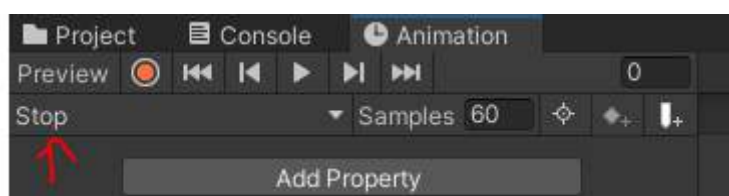
Esta ventana nos indica cómo se están enlazando las animaciones de nuestro objeto de juego. Por defecto lo que nos dice es que nada más entrar en escena (**Entry**) pasa al estado Move, que es la única animación que Unity ha encontrado para ese objeto y que es, de hecho, la que hemos creado. Podemos deducir entonces que ahora necesitamos una animación que deje al personaje quieto. Es decir que utilice únicamente el **bloque 8x8** central de nuestro Sprite. Para ello acudimos de nuevo a nuestra carpeta **Animations/Player** y creamos una nueva animación que llamaremos **Stop**:



De nuevo arrastramos esta animación al objeto **Player** para vincularla a su controlador. Si el enlace ha funcionado debería aparecernos en la ventana **animator**:



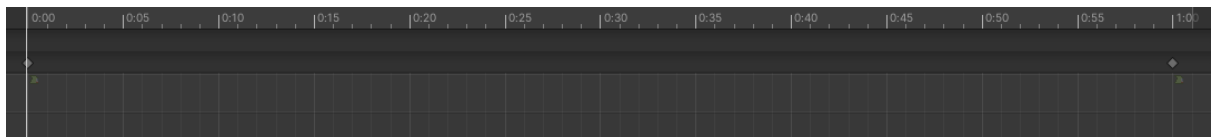
Ahora volvemos a la ventana **Animation** y seleccionamos la animación **Stop**:



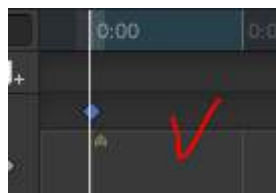
Queremos que siempre que se use esta animación se aplique el Sprite centrar, por lo que pulsamos en **Add property -> Sprite Renderer -> Sprite**



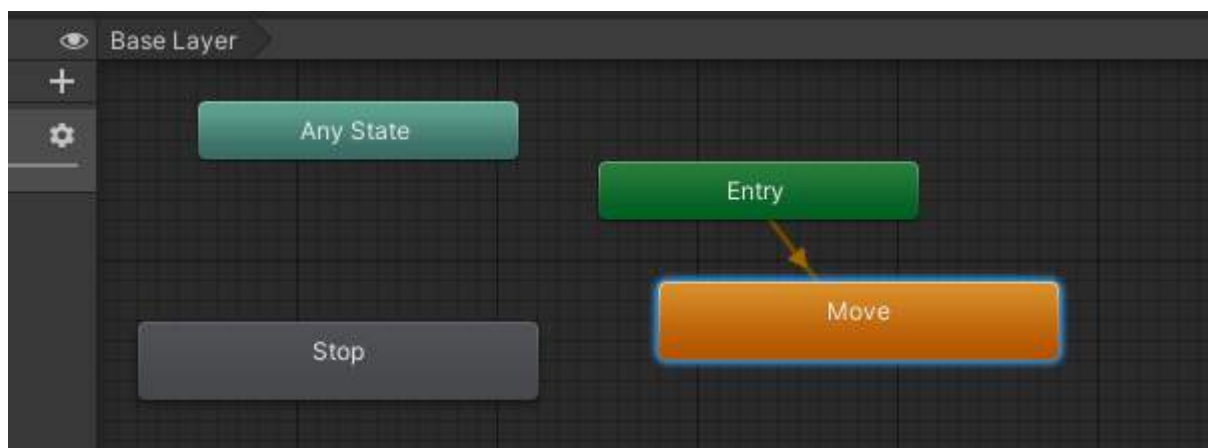
Por defecto ya dijimos que **Animation** trabaja con animaciones a **60 FPS**, que quiere decir que se ejecuta **60** veces en un segundo. Es decir que la animación dura por defecto **1 segundo**, por lo que tiene dos frames marcados: El **inicial** y el **final**:



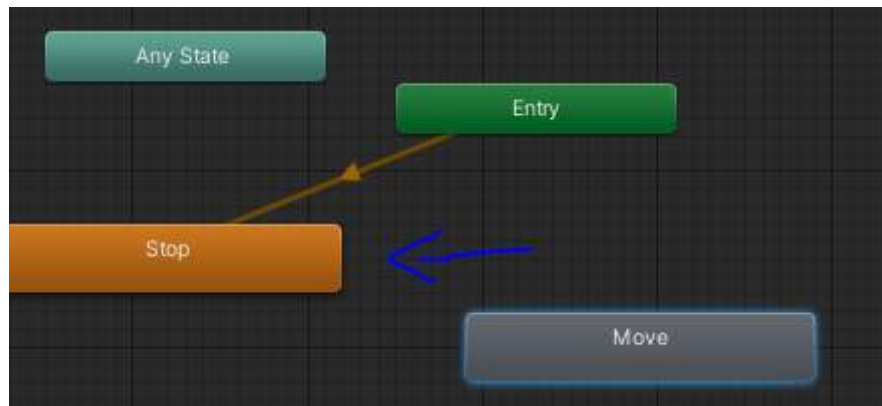
Nosotros queremos una animación que no dure una cantidad de tiempo **X**, sino que sea un **Sprite** eterno. Por ello seleccionamos el **frame final (1s)** y lo borramos con el **botón suprimir del teclado**. De este modo solo tendremos la inicial y debemos asegurarnos de que se ha marcado el **Sprite** centrar:



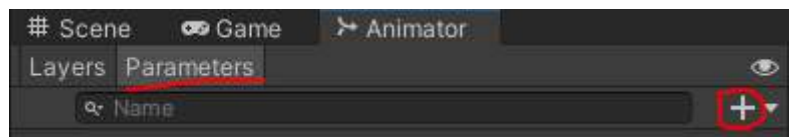
Ahora volvemos a la ventana **animator**:



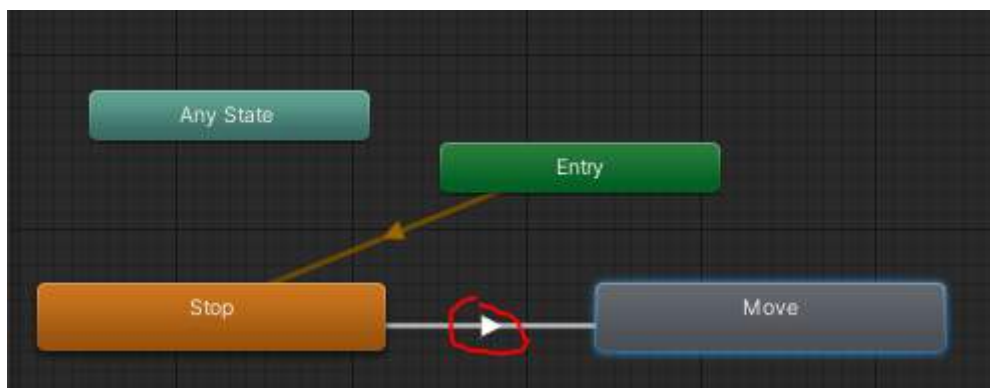
Hacemos click derecho sobre la animación **Stop** y la convertimos en el estado por defecto de nuestro objeto (**Set as Layer default state**).



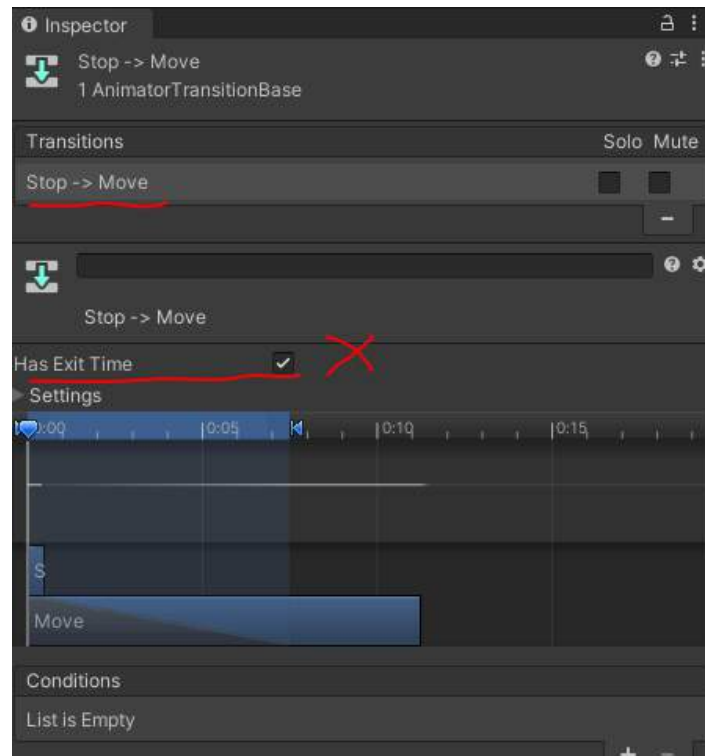
Ahora tenemos que **relacionar** las **animaciones** entre sí. Para ello la ventana **Animator** cuenta con la posibilidad de agregar parámetros relacionales. Para ello pulsamos en la pestaña **Parameters** y en la **Cruz** de agregar:



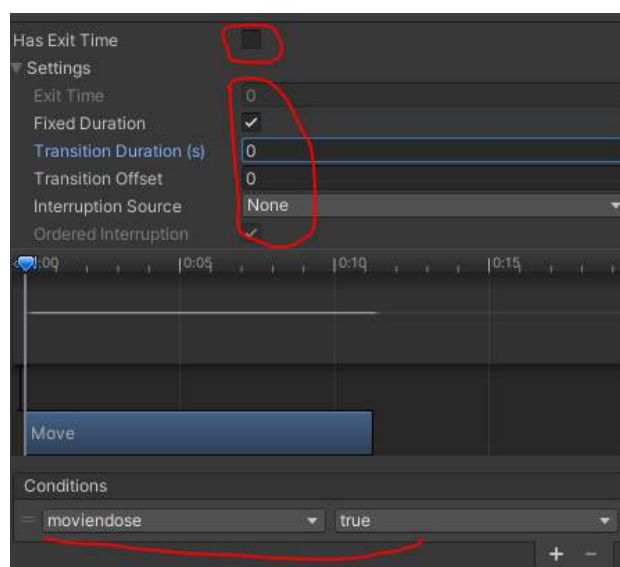
Agregamos un parámetro de tipo booleano al que llamaremos "**moviendose**". Ahora nos vamos a nuestra **animación Stop** y haciendo click derecho pulsamos en **Make Transition**. Nos aparecerá una flecha que podremos apuntar a otra animación. En nuestro caso apuntaremos a la animación **Move**, de modo que queden conectadas:



Seleccionamos en la flecha de transición o enlace y veremos como se desglosa a la derecha su página de inspector, con los datos relevantes de la relación:



Como podemos ver por defecto viene establecido o marcado la opción **Has Exit Time**, lo cual quiere decir que la animación tiene un tiempo de finalización. Además de esto si vemos en la parte inferior la lista de condiciones aplicadas a esa animación se encuentra vacía. Nosotros no queremos eso, sino que la animación funcione eternamente siempre que se encuentre en **TRUE** el parámetro booleano “**moviendose**” que acabamos de crear, por ello debemos dejar la configuración de este modo:





Con esto estaremos diciendo: “Siempre que estemos en la animación **Stop** y el parámetro **moviéndose** se encuentre en **TRUE**, se hará la transición a la animación **Move**”. Pero con esto estaremos haciendo una **transición unidireccional**, lo cual no tiene sentido para nosotros, ya que queremos que las animaciones funcionen de forma **cíclica**. Por ello tendremos que acudir a la animación **Move** y con click derecho crear una nueva relación pero que ahora apunte a la animación **Stop**. La configuración será igual que la anterior, solo que la verificación del parámetro moviéndose será **FALSE**, de modo que cuando nos encontramos en la animación **Move**, si moviéndose es **FALSE** se procederá a activar la animación **Stop**.

Una vez estemos manejando el **control del personaje** mediante **Scripts** de **C#** simplemente tendremos que preocuparnos de **alterar** el **valor** del **booleano** y las animaciones se activarán y comunicarán por sí mismas.

Con todo lo que hemos visto deberíamos ser capaces de poner en marcha la creación de un nivel de videojuegos de plataformas en 2D, que es el siguiente paso a dar.



## 8.- DESARROLLO DE UN NIVEL DE VIDEOJUEGO DE PLATAFORMAS

A la hora de plantear el desarrollo de un nivel de videojuego de plataformas una de las primeras cosas que debemos entender es el compromiso que el diseñador hace con el jugador planteando unas mecánicas simples, reconocibles y que desde el inicio ya obliguen al jugador a entender el funcionamiento básico del videojuego. Para ello podemos apoyarnos en una breve porción de una entrevista realizada a **Shigeru Miyamoto** sobre el diseño de primer nivel de **Super Mario Bros**, la cual podemos encontrar en el siguiente enlace:



<https://www.youtube.com/watch?v=K-NBcP0YUQI>

En esa entrevista se nos cuenta en esencia que un nivel de plataformas debe centrarse en **una o dos mecánicas básicas** planteadas de formas **simples** pero que a medida que va avanzando el nivel, estas se vayan **complicando o uniéndose**. La idea es conseguir que el jugador entienda rápidamente el funcionamiento básico del videojuego y que experimente con lo aprendido para sortear los obstáculos que se vienen, pero siempre aportando un **diseño acorde** a lo **enseñado**. Para **fomentar** la **experimentación** y la **exploración** Shigeru nos insta a utilizar recursos de **puntuación** que el jugador quiera recoger, ya sea por obtener un beneficio directo o por simplemente plantear el reto.





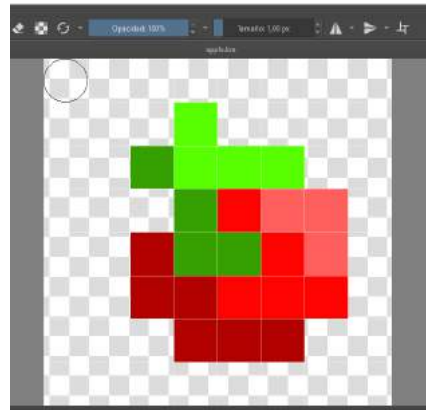
## 8.1.- Definiendo nuestros objetivos del nivel

Siempre que se plantea el diseño de niveles en un videojuego se deben fijar unos **objetivos a cumplir**, de modo que sepamos cuando el nivel está completo o no. En nuestro caso tenemos que asegurarnos de lo siguiente:

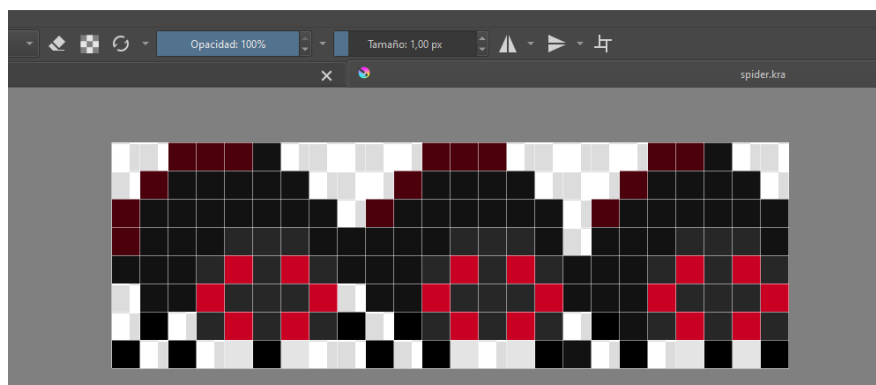
- **El nivel cuenta con un inicio y un final**, es decir una ubicación dentro de la escena donde aparece nuestro personaje principal y otra ubicación donde al momento de llegar el juego finalice y se nos informe de ello.
- **Los objetos de juego deben encontrarse definidos**. Estos objetos serán: El **personaje** principal, **objetos** de recolección, el **escenario** y la **cámara** principal
- **Debemos tener definidos los Assets a utilizar**. Hasta ahora hemos desarrollado assets de sonido y gráficos de nuestro personaje. Tendremos que desarrollar también un assets para el objeto de recolección, ya que para el escenario podemos aplicar **Assets** internos de **Unity**. También necesitamos un **Assets** que aplique como fondo del nivel.
- **Debemos definir el comportamiento de los objetos de juego y del sistema de puntuación**. A la hora de elaborar el nivel declararemos un personaje principal que sea capaz de desplazarse, saltar, morir y recolectar manzanas. Las **manzanas** se desperdigarán por el nivel marcando indicaciones visuales de cómo proceder por éste así como un incremento de puntuación por su recogida por valor de **una unidad**. así mismo por el nivel se van a desplegar **arañas enemigas** que serán eliminables al saltar sobre ellas y que ofrecerán un **bonus de 10 puntos por eliminación**. Debe existir una interfaz de usuario sencilla que indica la puntuación actual del personaje así como la finalización del nivel.

## 8.2.- Gestión de Assets y animaciones

Una vez que ya tenemos definidos nuestros objetivos del nivel lo primero que debemos hacer es comprobar que disponemos de los recursos necesarios para su creación. Hasta el momento hemos desarrollado el Sprite del personaje principal y lo hemos implementado en **Unity**. Debemos agregar también un objeto de juego **recolectable**, que en este caso serán **manzanas**. Para su creación haremos uso de nuevo de la herramienta **Krita**, como ya hicimos con el peor mensaje. Respetaremos la dimensión **8x8** para un bloque unitario, de modo que nos quede tal que así:

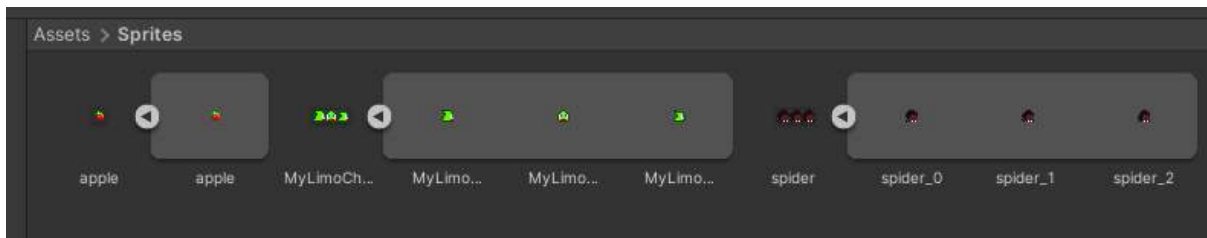


Para las arañas enemigas buscaremos algo un poco más complejo, ya que vamos a darles una animación concreta. En este caso buscamos que las arañas se desplacen de izquierda a derecha, por lo que necesitaremos una animación fluida de un mínimo de 3 pasos. Para aprovechar la explicación de como crear la animación del personaje principal vamos a crear un **Sprite Sheet** para nuestra araña de **24x8**:



De este modo ya tendríamos todos los **Sprites** visuales necesarios. A continuación los exportamos a **Unity** del mismo modo que ya hicimos con el personaje principal. Con respecto a las manzanas no requiere de ningún tipo de configuración extra, pero recordemos que la araña debe ser dividida en 3 imágenes estableciéndose como un **Sprite múltiple** y dividiéndolas con la opción **Slice** del editor de sprites, exactamente del mismo modo que ya hicimos con el personaje principal.

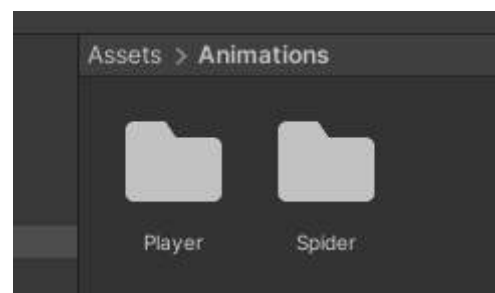
Para evitar que Unity nos implemente modificaciones en nuestros Assets gráficos a la hora de la implementación es importante que cuando los agreguemos especifiquemos que **no queremos aplicar ningún tipo de compresión**, que **cada pixel de la imagen sea un píxel de la escena** y que el **alpha** que se tengan en consideración sea preferentemente el **de la imagen importada** (Es preferible siempre exportarlas imágenes desde **Krita** en formato **PNG**). Si hemos seguido los pasos indicados deberíamos tener los siguientes recursos en nuestra carpeta de Sprites en Assets:



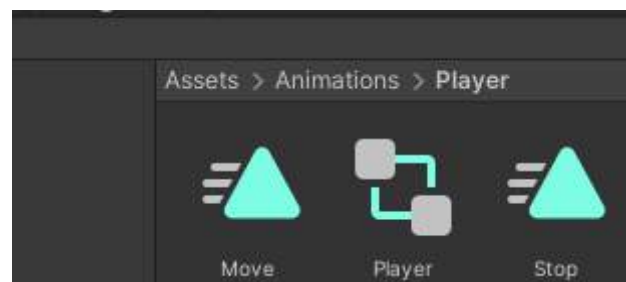
Asimismo cuando estuvimos creando nuestra melodía principal en **LMMS** explicamos cómo agregar assets de audio en **Unity**, y de hecho agregamos todos los necesarios, por lo que deberíamos ya tenerlos implementados en nuestro proyecto:



Para finalizar también deberíamos tener implementada la animación del personaje principal, la cual realizamos en el aparato de creación de animaciones en **Unity**. Siguiendo esos mismos pasos crearemos la animación de las arañas enemigas. Al ser una animación compartida bastará con que creemos una única animación, por lo que en la carpeta **Animations** deberíamos tener finalmente dos carpetas:

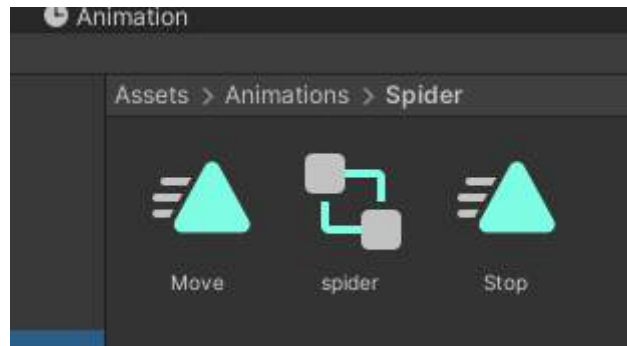


La carpeta **Player** debería contener las siguientes animaciones:

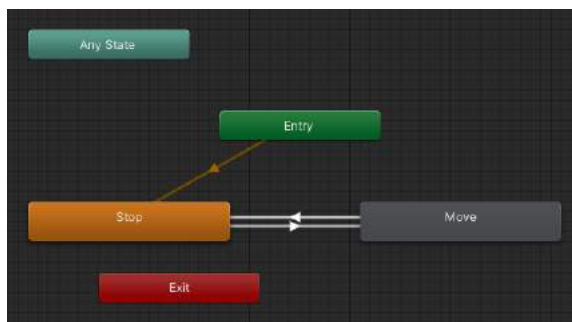




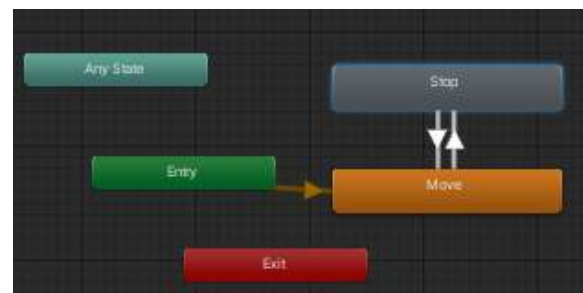
Y la carpeta **Spider**:



Como podemos ver la configuración que siguen ambas animaciones es la misma en ambos objetos de juego, cambiando únicamente el nombre. De todos modos a continuación se expone el enlazado de estados de ambos objetos para mayor comodidad y comprensión:



*Estados de Player*

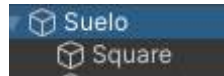


*Estados de Spider*

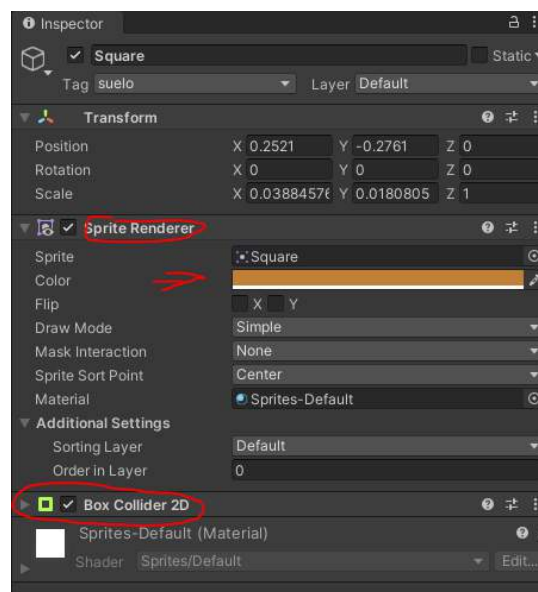
### 8.3.- Desarrollo del escenario

A la hora de desarrollar un nivel de plataformas es muy importante plantear unas **mecánicas jugables concretas** con las que el jugador pueda avanzar por éste de un modo satisfactorio y que vaya pidiendo más al jugador pero siempre de un modo que de pie a una lectura rápida. En nuestro caso queremos un nivel de plataformas que establezca al personaje principal a la izquierda del escenario y que por el propio diseño del escenario y sus objetos se deduzca como proceder por éste.

Para empezar debemos definir un objeto de juego dentro de la escena que contenga cada uno de los bloques del escenario. Como estos **bloques** van a ser **pisables** crearemos un objeto de juego padre el que denominaremos **Suelo** y que se va a componer de Sprites cuadrados de **Unity** (Para ello creamos el objeto en la escena, lo llamaremos **Suelo** y pulsando en dicho objeto haremos **click derecho -> Create 2D Object -> Sprite -> Square**):

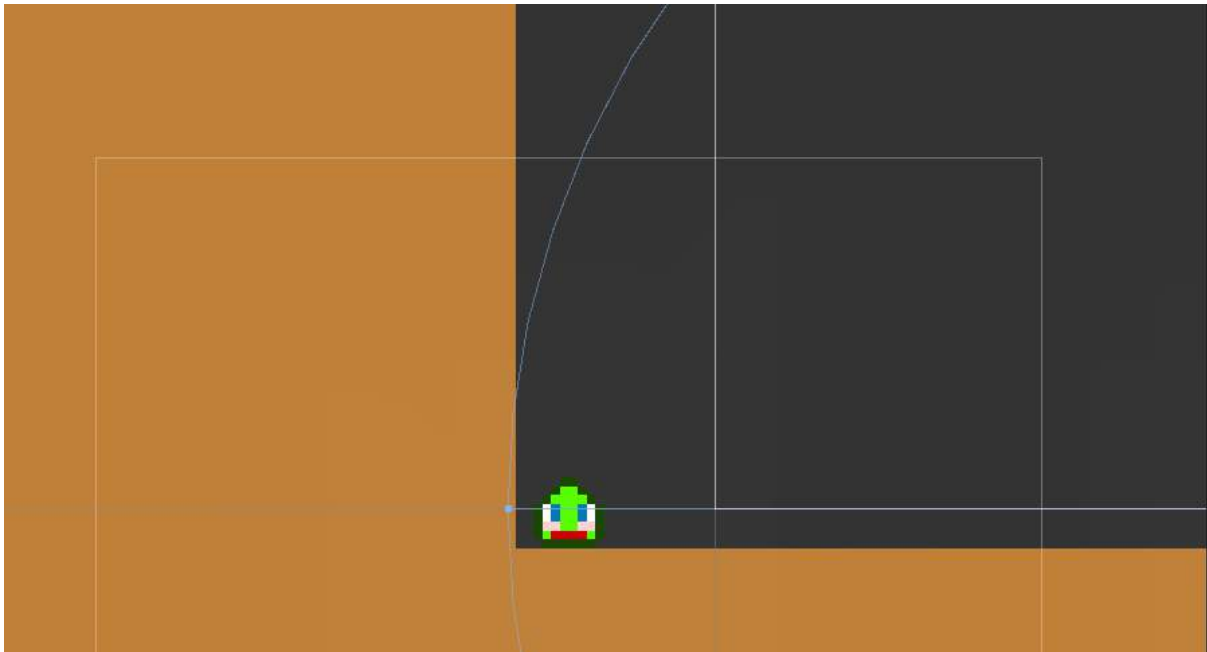


Si pulsamos sobre el objeto hijo **Square** veremos que se componen de un componente **Transform** que indica su posición en la escena y de un **Sprite Renderer** de tipo cuadrado de color blanco. Escogeremos un color **anaranjado** y le agregamos un componente de caja de colisiones, para que el jugador pueda interactuar con el suelo y pisarlo. No es necesario agregar un componente **RigidBody** porque el escenario no va a desplazarse sino que queremos que se mantenga fijo:



Asimismo vamos a agregarle una etiqueta a nuestro objeto **Square**. Para ello pulsamos sobre el selector **TAG** que se encuentra debajo del nombre en el inspector del elemento. Se nos pedirá que elijamos una **TAG** de entre las disponibles o que creamos una. En nuestro caso vamos a crear una nueva a la que llamaremos “**suelo**”, de modo que la configuración quede como en la imagen indicada arriba.

Ahora que tenemos listo nuestro primer bloque naranja lo situamos en la escena, de modo que quede a la izquierda del personaje. Una vez posicionado copiaremos el objeto **Square** y se nos creará debajo otro objeto **Square** con la misma configuración (lo cual nos va a ahorrar muchísimo el trabajo) y lo posicionamos debajo del personaje principal:



Con esto estaremos indicando al jugador que a la fuerza va a tener que desplazarse hacia la derecha para avanzar por el nivel, ya que al intentar la tarea opuesta se chocará con la pared.

Crear un escenario únicamente con bloques es un poco simple, por lo que agregaremos dos elementos extras que van a dar más jugo a los saltos y a la exploración: **Elevadores y Zonas secretas**.

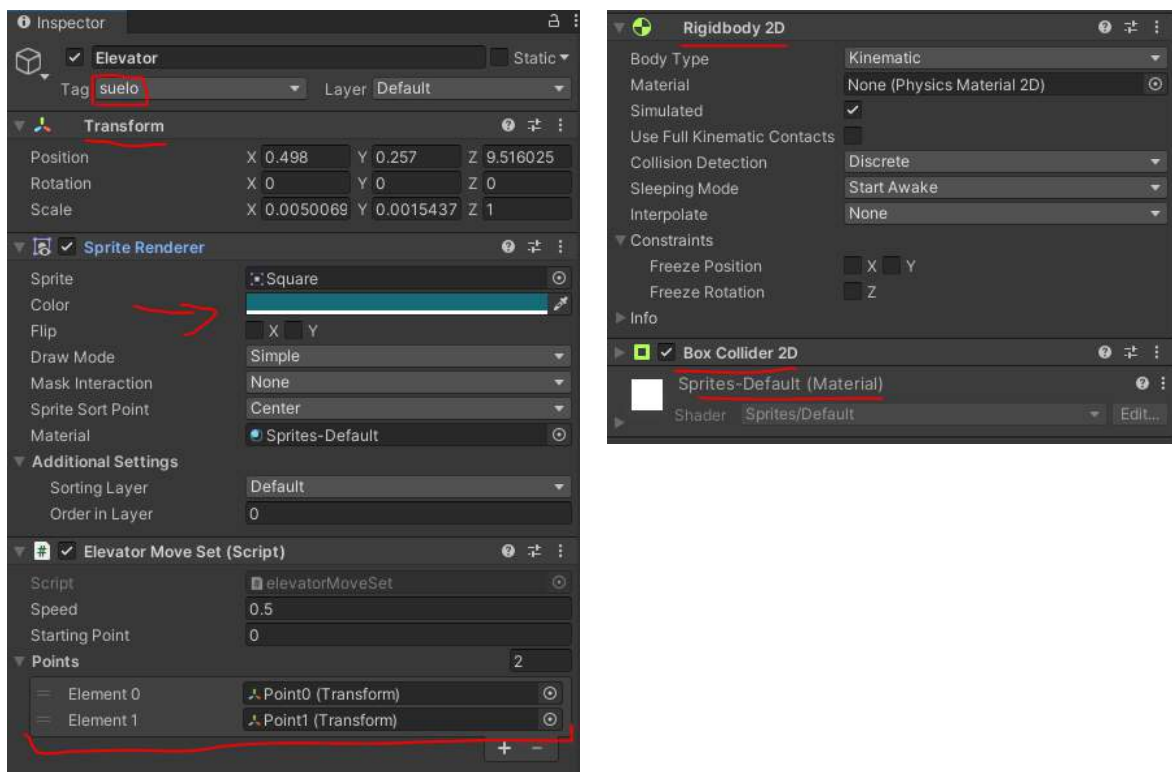
Cuando hablamos de un elevador dentro de un videojuego nos referimos a una **plataforma que se desplaza entre puntos** (normalmente de arriba a abajo y viceversa, de ahí el nombre. Pero no es necesario que siempre se desplaza en esas direcciones ni que solo se desplace entre 2 puntos únicamente). Para crear nuestro primer elevador crearemos un objeto vacío dentro de nuestro objeto padre **Suelo**. A este objeto vacío lo llamaremos **ElevadorA** y dentro de este crearemos 3 objetos vacíos: **Elevator**, **Point1** y **Point0**:



- **Elevator**: definirá físicamente al elevador, con una posición inicial concreta.
- **Point1** y **Point0**: Define los puntos por los que pasará el elevador. Básicamente son objetos de la escena que se establecen en posiciones concretas. La idea es que programemos un comportamiento sobre nuestro elevador mediante la cual se genere un **array de puntos** por lo que deberá desplazarse el elevador.



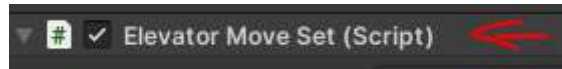
El objeto padre no requiere de ninguna configuración específica, por lo que puede dejarse con su componente **Transform** por defecto, indicando su posición inicial, pero realmente es poco relevante ya que las posiciones que verdaderamente importan son las de sus elementos hijos. El objeto hijo **Elevator** tendrá la siguiente configuración:



Como podemos ver debe tener marcado el **TAG suelo**, ya que a efectos prácticos es una plataforma pisable para el jugador. Debe tener un componente **Transform** que indique la posición dentro del escenario, un **Sprite Renderer** de cuadrado (igual que hicimos con los suelos), pero de un color que lo diferencia de los suelos comunes (En este caos como los suelos ordinarios son naranjas se ha optado por una **tonalidad azulada**). También debe tener un componente **Script** que defina su comportamiento y que veremos más adelante (En la imagen se puede apreciar como se define un **array** llamado **Points** y que se compone de 2 elementos **Points**, que son los que comentamos anteriormente. El Script lo veremos enseguida).

Finalmente el objeto **Elevator** debe tener un **RigidBody 2D** (Ya que va a desplazarse por el escenario) y una **caja de colisiones** que le permite interactuar tanto con los **Points** como con el propio **jugador**.

Nuestro siguiente paso es crear el **Script** de **movimientos** del **elevator** para agregarlo a su componente **Script**. Para ello iremos a nuestra carpeta de **Script** y creamos un nuevo **Asset** de tipo **Script** al que denominaremos **elevatorMoveSet**. Cuando se haya creado lo arrastramos hasta el componente Script de **Elevator**:



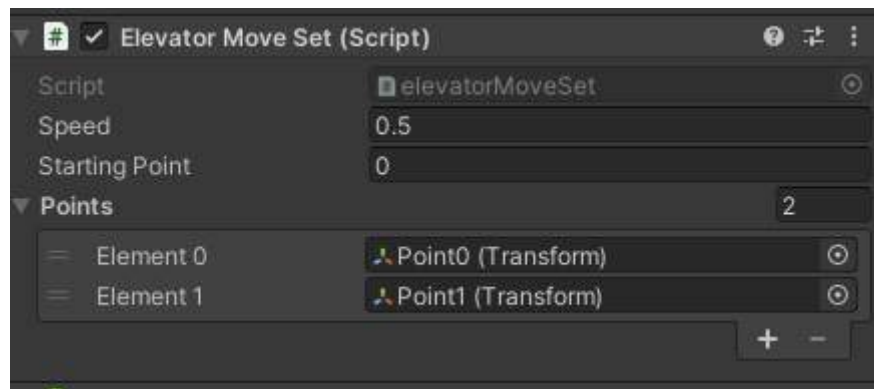
Si hacemos doble click sobre el **Script** se nos abrirá **Visual Studio Code**, para que podamos empezar a agregar el código de comportamiento. Agregaremos lo siguiente:

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

public class elevatorMoveSet : MonoBehaviour{
    public float speed;
    public int startingPoint;
    public Transform[] points;
    private int i;

    private void Start(){
        transform.position = points[startingPoint].position;
    }
    void Update(){
        if(Vector2.Distance(transform.position,points[i].position) < 0.02f)
        {
            i++;
            if (i == points.Length)
            {
                i = 0;
            }
        }
        transform.position = Vector2.MoveTowards(transform.position,
points[i].position, speed * Time.deltaTime);
    }
    private void OnCollisionEnter2D(Collision2D collision)
    {
        collision.transform.SetParent(transform);
    }
    private void OnCollisionExit2D(Collision2D collision)
    {
        collision.transform.SetParent(null);
    }
}
```

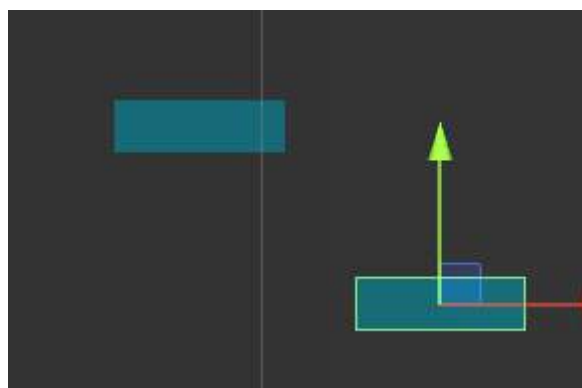
Es **importante** que tanto la **velocidad**, el **Point** inicial y el **array** de **Points** sean **públicos**, ya que es lo que nos va a permitir acceder a ellos desde el inspector de objetos:



Si nos fijamos el código simplemente define un array de **Points**, el cual es rellenado desde el inspector de **Elevator**. Para rellenarlo se nos despliega un selector y en él escogemos los ojos **Point1** y **Point0** que creamos anteriormente. Al iniciarse la escena se ejecutará el método **Start**, el cual hará que **Elevador** se dirija en la dirección del primer Point indicado. Posteriormente se irá llamando en cada frame a la función **Update**, la cual irá comprobando el siguiente punto mediante el **bucle for** hasta que este sea alcanzado y por tanto se cambie de punto. En el caso de alcanzarse el último punto o posición del array, se vuelve a la posición 0, por lo que el elevador realizará su función indefinidamente.

Finalmente si nos fijamos en la parte última del código estamos indicando que cuando un elemento colisione con nuestro elevador, dicho elemento pase a ser hijo del elevador. Esto lo hemos hecho así porque **en Unity los objetos de juego Padre someten a los Hijos**, por lo que si un objeto padre se desplaza, el hijo lo hará en consecuencia. Si no hiciéramos esto, el elevador se desplazaría entre los puntos pero el personaje no, por lo que con esa simple implementación logramos con éxito el objetivo. asimismo cuando el jugador salga del elevador dejará de ser hijo del elevador.

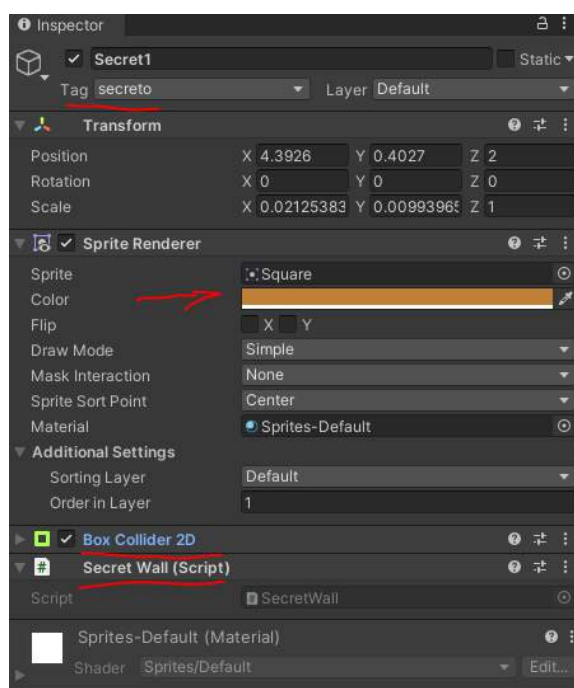
Con esto podremos crear tantas elevadores como queramos, siempre que le creemos sus correspondientes puntos por los que moverse:





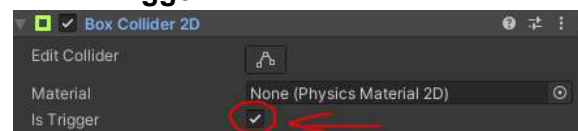
Finalmente quedaría haber de como crear una **zona secreta**. Una zona secreta en un videojuego es una zona que aparentemente es conocida pero que a la hora de interactuar con ella de alguna forma, muestra un diseño oculto. En nuestro caso vamos a crear una zona que visualmente sea igual que un bloque de suelo ordinario, pero que cuando colisione con el jugador se muestre que realmente era una zona hueca llena de **manzanas**.

Lo primero que haremos será crear dentro de nuestro objeto padre **Suelo** un nuevo objeto de juego vacío llamado **Secret1**. Este deberá tener la siguiente configuración:



Es decir:

- Debemos crearle un **TAG** propio al que llamaremos **"secreto"**.
- Le aplicaremos un **Sprite Renderer** del mismo color y tipo que un objeto **Square** de suelo ordinario.
- Le aplicaremos un componente **Script**, para programar su funcionalidad.
- Le aplicaremos un componente de caja de colisión **2D**, la cual debe ser un activador. Para esto convertiremos su caja en un **Trigger**:



Crearemos un nuevo Script al que llamaremos **SecretWall** y lo aplicaremos dentro del componente Script de **Secret1**. Abrimos el **Script** con **Visual Studio Code** y agregaremos el siguiente código:

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

public class SecretWall : MonoBehaviour
{
    Color orangeFull = new Color32(255, 253, 97, 255);
    Color orangeWithAlpha = new Color32(192, 128, 56, 150);
}
```



```
private void OnTriggerEnter2D(Collider2D collision)
{
    if (collision.transform.tag == "Player")
    {
        gameObject.GetComponent<Renderer>().material.color =
orangeWithAlpha;
    }
}
private void OnTriggerExit2D(Collider2D collision)
{
    gameObject.GetComponent<Renderer>().material.color = orangeFull;
}
}
```

Inicialmente se definen 2 objetos **Color32**, los cuales hacen referencia al color inicial del bloque (**naranja**) y al color que tendrá el bloque cuando el jugador active el secreto. Para activar el secreto se utilizan las funciones **OnTriggerEnter** y **OnTriggerExit**. La primera se ejecutará cuando un objeto colisione con el secreto y la segunda cuando un objeto que se encuentra activando el **secreto** deje de activarlo, saliendo de su **caja de colisión**. Cuando salte el activador del secreto se comprueba si el objeto de juego colisionador es el jugador. En caso afirmativo cambiará la tonalidad del bloque, revelando que no es un bloque de suelo, sino una **zona secreta llena de manzanas** (las cuales agregaremos en su sección correspondiente, pero de momento podemos agregar simplemente **Sprites no funcionales** de manzanas para que se vea el futuro resultado final):



*Secreto desactivado*



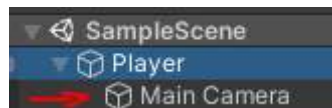
*Secreto activado*

Y con esto ya tendríamos todos los elementos necesarios para crear el escenario. A continuación veremos cómo implementar el resto de **Assets** y sus respectivas funcionalidades.



## 8.4.- Implementando nuestro Asset Player

Lo primero que deberemos hacer es crear un objeto de juego en la escena al que llamaremos **Player**. A continuación seleccionaremos el objeto de juego **Main Camera** que se genera con cada escena de Unity y lo arrastraremos hacia nuestro objeto Player, con lo que conseguiremos que el objeto **Player** pase a ser un **objeto de juego padre** y la cámara pase a ser un objeto de juego hijo.



Tenemos que pensar que Unity es un entorno donde desplegamos objetos de juego en posiciones concretas. Cada uno de estos objetos de juego en principio tiene autonomía propia, por lo que salvo que creamos eventos concretos, si desplazamos un objeto de juego no debería afectar al resto. En nuestro caso buscamos crear un objeto manejado directamente por el jugador, por lo que el usuario debe ser capaz de poder verlo en todo momento. Para conseguir esto tenemos que hacer que siempre el objeto Player se desplace, se desplace también la cámara siguiendo sus movimientos. Esto es precisamente lo que conseguimos al crear esta **herencia** de objetos de juego, ya que **los cambios del componente transform de un objeto padre se heredan a sus hijos**.

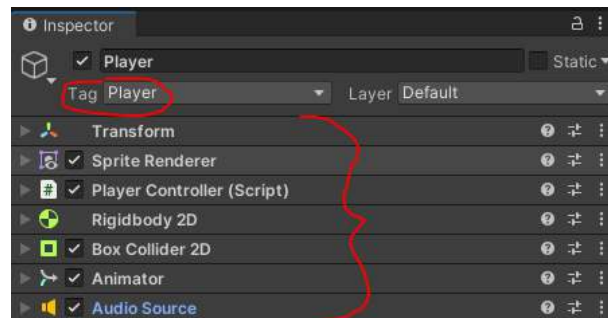
Ahora tenemos que plantearnos los componentes que va a necesitar nuestro objeto Player para cumplir con su cometido. Como queremos que muestre un Sprite o imagen identificativo deberemos crear un componente **Sprite Renderer**, el cual va a contener el Sprite que ya diseñamos en su momento en Krita. También necesitaremos aplicarle un **BoxCollider2D** (para que pueda interactuar con el resto de elementos del escenario), un componente **RigidBody2D** (ya que para las mecánicas jugables vamos a hacer uso directo del motor de físicas de Unity y, por tanto, necesitamos adjuntarle este componente para obligar al objeto a someterse al motor) y un componente Script que contenga el código de manipulación del objeto.

Como hemos creado una animación para el jugador deberemos añadirle también un componente **Animator**, así como un componente **Audio Source** para agregarle un sonido concreto (En nuestro caso ese recurso de sonido será *"playerJump"*, el cual ya vimos anteriormente cuando implementamos los recursos en Unity).

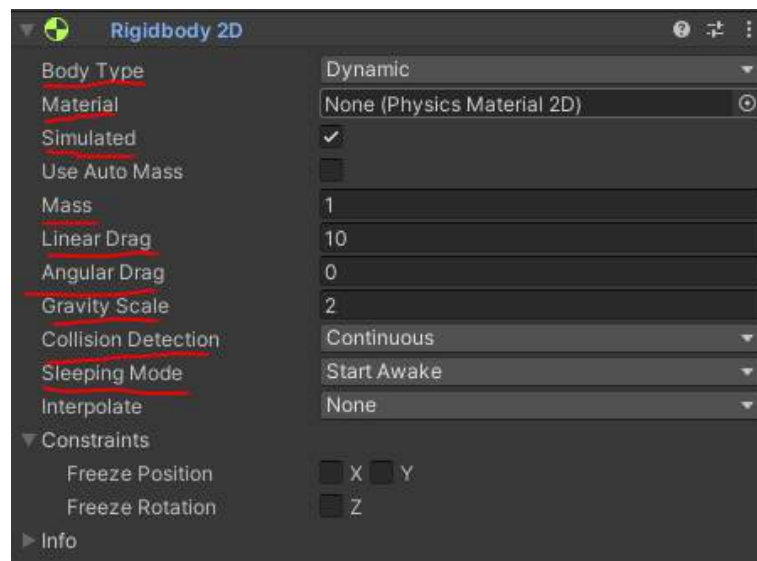
Finalmente, como estamos creando un objeto concreto y que es conveniente tener referenciado en todo momento vamos a añadirle una etiqueta o Tag. En este caso le pondremos la etiqueta **Player**, la cual viene creada por defecto en Unity.

Deberíamos tener, por tanto, un objeto de juego con la siguiente configuración:





A continuación explicaremos un poco el funcionamiento del componente **Rigidbody**, ya que es necesaria su comprensión para su correcta implementación en cada caso. Como hemos comentado anteriormente Unity cuenta con un motor de físicas incorporado. Este motor se encarga de simular unas “**leyes físicas**” configurables y que son aplicadas a todo objeto de juego que cuente con un componente **Rigidbody**. Cuando un objeto se le aplica este componente se le otorgan atributos concretos, como un Tipo de comportamiento (**Body Type**), Material, Masa (**Mass**), Rozamiento angular (**Angular Drag**) y lineal (**Linear Drag**), multiplicador del efecto gravedad (**Gravity Scale**), método de detección de colisiones, y otros atributos menos importantes. Observemos la configuración que le vamos a dar a nuestro personaje:



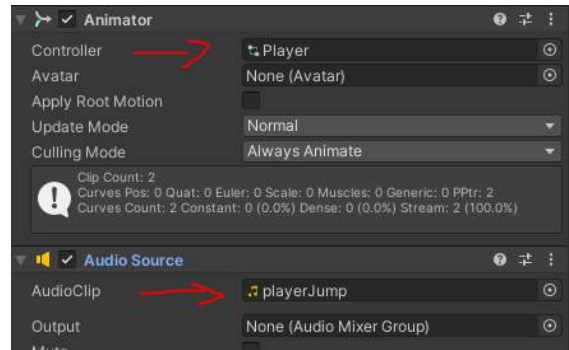
1. **Body Type:** Estableceremos un comportamiento dinámico para los cuerpos. Esto se utiliza para que la fuerza no se aplique del mismo modo siempre que se aplique al objeto. Por ejemplo, si un objeto se encuentra dentro del agua podemos utilizar este atributo para manipular en tiempo real su escala de gravedad, de modo que visualmente parezca que “bucea” cuando realmente está flotando.
2. **Material:** Lo dejaremos por defecto, ya que estamos trabajando en un escenario en 2D. Básicamente sirve para alternar entre el material aplicable al objeto de juego. Por ejemplo, no es igual un material físico que uno líquido o un efecto de vapor. En



caso de usar la opción por defecto estaremos diciendo que el comportamiento del objeto no se define por el material, sino por el código o script a aplicar sobre el objeto (Es precisamente por eso que lo hemos puesto con un comportamiento dinámico previamente).

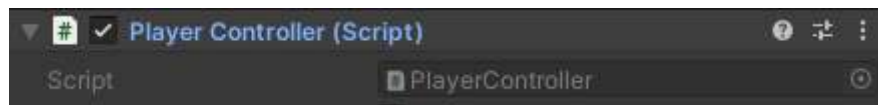
3. **Simulated:** Este es un parámetro opcional y que sirve para que podamos hacer uso del motor de físicas sin tener que hacer una build ejecutable del juego, sino aplicando directamente sobre el ejecutor de escenarios de Unity. Lo dejaremos marcado para que no tengamos que perder tiempo construyendo proyectos para cada prueba.
4. **Mass:** Define la masa del objeto de juego. En nuestro caso le hemos dado el valor de 1 unidad, lo que quiere decir que es un objeto de masa ordinaria. A más masa, más peso y por tanto más fuerza debemos aplicar para desplazar.
5. **Linear Drag y Angular Drag:** En ambos casos estamos hablando de la resistencia que va a tener tanto a la hora de desplazarse linealmente como a la hora de rotar. Como queremos que el personaje no rote estableceremos su Angular Drag en 0, y como queremos que al aplicarle fuerza no se deslice como si caminara sobre hielo lo pondremos un valor de 10 de linear Drag. Estos valores variarán en base a la masa del objeto.
6. **Gravity Scale:** Define el multiplicador de gravedad que se va a aplicar sobre el objeto. Como sabemos, por defecto la gravedad aplica una fuerza sobre todo objeto. Si queremos que esa fuerza sea igual que la aplicada según nuestras leyes físicas estableceremos como valor 1 unidad. En nuestro caso estamos hablando de un objeto de juego que, aunque tiene una masa ordinaria, es pequeño y tiene un alto rodamiento lineal, por lo que nos aseguraremos de ponerle un poco más de gravedad de lo normal. Con un valor de 2 unidades será suficiente.
7. **Collision Detection:** En Unity todos los objetos que tengan un componente BoxCollider son por defecto activadores. En Unity la comprobación de colisiones se hace por defecto de frame en frame de forma continuada, lo cual puede llegar a consumir bastante memoria si se aplica esa comprobación sobre demasiados objetos a la vez. Por ello se nos permite la posibilidad de alternar entre estos modos de detección de colisiones. Nosotros deberemos dejarlo como comprobación continua, ya que la comprobación de este tipo de activaciones con respecto al personaje principal es crucial para nuestra tarea.
8. **Sleeping Mode:** En Unity los objetos tienen estados asignados en base a su ciclo de vida (Estos estados ya los vimos cuando hicimos las animaciones de enemigos y del personaje anteriormente). En nuestro caso queremos que desde que se inicie la aplicación se aplique en la escena nuestro personaje, por lo que iniciaremos su comportamiento desde el mismo inicio con la opción Start Awake.

Con respecto a los componentes de **Animator** y **Audio Source** le aplicaremos esta configuración:



Como podemos ver al componente **Animator** le anexamos la animación *Player*, que ya creamos anteriormente. Al componente **Audio Source** le aplicaremos el clip de audio de salto del personaje, que como dijimos se llama *playerJump*.

Finalmente crearemos un Script al que llamaremos *PlayerController* y lo arrastraremos hasta el componente Script de nuestro objeto *Player*, de modo que queden vinculados:



Una vez haya quedado vinculado haremos doble click sobre el Script para que se abra en **Visual Studio Code**.

En el documento añadiremos el siguiente código **C#**:

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

public class PlayerController : MonoBehaviour
{
    public bool canJump = true;
    public AudioSource clipJump;
    void Update(){
        if (Input.GetKey("left"))
        {
            gameObject.GetComponent<Rigidbody2D>().freezeRotation = 
true;
            gameObject.GetComponent<Rigidbody2D>().AddForce(new
```



```

Vector2(-700f * Time.deltaTime, 0));
        gameObject.GetComponent<Animator>().SetBool("moviendose",
true);
        gameObject.GetComponent<SpriteRenderer>().flipX = true;
    }
    if (Input.GetKey("right"))
    {
        gameObject.GetComponent<Rigidbody2D>().freezeRotation =
true;
        gameObject.GetComponent<Rigidbody2D>().AddForce(new
Vector2(700f * Time.deltaTime, 0));
        gameObject.GetComponent<Animator>().SetBool("moviendose",
true);
        gameObject.GetComponent<SpriteRenderer>().flipX = false;
    }
    if(!Input.GetKey("left") && !Input.GetKey("right"))
    {
        gameObject.GetComponent<Animator>().SetBool("moviendose",
false);
    }
    if (Input.GetKey("up") && canJump)
    {
        clipJump.Play();
        gameObject.GetComponent<Rigidbody2D>().freezeRotation =
true;
        canJump = false;
        gameObject.GetComponent<Rigidbody2D>().AddForce(new
Vector2(0,200f));
    }
}
private void OnCollisionEnter2D(Collision2D collision)
{
    if (collision.transform.tag == "suelo")
    {
        gameObject.GetComponent<Rigidbody2D>().freezeRotation =
true;
        canJump = true;
    }
}
}

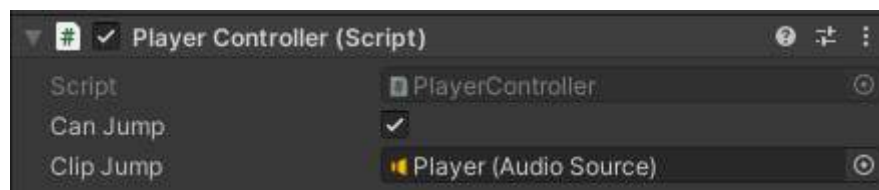
```



A continuación vamos a explicar cada parte de este. Lo primero que tenemos que ver es que se han declarado **dos variables públicas**. La primera es un **booleano** que define si el **personaje** va a **poder saltar o no**. Como queremos que el personaje pueda saltar siempre que haya tocado previamente el suelo y, por defecto, ya se encuentra en el suelo, le pondremos **True** como valor por defecto. La segunda es objeto de tipo **AudioSource** la que llamaremos **clipJump**. Recordemos que es muy importante que estas variables se declaren como públicas, ya que buscamos que sean accesibles desde la propia ventana de componentes de **Unity**:

```
public bool canJump = true;  
public AudioSource clipJump;
```

De este modo, si volvemos a **Unity**, veremos cómo estas variables son modificables. Ahora arrastraremos nuestro componente **Audio Source** a esta variable, para que quede vinculada la variable con ese **Audio Source** concreto de salto:



Lo siguiente que encontramos es el método **Update**, que recordemos que se ejecuta cada frame. Como estamos codificando el comportamiento de nuestro personaje, lo primero que definiremos serán sus movimientos. Queremos que se pueda **desplazar por el escenario de izquierda a derecha** dependiendo de la **flecha** del teclado que se oprima. Para ello capturaremos la tecla con el método **GetKey** de la clase **Input**.

En el caso de que se oprima una de esas 2 teclas, se tomará el componente **RigidBody2D** del objeto actual (**Player**) y se le aplicará una fuerza con el método **AddForce**. Este método recibe como parámetro un **Vector3** o un **Vector2**, dependiendo del tipo de dimensionalidad que estemos manejando. Como estamos diseñando para un escenario en dos dimensiones usaremos un **Vector2**, el cual recibe 2 parámetros:

- **Fuerza a aplicar en dirección horizontal (EJE X)**, de tipo **float**. Si usamos un valor en positivo la fuerza se aplicará hacia la derecha y en negativo hacia la izquierda de la escena. Si nos fijamos estamos multiplicando este valor por **Time.deltaTime**, ya que recordemos que aunque el método **Update** se ejecuta cada frame, nosotros queremos que la fuerza se someta al **tiempo** para no depender de la tasa de refresco de las pantallas.
- **Fuerza a aplicar en dirección vertical (EJE Y)**, de tipo **float**. Tanto para el desplazamiento a izquierda y derecha lo dejaremos en 0, ya que no debe verse afectado.



Asimismo si se pulsa alguna de las tablas tomaremos el componente **Animator** para establecer el valor de su booleano "**moviéndose**" (que ya vimos cuando creamos las animaciones) en **True**. Para ahorrar memoria y no repetir imágenes en una dirección y en otra utilizaremos un truco de optimización de **Sprites** que consiste en rotar los **Sprites** sobre el eje X, de modo que cambien su orientación. Con todo el código quedaría así:

```
if (Input.GetKey("left"))
{
    gameObject.GetComponent<Rigidbody2D>().freezeRotation =
true;
    gameObject.GetComponent<Rigidbody2D>().AddForce(new
Vector2(-700f * Time.deltaTime, 0));
    gameObject.GetComponent<Animator>().SetBool("moviendose",
true);
    gameObject.GetComponent<SpriteRenderer>().flipX = true;
}
if (Input.GetKey("right"))
{
    gameObject.GetComponent<Rigidbody2D>().freezeRotation =
true;
    gameObject.GetComponent<Rigidbody2D>().AddForce(new
Vector2(700f * Time.deltaTime, 0));
    gameObject.GetComponent<Animator>().SetBool("moviendose",
true);
    gameObject.GetComponent<SpriteRenderer>().flipX = false;
}
```

En el caso de que no se pulse ninguna de las 2 teclas indicadas el Script debe modificar el valor del booleano de la animación, para parar su funcionamiento. Para eso aplicaremos esta línea de código:

```
if(!Input.GetKey("left") && !Input.GetKey("right"))
{
    gameObject.GetComponent<Animator>().SetBool("moviendose",
false);
}
```

Por supuesto el personaje no va solo a desplazarse, sino que debe también ser capaz de saltar. Para ello capturaremos la tecla de flecha hacia arriba y comprobaremos el valor del booleano **canJump**. En caso de que se pulse la tecla y la variable se encuentre con valor True iniciaremos el **clip de sonido** del objeto, con el método **Play** de **Audio Source**. Para hacer que el personaje salte le aplicaremos una fuerza sobre el **eje Y** y estableceremos la variable **canJump** en **False**, para que no pueda saltar indefinidamente.





```

if (Input.GetKey("up") && canJump)
{
    clipJump.Play();
    gameObject.GetComponent<Rigidbody2D>().freezeRotation =
true;
    canJump = false;
    gameObject.GetComponent<Rigidbody2D>().AddForce(new
Vector2(0,200f));
}

```

Destacar que en cualquier pulsación de tecla de las comentadas se está aplicando también una restricción al **RigidBody** para que este no pueda rotar. Para eso se hace uso del método **freezeRotation**.

Finalmente queda indicarse al objeto que cuando colisione con otro objeto que tenga la etiqueta **suelo**, la variable *canJump* vuelva a ser **True**. Para conseguir esto utilizamos un método denominado **OnCollisionEnter2D**, que recibe un objeto de tipo **Collision2D**:

```

private void OnCollisionEnter2D(Collision2D collision)
{
    if (collision.transform.tag == "suelo")
    {
        gameObject.GetComponent<Rigidbody2D>().freezeRotation =
true;
        canJump = true;
    }
}

```

Con esto ya tendríamos definido el comportamiento y configuración completa de nuestro **Asset Player**. A continuación veremos cómo implementar assets de recolección y cómo hacer uso de un nuevo tipo de objetos, los **Prefabs**.

## 8.5.- Implementando Assets de recolección

Un **Asset** de recolección es un activo del videojuego que se dispone a lo largo del escenario y que el jugador puede recoger para ganar puntos extra o desbloquear contenido. Son algo inherente de los videojuegos de tipo arcade y de plataformas, desde el mítico **Pac-Man** que recogía puntos por su laberinto hasta videojuegos más actuales como **Hollow Knight**, donde se recogen una moneda propia del juego denominada "**Geo**".

Algo que tienen en común todos los **assets de recolección** es que sirven para un fin **concreto**, muy **básico**, requieren **pocos recursos** de proceso a nivel individual y se someten únicamente al **evento de ser recogidos**. Cuando tenemos este conjunto de



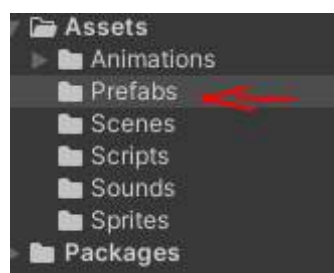
atributos lo más normal es **abstraer** el asset a una posición más abstracta denominada **Prefab**. Un **Prefab** no es más que una “*Plantilla*” de creación y actuación común que mantiene unido y sincronizado el comportamiento del conjunto de elementos que genera.

Supongamos que tenemos en nuestro escenario **50 monedas** a recolectar, cada una con su **Sprite**, **tamaño** y **Script** de recolección. A poco que empecemos a crear niveles con este patrón, los recursos se consumen más y más rápido, ya que **cada moneda se interpretará como un Asset o Activo** dentro del videojuego, haciendo que el gasto de recursos sea excesivo.

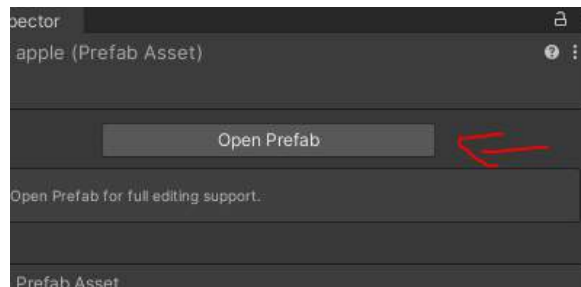
Si por el contrario creamos un **Prefab**, podremos adjudicar un único **Script**, un **Sprite** concreto y los componentes que estimemos, así como invocar tantas instancias del prefab como queramos, consumiendo muchos menos recursos en el proceso, ya que siempre que se active una moneda **descendiente del Prefab** lo que realmente hará su colisión en llamar al **Script** del **Prefab** del que deriva. Lo mismo ocurrirá con sus recursos de audio.

Por supuesto **no todos son ventajas**. Algo muy negativo del **Prefab** es que define un único tipo de comportamiento para todos sus descendientes. Esto quiere decir que si queremos que cada objeto de recolección tenga animaciones, colores o un aspecto diferente, no podremos hacerlo mediante Prefabs.

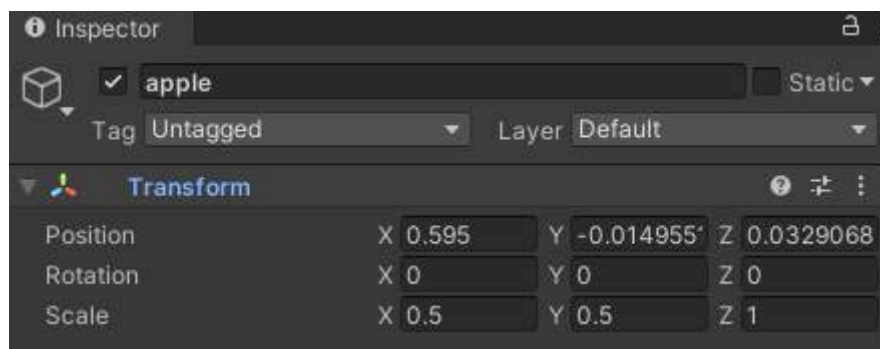
Una vez comentado lo que es un **Prefab** vamos a ver como crear el nuestro. Lo primero es crear una nueva carpeta dentro de la carpeta de **Assets** a la que llamaremos **Prefabs**:



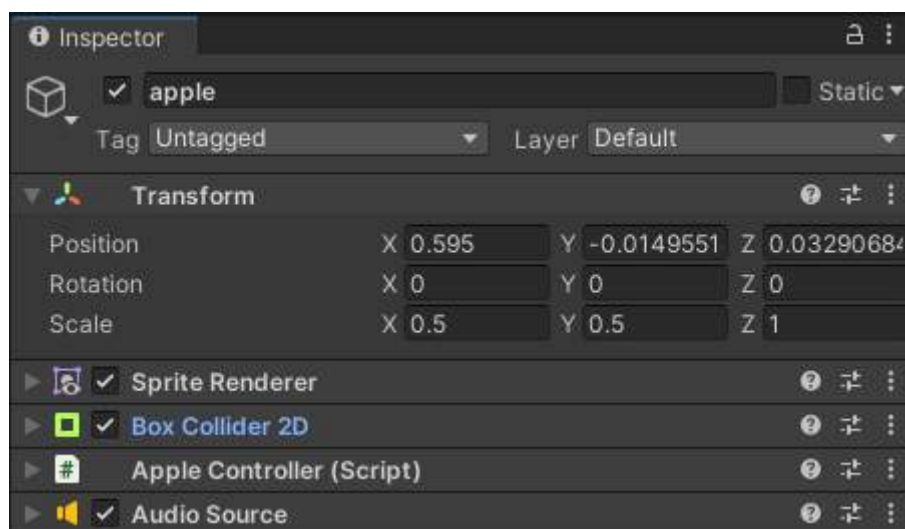
Dentro de esa carpeta haremos **clic derecho -> create -> Prefab**. Nos pedirá que le pongamos un nombre, que en nuestro caso será “**apple**”, ya que nuestro objeto de recolección serán las manzanas que ya dibujamos con **Krita** mediante pixel art previamente. Lo siguiente que haremos será pulsar en el botón que nos aparecerá en la parte superior, para abrir el **Prefab**:



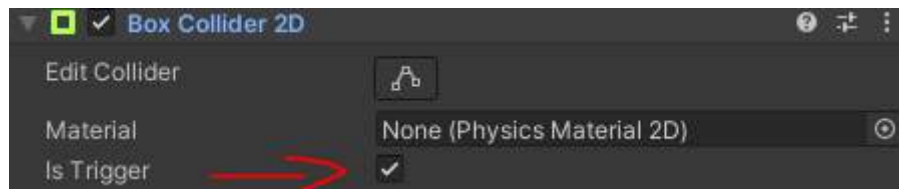
Veremos un objeto de juego normal y corriente, como los que ya hemos tratado anteriormente. Como sabemos estos cuentan por defecto con el componente **Transform**, que define su posición en la escena:



A continuación agregaremos los componentes a nuestro objeto de juego. Primero agregaremos un componente **Sprite Renderer**, a cual aplicaremos el **Sprite** de **apple** desarrollado en **Krita**. Posteriormente un componente **Box Collider 2D**, para que sea interactuable con el jugador. Finalmente agregaremos un componente **Script** para su código de funcionamiento y un componente **Audio Source**, para establecer el sonido de recogida distinguible. Es decir, deberíamos tener un objeto de juego tal que así:

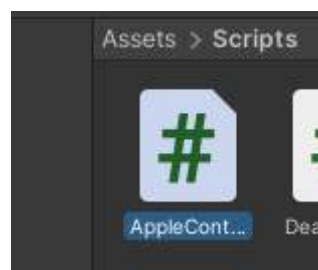


Una vez definido vamos a abrir el componente de **Box Collider** o **Caja de Colisión** y vamos a activar **“is Trigger”**:



Esto hará que el objeto de juego que nazca de nuestro prefab se considere un **activador**, de modo que cada vez que salte dicho activador podamos ejecutar métodos del videojuego, tanto referentes al objeto de recogida como al propio videojuego y su interfaz de usuario (**UI**). Por ejemplo esto nos servirá para que la UI pueda incrementar los puntos por cada manzana recogida.

Recuerda además agregar el **clip de audio** al componente **Audio Source**, como ya hicimos con el **Asset Player**. A continuación vamos a abrir nuestra carpeta de **Scripts** y vamos a crear un **Script** para nuestro **Prefab**. En este caso lo llamaremos por ejemplo **AppleController**:



Lo arrastraremos al componente **Script** del **Prefab**, para vincularlo con su componente y posteriormente lo abriremos con **Visual Studio Code**.

El código a implementar sería el siguiente:

```
public class AppleController : MonoBehaviour
{
    public GameObject applePrefab;
    public AudioSource clip;
    private void OnTriggerEnter2D(Collider2D collision)
    {
        if (collision.transform.tag == "Player")
        {
            clip.Play();
            GameObject scripter = GameObject.Find("Scripter");
            scripter.GetComponent<ScoreManagerScript>().increment(1);
        }
    }
}
```



```

        Invoke("deleteObject", 0.1f);
    }
}
private void deleteObject()
{
    Destroy(gameObject);
}
}

```

Como podemos ver lo primero que haremos será declarar un par de variables. La primera sería de tipo **GameObject** y es la que va a vincular el **Prefab** con nuestro objeto de juego. La segunda sería una de tipo **Audio Source**, la cual como vimos cuando desarrollamos el **Asset Player**, sirve para vincularle un componente **Audio Source** desde Unity y, con ello un clip de sonido concreto.

Posteriormente comentamos con el primer método, el cual se denomina **OnTriggerEnter2D** y recibe un objeto de tipo **Collider2D**. Este método se ejecutará siempre que cualquier objeto del juego nacido del **Prefab** haga saltar su activador por parte de un collider externo, pero siempre que ese collider lleve vinculado el tag "**player**". de este modo nos aseguraremos de que solo el jugador pueda recolectar las manzanas y no los enemigos. En caso de que el jugador interactúe con la manzana, se ejecutará el clip de audio **clip** mediante el método **Play** de la clase **AudioSource** y se ejecutará el método **increment** del objeto de juego **Scripter**, que es un objeto del juego que crearemos más adelante y que básicamente va a manejar funcionalidades que unen procesos de juego con la interfaz de usuario. Para finalizar, podremos utilizar el método **Invoke** de la clase **MonoBehaviour** para invocar el método de destrucción del objeto manzana en un plazo determinado de tiempo (En este caso 0.1 segundos).

El método **deleteObject** no tiene mucho que explicar. Simplemente utiliza el método **Destroy** de la clase **Monobehaviour** para destruir el propio objeto de juego que tiene vinculado el **Script** actual. De este modo se estaría eliminando el objeto de juego nacido del prefab, pero no el prefab en sí.

A continuación vamos a crear el objeto de juego **Scripter** que hemos comentado previamente. Usualmente se conoce como **Scripter** un objeto de juego invisible para el jugador pero que va a contener (Salvo el componente **Transform** que todo objeto tiene) únicamente componentes **Scripts** relativos a la **UI** del videojuego (Como por ejemplo el manejo de puntos) y los relativos a objetos de juego nacidos de un **Prefab**, por su especial condición. Para crear un objeto **Script** simplemente nos iremos a nuestra escena y haciendo **click derecho** -> **Create new GameObject** le pondremos por nombre **Scripter**.

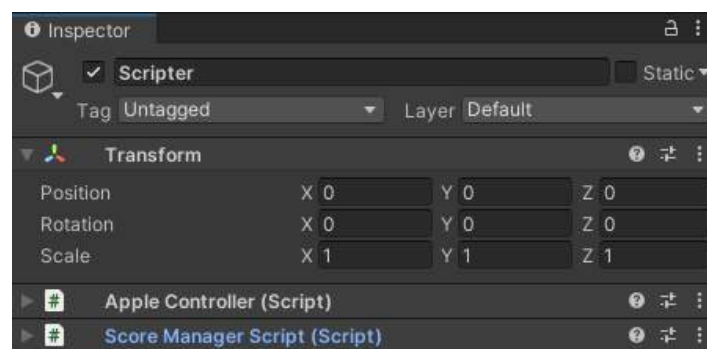




Le agregaremos **dos componentes de tipo Script**. En uno anexamos el **Script** de las manzanas, que recordemos que se llama **AppleController**. En el otro anexamos uno nuevo que vamos a crear para la **UI** del videojuego y que va a llamarse **ScoreManager**, ya que va a encargarse de manejar y pintar la puntuación, así como el mensaje de finalización del nivel. Abrimos el **Script** en **Visual Studio Code** y añadiremos el siguiente código:

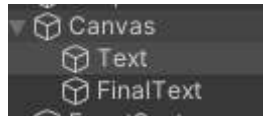
```
using UnityEngine.UI;
public class ScoreManagerScript : MonoBehaviour
{
    int score = 0;
    public Text gameScore;
    public Text finalText;
    public void increment(int points)
    {
        score += points;
        gameScore.text = score.ToString();
    }
}
```

Como podemos ver es bastante simple. Definimos **3 variables** y un método llamado **increment** que recibe un **entero** (Recordemos que este método es al que estamos invocando desde **AppleController**). Siempre que se llame al método se incrementará la variable **Score** y se almacenará en formato de tipo **Text** (Objeto de la clase UI y que define textos a mostrar en la interfaz de usuario) en la variable **gameScore**, la cual se va a visualizar en pantalla.



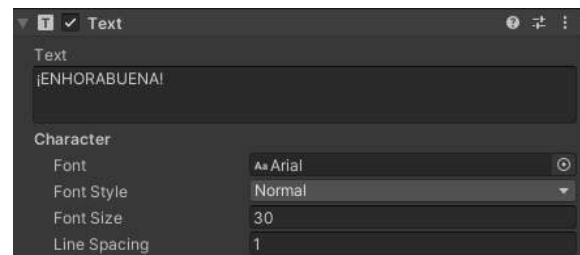
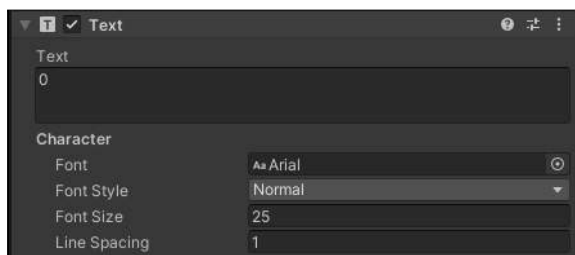
Pero con esto no hemos terminado, ya que como hemos dicho, un objeto **Scripter** no muestra nada por pantalla. Por lo que si queremos que se vean elementos **UI** definidos en un **Scriptes** necesitaremos un **lienzo** donde pintarlos. Para ello crearemos un objeto de juego normal y corriente en escena y lo denominaremos "**Canvas**". Dentro meteremos dos nuevos objetos de juego, uno llamado **Text** y otro llamado **Final Text**:





En ambos casos vamos a agregarles 3 componentes nuevos:

1. **Rect Transform**: componente que sirve para establecer un ancho y alto concreto a un elemento, sin necesidad de aplicarle una caja de colisión. Se utiliza para pintar texturas y elementos en pantalla.
2. **Canvas Renderer**: Competente que se une para poder fijar objetos en un Canvas.
3. **Text**: Competente para indicar un texto, pudiendo seleccionar color, fuente, tamaño, tipo, valor, etc. Al objeto Text le pondremos el valor 0 por defecto, y al final Text le pondremos el valor **ENHORABUENA**.



Ahora nos aparecerán en pantalla tanto el **0** como la palabra **ENHORABUENA**. Con el ratón los posicionamos en el lugar donde queramos que se queden fijados. En mi caso voy a colocar el 0 en la **zona superior de la pantalla** y la palabra enhorabuena un poco por debajo. La única diferencia es que voy a establecer que el color de la palabra **ENHORABUENA** sea por defecto **transparente**, de modo que no se vea nada. Posteriormente mediante un método llamaremos a este componente para que cambie su color a blanco, una vez el jugador llegue al final del nivel.

Con esto ya tendremos definidos los objetos de recolección, la UI y la puntuación. Solo quería empezar a disponer las manzanas desde el prefab a la escena. Para mantenerlas todas en cierto orden vamos a crear un objeto en la escena a la que llamaremos **Collectables**. Todos los objetos del prefab de manzanas los pondremos ahí y con el ratón los iremos disponiendo por el escenario:

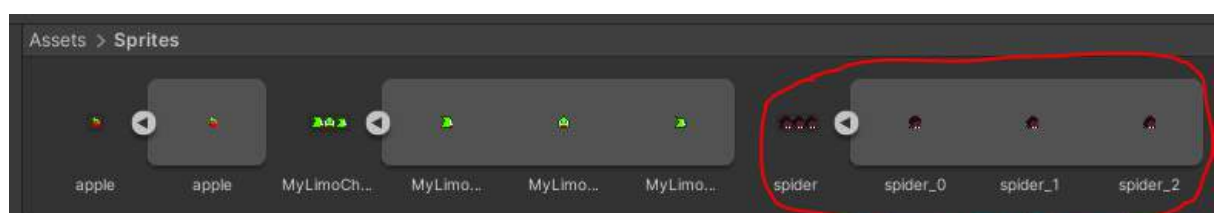


Sabremos que estamos invocando objetos de un **Prefab** porque se dispondrá en la jerarquía con un color de fuente **azulado**.

## 8.6.- Implementando comportamientos

Hasta ahora tenemos definidos algunos comportamientos en la escena, como el movimiento del personaje principal, su acción de salto y la posibilidad de recolección de manzanas. Quedaría la inclusión de enemigos y los condicionantes de nivel, aunque estos últimos los veremos en el siguiente apartado.

Centrándonos en los enemigos, lo primero que haremos es asegurarnos de que el Sprite de **araña** que creamos anteriormente en **Krita** se encuentra en nuestra carpeta de recursos de Sprite y que además la hemos dividido en partes equitativas aplicando el estado **Múltiple**, como ya hicimos con el **Sprite de Player**:



A continuación crearemos un objeto de juego en la escena y lo llamaremos **Enemies**. Este objeto será un objeto padre sin ningún tipo de definición particular pero que va a contener todos los enemigos del nivel. Es una técnica simplemente de ordenación de elementos en pantalla.

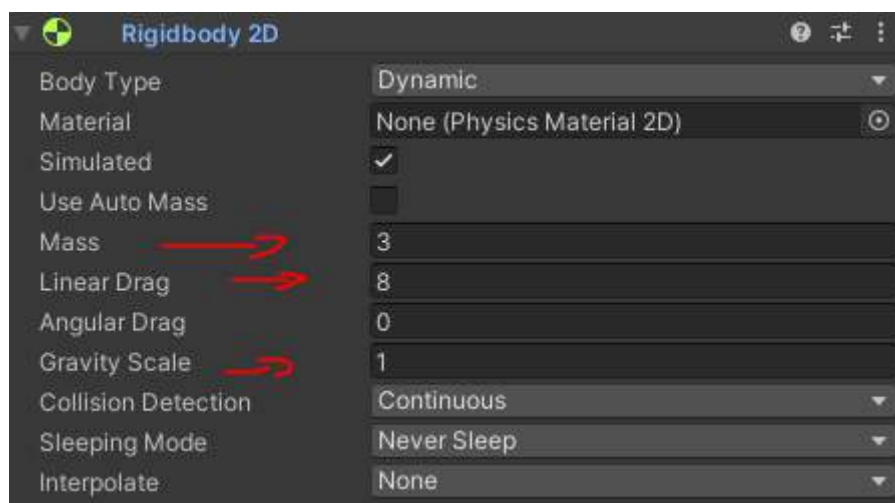
Nuestras arañas van a necesitar los siguientes componentes:

- **Transform:** Componente que viene por defecto con cada objeto y marca su posición en el escenario.



- **Sprite Renderer:** Lo utilizaremos para indicarle al objeto que Sprite utilizar. Como hemos comentado anteriormente se le aplicará el Sprite de araña definido como **Múltiple**.
- **Animator:** Si recordamos anteriormente hicimos una animación tanto para Player como para las arañas enemigas. Con este componente le aplicaremos esa animación.
- **Rigidbody2D:** Componente que une al objeto de juego con el motor de físicas de Unity. Lo utilizamos para el desplazamiento de las arañas, ya que van a desplazarse mediante impulsos de fuerza, del mismo modo que ocurría con Player.
- **BoxCollider2D:** Caja de colisiones para el objeto de juego, lo cual nos permitirá hacer que interactúe con otros. Con ello conseguiremos que Player pueda interactuar con la araña, saltando encima de ella y destruyéndola.
- **Script:** Para definir el comportamiento en C#.
- **Audio Source:** Para agregar el clip de audio que identifique la muerte del enemigo.

Como vamos a aplicar fuerzas para el movimiento del enemigo tenemos que configurar la masa de este. Para el diseño de las arañas queremos que su movimiento sea **lento y pesado**, de modo que sea más sencillo para el jugador el hecho de derrotarlas. Para ello vamos a establecerle una **masa** superior, de **3 unidades**. Al tener tanta masa no será necesaria una alta **escala de gravedad**, por lo que podremos dejarla en **1 unidad**. Con respecto a la configuración de **Linear Drag**, lo estableceremos en **8 unidades**, un poco por debajo de player, ya que con el aumento de masa ya se compensa la diferencia:



Si nos fijamos el resto de configuración es la misma que ya hicimos en **Player**, por lo que poca explicación quedaría por dar con respecto a este componente. A continuación vamos a nuestra carpeta **Script** y vamos a crear un nuevo **Script** para las arañas al que llamaremos **SpiderMoveSet**. Una vez lo hayamos creado lo vinculamos con el componente Script de nuestra araña y lo abriremos en **Visual Studio Code** y le aplicaremos el siguiente código:

```
public class SpiderMoveSet : MonoBehaviour
{
    public float timeRemaining;
    int direction = 1;
    public AudioSource clipDeath;
    public void Start()
    {
        timeRemaining = 3;
        direction = 1;
        gameObject.GetComponent().flipX = false;
    }
    // Update is called once per frame

    public void Update()
    {
        if (timeRemaining > 0)
        {
            timeRemaining -= Time.deltaTime;
            if (direction == 1)
            {
                gameObject.GetComponent().AddForce(new
Vector2(1f * Time.deltaTime, 0).normalized);
            }
            else
            {
                gameObject.GetComponent().AddForce(new
Vector2(-1f * Time.deltaTime, 0).normalized);
            }
        }
        else{
            timeRemaining = 3;
            if (direction == 1)
            {
                direction = 0;
                gameObject.GetComponent().flipX = true;
            }
        }
    }
}
```



```

        else
        {
            direction = 1;
            gameObject.GetComponent<SpriteRenderer>().flipX = false;
        }
    }
}

private void OnTriggerEnter2D(Collider2D collision){
    if (collision.transform.tag == "Player")
    {
        if (!collision.GetComponent<PlayerController>().canJump)
        {
            clipDeath.Play();
            Invoke("deleteObject", 0.1f);
            GameObject scripter = GameObject.Find("Scripter");

scripter.GetComponent<ScoreManagerScript>().increment(10);
            collision.GetComponent<Rigidbody2D>().AddForce(new
Vector2(0, 200f));
        }

    }

}

private void deleteObject()
{
    Destroy(gameObject);
}
}

```

Como podemos ver inicialmente declaramos 3 variables a utilizar. La primera es **timeRemaining**, que es un float que va a indicar el tiempo que queda hasta que la araña se dé la vuelta (Lo que buscamos es que la araña se mueva de izquierda a derecha en un ciclo infinito). La segunda variable es **direction**, que define la dirección en la que va a caminar la araña. Finalmente tenemos **AudioSource**, objeto que como ya sabemos de anteriores objetos, almacena clips de audio del objeto.

Cómo va a almacenar el clip de muerte lo denominaremos **clipDeath**. Todas estas variables salvo **direction** serán **públicas** y , por tanto, accesibles desde la interfaz de **Unity**. De este modo podremos vincular los datos con facilidad.



Lo siguiente que tenemos es el método **Start**, que recordemos que se ejecuta al inicio de la vida del objeto y básicamente define el comportamiento base inicial de las variables comentadas.

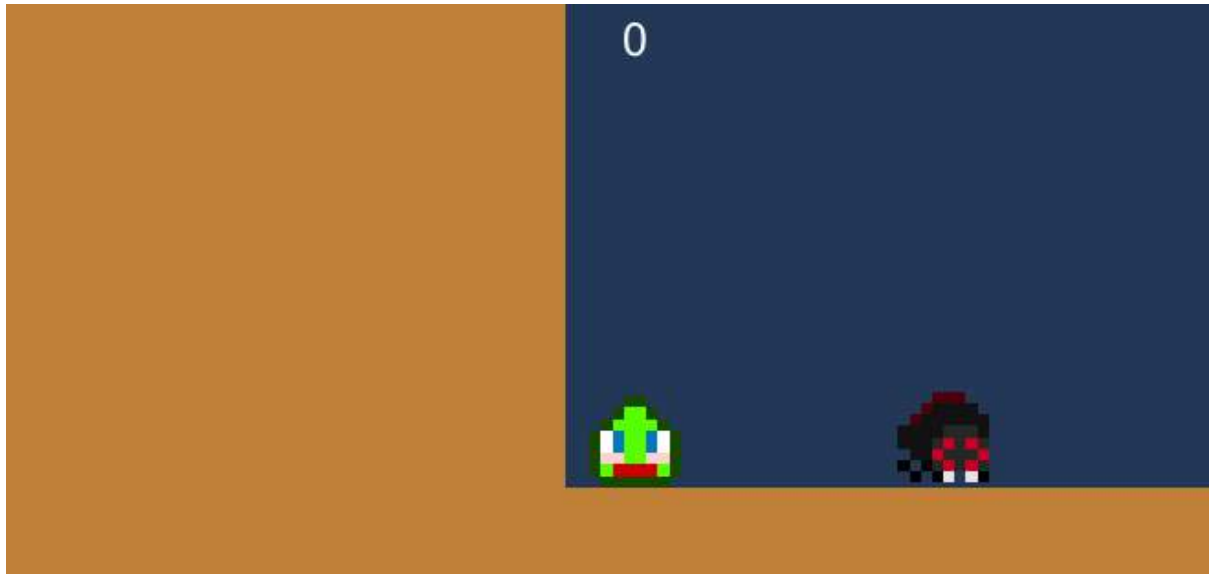
A continuación tenemos el método **Update**, que recordemos que se ejecuta en cada frame, pero del mismo modo que ocurría con **Player**, vamos a utilizar cálculos que utilicen **Time.deltaTime** para que las fuerzas a aplicar sobre el motor de físicas se aplican por tiempo en vez de por frames, como ya explicamos anteriormente. Para el movimiento de la araña simplemente comprobamos la variable **timeRemaining** y en caso de que sea superior a 0 (Es decir que no haya acabado el tiempo de giro) se mantendría con una fuerza aplicada concreta y en una dirección y, en caso contrario, se rotará la dirección y las fuerzas se invertirán (De positivo a negativo y de negativo a positivo, dle mismo modo que con Player).

Si hay algo que destaque en esta aplicación de fuerzas es el uso del método **normalized**, de la clase **RigidBody2D**. La utilidad del método es simple: Cuando en el mundo real le aplicamos una fuerza a un objeto, inicialmente la intensidad es alta pero al chocar con el objeto de destino, la masa y peso de dicho objeto reducirá esa fuerza inicial. En **Unity** ocurre algo similar, ya que aunque nosotros apliquemos una fuerza al objeto, si este tiene demasiada masa o una configuración pesada, la fuerza puede descompensarse entre inicio y fin de aplicación. Para evitar esto y hacer que la fuerza se aplique compensada desde el inicio haremos uso de este método **normalized**, consiguiendo un movimiento homogéneo todo el tiempo.

Finalmente quedan por comentar dos métodos que ya pudimos ver cuando analizamos el código de la recolección de manzanas. En caso de que la araña colisione con el objeto **Player** y este no pueda saltar (su variable **canJump** sea falsa) querrá decir que el jugador previamente ha saltado antes de colisionar, por lo que le ha saltado encima. Cuando ocurra esto se buscará al objeto de juego **Scripter** para incrementar los puntos de la partida en **10 puntos** y se aplicará una fuerza vertical hacia arriba al objeto **Player**, de modo que rebote al matar a la araña. Al ocurrir esto, se ejecutará el clip de audio de muerte de la araña y se llamará al método **deleteObject**, que destruirá el objeto araña.

Finalmente situaremos las **arañas** en las posiciones del escenario que estimemos oportunas. Una la dejaremos cerca de la posición inicial del jugador, de modo que desde el inicio tenga constancia de que existen y de su patrón de movimientos predecible:

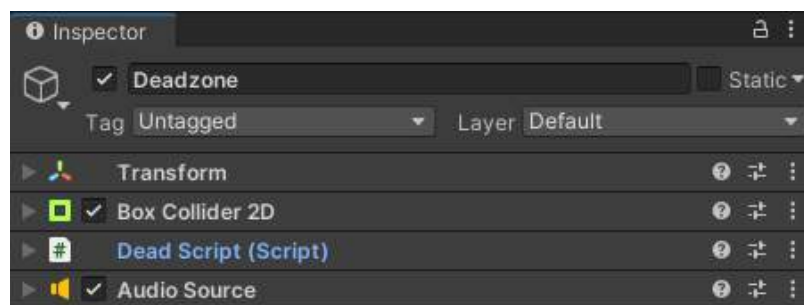




## 8.7.- Definiendo condicionantes del nivel (Muerte - Fin)

Tenemos que definir la condición que debe cumplirse dentro del nivel de juego para que el objeto **Player** pierda. A lo largo del escenario hemos dispuesto plataformas y elevadores que pueden provocar la caída del jugador fuera del escenario. En caso de que esto pase queremos que el jugador muera y vuelva a su posición inicial, por lo que tendremos que definir un objeto de juego que se sitúe por debajo de todo el escenario y que en caso de que el jugador colisione con este, se produzca el efecto comentado. Para ello vamos a crear un objeto de juego en la escena al que llamaremos **Deadzone**:

Este objeto debe tener el componente **Transform** por defecto, pero situado por debajo del escenario. No le aplicaremos un componente **SpriteRenderer**, ya que queremos que sea totalmente invisible. Necesitaremos aplicar un componente **Script** para definir el comportamiento, un **Audio Source** para establecer el clip de muerte del objeto **Player** y un **BoxCollider2D** para permitir que **Player** y **Deadzone** puedan interactuar mediante colisiones.



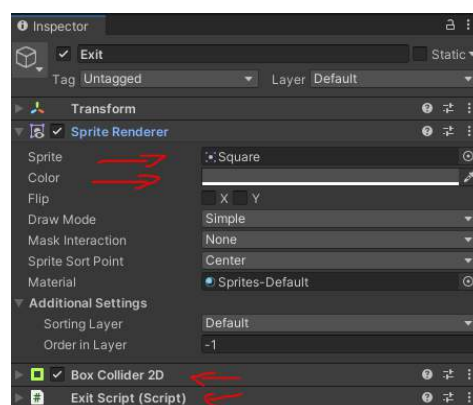


Ahora tenemos que crear un **Script**, al que llamamos **DeadScript** y lo aplicaremos al componente **Script** del objeto. Lo abriremos en **Visual Studio Code** y establecemos el siguiente código:

```
public class DeadScript : MonoBehaviour
{
    public AudioSource clip;
    // Start is called before the first frame update
    private void OnTriggerEnter2D(Collider2D collision)
    {
        if (collision.transform.tag == "Player")
        {
            collision.transform.position = new Vector3(0, 0, 0);
            clip.Play();
        }
    }
}
```

Como podemos ver el comportamiento es muy simple. Definimos una variable **AudioSource** para el clip de muerte del jugador y un método **OnTriggerEnter2D**. En el caso en el que el objeto con el tag **Player** colisione con el objeto **Deadzone**, se dispondrá al objeto **Player** en la posición 0,0,0 y se ejecutará el clip de muerte.

Finalmente queda hablar del condicionante de finalización del nivel. Para ello vamos a crear un objeto de juego al que llamaremos **Exit** y que va a tener los mismos componentes que el objeto de juego anterior, solo que esta vez no va a hacer falta el componente **Audio Source** y vamos a aplicarle un componente **Sprite Renderer** que va a tener asignado un Sprite básico de **Unity**, el Sprite **Square** (Si recordamos es el mismo sprite que hemos usado para crear el escenario, solo que mientras que el escenario es de color naranja, nuestro objeto Exit va a ser de color grisáceo). Por tanto deberíamos tener un objeto de juego tal que así:

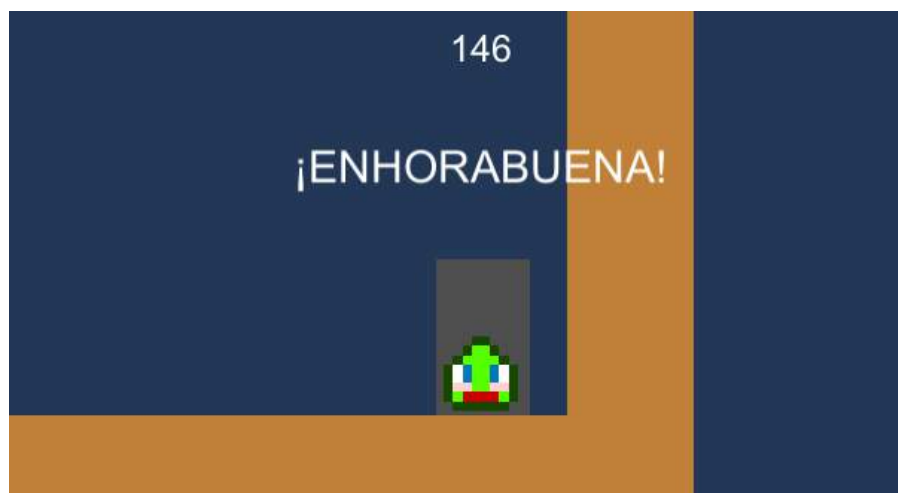




Si nos fijamos, hemos creado también un Script llamado **ExitScript** y lo hemos anexoado al componente **Script** del objeto. Lo abrimos con **Visual Studio Code** y añadimos el siguiente código:

```
public class ExitScript : MonoBehaviour
{
    private void OnTriggerEnter2D(Collider2D collision)
    {
        if (collision.transform.tag == "Player")
        {
            GameObject scripter = GameObject.Find("Scripter");
            scripter.GetComponent<ScoreManagerScript>().finalText.color=
Color.white;
        }
    }
}
```

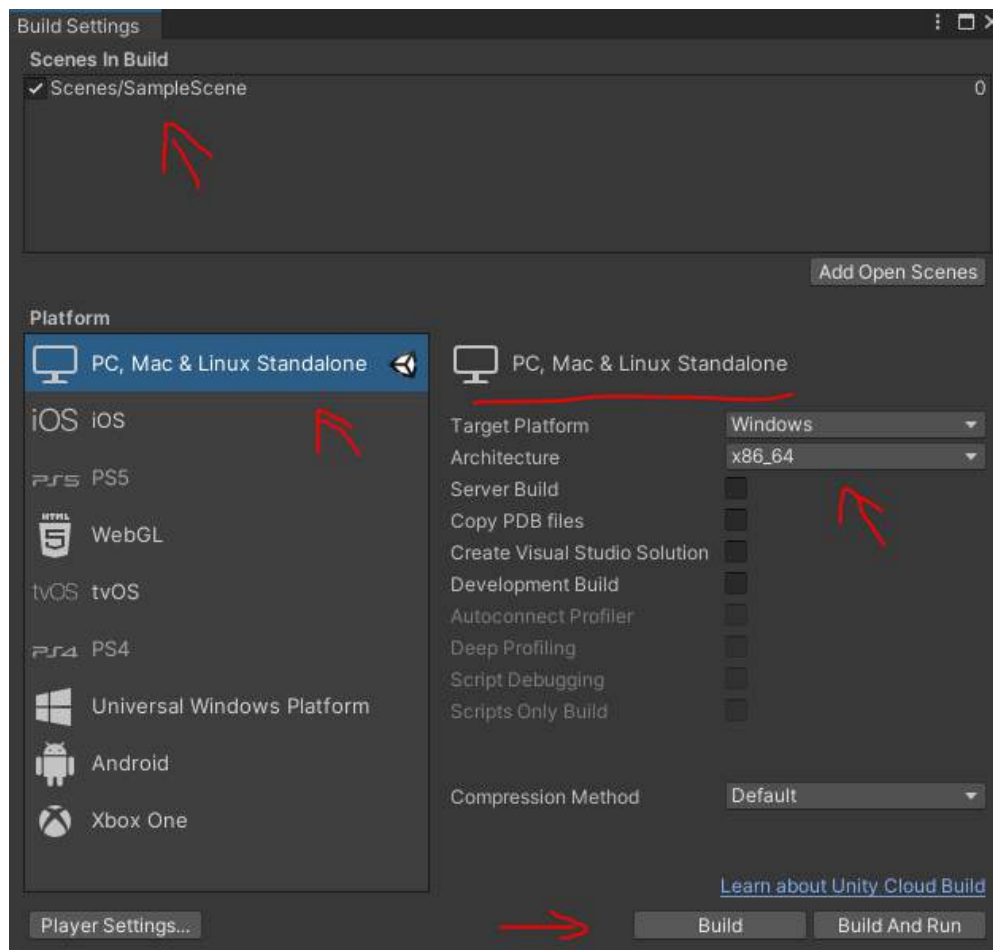
Como podemos ver es muy similar al anterior, solo que en este caso está llamando al objeto **Scripter** de la escena para hacer que la variable **finalText** de su **Script** tenga un color **blanco**, ya que si recordamos le habíamos puesto un tono **transparente** para que no se viese en la **UI**. De este modo cuando el jugador colisione con el ojo **Script** aparecerá el mensaje **¡ENHORABUENA!** en pantalla. Solo quedaría posicionar el objeto Exit al final de nivel y cuando el jugador llegue aparecerá el mensaje en pantalla, tal que así:



Con esto ya tendríamos el nivel completamente finalizado. Tan solo quedaría generar el ejecutable final del proyecto.

## 9.- GENERACIÓN DEL EJECUTABLE






Generar un **Ejecutable** en Unity es muy sencillo, ya que la herramienta se encarga de todo el proceso de construcción y comprensión del proyecto. Antes de la generación tenemos que acudir a **File-> Build Settings** para configurar las opciones de creación del ejecutable. Al acceder veremos una ventana como esta:



En el apartado de **Scenes in Build** seleccionamos las escenas que hemos creado y que vamos a establecer dentro de la aplicación. Como todo el nivel es una única escena y no hemos creado otras no tendremos que retocar nada. En la zona de la izquierda establecemos la plataforma sobre la que queremos que nos haga la compilación. En nuestro caso es un videojuego planteado únicamente para **ordenadores** ya que hemos establecido **inputs** de tecla únicamente para botones de un **teclado**. Por ello elegiremos la opción de **PC** y marcaremos una plataforma objetivo para la construcción. Como toda la creación y testeo se ha realizado en base a Windows 10 en una arquitectura de 64 bits, dejaremos las opciones mostradas por defecto y simplemente pulsamos en **Build**.



Se nos abrirá un explorador de carpetas y escribimos el nombre de la carpeta destino para el ejecutable. En mi caso voy a crear una carpeta llamada **MyBuilds** y la indicaré como destino. Una vez termine la creación del ejecutable lo tendremos disponible para su uso y distribución:

Nombre	Fecha de modificación	Tipo	Tamaño
 MonoBleedingEdge	03/05/2022 21:48	Carpeta de archivos	
 ProyectoFinalPlataformas_Data	03/05/2022 21:48	Carpeta de archivos	
 ProyectoFinalPlataformas	03/04/2022 10:25	Aplicación	639 KB
 UnityCrashHandler64	03/04/2022 10:31	Aplicación	1.206 KB
 UnityPlayer.dll	03/04/2022 10:31	Extensión de la ap...	27.614 KB



## 10.- CONCLUSIONES

Como hemos podido comprobar a la hora de realizar el nivel, es completamente necesario entender el funcionamiento de la herramienta con la que se trabaja y los elementos que utiliza, ya que aunque el comportamiento de los objetos se define mediante **Scripts** en **C#**, los propios objetos de juego deben seguir las normas establecidas por el motor tanto en la vertiente de diseño **2D**, **3D** o de cualquier tipo de **Asset** a crear, como ya vimos para el caso de los recursos de audio.

El ejercicio de creación de nivel planteado y desarrollado es ciertamente una forma de trabajar, pero no es en absoluto la única ni la más óptima, ya que este proyecto se enfoca más en conocer la herramienta y dar nuestros primeros pasos en el mundo del desarrollo de videojuegos mediante un motor gráfico concreto. En un futuro sería interesante aprender a manejar flujos de cámara y cinemáticas, creación de **IAs** más avanzadas, doblaje de personajes y gestión de elementos internos al videojuego, como un sistema de inventario o de misiones.





## ANEXO.- BIBLIOGRAFÍA

1. OXO - el primer videojuego:
  - a. <https://es.wikipedia.org/wiki/OXO>
  - b. <https://history-computer.com/oxo-game-guide/>
  - c. <https://hipertextual.com/2011/07/oxo-un-videojuego-para-uno-de-los-primeros-computadores-de-la-historia>
2. Alexander S. Douglas - El creador del primer videojuego:
  - a. <https://www.exevi.com/tag/alexander-s-douglas/>
3. Steve Russel - Creador de Spacewar!:
  - a. [https://es.wikipedia.org/wiki/Steve\\_Russell](https://es.wikipedia.org/wiki/Steve_Russell)
4. Spacewar! - El primer videojuego de disparos:
  - a. <https://es.wikipedia.org/wiki/Spacewar!>
  - b. <https://proyectoidis.org/spacewar/>
5. Magnavox Odyssey - La primera videoconsola doméstica.
  - a. [https://es.wikipedia.org/wiki/Magnavox\\_Odyssey](https://es.wikipedia.org/wiki/Magnavox_Odyssey)
  - b. <https://retromaquinitas.com/consolas/consolas-philips/odyssey/>
6. Ralph Hendrikson Baer - Creador de la Magnavox Odyssey
  - a. [https://es.wikipedia.org/wiki/Ralph\\_H.\\_Baer](https://es.wikipedia.org/wiki/Ralph_H._Baer)
  - b. <https://www.xataka.com/historia-tecnologica/a-le-debemos-primera-consola-videojuegos-historia-ralph-baer-gigante-ingenieria-vida-apasionante>
7. Pong - El primer videojuego de éxito comercial:
  - a. <https://es.wikipedia.org/wiki/Pong>
  - b. <https://hipertextual.com/2011/08/pong-videojuego-creo-industria>
8. Nolan Bushnell - Creador de Pong y fundador de Atari:
  - a. [https://es.wikipedia.org/wiki/Nolan\\_Bushnell](https://es.wikipedia.org/wiki/Nolan_Bushnell)
  - b. <https://forohistorico.coit.es/index.php/personajes/personajes-internacionales/item/bushnell-nolan-kay>
9. El crack de 1983 - La primera crisis del videojuego:
  - a. <https://khronoshistoria.com/go/historia-contemporanea/cultura-popular/crack-de-1983-videojuegos/>
  - b. [https://es.wikipedia.org/wiki/Crisis\\_del\\_videojuego\\_de\\_1983](https://es.wikipedia.org/wiki/Crisis_del_videojuego_de_1983)
  - c. <https://www.vidaextra.com/industria/el-et-de-atari-y-el-crack-del-83>
10. Game & Watch - La primera consola LCD de prestigio:
  - a. [https://es.wikipedia.org/wiki/Game\\_%26\\_Watch](https://es.wikipedia.org/wiki/Game_%26_Watch)
  - b. <https://www.vidaextra.com/hardware/game-watch-primera-gran-revolucion-nintendo-mundo-videojuegos>
11. Nintendo - La Gran N:
  - a. <https://es.wikipedia.org/wiki/Nintendo>
  - b. <https://www.nintendo.es/Hardware/La-historia-de-Nintendo/La-historia-de-Nintendo-625945.html>
  - c. <https://hablemosdeempresas.com/empresa/historia-nintendo/>
12. Sega - La reina de los arcades:
  - a. <https://es.wikipedia.org/wiki/Sega>
  - b. <https://www.tokioschool.com/noticias/historia-de-sega/>



- c. [https://as.com/meristation/2020/11/14/reportajes/1605344848\\_171306.html](https://as.com/meristation/2020/11/14/reportajes/1605344848_171306.html)
13. BASIC - El lenguaje de programación de alto nivel más reconocido:
  - a. <https://es.wikipedia.org/wiki/BASIC>
  - b. <https://internetpasoapaso.com/lenguaje-programacion-basic/>
14. Sello de calidad Nintendo - El primer sello de calidad de la industria del videojuego:
  - a. [https://es.wikipedia.org/wiki/Nintendo\\_Seal\\_of\\_Quality](https://es.wikipedia.org/wiki/Nintendo_Seal_of_Quality)
  - b. <https://www.vidaextra.com/industria/como-el-sello-de-calidad-de-nintendo-cambio-para-siempre-la-industria>
  - c. [https://nintendo.fandom.com/es/wiki/Nintendo\\_Seal\\_of\\_Quality](https://nintendo.fandom.com/es/wiki/Nintendo_Seal_of_Quality)
15. Los hermanos Stamper - Los padres de Rare:
  - a. [https://es.wikipedia.org/wiki/Tim\\_%26\\_Chris\\_Stamper](https://es.wikipedia.org/wiki/Tim_%26_Chris_Stamper)
  - b. <https://portal.33bits.net/ultimate-play-the-game-el-pasado-de-rare/>
  - c. [https://en.wikipedia.org/wiki/Ultimate\\_Play\\_the\\_Game](https://en.wikipedia.org/wiki/Ultimate_Play_the_Game)
16. Yu Suzuki - El genio del arcade:
  - a. [https://es.wikipedia.org/wiki/Y%C5%AB\\_Suzuki](https://es.wikipedia.org/wiki/Y%C5%AB_Suzuki)
  - b. <https://www.tokioschool.com/noticias/yu-suzuki/>
17. CDs - El primer disco óptico comercial:
  - a. [https://es.wikipedia.org/wiki/Disco\\_compacto](https://es.wikipedia.org/wiki/Disco_compacto)
  - b. <https://computerhoy.com/reportajes/tecnologia/historia-cd-677439>
18. Ken Kutaragi - El padre de la primera Sony PlayStation:
  - a. [https://es.wikipedia.org/wiki/Ken\\_Kutaragi](https://es.wikipedia.org/wiki/Ken_Kutaragi)
  - b. <https://sistemas.com/11631.php>
19. PlayStation - La primera videoconsola de Sony:
  - a. <https://redhistoria.com/historia-del-playstation-origen-evolucion-y-consolas/>
  - b. [https://es.wikipedia.org/wiki/PlayStation\\_\(marca\)](https://es.wikipedia.org/wiki/PlayStation_(marca))
20. Hiroshi Yamauchi - El tercer presidente de Nintendo:
  - a. [https://es.wikipedia.org/wiki/Hiroshi\\_Yamauchi](https://es.wikipedia.org/wiki/Hiroshi_Yamauchi)
  - b. [https://nintendo.fandom.com/es/wiki/Hiroshi\\_Yamauchi](https://nintendo.fandom.com/es/wiki/Hiroshi_Yamauchi)
  - c. <https://kriplekit.com/hiroshi-yamauchi/>
21. Phillips CD-i - El primer gran fracaso de Nintendo:
  - a. <https://www.gamerfocus.co/juegos/expedientes-gf-el-oscuro-capitulo-de-nintendo-y-philips-cd-i/>
  - b. <https://retromaquinitas.com/consolas/consolas-philips/cdi/>
22. Parties empresariales - Las puntas de lanza de la industria:
  - a. <https://arstechnica.com/civis/viewtopic.php?t=206689#:~:text=3rd%20party%20devs%20are%20independently.is%20partly%20owned%20by%20Sony.>
  - b. <https://www.unocero.com/videojuegos/gaming/que-son-los-juegos-first-party-second-party-y-third-party/>
  - c. <https://www.levelup.com/blogs/274507/Que-son-y-que-significa-First-Party-Second-Party-y-Third-Party>
23. Unreal Engine - El primer motor de Epic Games:
  - a. [https://es.wikipedia.org/wiki/Unreal\\_Engine](https://es.wikipedia.org/wiki/Unreal_Engine)
  - b. <https://www.tokioschool.com/noticias/unreal-engine-epic-games-historia/>
24. MicroDVD - El formato de almacenamiento de la Nintendo GameCube:
  - a. <https://hmong.es/wiki/MicroDVD>



- b. <https://es.wikipedia.org/wiki/MicroDVD>
- 25. DVDs - La evolución natural del CD:
  - a. [https://es.wikipedia.org/wiki/DVD#:~:text=El%20est%C3%A1ndar%20del%20DVD%20surg%C3%B3,permiten%20grabar%20y%20luego%20borrar\).](https://es.wikipedia.org/wiki/DVD#:~:text=El%20est%C3%A1ndar%20del%20DVD%20surg%C3%B3,permiten%20grabar%20y%20luego%20borrar).)
  - b. <https://www.escuelapedia.com/historia-del-dvd/>
  - c. <https://redgrafica.com/la-historia-del-dvd/>
- 26. Xbox - La primera videoconsola de Microsoft:
  - a. <https://es.digitaltrends.com/videojuego/la-historia-de-xbox/>
  - b. [https://es.wikipedia.org/wiki/Xbox\\_\(marca\)](https://es.wikipedia.org/wiki/Xbox_(marca))
  - c. <https://www.xbox.com/es-ES/power-on>
- 27. Nintendo Wii - El gran giro de timón de Nintendo:
  - a. <https://es.wikipedia.org/wiki/Wii>
  - b. <https://dossierinteractivo.com/nintendo-wii-historia-datos/>
  - c. <https://www.filo.news/gaming/Hace-13-anos-salia-Nintendo-Wii---la-historia-d-e-un-exito-y-un-fracaso-20191119-0073.html>
- 28. DLC - El poder de las microtransacciones:
  - a. [https://es.wikipedia.org/wiki/Contenido\\_descargable](https://es.wikipedia.org/wiki/Contenido_descargable)
  - b. <https://elsabio.es/que-es-un-dlc/>
  - c. [https://www.diariodesevilla.es/videojuegos/DLC-letras-caras-mundo-videojuegos\\_0\\_1428757274.html](https://www.diariodesevilla.es/videojuegos/DLC-letras-caras-mundo-videojuegos_0_1428757274.html)
- 29. Freemium - Cuando jugar gratis tiene un precio:
  - a. <https://economipedia.com/definiciones/modelo-freemium-que-es-y-como-funciona.html>
  - b. <https://www.questionpro.com/blog/es/que-es-el-modelo-freemium-y-por-que-deberias-considerarlo/>
  - c. <https://es.wikipedia.org/wiki/Freemium>
- 30. SSD - Velocidad y comodidad:
  - a. [https://es.wikipedia.org/wiki/Unidad\\_de\\_estado\\_s%C3%B3lido](https://es.wikipedia.org/wiki/Unidad_de_estado_s%C3%B3lido)
  - b. <https://www.beep.es/componentes-de-ordenador/discos-duros/discos-ssd>
- 31. PS Now - El servicio de entretenimiento por streaming de Sony:
  - a. <https://www.playstation.com/es-es/ps-now/getting-started/>
  - b. <https://www.pocket-lint.com/es-es/videojuegos/noticias/playstation/126394-playstation-ahora-explico-los-dispositivos-de-precio-como-conseguirlo>
  - c. <https://www.revistagq.com/noticias/articulo/ps-now-que-es-como-funciona-streaming-playstation>
- 32. Xbox Game Pass - El servicio de entretenimiento por streaming de Microsoft:
  - a. <https://www.xataka.com/basics/que-game-pass-que-ventajas-ofrece-suscripcion-microsoft>
  - b. <https://www.xbox.com/es-ES/xbox-game-pass>
  - c. [https://es.wikipedia.org/wiki/Xbox\\_Game\\_Pass](https://es.wikipedia.org/wiki/Xbox_Game_Pass)
- 33. El ciclo del desarrollo del videojuego:
  - a. <https://docs.hektorprofe.net/escueladevideojuegos/articulos/fases-del-desarrollo-de-videojuegos/>
- 34. PEGI - El sistema de autorregulación por edades en Europa:
  - a. <http://www.aevi.org.es/documentacion/el-codigo-peg/>



- b. <https://www.vidaextra.com/guias-y-trucos/pegi-esrb-cero-que-significan-etiquetas-edad-contenido-videojuegos>
- 35. Focus Testing - El feedback más rápido:
  - a. <http://kensheedlo.com/essays/why-focus-testing-is-awesome/>
  - b. <https://ahorasomos.izertis.com/globetesting/micro-focus-testing-funcional/>
- 36. Gold Game - El estado de lanzamiento del producto:
  - a. <https://www.hobbyconsolas.com/reportajes/significa-juego-ya-gold-hobby-basics-206308>
  - b. <https://www.gamerdic.es/termino/gold/>
- 37. C# - El lenguaje de .NET:
  - a. [https://es.wikipedia.org/wiki/C\\_Sharp](https://es.wikipedia.org/wiki/C_Sharp)
  - b. <https://docs.microsoft.com/es-es/dotnet/csharp/tour-of-csharp/>
- 38. Visual Studio Code - El IDE ideal para Unity:
  - a. [https://es.wikipedia.org/wiki/Visual\\_Studio\\_Code](https://es.wikipedia.org/wiki/Visual_Studio_Code)
- 39. Krita - Una herramienta gratuita y potente para crear Sprites:
  - a. <https://es.wikipedia.org/wiki/Krita>
- 40. Krita (Recurso) - Recurso The Legend Of Zelda
  - a. <https://www.purez.net/index.php?page=files&id=1561>
- 41. LMMS - Un editor y creador de sonidos muy recomendable:
  - a. <https://lmms.io/>
  - b. <https://es.wikipedia.org/wiki/LMMS>
- 42. Motores de videojuegos - La base de todo desarrollo actual:
  - a. [https://es.wikipedia.org/wiki/Motor\\_de\\_videojuego](https://es.wikipedia.org/wiki/Motor_de_videojuego)
  - b. <https://www.quora.com/What-was-the-first-game-development-engine-release-d-to-the-public>
- 43. Evolución e historia de la industria del videojuego - Contenido extra:
  - a. [https://www.amazon.es/Replay-historia-videojuegos-Tristan-Donovan/dp/8494816837/ref=asc\\_df\\_8494816837/?tag=googshopes-21&linkCode=df0&hvadid=267640459595&hvpos=&hvnetw=g&hvrnd=12893292561919043225&hvpo ne=&hvptwo=&hvqmt=&hvdev=c&hvdvcmdl=&hvlocint=&hvlocphy=1005421&hvtargid=pla-467834906862&psc=1](https://www.amazon.es/Replay-historia-videojuegos-Tristan-Donovan/dp/8494816837/ref=asc_df_8494816837/?tag=googshopes-21&linkCode=df0&hvadid=267640459595&hvpos=&hvnetw=g&hvrnd=12893292561919043225&hvpo ne=&hvptwo=&hvqmt=&hvdev=c&hvdvcmdl=&hvlocint=&hvlocphy=1005421&hvtargid=pla-467834906862&psc=1)
  - b. <https://www.nintendo.es/Hardware/La-historia-de-Nintendo/La-historia-de-Nintendo-625945.html#1889>
  - c. [https://en.wikibooks.org/wiki/NES\\_Programming/Introduction](https://en.wikibooks.org/wiki/NES_Programming/Introduction)
- 44. MonoBehavior - La clase principal de todo objeto de juego en Unity:
  - a. <https://docs.unity3d.com/ScriptReference/MonoBehaviour.html>
- 45. Softdisk - La primera piedra del camino al éxito de IdSoftware:
  - a. <https://en.wikipedia.org/wiki/Softdisk>
  - b. <https://hmong.es/wiki/Softdisk>
- 46. John Carmack y John Romero - Los padres del shooter moderno:
  - a. [https://es.wikipedia.org/wiki/John\\_Carmack](https://es.wikipedia.org/wiki/John_Carmack)
  - b. <https://www.biografias.es/famosos/john-carmack.html>
  - c. [https://es.wikipedia.org/wiki/John\\_Romero](https://es.wikipedia.org/wiki/John_Romero)
  - d. [https://doom.fandom.com/es/wiki/John\\_Romero](https://doom.fandom.com/es/wiki/John_Romero)
  - e. <https://allfamous.org/es/people/john-romero-19671028.html>



47. Commander Keen - La mascota del PC:
  - a. <https://vandal.elespanol.com/sagas/commander-keen>
  - b. [https://doom.fandom.com/es/wiki/Comandante\\_Keen](https://doom.fandom.com/es/wiki/Comandante_Keen)
  - c. [https://es.wikipedia.org/wiki/Commander\\_Keen](https://es.wikipedia.org/wiki/Commander_Keen)
48. IdSoftware - La cuna del motor gráfico actual:
  - a. [https://es.wikipedia.org/wiki/Id\\_Software](https://es.wikipedia.org/wiki/Id_Software)
  - b. <https://es.ign.com/id-software/170598/feature/30-anos-de-id-software-un-repa-so-a-la-historia-de-los-padres-de-los-videojuegos-modernos>
49. Wolfenstein 3D Engine - El primer motor de idSoftware:
  - a. [https://wolfenstein.fandom.com/es/wiki/Wolfenstein\\_3D\\_engine](https://wolfenstein.fandom.com/es/wiki/Wolfenstein_3D_engine)
  - b. [https://es.wikipedia.org/wiki/Wolfenstein\\_3D](https://es.wikipedia.org/wiki/Wolfenstein_3D)
  - c. <https://fabiensanglard.net/gebbwolf3d/>
50. Doom Engine - La evolución natural del W3DE:
  - a. [https://es.wikipedia.org/wiki/Doom\\_Engine](https://es.wikipedia.org/wiki/Doom_Engine)
  - b. [https://doom.fandom.com/es/wiki/Motor\\_de\\_Doom](https://doom.fandom.com/es/wiki/Motor_de_Doom)
51. Quake Engine - El padre de los motores de físicas y entornos 3D poligonales:
  - a. [https://es.wikipedia.org/wiki/Quake\\_engine](https://es.wikipedia.org/wiki/Quake_engine)
  - b. <https://es-academic.com/dic.nsf/eswiki/975518>
  - c. [https://www.wikiwand.com/es/Quake\\_engine](https://www.wikiwand.com/es/Quake_engine)
52. JMonkey - El motor de videojuegos en Java:
  - a. [https://es.wikipedia.org/wiki/JMonkey\\_Engine](https://es.wikipedia.org/wiki/JMonkey_Engine)
  - b. <https://www.adictosaltrabajo.com/2009/07/14/j-monkey-engine-primer-juego/>
53. GameMaker - El programa de animaciones con altas aspiraciones:
  - a. <https://gamemaker.io/es/gamemaker>
  - b. <https://gamemaker.io/es>
  - c. [https://es.wikipedia.org/wiki/GameMaker\\_Studio](https://es.wikipedia.org/wiki/GameMaker_Studio)
54. EpicGames - La empresa que dio origen al Unreal Engine:
  - a. [https://es.wikipedia.org/wiki/Epic\\_Games](https://es.wikipedia.org/wiki/Epic_Games)
  - b. [https://gearsofwar.fandom.com/es/wiki/Epic\\_Games](https://gearsofwar.fandom.com/es/wiki/Epic_Games)
  - c. [https://as.com/meristation/2021/11/29/reportajes/1638173667\\_712771.html](https://as.com/meristation/2021/11/29/reportajes/1638173667_712771.html)
55. Unity - El motor de videojuegos más utilizado en la industria actual:
  - a. <https://unity.com/es>
  - b. [https://es.wikipedia.org/wiki/Unity\\_Technologies](https://es.wikipedia.org/wiki/Unity_Technologies)
  - c. <https://www.capterra.es/software/158591/unity>
56. Godot - Un motor de videojuegos con enorme potencial:
  - a. <https://godotengine.org/>
  - b. <https://es.wikipedia.org/wiki/Godot>
57. Unity Hub - El gestor de proyectos en Unity:
  - a. <https://unity.com/es/unity-hub>
58. GameObject - La unidad del todo en una escena de Unity:
  - a. <https://docs.unity3d.com/560/Documentation/Manual/class-GameObject.html>
59. Component - Las partes de un GameObject:
  - a. <https://docs.unity3d.com/560/Documentation/Manual/Components.html>
60. Sprite - Los archivos de imágenes de un videojuego:



- a. <https://www.didactoons.com/que-es-un-sprite-y-para-que-sirve-en-el-desarrollo-de-videojuegos/>
- b. [https://es.wikipedia.org/wiki/Sprite\\_\(videojuegos\)](https://es.wikipedia.org/wiki/Sprite_(videojuegos))
- 61. Script - Documento de secuencias de comandos en un motor de videojuegos:
  - a. <https://www.seoestudios.es/blog/que-es-un-script/>
  - b. <https://es.wikipedia.org/wiki/Script>
- 62. Frames - La tasa de refresco de pantalla:
  - a. <https://www.xatakahome.com/curiosidades/fps-frecuencia-refresco-dos-factor-es-a-tener-cuenta-a-hora-optimizar-uso-monitor>
  - b. <https://www.xatakahome.com/televisores/lo-nombramos-muchas-veces-pero-sabemos-que-significan-los-hertzios-en-un-televisor-te-lo-explicamos>
- 63. Rigidbody - Los cuerpos físicos del motor de físicas de Unity:
  - a. <https://docs.unity3d.com/es/2018.4/Manual/class-Rigidbody.html>
- 64. Box Collider - El esquema de colisiones más utilizado:
  - a. <https://docs.unity3d.com/es/2018.4/Manual/class-BoxCollider.html>
- 65. Prefabs - Las plantillas de los objetos de juego en Unity:
  - a. <https://docs.unity3d.com/es/530/Manual/Prefabs.html>
- 66. Trigger - El activador de eventos:
  - a. <https://docs.unity3d.com/ScriptReference/Collider.OnTriggerEnter.html>