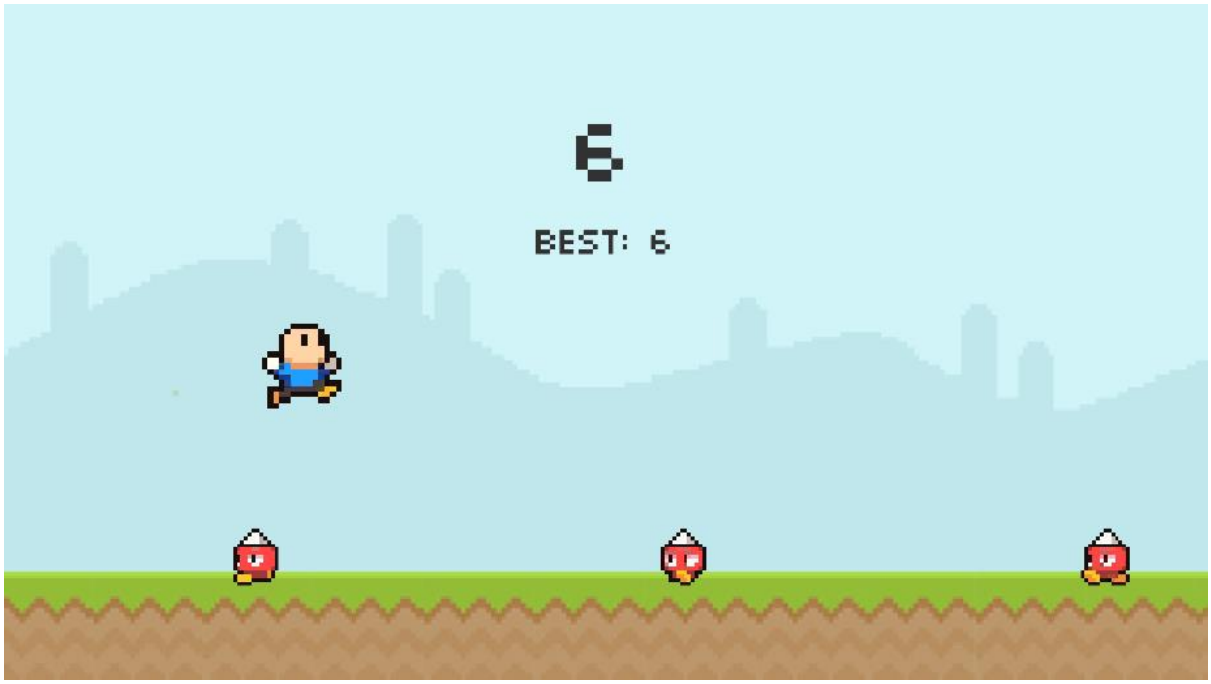
JUMPING GUY

Tu primer videojuego 2D completo y multiplataforma en Unity 5



Un curso para Escuela de Videojuegos

creado por Héctor Costa

Índice

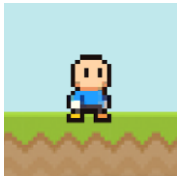
Bienvenida

Hola muy buenas! Me llamo Héctor y soy aficionado al desarrollo de videojuegos, programador de profesión y apasionado por enseñar a los demás. Tengo unos 10 años de experiencia en este mundo y actualmente ejerzo de instructor de programación en la plataforma Udemy. Mi lenguaje favorito es Python y mi especialidad es el desarrollo web full-stack con Django y Angular.

El mundo de los videojuegos siempre me ha gustado, pero nunca me había aventurado a crear un juego por temor a que fuera demasiado difícil. Sin embargo al final tomé la decisión de aprender sobre el tema y como consecuencia de juntar mis tres pasiones (la programación, enseñar y los videojuegos) a mediados de 2015 fundé una escuela de videojuegos con la excusa de seguir aprendiendo, centrada en tutoriales para Youtube.

Al principio la escuela estaba centrada en el programa GameMaker Studio 1.4, y así he pasado 2 años compartiendo todo tipo de vídeos y tutoriales mientras yo mismo aprendía, pero con el lanzamiento de GameMaker Studio 2, el desánimo por la falta de nuevas funcionalidades se apoderó de mí, y como consecuencia dí un cambio brusco hacia un motor mucho más potente llamado Unity.

Esta nueva etapa con el motor Unity ha supuesto un antes y un después en mi forma de entender tanto el mundo del desarrollo de videojuegos como mi propia escuela. He decidido ir un paso más allá y tomarme todo este proyecto mucho más en serio, esforzándome mucho más en cada vídeo con la intención de ayudar a más gente para por qué no llegar algún día a vivir haciendo lo que me gusta.



Ahora, después de 3 meses aprendiendo los fundamentos de este motor he creado este curso como culminación de la etapa de pre-aprendizaje, cubriendo de forma práctica y conectando todos esos conocimientos adquiridos y muchos otros que surgen durante el desarrollo de este juego, mi primer juego 2D completo y multiplataforma con Unity. Si quieres probarlo lo encontrarás para Windows, en la Web y en Android.

¡Espero que aprendas mucho a lo largo de este pequeño viaje!

Héctor Costa,
Escuela de Videojuegos
21 de Marzo de 2017

Requisitos

Tengo que empezar siendo contundente y claro, este curso no es para todo el mundo.

Está dirigido a todos los alumnos y seguidores de la escuela que me han seguido desde el primer vídeo sobre Unity, que ya conocen el uso del motor y que están familiarizados con el lenguaje orientado a objetos C#.

Por lo que se requiere:

- **Fundamentos de programación orientada a objetos**
- **Fundamentos del motor Unity 5**
- **Fundamentos del lenguaje C#**

No te recomiendo seguir el curso si no cumples los 3 requisitos porque te vas a perder fácilmente y sentirás frustración, la programación es muy puñetera y Unity no es precisamente un programa fácil de dominar.

Por suerte para tí todos mis vídeos de fundamentos de Unity son gratuitos, incluido un repaso por el lenguaje C# de 2 horas de duración. Los puedes encontrar en esta lista de reproducción.

Recursos

Todo lo que necesitas para realizar el videojuego, texturas, imágenes, sonidos, música, fuentes… Te lo proporcionaré en un fichero comprimido para que puedas centrarte en lo realmente importante: aprender.

Los encontrarás en esta url: https://escueladevideojuegos.net/download/3848/

Como personalmente no soy artista, ni dibujante, ni compositor, para conseguir un juego con un mínimo de calidad debo hacer uso de recursos de otras personas. Por suerte hay buena gente en el mundo dispuesta a compartir su arte sin pedir nada a cambio, pero aún así creo que es de obligación moral citar al autor de cada recurso:

Fichero	Autor	Descripción
Fuente	04	http://www.04.jp.org
Texturas	Kenney	https://kenney.nl/assets
Sprites	Grafxkid	http://opengameart.org/content/classic-hero-and-baddies-pack
Tema musical	Joth	http://opengameart.org/content/8bit-theme
Melodía al perder	LRSF	https://freesound.org/people/LittleRobotSoundFactory/270329
Sonidos SFX		http://www.drpetter.se/project_sfxr.html

Tampoco olvides que este documento es sólo un apoyo para el curso en vídeo que encontrarás en la web de la escuela.

Dicho esto, podemos continuar.

Diseño multiplataforma

Antes de ponernos manos a la obra con la creación de un juego 2D multiplataforma es muy importante tener en cuenta varios factores para crear un buen diseño del escenario.

Como sabéis existen pantallas de todas las formas y colores, con un montón de resoluciones diferentes. Desde pantallas cuadradas, rectangulares, super anchas… Esto en juegos con un gran escenario no suponen un problema porque la cámara se adapta a la pantalla tomando sólo una parte de ese escenario, pero cuando creamos juegos cuyo fondo es el límite del propio escenario, la cosa cambia.

Relación de aspecto

El primer concepto que tiene que quedarnos claro es el de relación de aspecto. La relación de aspecto es un cálculo que indica la proporción entre el tamaño horizontal y el tamaño vertical de una imagen. Por ejemplo, una relación 4/3 indica que por cada 4 unidades de anchura, la altura sería de 3 unidades, independientemente de lo que valga una unidad. Pueden ser píxeles, pulgadas, centímetros…

Algunas de las relaciones de aspecto más famosas para pantallas son:

Relación	Valor	Descripción
5/4	1.25	Estándar en monitores de ordenador prácticamente cuadrados.
4/3	1.33	Estándar del formato PAL para televisores, pantallas de ordenador y iPads.
3/2	1.5	Estándar del formato NTSC para televisores y algunos smartphones.
16/9	1.77	Estándar de pantallas HD, conocido como formato panorámico o widescreen.

Por tanto, si conseguimos adaptar nuestro juego para cualquier proporción entre 5/4 (1.25) y 16/9 (1.77), éste se verá bien en casi cualquier dispositivo, sin importar que sea un Smartphone, una Tablet o iPad, una pantalla de PC o una TV.

Resolución base y zona segura

Una vez tenemos claro qué es la relación de aspecto es hora de empezar el diseño de nuestro juego, para ello necesitamos decidir una resolución base. La resolución no es tan importante como la relación de aspecto, y aunque es cierto que cuanta más resolución tiene un sprite mejor se ve, también hay que tener en cuenta que motores como Unity son capaces de escalar y suavizar los gráficos bastante bien. ¿Entonces qué resolución hay que elegir? Pues cualquiera resolución de 16/9:

Nombre	**Resolución**
Full HD	1920 x 1080
HD+	1600 x 900
HD	1280 x 720
qHD	960 x 540
nHD	640 x 360

Para un juego pixel-art una resolución nHD o qHD suele servir, si no es pixel-art una HD o HD+ escalada suele verse muy bien en todos los dispositivos. Y si buscamos la excelencia, Full HD siempre le dará un toque muy pulido al juego. Como el que vamos a desarrollar es de estilo pixelado he decidido ir a por una resolución nHD de 640 x 360:

Muy bien, ahora deberíamos ponernos a diseñar el juego, pero antes hay que aprender un concepto importantísimo, me refiero a la zona segura.

La zona segura hace referencia a un espacio del escenario que será visible en cualquier dispositivo, independientemente de su relación de aspecto o resolución. Para identificar el tamaño horizontal de la zona segura tenemos que seguir una fórmula:

NUEVO ANCHO = ALTO * RELACIÓN DE ASPECTO MÍNIMA

Por ejemplo en base a las resoluciones 16/9 suponiendo que la relación de aspecto mínima es 5/4 (1.25) el nuevo ancho es:

Nombre	**Resolución**	**Zona Segura**
Full HD	1920 x 1080	1350 x 1080
HD+	1600 x 900	1125 x 900
HD	1280 x 720	900 x 720
qHD	960 x 540	675 x 540
nHD	640 x 360	480 x 360

Una vez tenemos la resolución de la zona segura sólo la tenemos que centrar sobre la resolución base y así la diferencia entre ambas representará la zona no segura.

Ahora sí que podemos diseñar nuestro juego, siempre que todos los elementos no sacrificables del mismo se sitúen sobre la zona segura.

Cuento con un fondo y una plataforma que nos servirán como texturas de base.

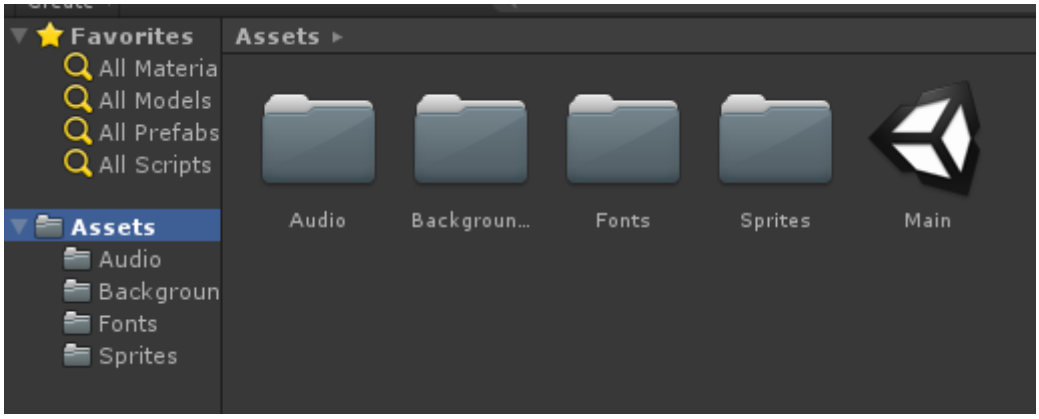
La diferencia entre una textura y un sprite es que una textura es como un patrón que se puede repetir, en ocasiones horizontalmente, otras verticalmente y a veces en ambas disposiciones. Esto lo he decidido así porque para generar un efecto Parallax (lo veremos más adelante), es necesaria una textura.

En base a este escenario he hecho un pre-diseño ubicando todos los elementos importantes sobre la zona segura, no es definitivo pero como guía se ve por donde van los tiros.

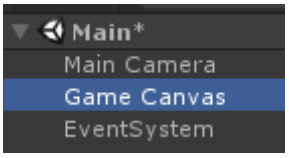
Creando la escena

Sin proyecto no hacemos nada, así que vamos a empezar creando un nuevo proyecto 2D llamado Jumping Guy, y lo primero será establecer la cámara en aspecto libre.

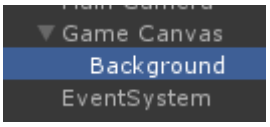
Bien, ahora vamos a importar los assets, sólo tenéis que arrastrar el contenido de la carpeta que os facilito:



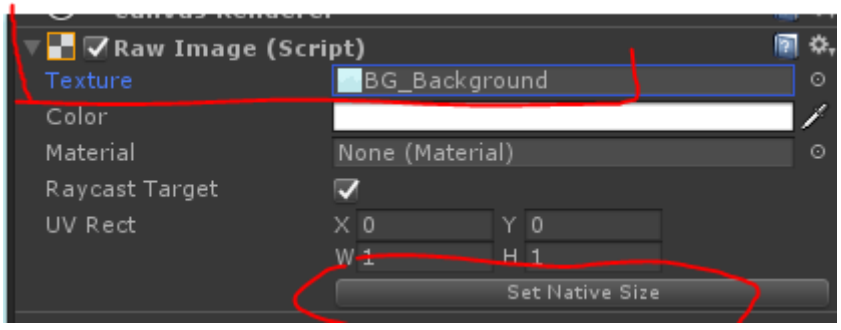
Seguidamente crearemos un objeto Canvas (Game Canvas), dentro pondremos dos Raw Images para el fondo y la plataforma. Lo hago así porque es la única forma que he encontrado para hacer un efecto Parallax de forma sencilla:



La primera RawImage la llamaré Background:

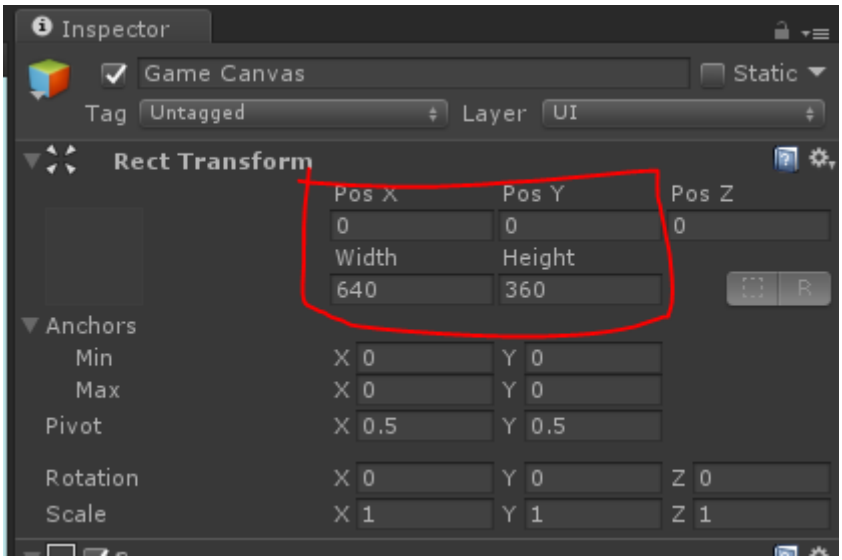


Aquí seleccionaremos la imagen de fondo y le daremos a Set Native Size, así se rellenará el espacio:



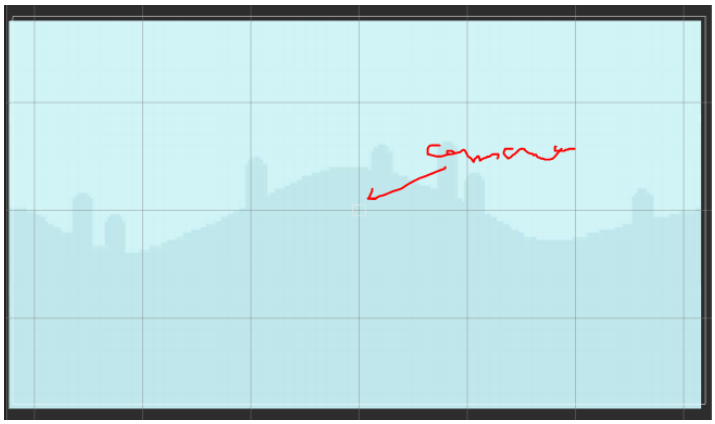
Bien, ahora ya tenemos algo con lo que empezar a juguetear. Lo primero que haremos es cambiar el **Render Mode** a **World Space**. Esto lo haremos porque el fondo del canvas nos hará de fondo del juego, será el mundo.

Ahora se nos han habilitado las opciones de redimensión, así que vamos a adaptar el canvas a la resolución de referencia de nuestro juego: **640 x 360**, también centraremos las posiciones:



Al hacer esto, si previsualizamos el juego veremos un zoom gigantesco, ¿que está sucediendo? Bueno, pues resulta que la cámara ortográfica divide el mundo en algo llamado unidades del mundo.

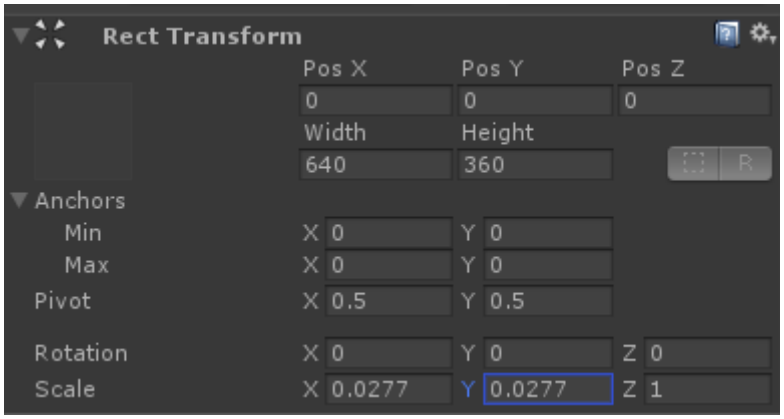
Según la configuración actual la cámara tiene un tamaño de 5, eso es un tamaño muy pequeño para nuestro canvas:



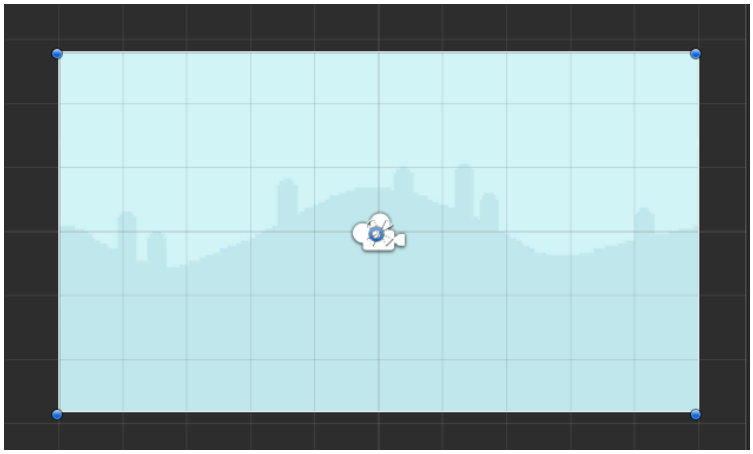
¿Qué hacemos entonces? Bueno, por regla general no es buena idea hacer que el mundo esté dividido en demasiadas unidades y menos para un juego simple como este. La alternativa es reescalar el canvas, así que vamos a hacerlo. ¿Pero a qué cantidad lo reescalamos?

Aquí tenemos que realizar unos cálculos. Si el tamaño de nuestra cámara está configurada a **5**, por tanto **10** unidades verticales de mundo, lo cual me parece bien, tenemos que dividir el número de unidades del mundo entre la anchura del canvas, es decir: **10 / 360 = 0.0277**

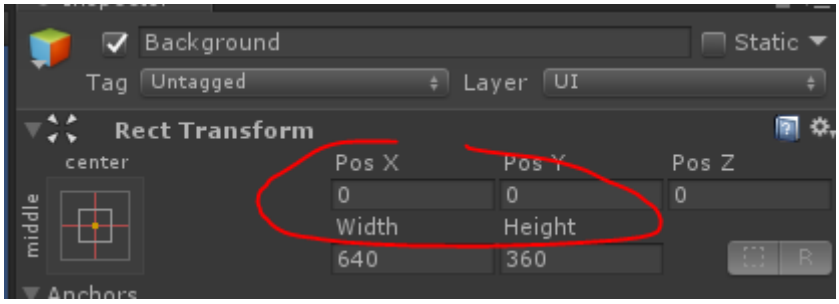
Ese es nuestro número:



Bien, ahora ya tenemos el canvas ajustado perfectamente a la cámara y el fondo dividido en 10 unidades del mundo, una para cada cuadrado:

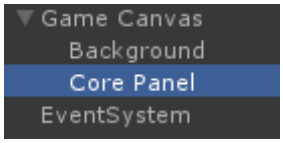


Si no lo tenéis bien centrado repasad que las posiciones del background a 0, a veces se modifican sin darnos cuenta:

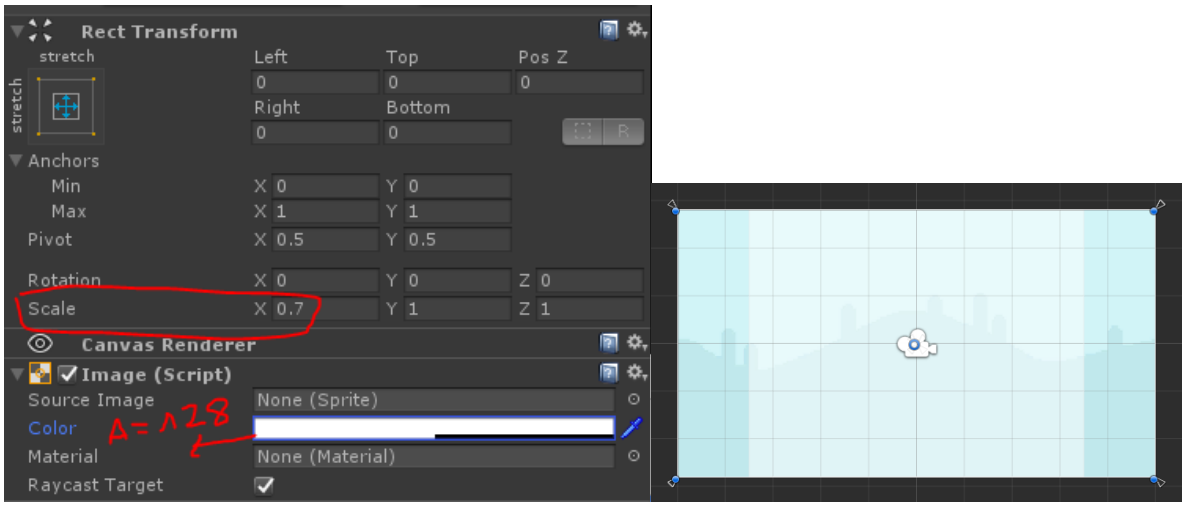


Estupendo. Ya tenemos configurado, incluso si cambiamos entre varios aspectos podréis apreciar como se recorta la escena a partir del centro, siendo el 5:4 el aspecto más cuadrado (un poco más que los iPads). Este espacio corresponde a la zona segura, es decir, tendremos que evitar poner los objetos de juego fuera de este área.

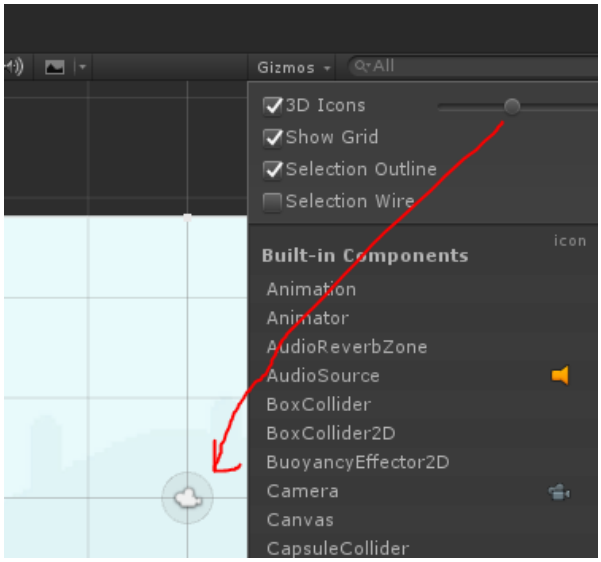
De hecho antes de continuar es buena idea crear un Core Panel, una guía de referencia para tener siempre en cuenta este espacio. Hagámoslo (crear UI > Panel):



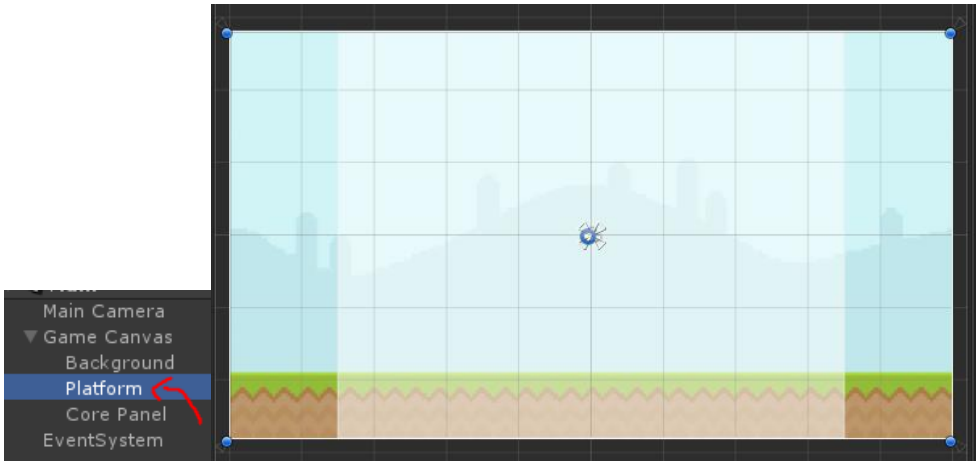
Por defecto el panel ocupa todo el canvas, sólo lo tenemos que escalar un poco, 0.7 en el eje X hará que ocupe el 70% centrado al medio, y eso nos hará el truco. Si además añadimos una transparencia al color blanco tendremos nuestro panel limitando el espacio seguro.



Genial, puede ser un buen momento para reducir el tamaño de los gizmos, así nos molestará menos:



Hecho esto vamos a añadir la plataforma, será muy fácil. Sólo tenemos que clonar el Background y cambiar la imagen (recordad ponerlo detrás del Core Panel):

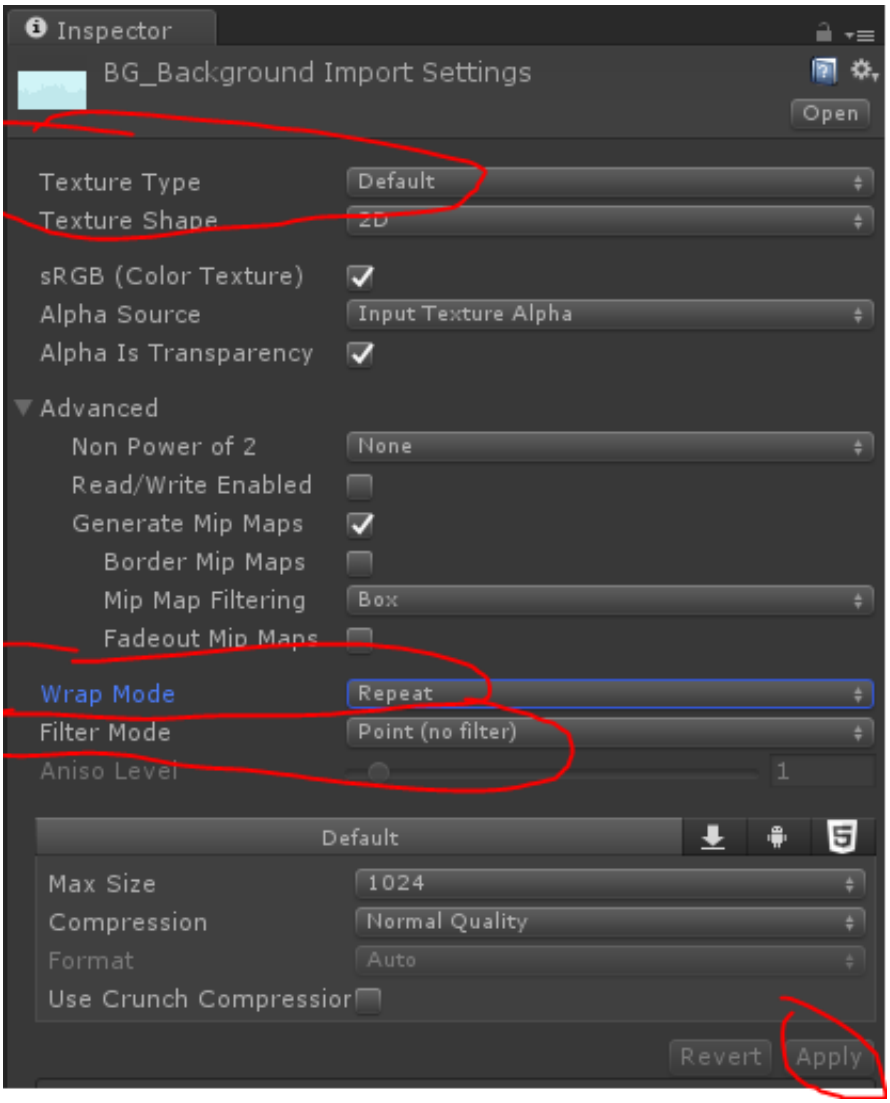


Muy bien, ya estamos listos para agregar el efecto parallax.

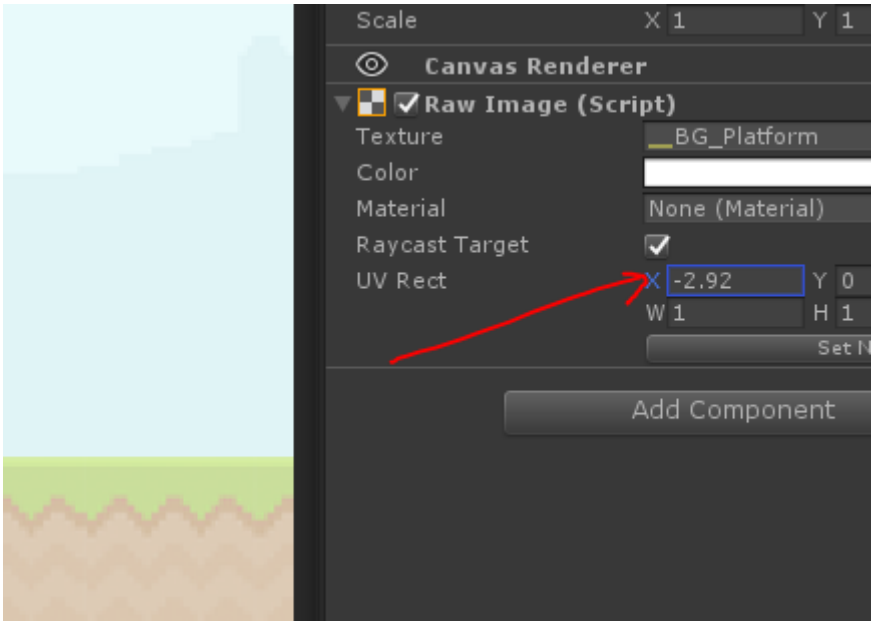
Efecto Parallax de fondo

Los efectos parallax se utiliza para generar sensación de profundidad. En nuestro caso añadiremos un movimiento al fondo y a la plataforma para que parezca que se mueven a velocidades distintas.

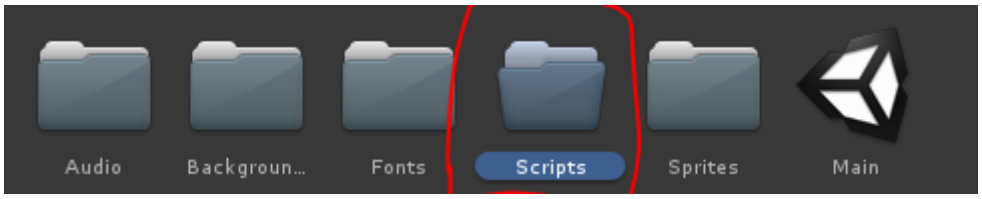
Para lograr el efecto jugaremos con el componente Raw Image y su propiedad UV Rect, sin embargo para utilizarlo antes tenemos que transformar las dos imágenes a Texturas, hacer que se repitan y poner el filtrado en Point para que escale los píxeles sin suavizarlos:



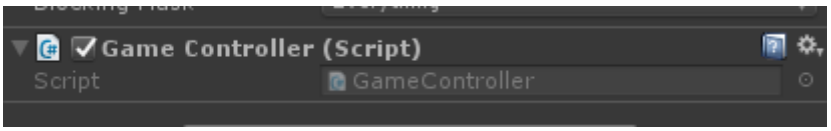
Habiendo hecho esto, si toqueteamos la propiedad X del UV Rect, veremos cómo añadir un margen u offset al fondo y este se repite:



La gracia entonces es hacer que el juego incremente esta propiedad X un poco en cada fotograma, tanto para el fondo como para la plataforma, pero con cantidades distintas. Así lograremos ese efecto Parallax. Necesitaremos un Script, así que vamos a crear una carpeta para guardarlo:



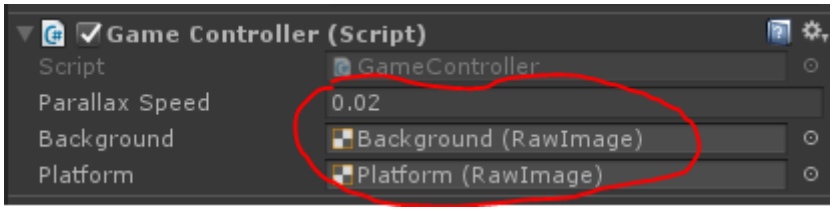
El Script lo vamos a añadir al Game Canvas. Al ser el objeto básico será el encargado de manejar funcionalidades generales. Le llamaré **GameController**:



Este tomará una variable flotante pública gracias a la que podremos graduar la velocidad, y también crearemos dos variables más de tipo RawImage para enviarle el Background y la Plataforma fácilmente:

```
1  using System.Collections;
2  using System.Collections.Generic;
3  using UnityEngine;
4  using UnityEngine.UI;
5
6  public class GameController : MonoBehaviour {
7
8      public float parallaxSpeed = 0.02f;
9      public RawImage background;
10     public RawImage platform;
11
12     void Start () {
13
14     }
15
16     void Update () {
17         float finalSpeed =  parallaxSpeed * Time.deltaTime;
18         background.uvRect = new Rect(background.uvRect.x + finalSpeed, 0f, 1f, 1f);
19         platform.uvRect = new Rect(platform.uvRect.x + finalSpeed * 4, 0f, 1f, 1f);
20     }
21 }
```

No olvides asignar los RawImage:



Y si ahora probamos el juego ya lo tendremos. Además podemos graduar fácilmente la velocidad. Por cierto, voy a introducir un TIP de Unity. Podemos agregar una barra de selección para una variable pública. Es muy fácil, hemos de añadir un rango de posibilidades justo encima de la variable en cuestión:

```
[Range(0f, 0.20f)]
public float parallaxSpeed = 0.02f;
```

De esta forma nos será más cómodo ajustar una velocidad de testeo:

Vamos a dejar una velocidad base de 0.025, y con esto dejamos por finalizada la escena y el efecto parallax.

Portada animada e inicio

Ahora que tenemos el fondo, vamos a suponer que por defecto el juego está parado. Entonces cuando el usuario presione una tecla empezará el juego.

Para lograr este comportamiento debemos controlar de alguna forma si el juego ha empezado o no. Creo que lo mejor en este punto es utilizar un enumerador. Un enumerador es una estructura que nos permitirá definir distintas opciones. Nosotros vamos a utilizar un enumerador para almacenar el estado del juego, por ahora tendremos 3: Idle y Playing:

```
public enum GameState {Idle, Playing};
```

Ahora crearemos una variable público de tipo GameState con el valor por defecto Idle, esperando:

```
[Range(0f, 0.20f)]
public float parallaxSpeed = 0.02f;
public RawImage background;
public RawImage platform;
public GameState gameState = GameState.Idle;
```

El truco consistirá en cambiar el valor a Playing cuando presionemos una tecla o presionemos el ratón (así permitimos jugar de dos formas). Pero la condición fundamental es que el estado sea ya Idle:

```
// Empieza el juego
if(gameState == GameState.Idle && (Input.GetKeyDown("up") || Input.GetMouseButtonDown(0))){
    gameState  = GameState.Playing;
}
```

Muy bien, por tanto ahora tenemos una variable para controlar si el juego ha empezado, así que nuestra primera tarea puede ser iniciar el efecto parallax sólo si ha empezado el juego:

```
// El juego está en marcha
if(gameState == GameState.Playing){
    // Efecto parallax
    float finalSpeed =  parallaxSpeed * Time.deltaTime;
    background.uvRect = new Rect(background.uvRect.x + finalSpeed, 0f, 1f, 1f);
    platform.uvRect = new Rect(platform.uvRect.x + finalSpeed * 4, 0f, 1f, 1f);
}
```

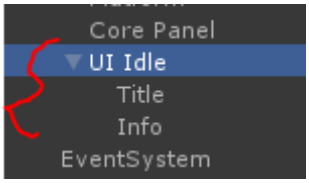
Esto funciona perfecto, pero vamos a refactorizar el código en un método a parte:

```
// Si el juego está en marcha
else if (gameState == GameState.Playing) {
    Parallax();
}
```

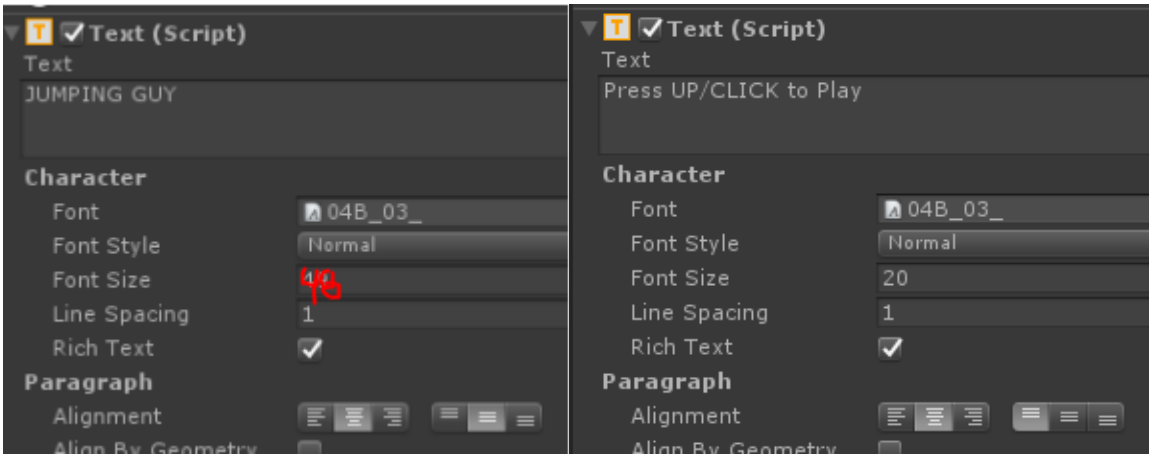
```
void Parallax(){
    // Efecto parallax
    float finalSpeed =  parallaxSpeed * Time.deltaTime;
    background.uvRect = new Rect(background.uvRect.x + finalSpeed, 0f, 1f, 1f);
    platform.uvRect = new Rect(platform.uvRect.x + finalSpeed * 4, 0f, 1f, 1f);
}
```

Vale, pues vamos a centrarnos en añadir la interfaz inicial a modo de portada. Un par de textos nos servirán, uno con el título del juego y otro con una descripción que diga al jugador como jugar.

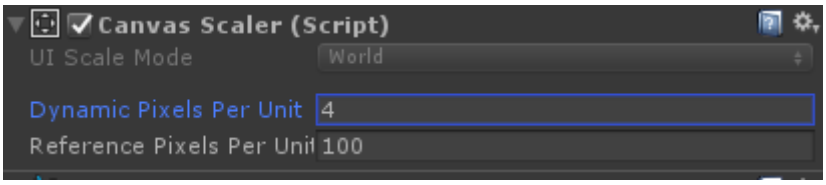
Como ambos textos corresponderan al estado Idle, el de inicio, podríamos crear un objeto padre llamado UI Idle y ponerlos dentro para manejarlos en conjunto:



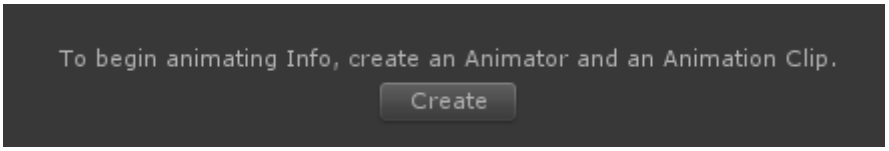
Recordad que tenemos que jugar con el espacio seguro, así que podemos hacer una disposición sencilla:



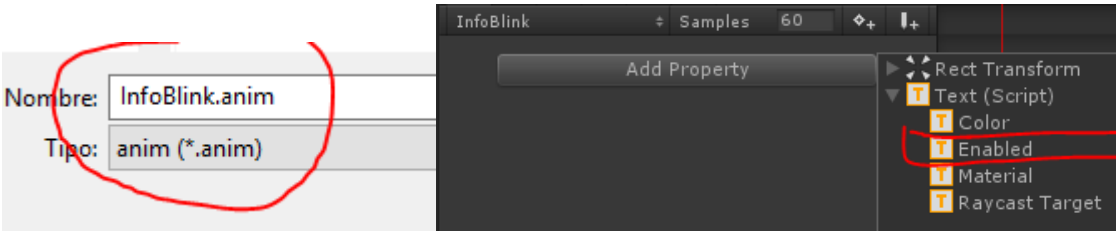
Muy bien, ahora fijaros en un detalle importante y es que si probamos el juego maximizado la calidad del texto es bastante mala, para mejorarlo podemos aumentar la cantidad de píxeles por unidad al escalar el canvas. Poniendo 4 lograremos que aumente la definición del texto:



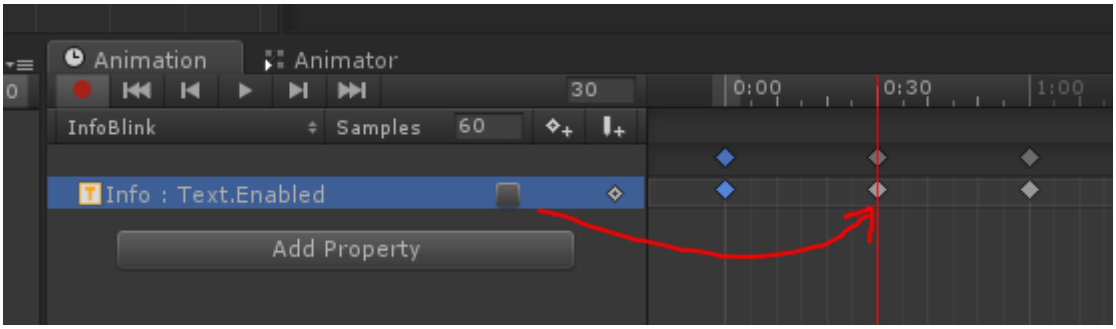
Ahora aprovecharía para añadir una animación al texto de info para que haga el efecto de parpadear. Esto es tan sumamente sencillo que sólo tenemos que activar y desactivar el render:



Podemos crear el directorio Animations para tenerlo todo ordenado:



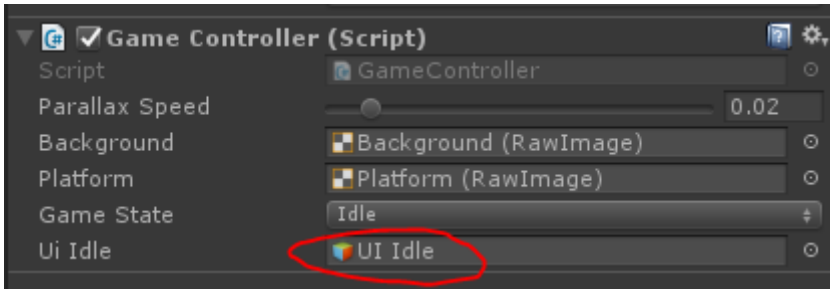
Añadimos un Frame Set desactivando el texto y listo:



Por último tenemos que desactivar esta UI al empezar el juego, es facilisimo enviando al objeto al GameController y desactivandolo con el método SetActive que tienen todos los objetos:

```
public GameObject uiIdle;
```

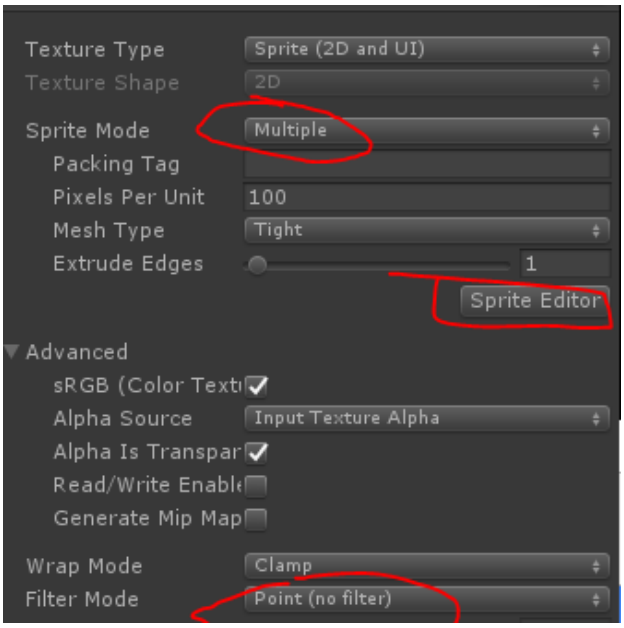
```
if (Input.GetKeyDown("up") || Input.GetMouseButtonDown(0)) {
    gameState = GameState.Playing;
    uiIdle.SetActive(false);
}
```



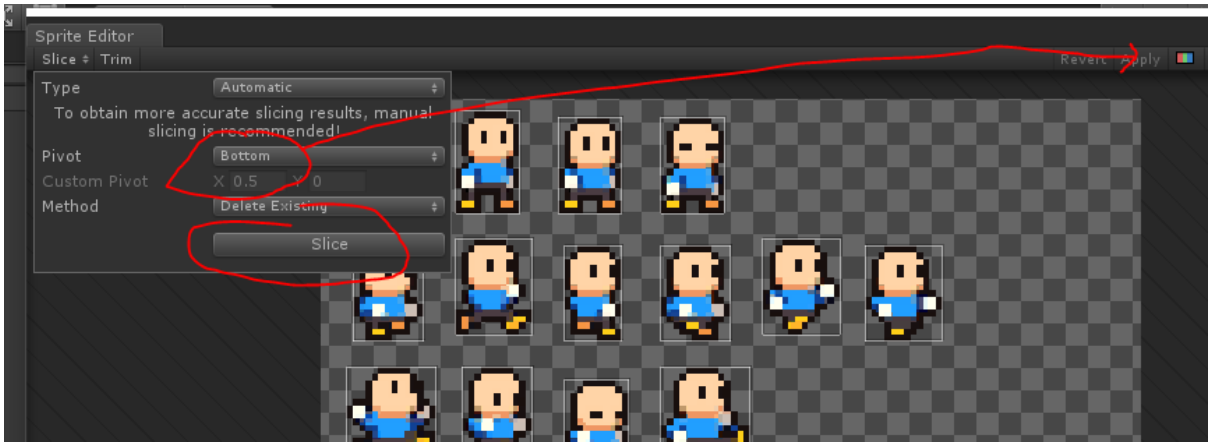
Listo!

Creando al protagonista

Muy bien, se supone que ha empezado el juego, ¿pero dónde está nuestro protagonista? Vamos a añadirlo como sprite 2D múltiple y lo troceamos:

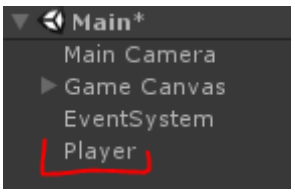


A la hora de trocearlo es muy importante poner el pivote abajo, ya que nuestro personaje tiene su punto de apoyo en los pies:



Bien, ahora esto es muy importante. Tanto nuestro protagonista como los enemigos y los demás objetos NO van a formar parte del canvas, van a ser EXTERNOS

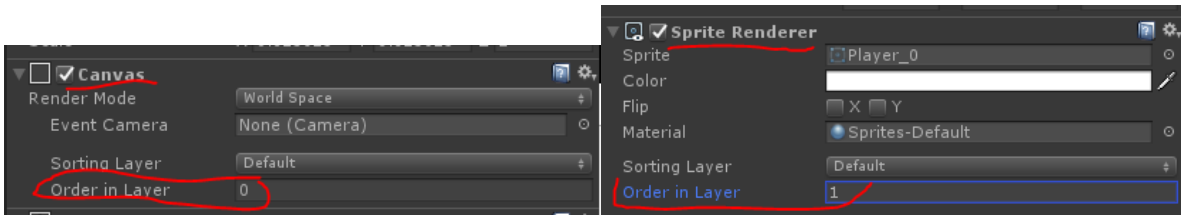
Así que vamos a crear un objeto llamado **Player**, fuera del canvas:



Ahora a este Player le vamos a crear un subobjeto llamado Body, podemos arrastrarlo directamente desde la primera imagen, sólo será de referencia:

Ahora fijaos en el detalle, nuestro personaje es bastante pequeño. Esto es debido a su configuración de píxeles por unidad. Sabéis que tenemos el mundo dividido en 10 unidades de alto, cada una de 36 píxeles (360/10). Por otro lado el personaje mide 30 píxeles de ancho, y tiene un PPU de 100. ¿Esto qué significa? Pues que se piensa que en cuadro hay 100 píxeles de ancho, pero no es cierto, el tamaño de cada unidad habíamos quedado que era 36 píxeles. Ese es el PPU correcto, 36:

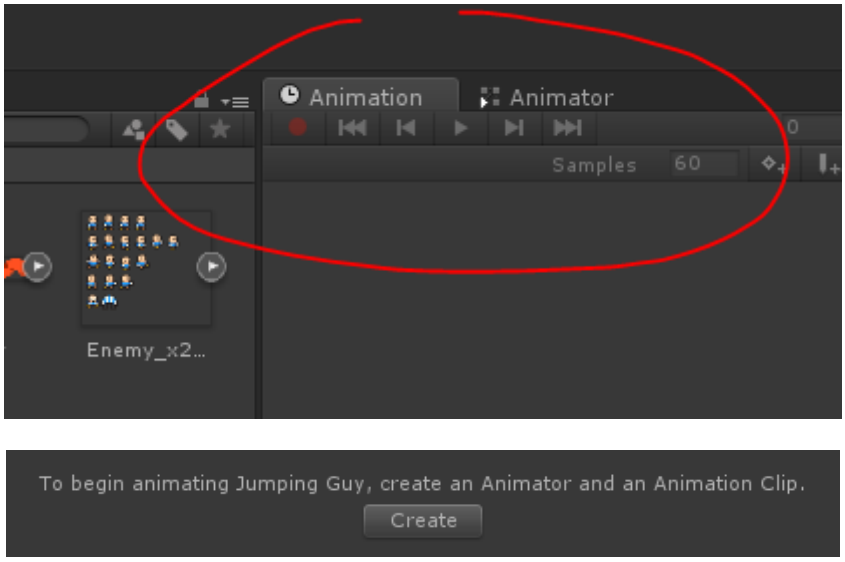
Muy bien. Antes de continuar algo muy importante es indicarle al Body que tiene más prioridad de dibuje que el canvas, así se dibujará por delante:



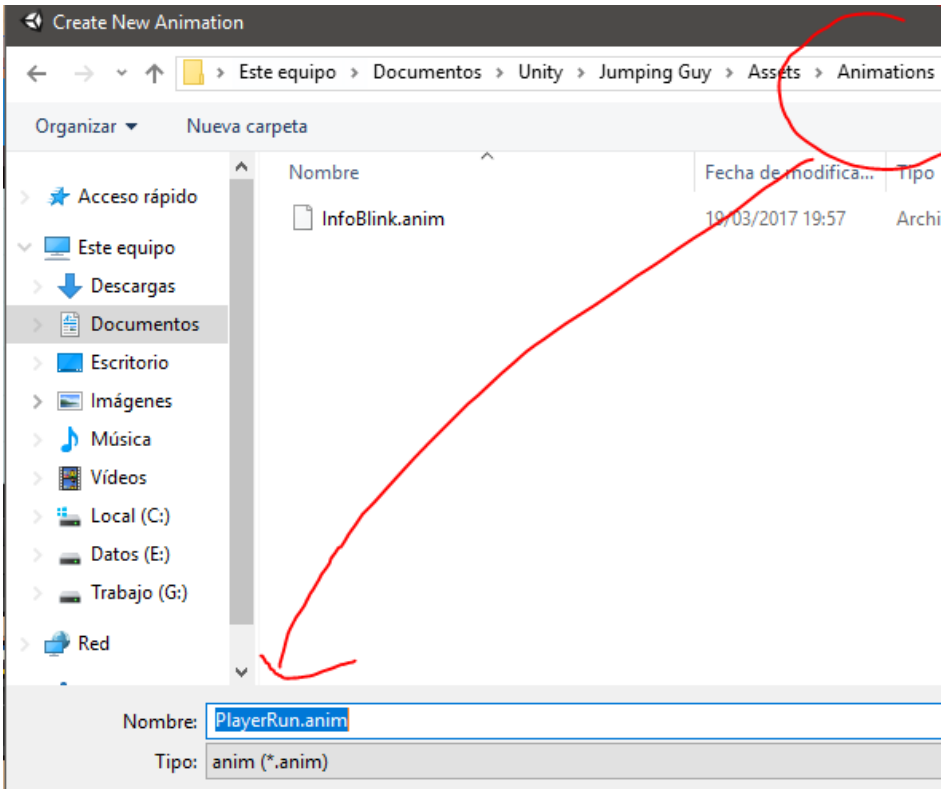
Todos los personajes y enemigos los dibujaremos por encima del canvas de fondo, debéis recordarlo.

Creando animación de correr

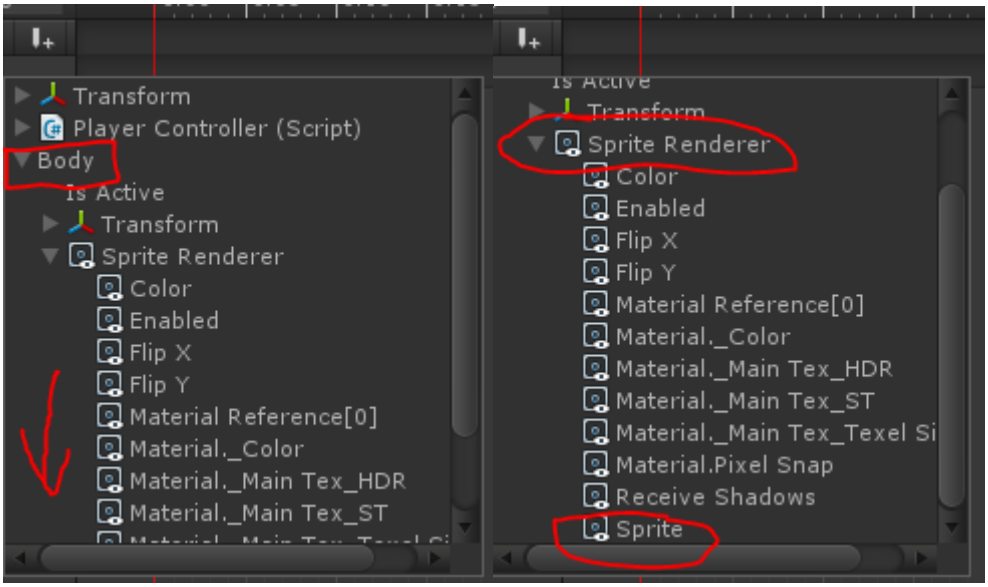
Bien, vamos a añadir animación a nuestro personaje para que parezca que corre hacia la derecha, aunque en realidad estará quieto en el mismo sitio (si es necesario configuramos las pestañas). Lo haremos sobre el objeto Player:



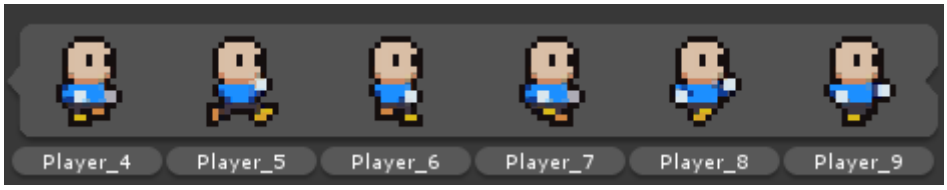
La crearé en el directorio Animations, PlayerRun:



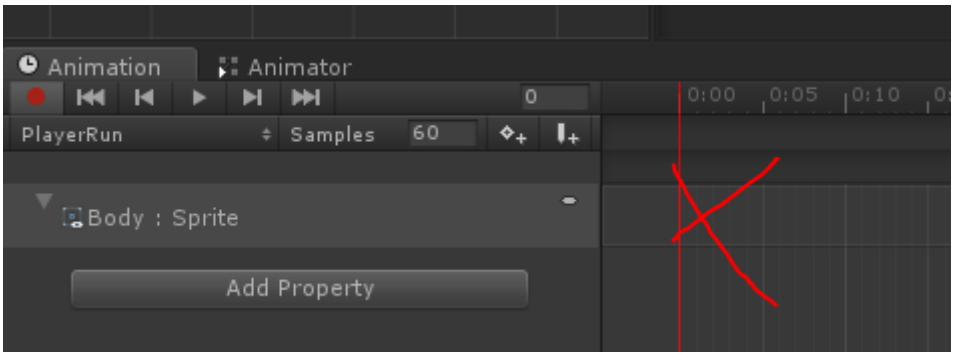
Ahora vamos a animar el Body de este Player, concretamente su sprite, así que tenemos que desplegar un poco el menú de propiedades, dentro de Sprite Renderer, Sprite:



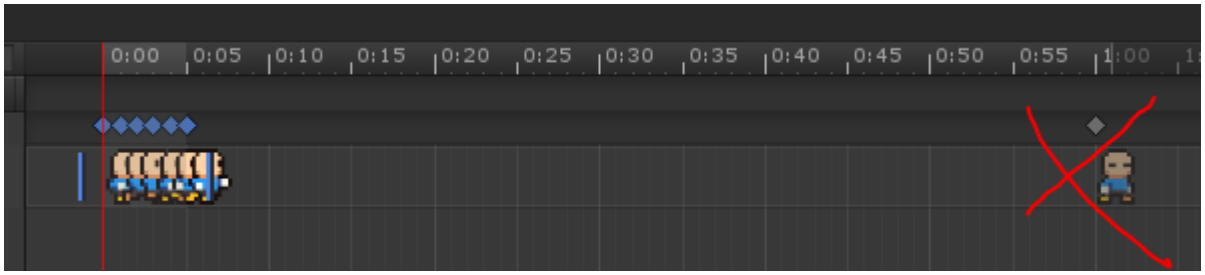
Ahora vamos a añadir la animación de correr, de la 4 a la 9 si no recuerdo mal:



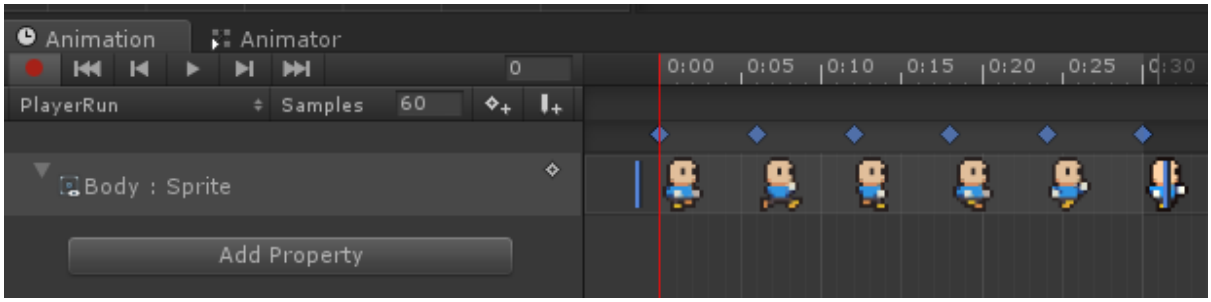
Para que sea más fácil vamos a borrar el primer key frame y arrastraremos las 6 imágenes:



Y borraremos la última:



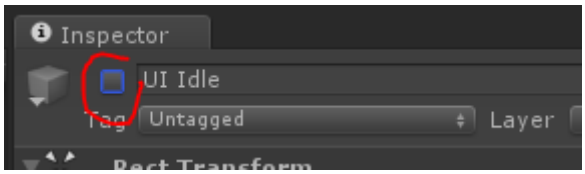
Las otras 6 haremos que duren 0.30 (medio segundo) de principio a fin:



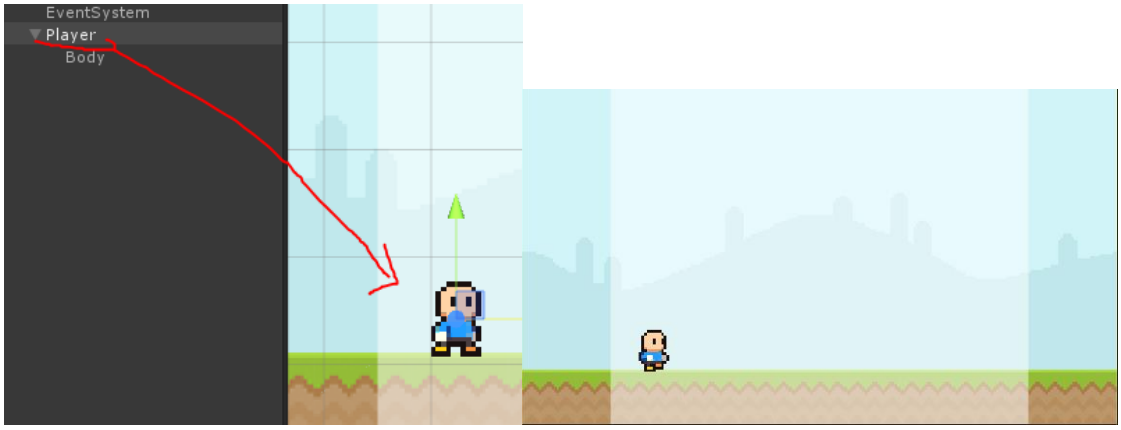
Esta es la forma de hacerlo más lógica y simple posible, animando desde el padre a los hijos.

Muy bien, a ver como queda:

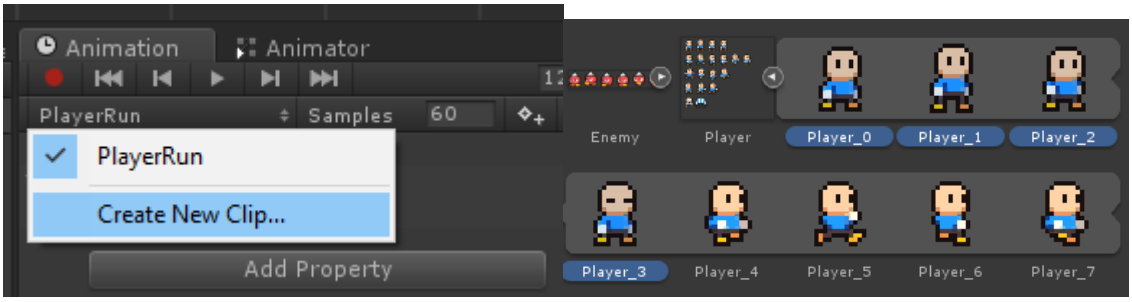
Vamos a desactivar manualmente la UI Idle para que no nos moleste por ahora:



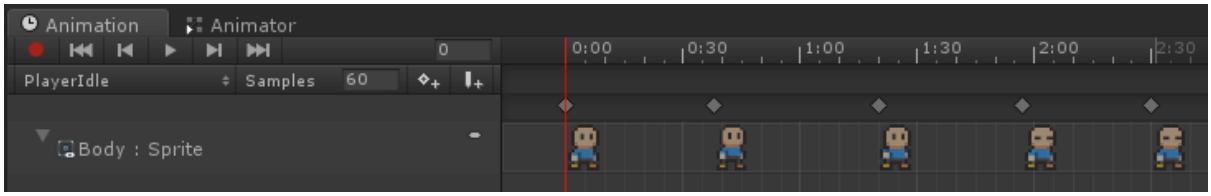
Ahora tenemos que decidir un lugar para nuestro personaje, siempre dentro de la zona segura. Según mi diseño debería estar más bien abajo a la izquierda, así tendremos tiempo de ver cómo vienen los enemigos. Lo ajustaremos utilizando el objeto **Player en X = -5** **Y = -3.5**:



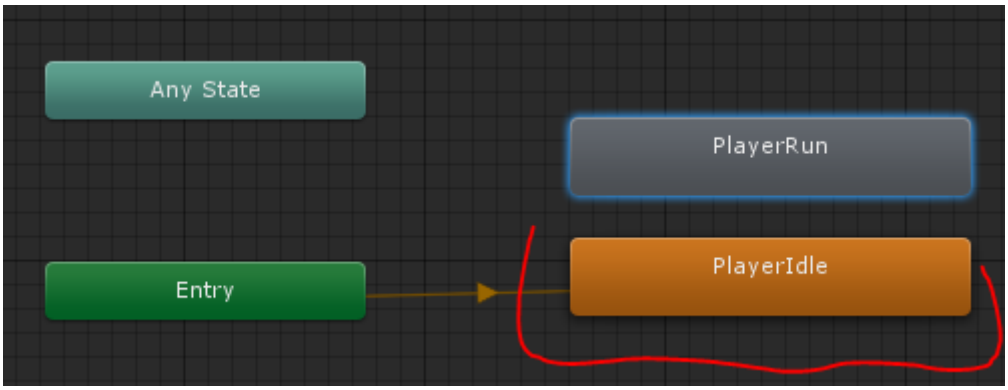
Bastante bien. Claro, nuestro personaje aparece ya corriendo desde el principio, podríamos afinar esto añadiendo una animación Idle inicial (**PlayerIdle**), recordad hacerlo en el Sprite Renderer del Body pero desde el Player:



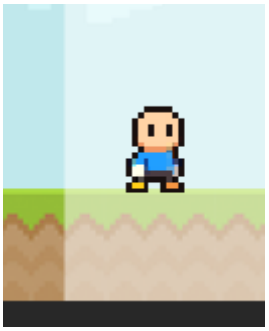
Que dure por ejemplo 2,5 segundos, alargando el parpadeo para que el efecto sea más interesante, pero podéis experimentar hasta encontrar una animación que os guste, faltaría más:



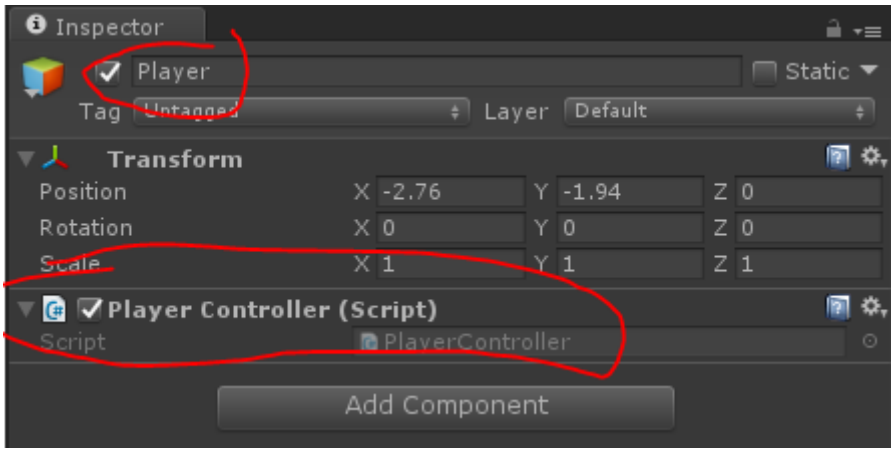
Y la hacemos default:



Vamos a probar el juego:



Ahora la cuestión es cambiar a la animación a correr al poner el juego en marcha. Esto lo podríamos hacer muy fácilmente a través del Script **GameController**, accediendo al animador de Body y cambiando el estado a PlayerRun, pero no me gusta esta idea, digamos que no es extensible para el futuro, es mejor tenerlo todo ordenado y que sea el propio Player el encargado de cambiar su estado a través de un método público, así que vamos a hacerlo de esta forma, creando un Script **PlayerController** que acceda al animador de Body.

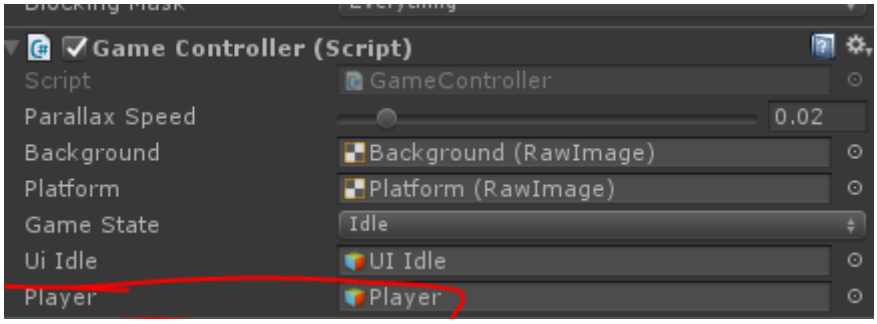



```
public class PlayerController : MonoBehaviour {

    private Animator animator;

    void Start () {
        animator = GetComponent<Animator>();
    }
                alternativa
    public void UpdateState(string state = null){
        if (state != null) {
            animator.Play(state);
        }
    }
}
```

Y ahora para llamarlo desde el GameController:

```
public GameObject player;
```

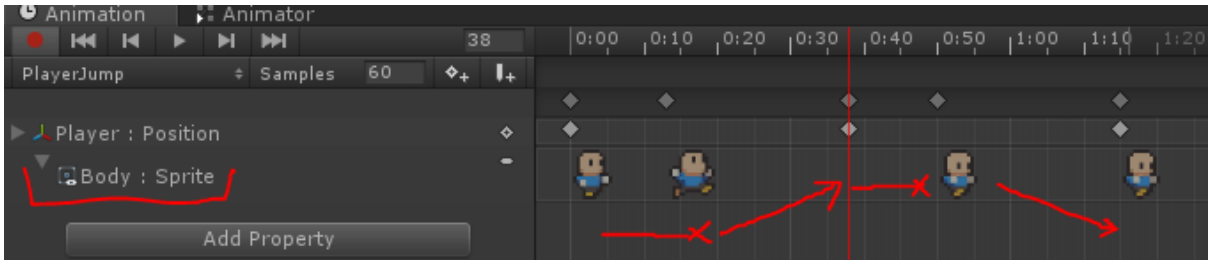
```
// Si el juego está parado
if (gameState == GameState.Idle) {
    if (Input.GetKeyDown("up") || Input.GetMouseButtonDown(0)) {
        gameState = GameState.Playing;
        uiIdle.SetActive(false);
        player.SendMessage("UpdateState", "PlayerRun");
    }
}
```

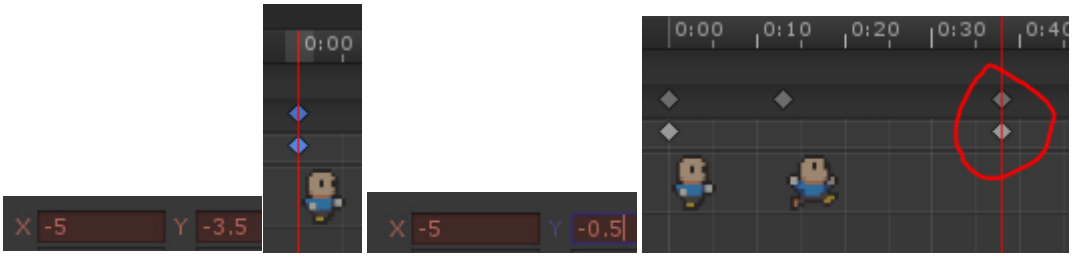


Ale, una cosa menos.

Creando animación de salto

Ahora vamos a crear una animación de salto para esquivar los enemigos, no por nada el juego se llama Jumping Guy. Por ahora haremos que dure **1.10** segundos, pero quizá luego tenemos que cambiarlo. Es muy importante mover la posición del Player, no del body a la hora de animarlo:

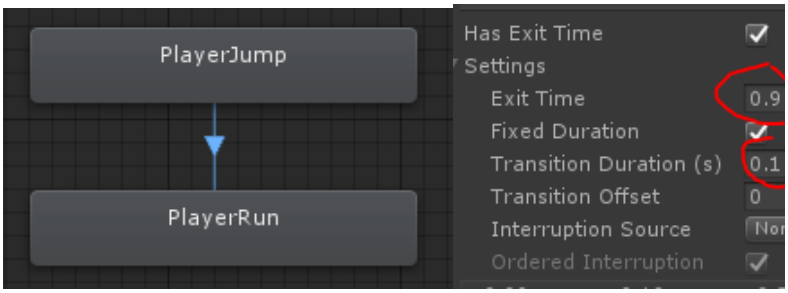




Una vez creada la animación vamos a configurar el salto al apretar la tecla arriba o al hacer clic en la pantalla. Esto lo detectaremos en el PlayerController:

```
void Update(){
    if (Input.GetKeyDown("up") || Input.GetMouseButtonDown(0)) {
        UpdateState("PlayerJump");
    }
}
```

Bien, ahora ya saltamos pero tenemos que crear una transición de vuelta a PlayerRun porque sino se queda parado, podemos ajustar la duración a nuestro gusto:

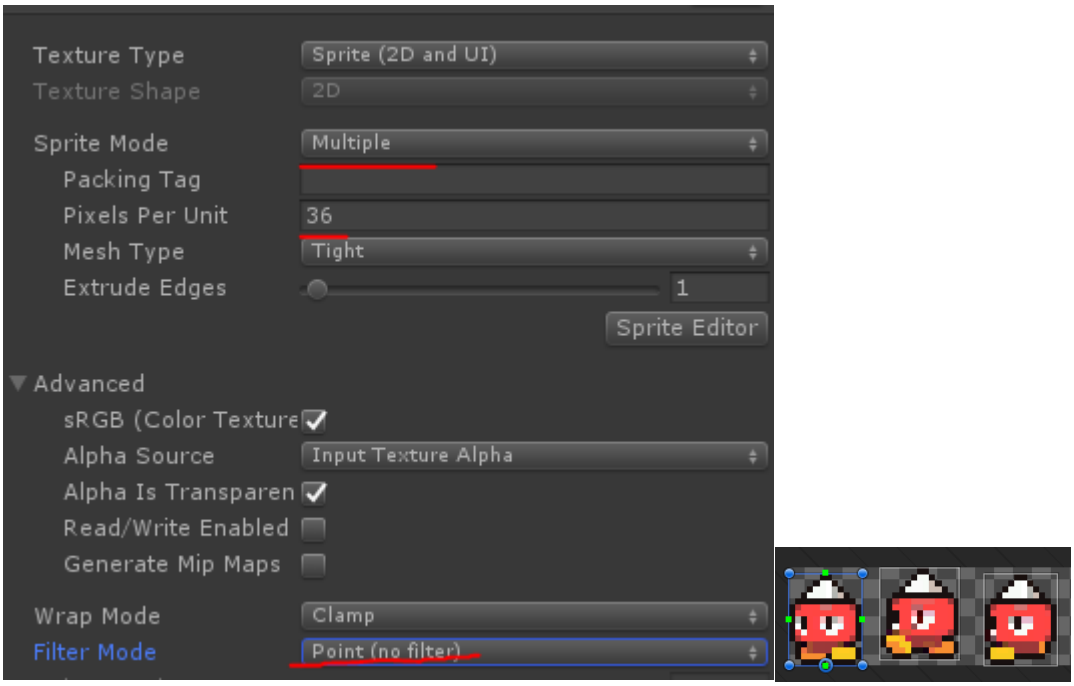


Perfecto, ya tenemos nuestro personaje saltando!

Creando al enemigo

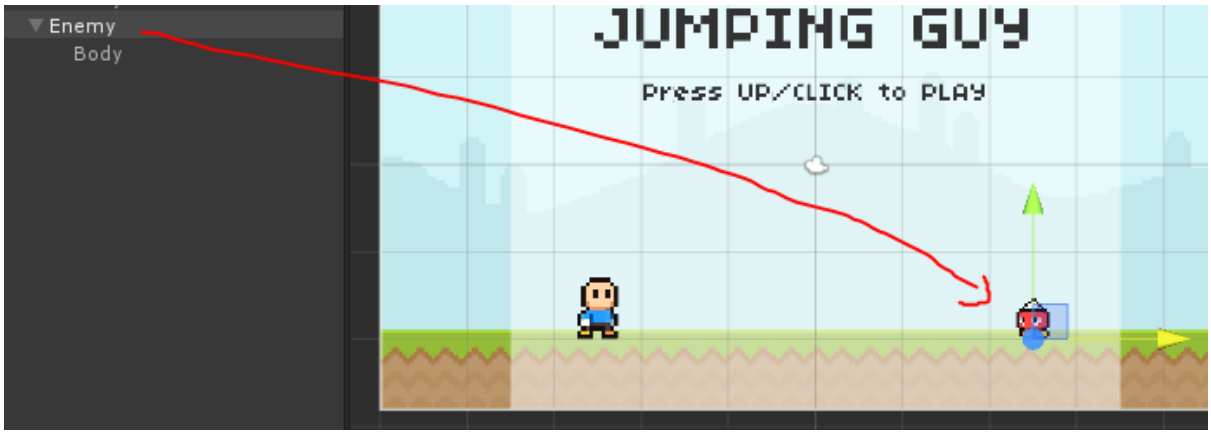
Si nuestro personaje se dedicara a saltar sin más no tendría mucho sentido el juego, así que vamos a añadir un divertido enemigo que aparezca por la derecha para saltarle por encima.

Igual que antes los PPU son 36, y trocearemos el sprite con el pivote abajo con el filtro en Point:

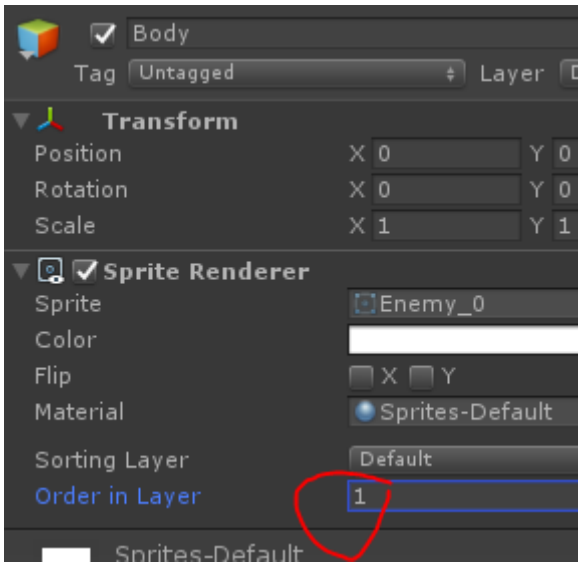


Para el objeto enemigo seguiremos la misma lógica que con el personaje. Creándolo fuera del canvas, con un objeto Enemy y dentro un body con el sprite:

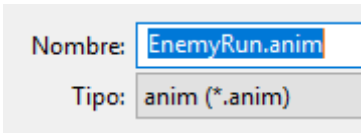
Para posicionar el enemigo en la escena siempre usaremos el objeto Enemy y tendremos el body centrado en 0,0 de forma relativa:



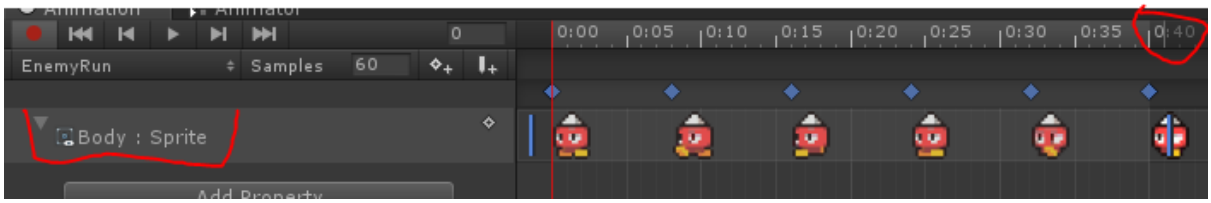
Recordad muy importante otorgarle la capa 1 a su body para que aparezca siempre por delante del canvas:



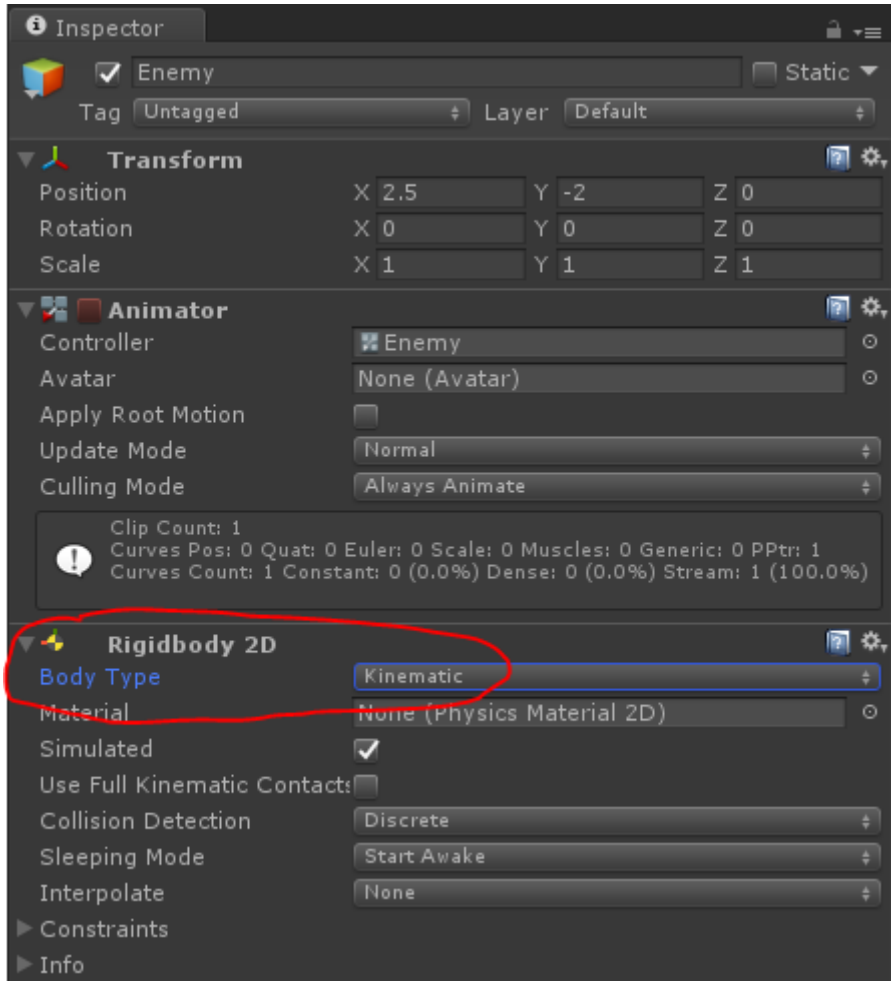
Ahora le añadiremos una animación simple de movimiento, recordad, la hacemos en Enemy pero animamos el Sprite Renderer del Body:



Con una duración de 0:40 (2 tercios de segundo) da bastante el pego:



Vale vamos a darle un movimiento horizontal. Para ello añadiremos un rigidbody cinemático (recordad que los cinemáticos son cuerpos a los que les afecta la velocidad pero no la gravedad ni la fuerza de las colisiones) con una velocidad de movimiento inicial:



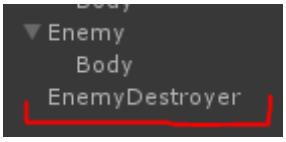
Vamos a otorgar una velocidad inicial a este enemigo usando un script EnemyController:

```
public float velocity = 2f;

private Rigidbody2D rb2d;

void Start () {
    rb2d = GetComponent<Rigidbody2D>();
    rb2d.velocity = Vector2.left * velocity;
}
```

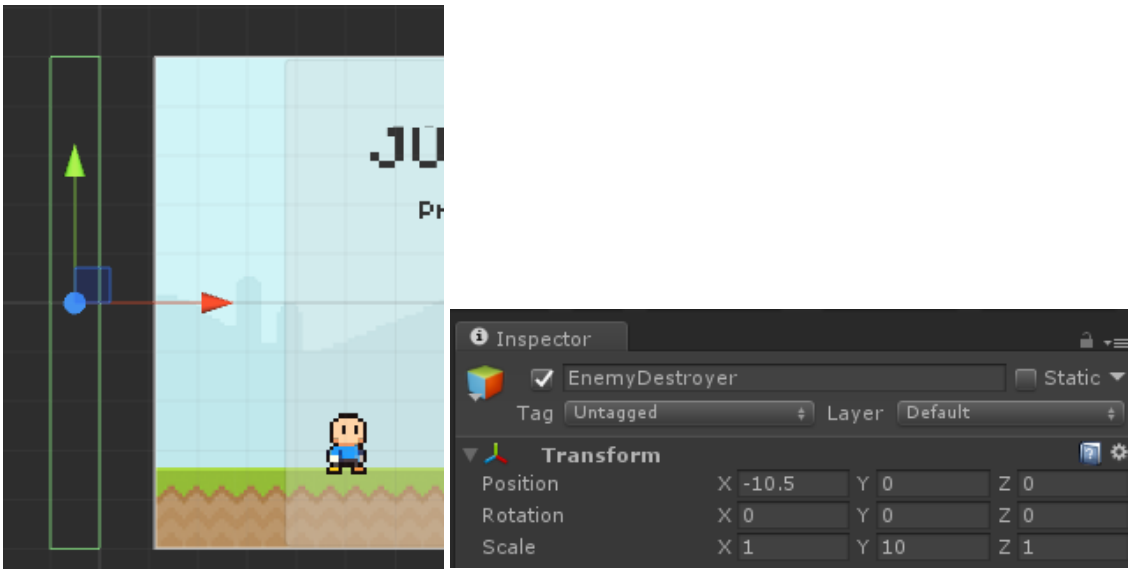
Bien, vamos a probar el juego… Ya se mueve y podemos saltarle por encima!

Autodestruir enemigos

Antes de continuar es esencial destruir los enemigos automáticamente una vez desaparecen para que no ocupen memoria innecesaria. La haremos creando una pared invisible, básicamente un objeto con un Box Collider 2D, que al chocar los enemigos contra él se destruyan. Le llamaré **EnemyDestroyer**

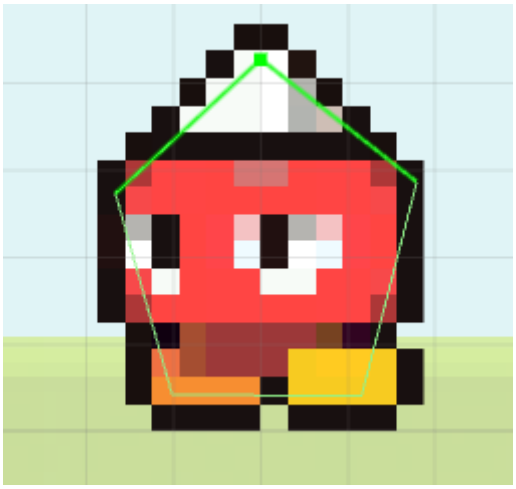


Le añadimos el collider:



Y hacemos que sea un Trigger:

Ahora añadimos otro collider en el enemigo, pero esta vez será poligonal y establecemos la punta correctamente editando los vértices (Edit Collider):

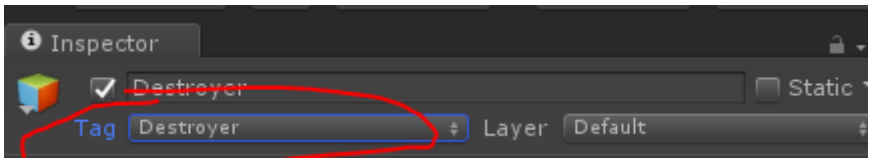


Este collider no es necesario que sea un trigger:

Ahora en el enemigo detectaremos la colisión contra un trigger y borramos al enemigo:

```
void OnTriggerEnter2D(){
    Destroy(gameObject);
}
```

Claro, con esto ya funciona pero es una implementación demasiado Genérica que se ejecutaría contra cualquier trigger. Lo idea es comprobar si al ocurrir este evento se trata de una colisión contra el Destructor, y sólo en ese caso borrar el objeto. No es muy difícil, podemos usar un TAG para identificar al destructor:



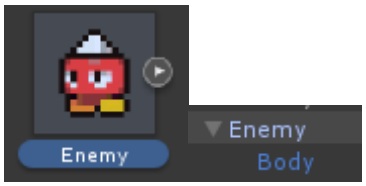
Ahora recibiremos al objeto contra el que ocurre la colisión y comprobar su TAG:

```
void OnTriggerEnter2D(Collider2D other) {
    if (other.gameObject.tag == "Destroyer") {
        Destroy(gameObject);
    }
}
```

Generador de enemigos

Muy bien, ya tenemos al personaje y al enemigo, pero para que el juego sea un reto tenemos que desarrollar un sistema capaz de generar enemigos de forma automática. Estos aparecerán en el lado derecho de la escena y la cruzaran hasta destruirse por el lado izquierdo.

Para hacerlo empezaremos transformando el enemigo en un prefab, así podremos crear nuevas instancias. Como siempre podemos crear una carpeta Prefabs para tenerlo todo ordenado:



El kit de la cuestión es crear un objeto EnemyGenerator encargado de crear instancias de este prefab cada cierto tiempo. Así que vamos a crear este objeto:

Vale, vamos a añadir un Script **EnemyGeneratorController**. Este generador tomará el Prefab del enemigo y un tiempo de generación con el que indicaremos cada cuantos segundos deberá crear una instancia:

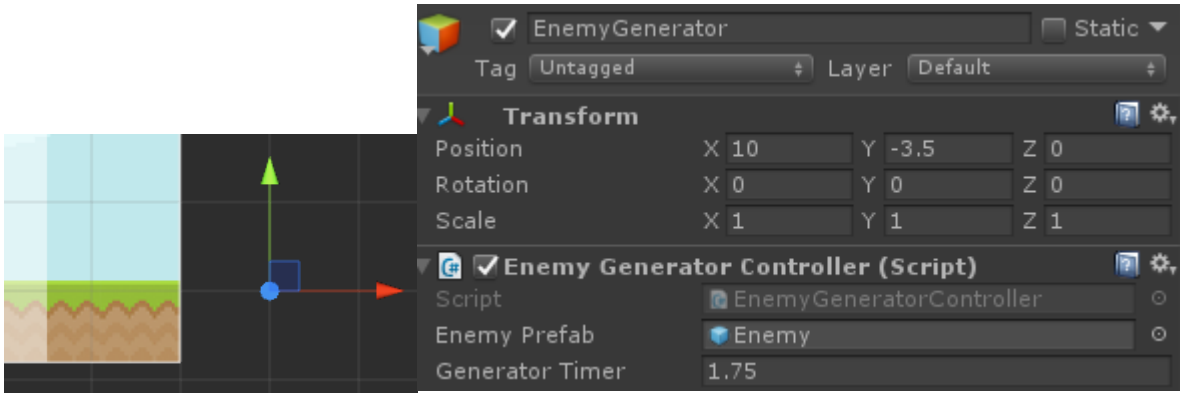
```
public GameObject enemyPrefab;
public float generatorTimer = 1.75f;
```

Lo más esencial de este Script es contar con un método para crear una instancia de un enemigo en su propia posición:

```
void CreateEnemy(){
    Instantiate(enemyPrefab, transform.position, Quaternion.identity);
}
```

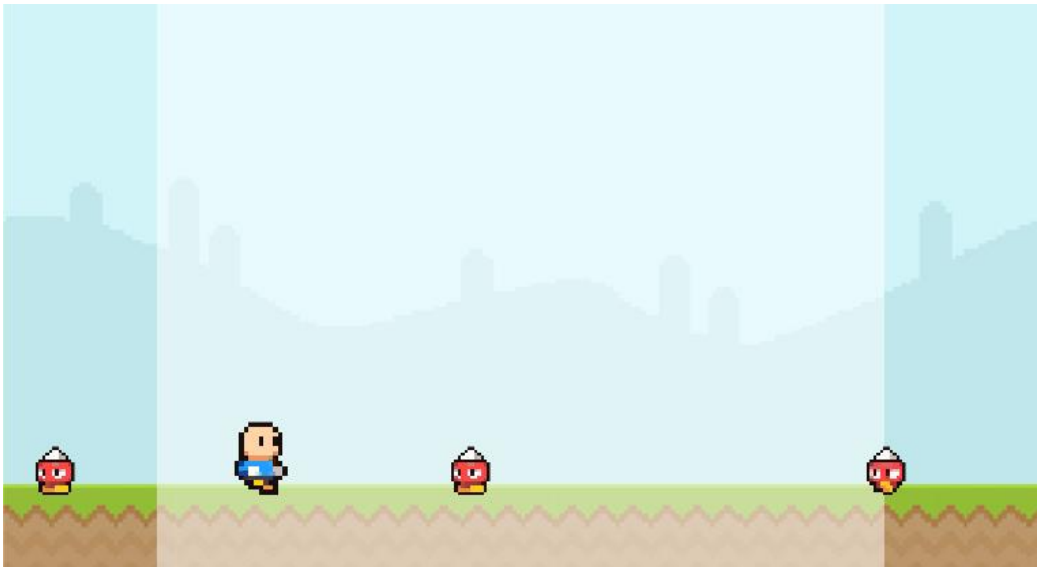
```
void Start(){
    CreateEnemy();
}
```

Vale, tenemos que posicionar correctamente el generador, fuera de la escena a la derecha:



Muy bien. ¿Ahora cómo hacemos para generar los enemigos automáticamente? Pues con una función que ya vimos anteriormente llamada InvokeRepeating:

```
void Start(){
    InvokeRepeating("CreateEnemy", 0f, generatorTimer);
}
```



Ya lo tenemos! Pero claro… Esto se pone en marcha desde el principio y queremos que inicie la generación al empezar el juego… Bueno, pues siempre podemos crear unos métodos públicos para controlar el generador desde el script **GameController**:

```
void CreateEnemy(){
    GameObject enemy = Instantiate(enemyPrefab, transform
}

public void StartGenerator(){
    InvokeRepeating("CreateEnemy", 0f, generatorTimer);
}

public void CancelGenerator(){
    CancelInvoke("CreateEnemy");
}
```

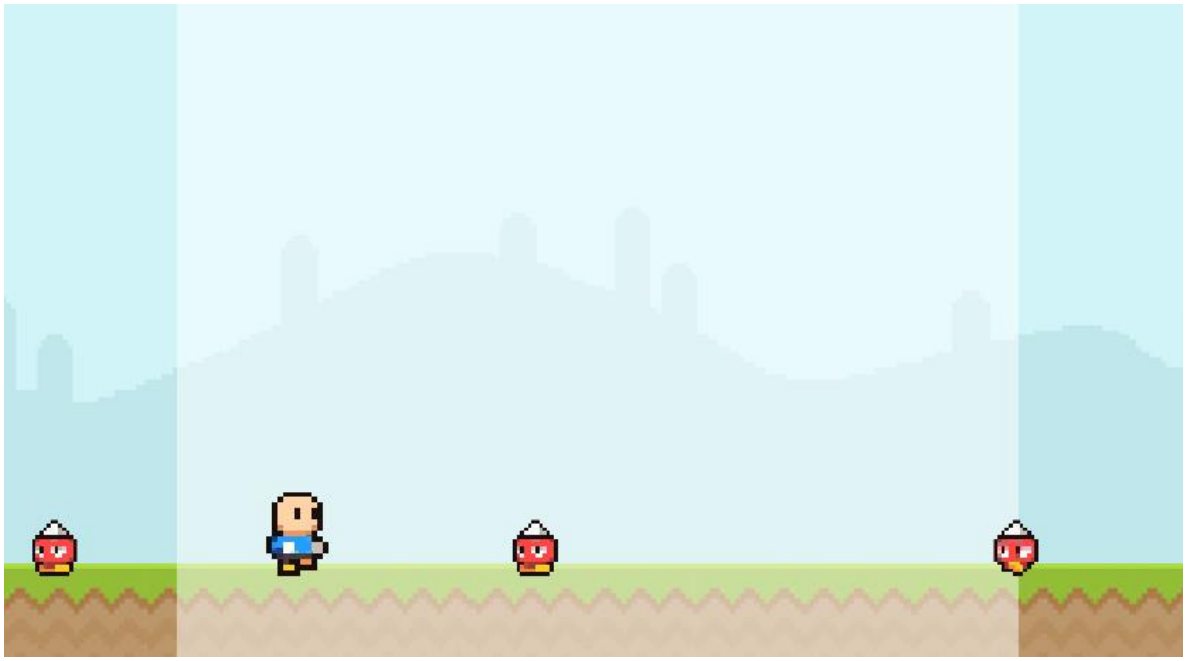
Ahora importamos el objeto EnemyGenerator desde **GameController** y llamar al método **StartGenerator** al empezar el juego:

```
public GameObject enemyGenerator;
```

```
if (Input.GetKeyDown("up") || Input.GetMouseButtonDown(0)) {
```

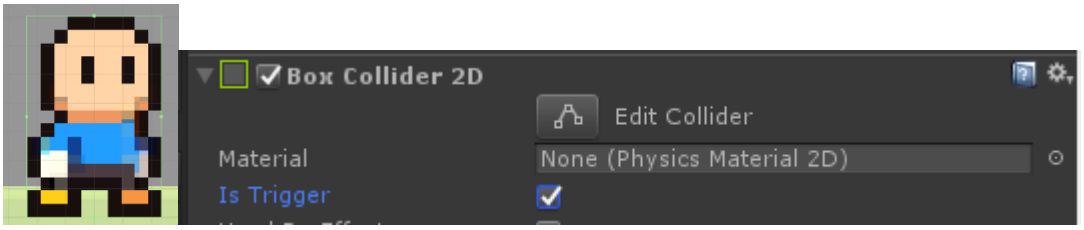
```
    gameState = GameState.Playing;
    uiIdle.SetActive(false);
    player.SendMessage("UpdateState", "PlayerRun");
    enemyGenerator.SendMessage("StartGenerator");
```

Vale, vamos a establecer el objeto y borrar la instancia del prefab Enemy antes de probar:

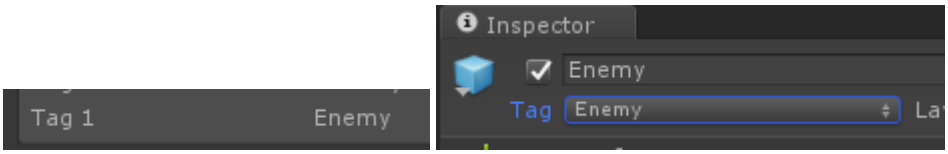


Creando animación de muerte

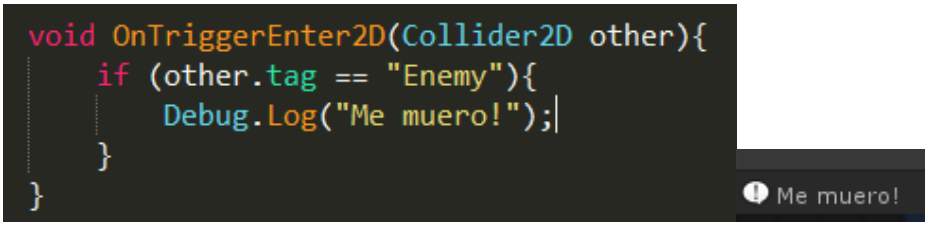
Lo primero que necesitamos es un collider 2D para detectar cuando nuestro personaje choca contra un enemigo. Lo crearemos en el Player y haremos que sea un trigger. Si véis que os cuesta definir el contorno podéis desactivar un momento el background:



Ahora añadiremos un Tag Enemy al enemigo, lo hacemos desde el prefab:

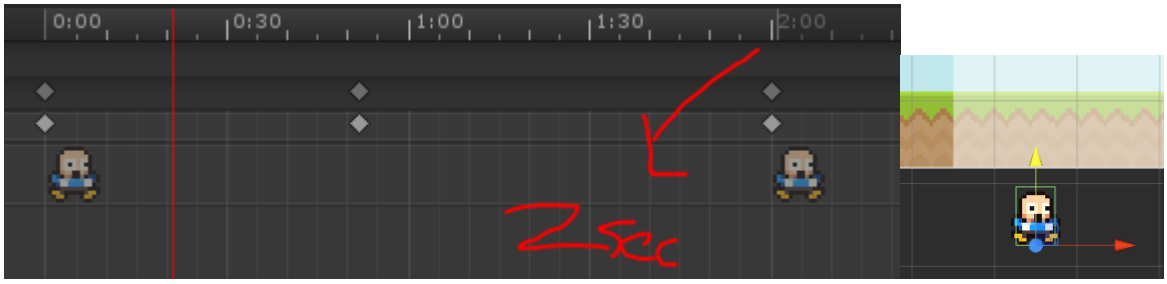


Ahora detectaremos la colisión entre el Personaje y el Enemigo con un trigger y haremos un debug:

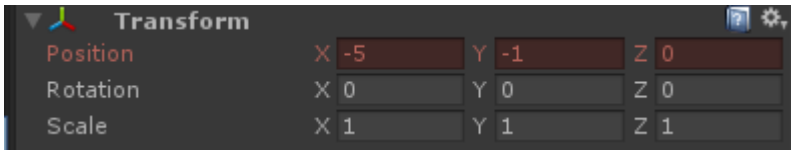


Vale, vamos a por la animación. Tenemos una imagen perfecta, podemos simular una animación como la de los primeros Super Mario, daba bastante el pego. Recordad que tenemos que crearla en Player y animador el Sprite Render del Body:

Podemos hacer que la imagen suba un poco y luego baje hasta desaparecer de la pantalla:

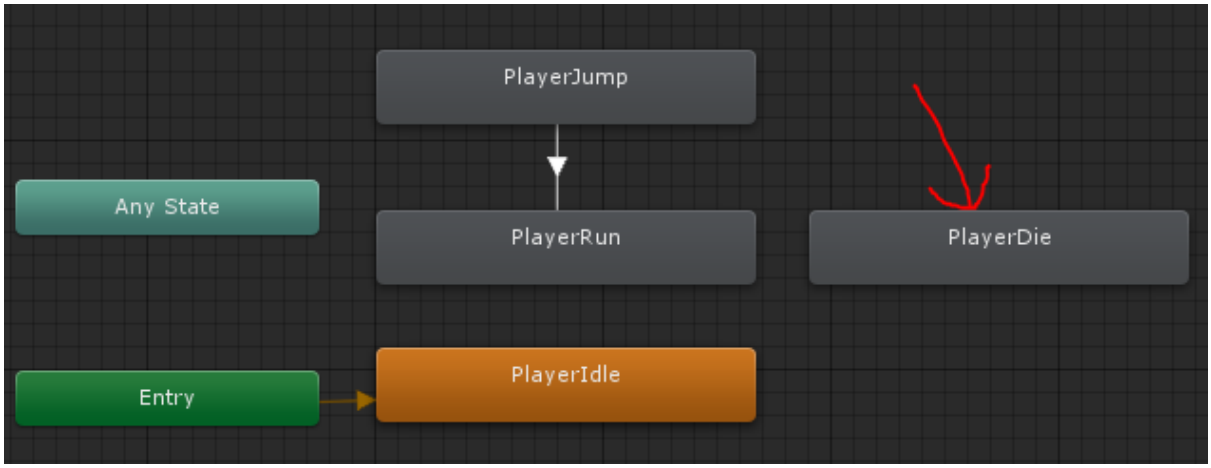


Posición superior



Para que no se repita la animación le desactivaremos el Loop:

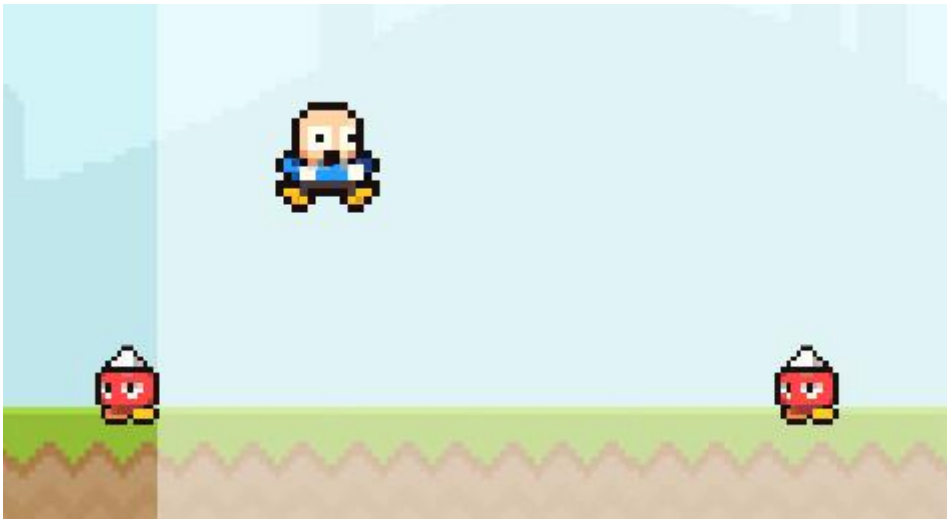
El caso entonces es que esta animación no está conectada con ninguna otra, simplemente haremos que se ponga por defecto cuando detectemos una colisión contra un enemigo:



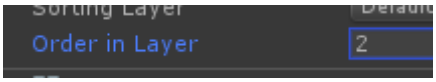
Para hacerlo utilizaremos nuestro método UpdateState:

```
void OnTriggerEnter2D(Collider2D other){
    if (other.tag == "Enemy"){
        UpdateState("PlayerDie");
    }
}
```

Vamos a probar como queda:



Muy bien! Fijaros que el personaje aparece por detrás de los enemigos, podemos cambiar orden sobre la capa para que aparezca por delante:



Básicamente ya lo tenemos pero tenemos que asegurarnos de no poder saltar de nuevo una vez hemos muerto. Mi recomendación es controlarlo con nuevo estado de juego llamado Ended (terminado), así que tendremos que importar el estado de juego dentro del PlayerController. Además para poder utilizar las opciones del enum tenemos que sacar la definición fuera de la clase GameController:

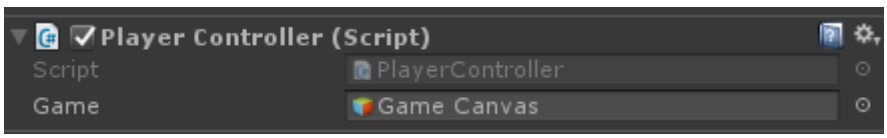
```
public enum GameState {Idle, Playing, Ended};

public class GameController : MonoBehaviour {
```

Ya de paso creamos la nueva condición en el if:

```
// Si el juego está terminado
else if (gameState == GameState.Ended) {
}
```

Y ahora nos ponemos con el código de PlayerController:

```
public GameObject game;
```



Y podemos acceder al estado de juego así como cambiarlo:

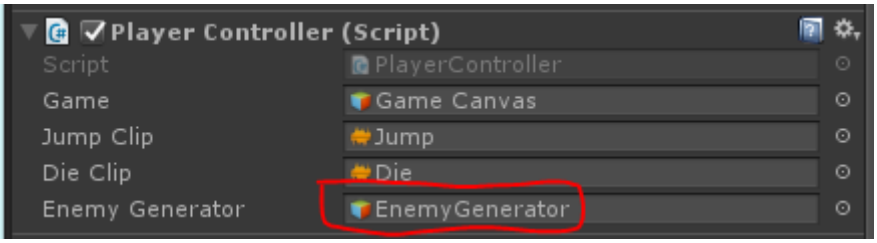
```
void OnTriggerEnter2D(Collider2D other){
    if (other.tag == "Enemy"){
        UpdateState("PlayerDie");
        game.GetComponent<GameController>().gameState = GameState.Ended;
    }
}
```

Lo único que nos falta es asegurarnos de que antes de saltar el estado sea Playing:

```
void Update () {
    bool gamePlaying = game.GetComponent<GameController>().gameState == GameState.Playing;
    if (gamePlaying && (Input.GetKeyDown("up") || Input.GetMouseButtonDown(0))){
        UpdateState("PlayerJump");
    }
}
```

Por último, y como paso opcional podemos parar el generador de enemigos:

```
public GameObject enemyGenerator;
```




```
void OnTriggerEnter2D(Collider2D other){
    if (other.tag == "Enemy"){
        UpdateState("PlayerDie");
        game.GetComponent<GameController>().gameState = GameState.Ended;
        enemyGenerator.SendMessage("CancelGenerator");
    }
}
```

Algo interesante es la posibilidad de eliminar o no los enemigos que quedan activos al parar el generador. Para ello podemos usar un parámetro de limpieza en el CancelGenerator y recorrer las instancias de objetos con el Tag Enemy, borrándolas:

```
public void CancelGenerator(bool clean = false){
    CancelInvoke("CreateEnemy");
    if (clean){
        Object[] allEnemies = GameObject.FindGameObjectsWithTag("Enemy");
        foreach(GameObject enemy in allEnemies) {
            Destroy(enemy);
        }
    }
}
```

```
enemyGenerator.SendMessage("CancelGenerator", true);
```

Reiniciando el juego

Una vez morimos también tenemos que permitir al jugador reiniciar el juego. Más adelante mostraremos un recuento y un récord de puntos, pero por ahora saldremos del paso temporalmente haciendo que al clicar la pantalla o apretando la flecha arriba se reinicie. Para reiniciar la escena necesitamos acceder al módulo SceneManagement y la clase SceneManager que tiene un método para Cargar escenas:

```
using UnityEngine.SceneManagement;
```

Podemos crear un método público RestartGame:

```
public void RestartGame(){
    SceneManager.LoadScene("Main");
}
```

Y comprobar si cuando el estado es Ended el usuario presiona una de las opciones:

```
// Si el juego está terminado
else if (gameState == GameState.Ended) {
    if (Input.GetKeyDown("up") || Input.GetMouseButtonDown(0)) {
        RestartGame();
    }
}
```

Con esto sería suficiente, pero como no me gusta ver código repetido sin sentido, os propongo refactorizar y detectar el momento de presionar los controles fuera del switch:

```
void Update () {

    bool userAction = Input.GetKeyDown("up") || Input.GetMouseButtonDown(0);
```

```
//Empieza el juego
if (gameState == GameState.Idle && userAction){
```

```
// Si el juego está terminado
else if (gameState == GameState.Ended) {
    if (userAction) {
```

Todo sea por las buenas prácticas!

EXTRA: Solucionar bug en Android

En algunas ocasiones los juegos pueden dar fallos o no funcionar de la forma esperada. En este caso vuelvo del futuro para explicaros un bug que ocurrirá en la versión Android si no hacemos nada para evitarlo, y es que cuando el jugador muere si mantiene el dedo sobre el juego se salta toda la escena de muerte del personaje y se reinicia de forma casi inmediata.

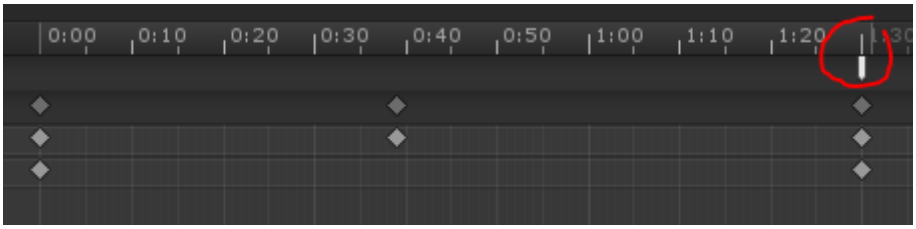
Para resolver esto podemos crear un nuevo estado de juego llamado Ready (preparado). Este estado indicará que el juego está preparado para reiniciarse, y lo estableceremos más o menos cuando finalice la animación de muerte del personaje con un evento de animación:

```
public enum GameState {Idle, Playing, Ended, Ready};
```

```
//Juego preparado para reiniciar
else if (gameState == GameState.Ready){
    if (userAction){
        RestartGame();
    }
}
```

```
void GameReady(){
    game.GetComponent<GameController>().gameState = GameState.Ready;
}
```

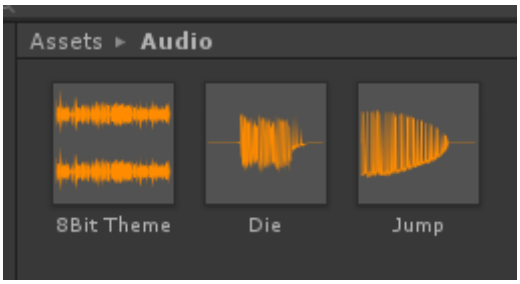
Creamos el evento en la animación de morir:



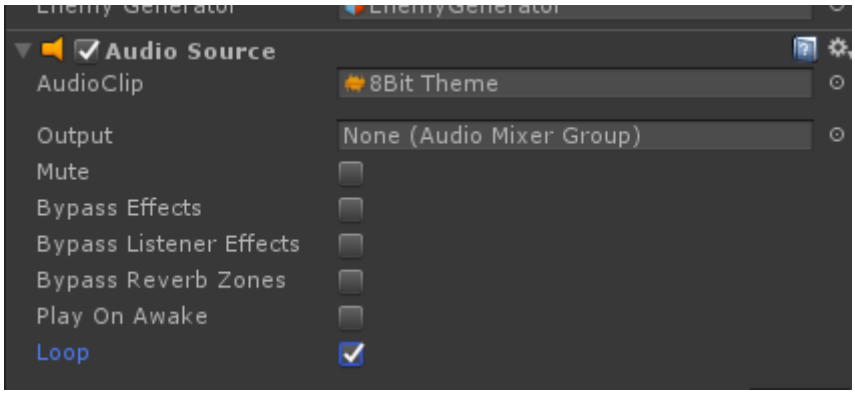
Y con esto ya habremos solucionado el bug.

Música y sonidos

Antes de continuar mejorando el juego vamos a añadir el audio. Tenemos 4 sonidos: un tema de fondo, uno de saltar, uno de morir y otro que utilizaremos luego al conseguir un punto, Point:



El tema lo arrastramos directamente a Game Canvas creando así un AudioSource, ponemos el Play on **Awake en false** y el Loop para que se vaya repitiendo:

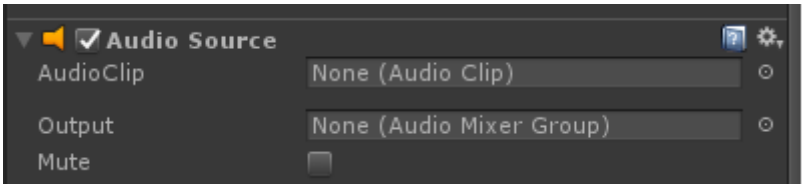


Para iniciar la música cuando se ponga el juego en marcha simplemente haremos play en el audiosource:

```
private AudioSource musicPlayer;

void Start(){
    musicPlayer = GetComponent<AudioSource>();
}
```

```
//Empieza el juego
if (gameState == GameState.Idle && userAction){
    gameState = GameState.Playing;
    uiIdle.SetActive(false);
    player.SendMessage("UpdateState", "PlayerRun");
    enemyGenerator.SendMessage("StartGenerator");
    musicPlayer.Play();
}
```

Bien. Ahora vamos a añadir un Audio Source a nuestro prota sin ningún sonido por defecto:

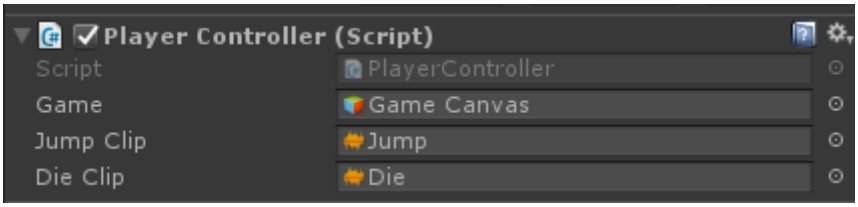


Ahora vamos al código de nuestro personaje y vamos a crear unas variables, el audioPlayer que es el audioSource lo buscaremos automáticamente en el Create:

```
private Animator animator;
private AudioSource audioPlayer;

public GameObject game;
public AudioClip jumpClip;
public AudioClip dieClip;
```

```
void Start () {
    animator = GetComponent<Animator>();
    audioPlayer = GetComponent<AudioSource>();
}
```

Para los dos clips arrastraremos los sonidos directamente:



Bien, ahora ya podemos reproducir los sonidos. Lo que haremos es por ejemplo al morir parar la música del juego (componente AudioSource del objeto game), y luego asignar el clip de morir y reproducirlo:

```
void OnTriggerEnter2D(Collider2D other){
    if (other.gameObject.tag == "Enemy"){
        UpdateState("PlayerDie");
        game.GetComponent<GameController>().gameState = GameState.Ended;
        enemyGenerator.SendMessage("CancelGenerator", true);

        game.GetComponent<AudioSource>().Stop();
        audioPlayer.clip = dieClip;
        audioPlayer.Play();
    }
}
```

Para el salto vamos a tener que jugar un poco, ya que tenemos que reproducirlo únicamente cuando nuestro el personaje está tocando el suelo. ¿Cómo lo hacemos? La forma fácil que se me ocurre es almacenar la posición Y inicial, la posición del personaje tocando suelo:

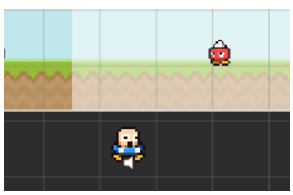
```
private float startY;

void Start () {
    animator = GetComponent<Animator>();
    audioPlayer = GetComponent<AudioSource>();
    startY = transform.position.y;
}
```

Ahora, justo en el salto comprobaremos si estamos tocando el suelo antes de reproducir la música, es decir, si la posición y actual es igual a la posición inicial cuando tocábamos el suelo:

```
void Update(){
    bool gameEnded = game.GetComponent<GameController>().gameState == GameState.Ended;
    bool userAction = Input.GetKeyDown("up") || Input.GetMouseButtonDown(0);
    bool isGrounded = transform.position.y == startY;

    if (!gameEnded && userAction && isGrounded) {
        UpdateState("PlayerJump");
        audioPlayer.clip = jumpClip;
        audioPlayer.Play();
    }
}
```

Si lo probamos tenemos un problema y es que en el momento de jugar, al apretar por primera vez un botón también suena el salto. Tendremos que modificar el código para que únicamente se reproduzca si el estado de juego es Playing:

```
void Update(){
    bool gamePlaying = game.GetComponent<GameController>().gameState == GameState.Playing;
    bool userAction = Input.GetKeyDown("up") || Input.GetMouseButtonDown(0);
    bool isGrounded = transform.position.y == startY;
    if (gamePlaying && userAction && isGrounded){
        UpdateState("PlayerJump");
        audioPlayer.clip = jumpClip;
        audioPlayer.Play();
    }
}
```



Si consideráis que los efectos se oyen demasiado fuerte podemos bajar un poco el volumen del AudioSource de Player:

Dificultad progresiva

El juego está más o menos listo, sólo nos falta que se incremente la dificultad poco a poco, que queda vez el ritmo de juego sea más rápido.

Esto podría ser todo un quebradero de cabeza, pero en Unity es imposible que sea más sencillo, ya que internamente hay una propiedad del juego capaz de manejar la velocidad el timing. De hecho esta técnica se utiliza para crear desde una simple opción de pausa, como efectos slow-motion.

Vamos a crear simplemente un método en el GameController que se encargará de aumentar el ritmo en X cantidad cada N segundos.

```
public float scaleTime = 6f;
public float scaleInc = .25f;
```

6 y 0.25 no son números aleatorios, sino que después de varias pruebas este es el ritmo que más me ha gustado, que vendría a ser, cada 6 segundos hacer que todo vaya un 25% más rápido:

```
void GameTimeScale(){
    Time.timeScale = Time.timeScale + scaleInc;
    Debug.Log("Ritmo Incrementando: " + Time.timeScale.ToString());
}
```

Para incrementar el ritmo lo haremos con un InvokeRepeating y una variable pública para controlarlo, por defecto yo le pondré 6 segundos, pero vosotros podéis experimentar:

```
// Si el juego está parado
if (gameState == GameState.Idle) {
    if (userAction) {
        gameState = GameState.Playing;
        uiIdle.SetActive(false);
        uiScore.SetActive(true);
        player.SendMessage("UpdateState", "PlayerRun");
        enemyGenerator.SendMessage("StartGenerator");
        musicPlayer.Play();
        InvokeRepeating("GameTimeScale", scaleTime, scaleTime);
    }
}
```

Ahora tenemos que cancelar la invocación cuando muramos:

```
public void ResetTimeScale(float newtimeScale = 1f){
    CancelInvoke("GameTimeScale");
    Time.timeScale = newtimeScale;
    Debug.Log("Ritmo Reestablecido: " + Time.timeScale.ToString());
}
```

Y lo llamamos al morir:

```
enemyGenerator.SendMessage("CancelGenerator", true);
game.SendMessage("ResetTimeScale");
```

Lo divertido de este método es que podemos enviar un factor de tiempo optativo al reiniciarlo, y si por ejemplo mandamos 0.5f, la animación de muerte se verá el doble de lenta:

```
game.SendMessage("ResetTimeScale", .5f);
```

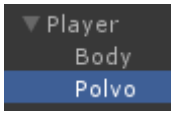
La pega es que al reiniciar la escena tendremos que reiniciar el ritmo a 1 sí o sí:

```
// Si el juego está terminado
else if (gameState == GameState.Ended) {
    if (userAction) {
        ResetTimeScale();
        RestartGame();
    }
}
```

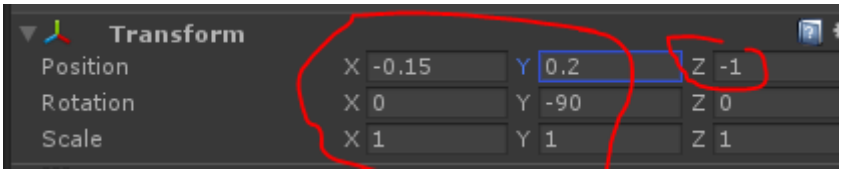
Y así ya lo tenemos listo.

Partículas de polvo al correr

Crear sistema dentro del player:



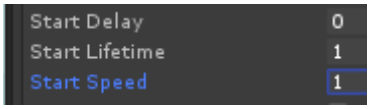
Posicionamiento



Position Z= -1 DELANTE SPRITE

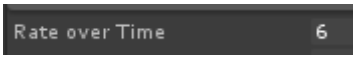
Rotation Y=-90 ENFOCAR HACIA ATRÁS

General

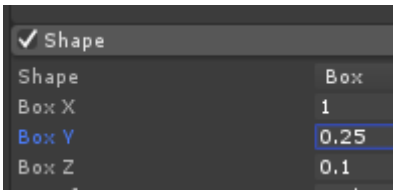


Color marrón de la plataforma

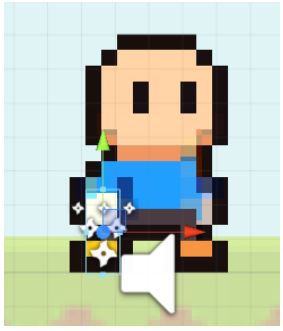
Emission



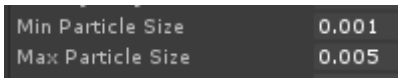
Shape



Lo hacemos pequeñito que expulse el polvo hacia la izquierda:



Renderer



Color Over Time

Módulo extra para que desaparezcan las micro-partículas, color transparente al final:

Bien, ya tenemos más o menos el sistema de partículas configurado, ahora es cuestión de activarlo cuando empieza el juego y mostrarlo sólo cuando estamos tocando el suelo, desde el mismo Player:

```
public ParticleSystem dust;
```

Desactivar de inicio

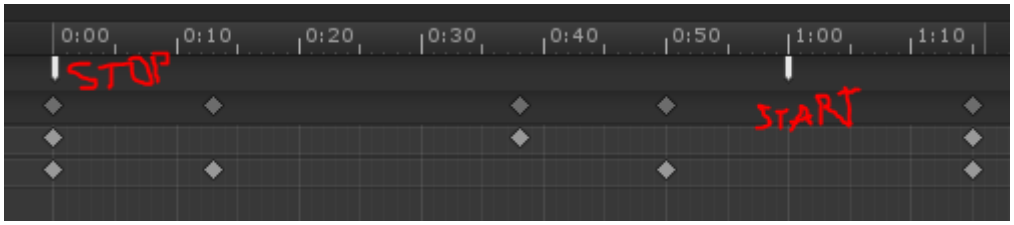
Ahora creamos dos métodos:

```
public void DustPlay(){
    dust.Play();
}

public void DustStop(){
    dust.Stop();
}
```

Primeramente iniciaremos el polvo al comenzar a jugar, desde el **GameController**:

```
InvokeRepeating("GameTimeScale", scaleTime, scaleTime);
player.SendMessage("DustPlay");
```

Y con dos eventos de animación desactivamos y activamos el polvo a voluntad ya en la animación de salto:



No olvidemos también desactivarlo manualmente al morir:

```
game.SendMessage("ResetTimeScale", .5f);
DustStop();
```

Y con esto ya tenemos nuestro efecto de polvo, aunque es sutil le da un buen toque.

Marcador de puntos

Vamos a crear un marcador que detecte cuantos enemigos hemos saltado y lo muestre por pantalla.

Lo más fácil para nosotros es duplicar el UI Idle y llamarle UI Score. Ahora desactivamos el UI Idle por ahora y del UI Score renombramos Title a Points y ponemos un 0. El info lo desactivamos también por ahora:

Ahora nos vamos a ir al GameController y vamos a crear una variable llamada points:

```
private int points = 0;
```

Y vamos a crear un método público llamado IncreasePoints():

```
public void IncreasePoints(){
    points++;
}
```

Ahora crearemos otra variable pública llamada pointsText y le arrastraremos el Texto de UI:

```
public Text pointsText;
```

Y en el propio método increasePoints le estableceremos el texto:

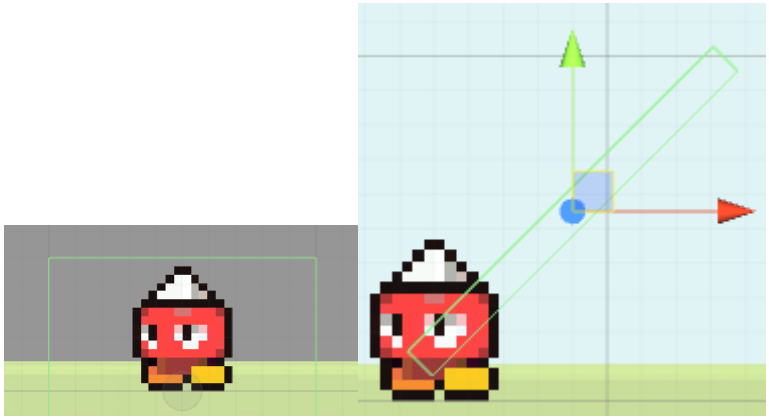
```
pointsText.text = points.ToString();
```

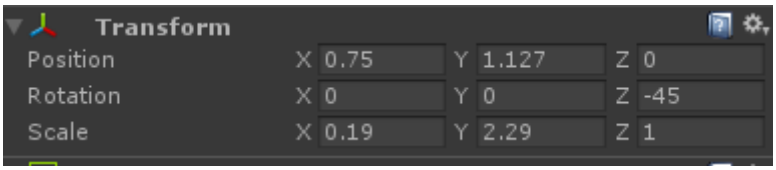
De hecho podemos simplificar ambas líneas en una:

```
public void IncreasePoints(){
    pointsText.text = (++points).ToString();
}
```

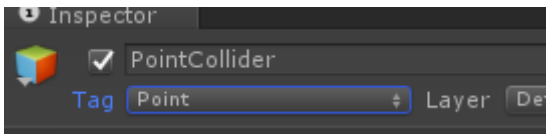
Vale ya tenemos el terreno preparado, ahora tenemos que detectar de alguna forma cuando el personaje gana un punto. ¿Cómo podemos hacerlo? Muy fácil, creando un subobjeto con un collider en el enemigo que se moverá junto a él, y cuando concuerden el jugador y ese collider incrementaremos el marcador.

Lo primero que haré pues es crear un subobjeto en el enemigo llamado **Point Collider**. Como no tenemos ninguna vamos a crear una instancia de prefab. Voy a añadirle un Box Collider 2D, y como véis es un poco difícil distinguir el borde, así que temporalmente desactivaré el background y pondré el Point Collider formando una línea muy delgada sobre el enemigo pero inclinada 45 grados atrás, eso para evitar que al morir no nos cuente un punto:

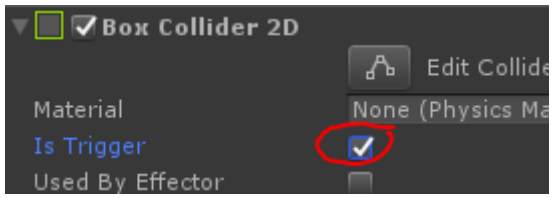




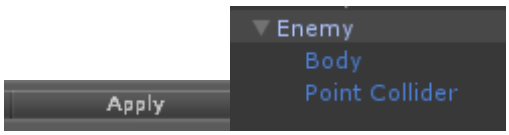
Vale, ahora vamos a añadir un TAG al Points Collider, sí, le llamaré PointCollider:



Y no olvidemos marcar la casilla Trigger:



Aplicamos los cambios al Prefab:



Y ahora ya podemos desarrollar la colisión desde el PlayerController y también reproducir el sonido de Punto:

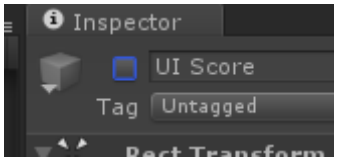
```
public AudioClip pointClip;
```


```
else if (other.tag == "PointCollider"){
    game.SendMessage("IncreasePoints");
    audioPlayer.clip = pointClip;
    audioPlayer.Play();
}
```

Y creo que todo debería funcionar. Aplicamos los cambios al Prefab, borramos y vamos a probar…

Todo funciona bien! Sólo tenemos que desactivar de inicio el objeto UI Score, y justo cuando desactivamos la UI Idle activar esta:

```
public GameObject uiScore;
```




```
uiIdle.SetActive(false);
uiScore.SetActive(true);
```

Vamos a ver… Perfecto!

Guardar récord con PlayerPrefs

Estamos a puntito de acabar el juego, sólo queda un detalle que lo hará un poco más adictivo, y ese es mostrar el récord de puntos. Pero claro, para hacer eso tenemos que guardar datos en un fichero para que sigan ahí la próxima vez que pongamos el juego en marcha… Bueno, pues podemos utilizar el módulo PlayerPrefs para guardar datos del jugador.

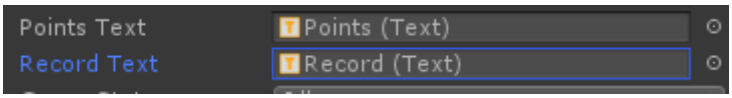
Sólo tenemos que ir al GameController y desarrollar dos métodos públicos, uno para consultar el récord máximo y otro para guardar el record actual:

```
public int GetMaxScore(){
    return PlayerPrefs.GetInt("Max Points", 0);
}

public void SaveScore(int currentPoints){
    PlayerPrefs.SetInt("Max Points", currentPoints);
}
```

Esto lo gestionará todo por nosotros. ¿Donde los ejecutamos? Bueno, primero necesitamos dibujar el récord en algún lugar, y para eso tenemos ese objeto Info en UI score, voy a renombrarlo a Record:

Tendríamos que eliminarle el animador que ha clonado del texto Info. Ahora lo vamos a importar:

```
public Text recordText;
```



Y vamos a establecer el valor desde el propio Start:

```
void Start(){
    musicPlayer = GetComponent<AudioSource>();
    maxPointsText.text = "BEST: " + GetMaxScore().ToString();
}
```

Con esto ya funcionará se mostrará la mejor puntuación, pero no es perfecto, ya que el récord no se actualizará a medida que se vaya superando. Así que compararemos los puntos en el momento de

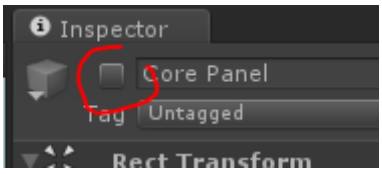
incrementar el marcador, y si el marcador es mayor que el récord actual, entonces lo actualizamos en la UI y lo guardamos en el fichero:

```
public void IncreasePoints(){
    pointsText.text = (++points).ToString();
    if (points >= GetMaxScore()){
        maxPointsText.text = "BEST: " + points.ToString();
        SaveScore(points);
    }
}
```

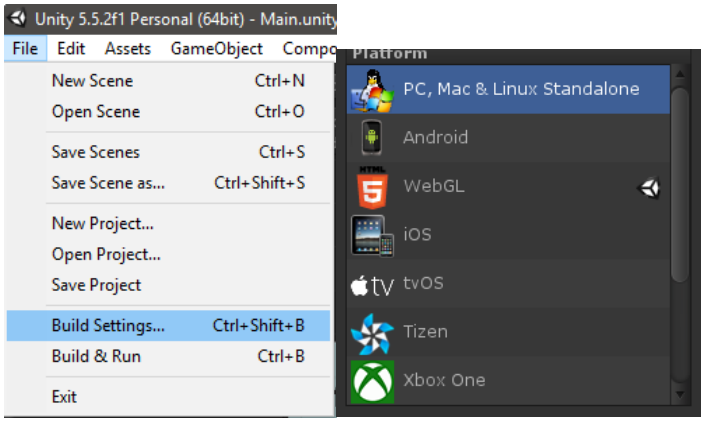
Y con esto hemos acabado el desarrollo de nuestro videojuego! Ya sólo queda exportar en PC, WebGL y Android.

Exportación multiplataforma

Genial! Pues nada, ya sólo falta generar los ejecutables, pero antes no podemos olvidarnos de lo más importante… **Desactivar el panel que nos ha estado ayudando durante todo el desarrollo!**



Y ahora sí, a generar los ejecutables! Vamos a abrir la ventana de generación, allí podremos seleccionar la plataforma y la configuración base:

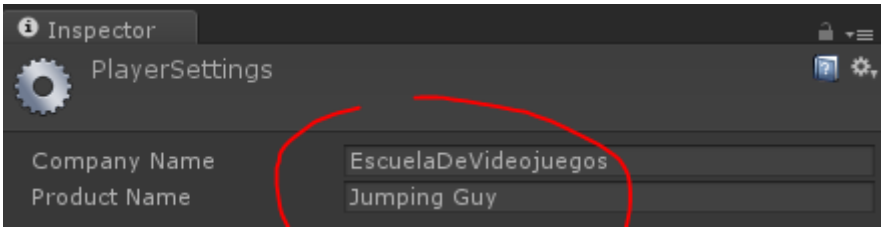


Para acceder a las configuraciones avanzadas tenemos el botón:

Desde ahí lo configuraremos todo.

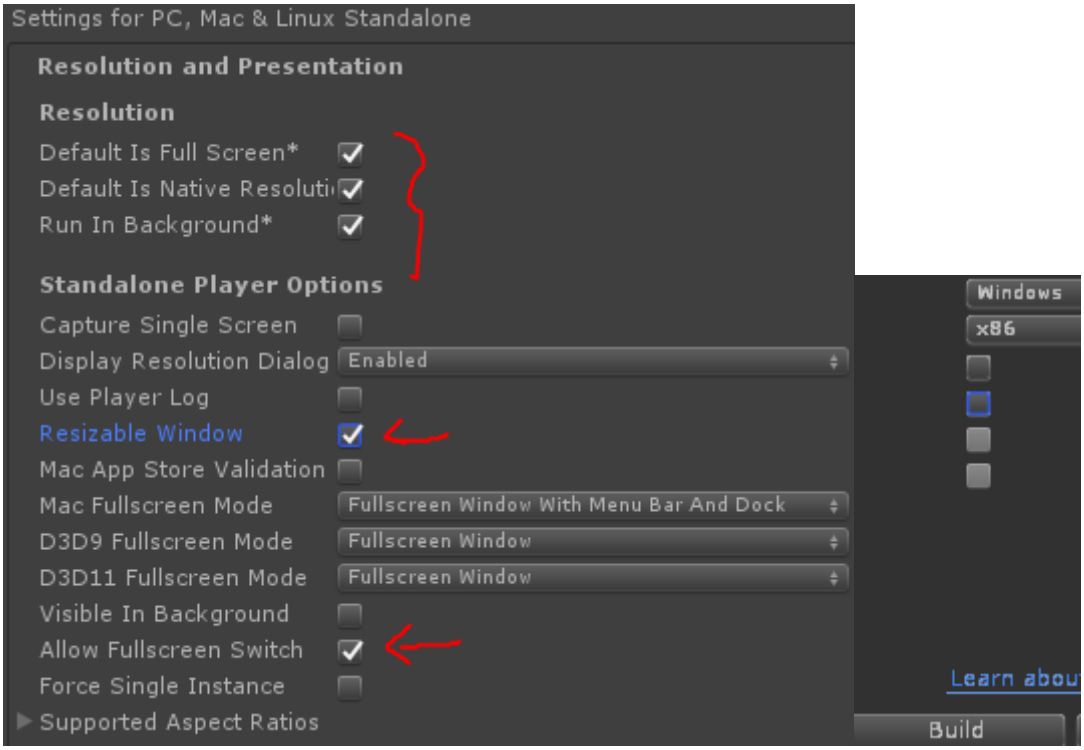
Windows

Nombre de la empresa y del producto:



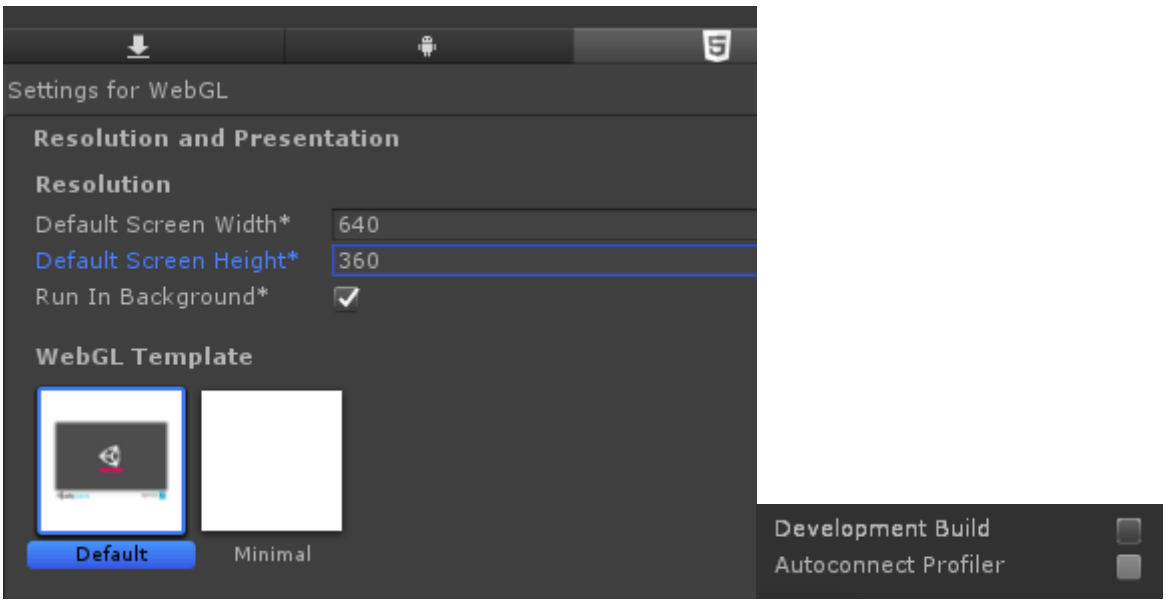
Icono, se sugiere un tamaño de 256 x 256 como mínimo:

Configuraciones generales:



WebGL

Sólo tenemos que descargar la extensión de Unity para poder exportar a web (tarda un rato).



Android

El procedimiento en Android es un poco complejo, te recomiendo el formato vídeo. Podrás encontrarlo en el curso de Udemy o en la Web http://escueladevideojuegos.net/academia-unity/

Scripts

GameController.cs

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

public enum GameState {Idle, Playing, Ended, Ready};

public class GameController : MonoBehaviour {

	[Range (0f, .20f)]
	public float parallaxSpeed = .02f;
	public float scaleTime = 6f;
	public float scaleInc = .25f;
	public GameObject uiIdle;
	public GameObject uiScore;
	public GameObject player;
	public GameObject enemyGenerator;
	public GameState gameState = GameState.Idle;
	public RawImage background;
	public RawImage platform;
	public Text pointsText;
	public Text recordText;

	private AudioSource musicPlayer;
	private int points = 0;

	void Start () {
		musicPlayer = GetComponent<AudioSource>();
		recordText.text = "BEST: " + GetMaxScore().ToString();
	}

	void Update () {
		bool userAction = Input.GetKeyDown("up") || Input.GetMouseButtonDown(0);

		//Empieza el juego
		if (gameState == GameState.Idle && userAction){
			gameState = GameState.Playing;
			uiIdle.SetActive(false);
			uiScore.SetActive(true);
			player.SendMessage("UpdateState", "PlayerRun");
			player.SendMessage("DustPlay");
			enemyGenerator.SendMessage("StartGenerator");
			musicPlayer.Play();
			InvokeRepeating("GameTimeScale", scaleTime, scaleTime);
		}
		//Juego en marcha
		else if (gameState == GameState.Playing){
			Parallax();
		}
		//Juego preparado para reiniciarse
		else if (gameState == GameState.Ready){
			if (userAction){
				RestartGame();
			}
		}
	}

	void Parallax(){
		float finalSpeed = parallaxSpeed * Time.deltaTime;
		background.uvRect = new Rect(background.uvRect.x + finalSpeed, 0f, 1f, 1f);
		platform.uvRect = new Rect(platform.uvRect.x + finalSpeed * 4, 0f, 1f, 1f);
	}
```

```
        public void RestartGame(){
                ResetTimeScale();
                SceneManager.LoadScene("Principal");
        }

        void GameTimeScale(){
                Time.timeScale += scaleInc;
                Debug.Log("Ritmo incrementado: " + Time.timeScale.ToString());
        }

        public void ResetTimeScale(float newTimeScale = 1f){
                CancelInvoke("GameTimeScale");
                Time.timeScale = newTimeScale;
                Debug.Log("Ritmo reestablecido: " + Time.timeScale.ToString());
        }

        public void IncreasePoints(){
                pointsText.text = (++points).ToString();
                if (points >= GetMaxScore()){
                        recordText.text = "BEST: " + points.ToString();
                        SaveScore(points);
                }
        }

        public int GetMaxScore(){
                return PlayerPrefs.GetInt("Max Points", 0);
        }

        public void SaveScore(int currentPoints){
                PlayerPrefs.SetInt("Max Points", currentPoints);
        }
}
```

PlayerController.cs

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

public class PlayerController : MonoBehaviour {

        public GameObject game;
        public GameObject enemyGenerator;
        public AudioClip jumpClip;
        public AudioClip dieClip;
        public AudioClip pointClip;
        public ParticleSystem dust;

        private Animator animator;
        private AudioSource audioPlayer;
        private float startY;

        void Start () {
                animator = GetComponent<Animator>();
                audioPlayer = GetComponent<AudioSource>();
                startY = transform.position.y;
        }

        void Update () {
                bool isGrounded = transform.position.y == startY;
                bool gamePlaying = game.GetComponent<GameController>().gameState ==
GameState.Playing;
                bool userAction = Input.GetKeyDown("up") || Input.GetMouseButtonDown(0);

                if (isGrounded && gamePlaying && userAction){
                        UpdateState("PlayerJump");
                        audioPlayer.clip = jumpClip;
                        audioPlayer.Play();
                }
        }

        public void UpdateState(string state = null){
                if (state != null){
                        animator.Play(state);
                }
        }

        void OnTriggerEnter2D(Collider2D other){
                if (other.gameObject.tag == "Enemy"){
                        UpdateState("PlayerDie");
                        game.GetComponent<GameController>().gameState = GameState.Ended;
                        enemyGenerator.SendMessage("CancelGenerator", true);
                        game.SendMessage("ResetTimeScale", 0.5f);

                        game.GetComponent<AudioSource>().Stop();
                        audioPlayer.clip = dieClip;
                        audioPlayer.Play();

                        DustStop();
                } else if (other.gameObject.tag == "Point"){
                        game.SendMessage("IncreasePoints");
                        audioPlayer.clip = pointClip;
                        audioPlayer.Play();
                }
        }

        void GameReady(){
                game.GetComponent<GameController>().gameState = GameState.Ready;
        }
```

```
        void DustPlay(){
                dust.Play();
        }

        void DustStop(){
                dust.Stop();
        }

}
```

EnemyController.cs

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

public class EnemyController : MonoBehaviour {

        public float velocity = 2f;

        private Rigidbody2D rb2d;

        void Start () {
                rb2d = GetComponent<Rigidbody2D>();
                rb2d.velocity = Vector2.left * velocity;
        }

        void OnTriggerEnter2D(Collider2D other){
                if (other.gameObject.tag == "Destroyer"){
                        Destroy(gameObject);
                }
        }
}
```

EnemyGeneratorController.cs

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

public class EnemyGeneratorController : MonoBehaviour {

        public GameObject enemyPrefab;
        public float generatorTimer = 1.75f;

        void CreateEnemy(){
                Instantiate(enemyPrefab, transform.position, Quaternion.identity);
        }

        public void StartGenerator(){
                InvokeRepeating("CreateEnemy", 0f, generatorTimer);
        }

        public void CancelGenerator(bool clean = false){
                CancelInvoke("CreateEnemy");
                if (clean){
                        Object[] allEnemies = GameObject.FindGameObjectsWithTag("Enemy");
                        foreach( GameObject enemy in allEnemies){
                                Destroy(enemy);
                        }
                }
        }
}
```

Mucho más en

Escuela de Videojuegos

¡Hasta pronto!