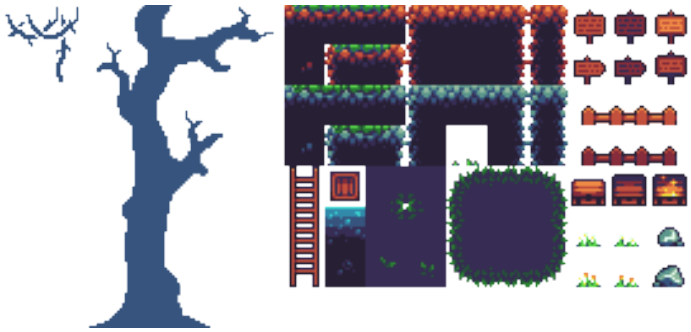
Tema 3. Un juego de plataformas con Unity

En este tema vamos a ver cómo hacer un segundo juego 2D, más complejo y más completo. Hablaremos de gravedad, scroll lateral, mapas de tiles y animaciones más complejas.

3.1. Diseño de un primer nivel. Gravedad.

Si no queremos capturar imágenes a partir de un juego existente, tenemos la alternativa de usar recopilaciones de sprites ya existentes. Es habitual que los distintos sprites se encuentren recopilados en una única imagen, formando lo que se conoce como una "spritesheet":



Hay páginas que contienen varias recopilaciones, como

http://opengameart.org

que tiene toda una categoría dedica a "platformers":

https://opengameart.org/content/openplatformers

Uno de los autores que colabora en OpenGameArt tiene aún más recopialciones en su página web:

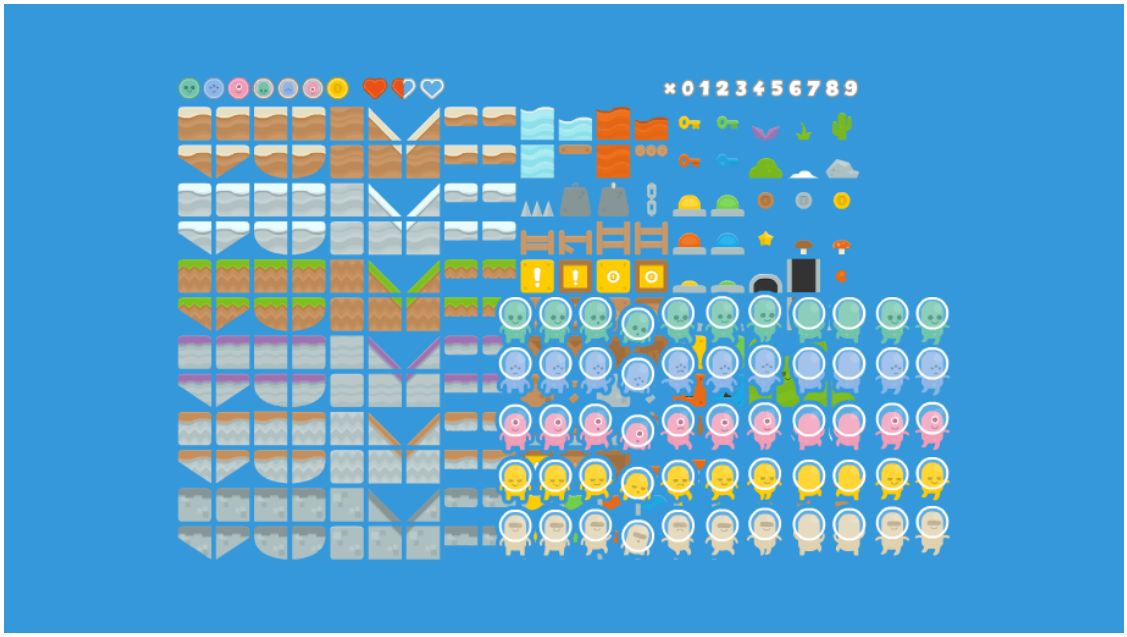
https://www.kenney.nl

donde el apartado de "2d" es:

https://www.kenney.nl/assets?q=2d

En este tema usaremos un extracto de su "Platformer Art Deluxe", que se puede usar libremente:

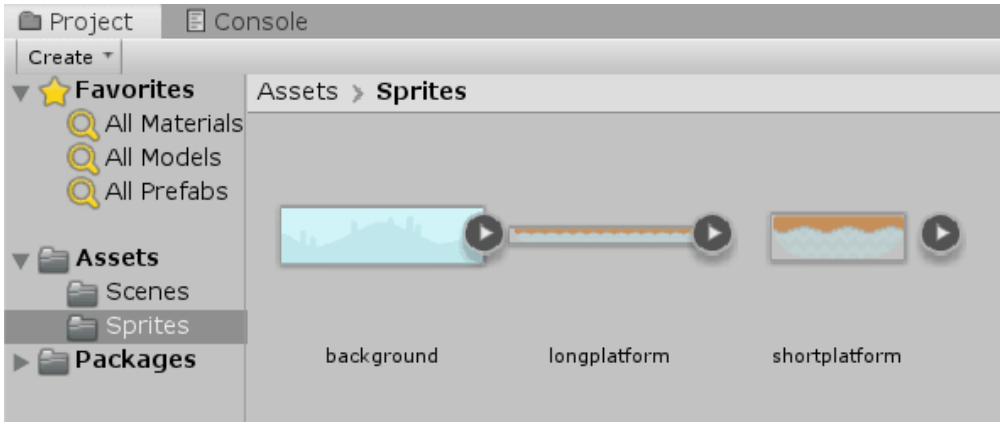
https://www.kenney.nl/assets/platformer-art-deluxe



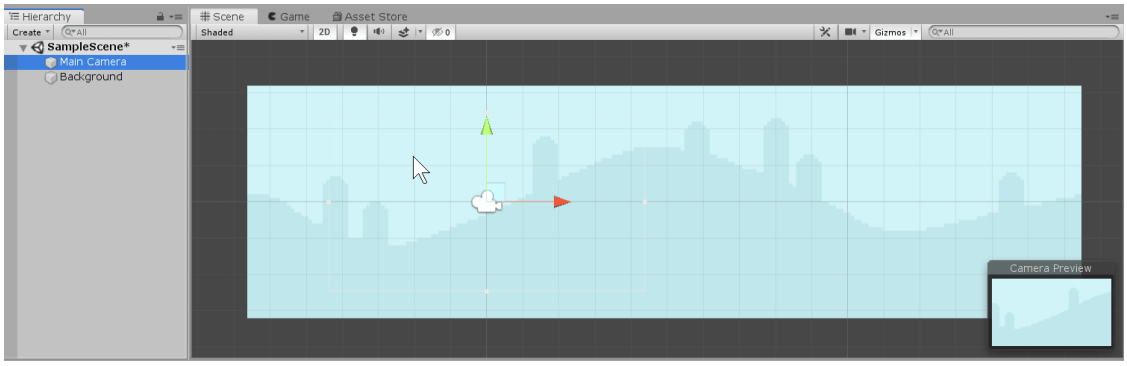
Eso sí, primero partiremos de algunas imágenes extraídas y, según el caso, combinadas para formar "plataformas grandes", hasta el momento en que aprendamos cómo usar un "spritesheet" desde Unity:

Los pasos que podríamos seguir para crear la estructura del primer nivel son:

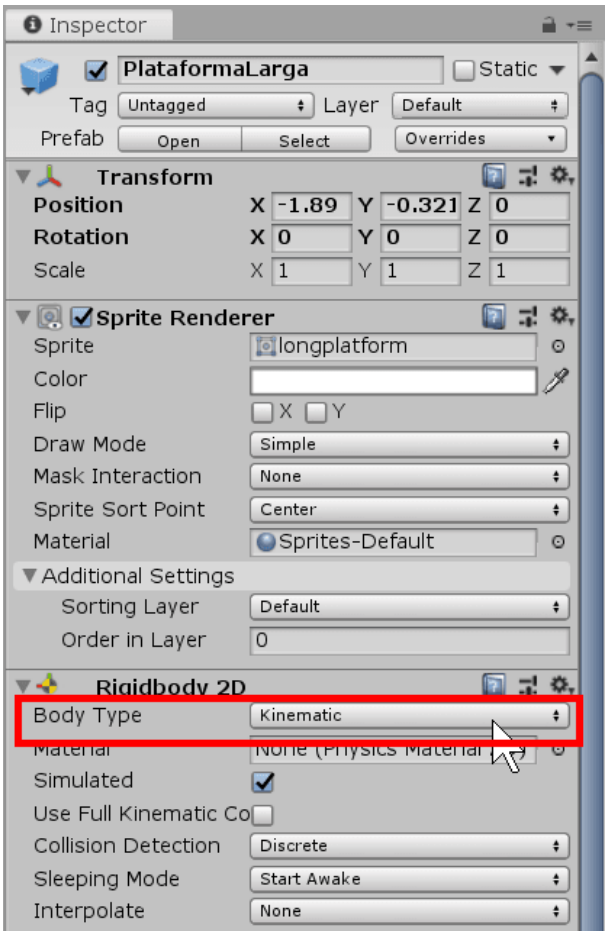
- Prepararemos imágenes para el fondo y para plataformas, en dos tamaños, y las arrastraremos a una carpeta de "Sprites".



- Colocaremos un fondo más grande (especialmente en anchura) que la cámara, para luego poder hacer scroll (un poco más adelante).

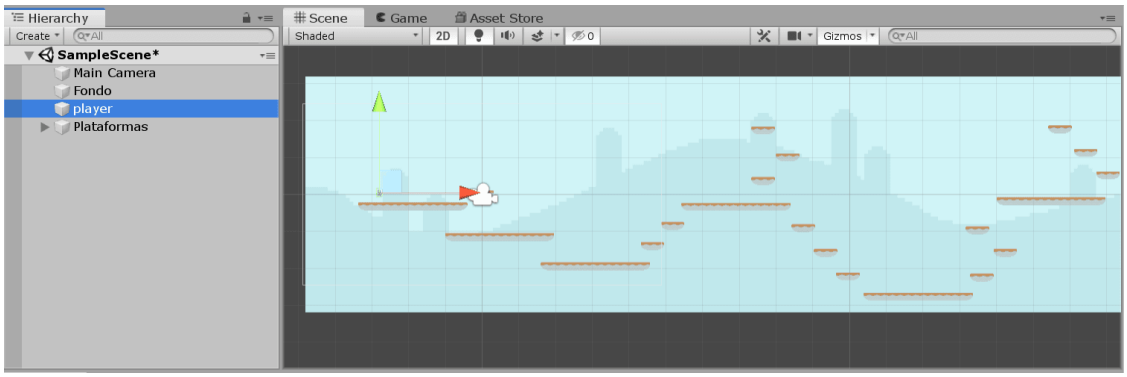


- Como tendremos varias plataformas de 2 tipos distintos, crearemos un "prefab" para cada una de ellas: arrastraremos la imagen a la carpeta de Sprites, de ahí a la escena, le añadiremos un "RigidBody2D", de tipo "Kinematic" para que no le afecte la gravedad, un BoxCollider y luego arrastraremos el objeto al panel inferior para crear un "prefab" a partir de él:



- Luego añadimos el personaje repitiendo los mismos pasos, excepto que para él sí habrá gravedad, por lo que no deberá ser un "cuerpo rígido cinemático" sino dinámico. Si lo situamos un poco por encima de la plataforma, caerá hasta quedar apoyado en ella:

- Y finalmente ya podemos añadir más plataformas hasta diseñar el nivel que nos guste:



- Sólo queda hacer que el personaje se mueva lado a lado, mirando el eje horizontal de los dispositivos de entrada, algo que ya sabemos hacer:

```
float horizontal = Input.GetAxis("Horizontal");
transform.Translate(horizontal * velocidad * Time.deltaTime, 0, 0);
```

Cuando se acabe una plataforma, el personaje caerá hasta la siguiente, si existe, o indefinidamente, si no hay ninguna por debajo. Lo solucionaremos más adelante.

Ejercicio propuesto 3.1.1: Crea un primer nivel a tu gusto

3.2. Saltos: AddForce

Para el salto usaremos "Input.GetAxis("Jump");", que por defecto corresponde a la tecla Espacio del teclado. Si se ha pulsado el botón de salto, añadiremos una fuerza hacia arriba (según el eje Y) al RigidBody2D del personaje:

```
float salto = Input.GetAxis("Jump");
if (salto > 0)
{
    Vector3 fuerzaSalto = new Vector3(0, velocidadSalto, 0);
    GetComponent<Rigidbody2D>().AddForce(fuerzaSalto);
}
```

donde el valor de "velocidadSalto" deberemos afinarlo según el efecto que queramos en nuestro juego, y podría ser algo como:

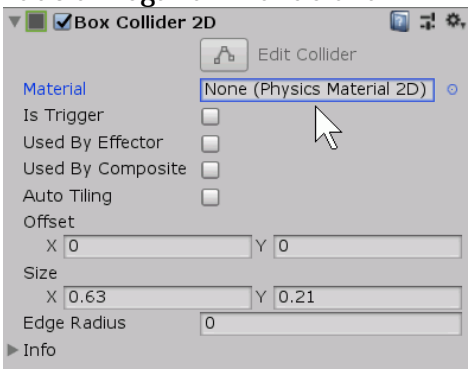
```
private float velocidadSalto = 20;
```

Ejercicio propuesto 3.2.1: Haz que tu personaje pueda saltar

(Como siempre, si queremos que se pueda personalizar desde el editor, lo etiquetaríamos con [SerializeField]).

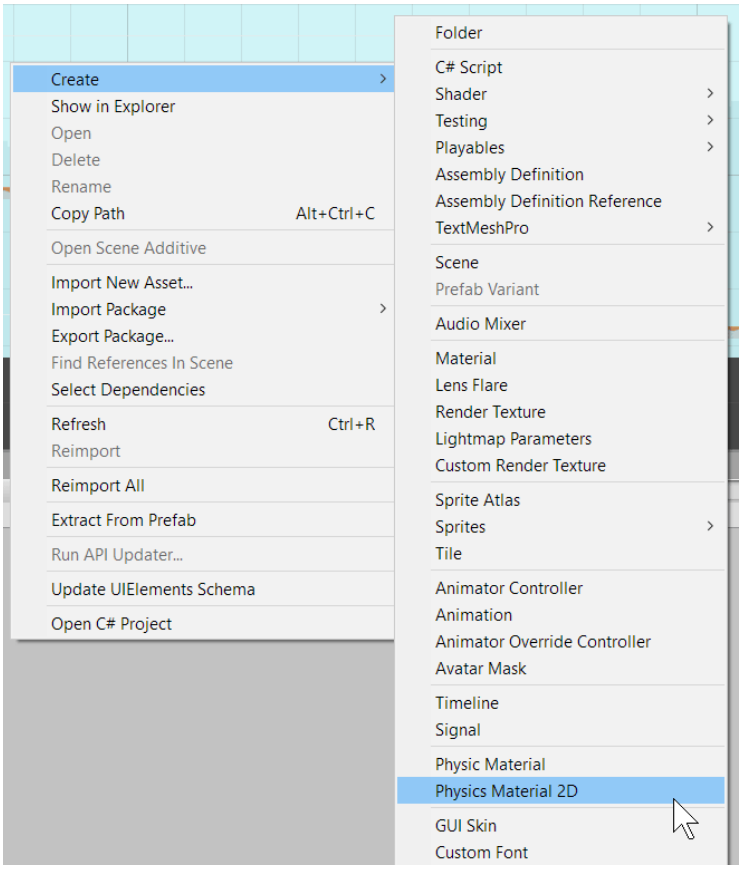
3.3. Fricción entre materiales

Ahora mismo, nuestro personaje se desplaza a una velocidad razonable cuando está en las plataformas, pero sale "disparado" mucho más rápido hacia el lado al llegar al final de una plataforma. Eso se debe a que hay "fricción" con la plataforma (pero no la hay con el aire).

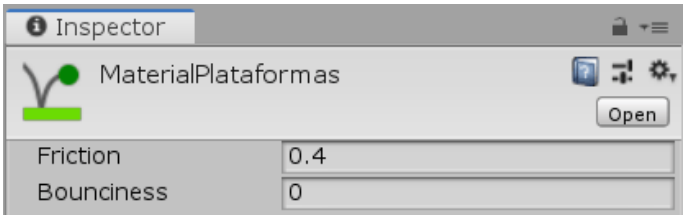


Vamos a solucionarlo. A cada "Collider" se le puede asociar un material (por defecto es "None"):

Pues bien, podemos pulsar el botón derecho sobre el panel inferior y decir que queremos



Y veremos que ese material tiene por defecto un valor de fricción (Friction) de 0.4, que podemos bajar a 0:



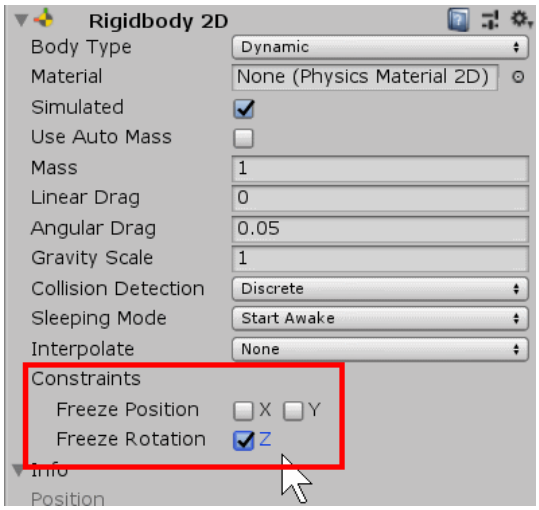
Y ya podríamos asignar ese material a los "prefabs" de las plataformas (e incluso al personaje).

Ahora que no hay fricción, deberemos rebajar la "velocidad" del personaje, o se moverá demasiado deprisa:

```
private float velocidad = 2;
```

Aún queda un problema adicional: en ocasiones puntuales, nuestro personaje puede "girar" tras una colisión:

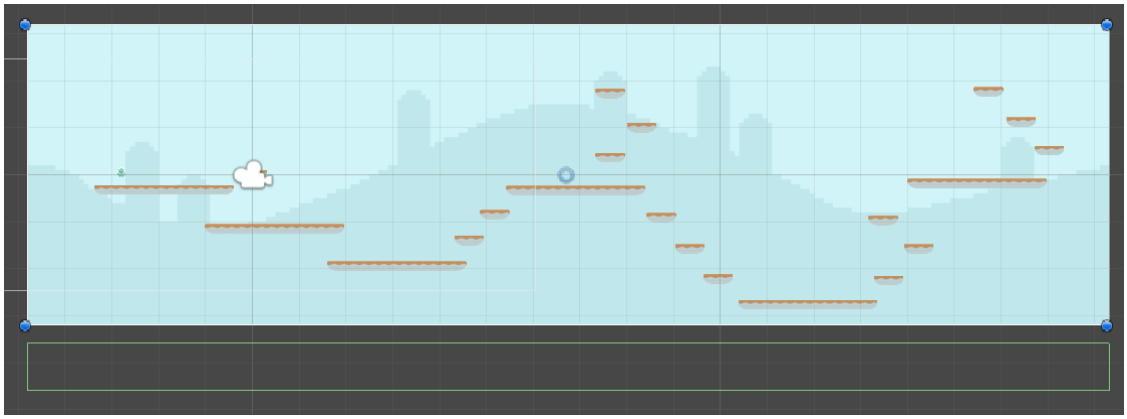
por lo que podemos bloquear su rotación, en el apartado "Constraints" de su RigidBody2D:



Ejercicio propuesto 3.3.1: Ajusta la fricción de tus plataformas.

3.4. Caídas al vacío

Si nuestro personaje cae al vacío, nada le detiene y la partida no termina. Una forma sencilla de evitarlo es añadir un "collider" rectangular por debajo de la zona visible del fondo, que actúe como "trigger".



Si el personaje llega a tocar ese collider, debería volver a su posición inicial y (más adelante) perder una vida, algo que ya habíamos hecho en el laberinto 3D:

```
float xInicial, yInicial;

void Start()
{
    xInicial = transform.position.x;
    yInicial = transform.position.y;
}

public void Recolocar()
{
    transform.position = new Vector3(xInicial, yInicial, 0);
}
```

Así que podemos añadir un script al fondo, de modo que pida al personaje que se recoloque si detecta una colisión con ese collider inferior:

```
private void OnTriggerEnter2D(Collider2D collision)
{
    FindObjectOfType<Personaje>().SendMessage("Recolocar");
}
```

(En un juego un poco más complejo, en el que hubiera otros elementos móviles, deberíamos al menos usar un "tag" para comprobar que es el personaje el que ha colisionado, y no algún otro objeto).

Ejercicio propuesto 3.4.1: No permitas caídas ilimitadas de tu personaje.

3.5. Evitando saltos múltiples: Raycasting

Nuestro juego permite saltos múltiples: si se vuelve a pulsar el botón de salto cuando está en el aire, salta desde esa posición. Eso es algo que no es aceptable en la mayoría de juegos, por lo que deberemos comprobar si está tocando el suelo antes de permitirle saltar.

Una forma de hacerlo es mirar a qué distancia está del suelo. Y una de las posibilidades que permite Unity es "lanzar un rayo" en una cierta dirección y ver a qué distancia toca al siguiente objeto:

```
RaycastHit2D hit = Physics2D.Raycast(transform.position,
    new Vector2(0, -1));
```

(donde un vector2(0,-1) es un vector descendente).

El valor anterior se puede comprobar como "bool", y sería "true" si hay colisión, o "false" si no la hay. En nuestro caso, no nos basta con saber si hay un objeto en esa dirección, sino que queremos ver si está cerca, lo que podríamos hacer con:

```
float distanciaAlSuelo = hit.distance;
bool tocandoElSuelo = distanciaAlSuelo < alturaPersonaje;
if (tocandoElSuelo)
{
    Vector3 fuerzaSalto = new Vector3(0, velocidadSalto, 0);
    GetComponent<Rigidbody2D>().AddForce(fuerzaSalto);
}
```

donde la altura del personaje se puede saber (por ejemplo) a partir del tamaño de su collider:

```
alturaPersonaje = GetComponent<Collider2D>().bounds.size.y;
```

Y, si queremos más seguridad, podríamos comprobar antes si realmente hay algo debajo, aunque es algo que nuestro juego no necesita, porque tiene un "collider" por debajo de todo, para descubrir es el personaje se ha caído:

```
if (hit.collider != null)
{
    float distanciaAlSuelo = hit.distance;
    ///...
```

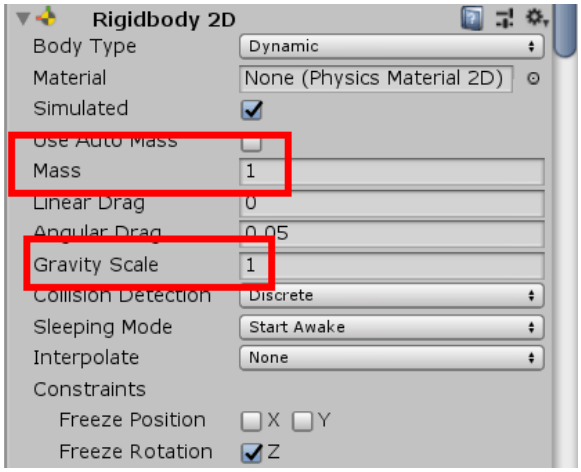
Con todos estos cambios, la clase personaje quedaría así:

```
public class Personaje : MonoBehaviour
{
    private float velocidad = 2;
    private float velocidadSalto = 50;
    float xInicial, yInicial;
    float alturaPersonaje;

    void Start()
    {
        xInicial = transform.position.x;
        yInicial = transform.position.y;
        saltando = false;
        alturaPersonaje = GetComponent<Collider2D>().bounds.size.y;
    }

    void Update()
    {
```

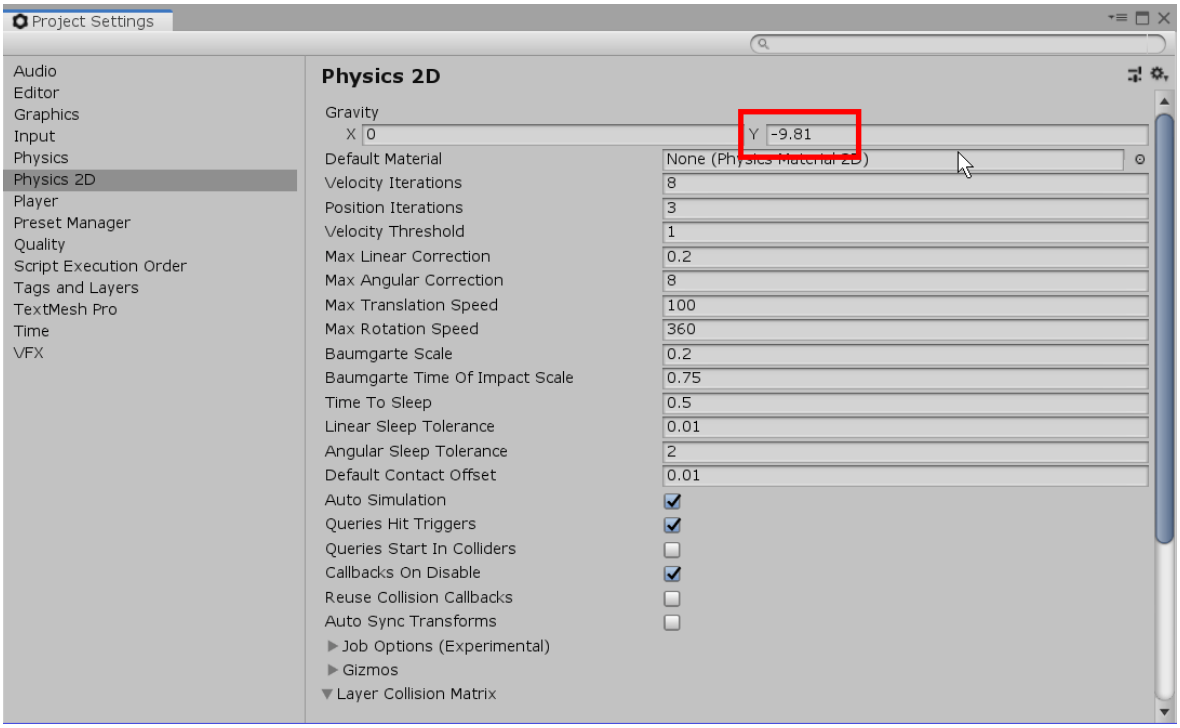
```
        float horizontal = Input.GetAxis("Horizontal");
        transform.Translate(horizontal * velocidad * Time.deltaTime, 0, 0);

        float salto = Input.GetAxis("Jump");
        if (salto > 0)
        {
            RaycastHit2D hit = Physics2D.Raycast(transform.position, new Vector2(0,
-1));
            if (hit.collider != null)
            {
                float distanciaAlSuelo = hit.distance;
                bool tocandoElSuelo = distanciaAlSuelo < alturaPersonaje;
                if (tocandoElSuelo)
                {
                    Vector3 fuerzaSalto = new Vector3(0, velocidadSalto, 0);
                    GetComponent<Rigidbody2D>().AddForce(fuerzaSalto);
                }
            }
        }
    }

    public void Recolocar()
    {
        transform.position = new Vector3(xInicial, yInicial, 0);
    }

}
```

Si no nos acaba de convencer la duración de los saltos, ya sea en vertical o en horizontal, podemos jugar con 3 datos:

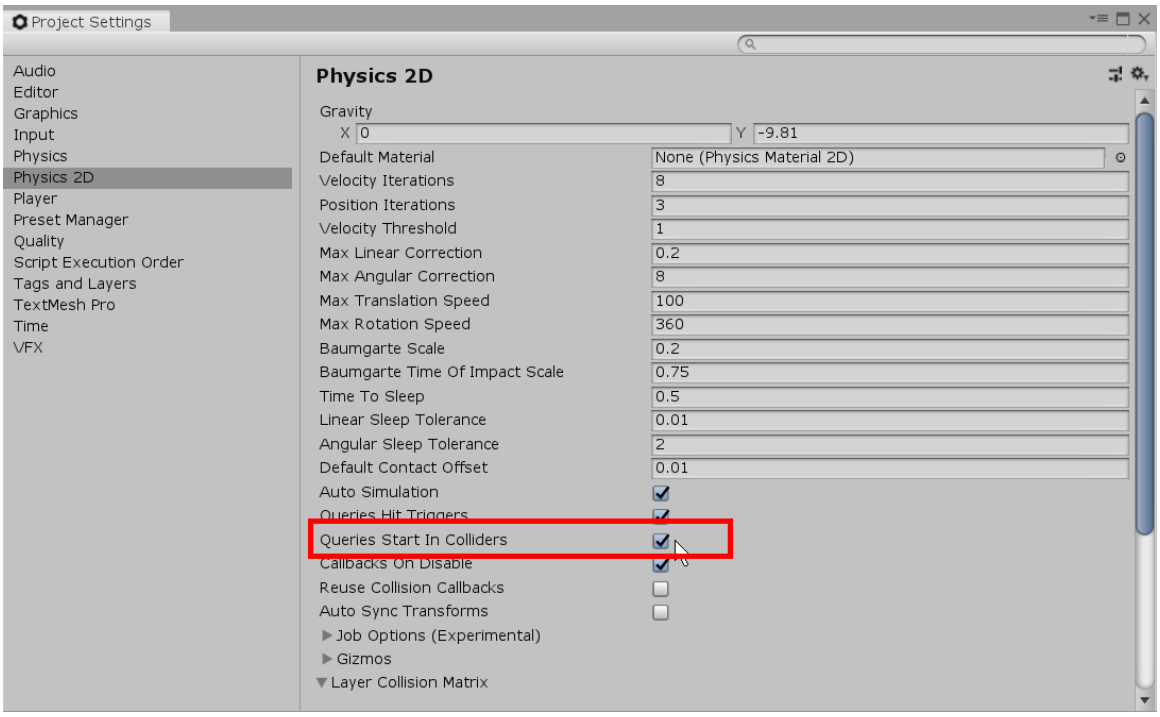
- La velocidad inicial que hemos dado al objeto, que es un atributo de éste.
- Su masa y su "escala de gravedad", que se pueden ver en el "Inspector".



- El valor global de la gravedad, que está en las propiedades del proyecto.



Un detalle adicional que hay que considerar con el "trazado de rayos": si los lanzamos desde el centro del objeto (y no desde fuera de él, o de una esquina), chocarán con el "collider" del propio objeto y nos darán una distancia de 0. Una forma rápida de solucionarlo puede ser decir desde las opciones de "Physics2D" que los rayos no choquen con los "colliders":



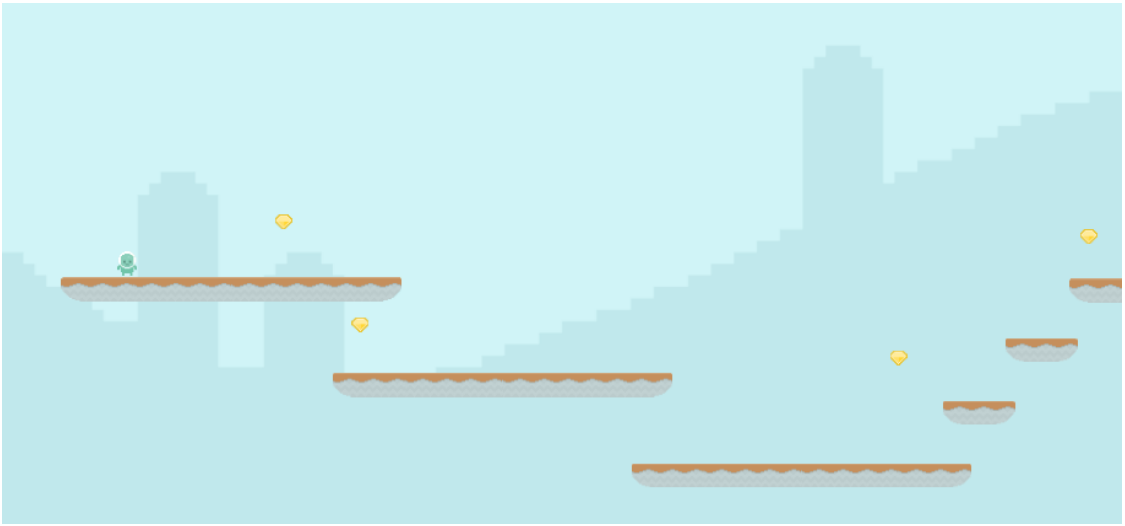
Ejercicio propuesto 3.5.1: No permitas que el personaje encadene saltos.

3.6. Items y puntos: GameController

Añadir Items que el usuario deba recoger es, en buena parte, parecido a las plataformas:

- Incluir el sprite en el proyecto
- Crear un objeto a partir de él
- Ajustar su "altura" dentro de la capa para que quede por delante del fondo
- Añadirle un "collider", que sea de tipo "trigger"
- Crear un "prefab" a partir de él
- Borrar el objeto de la jerarquía y, en su lugar, repartir varios basados en el "prefab"

El resultado podría ser algo como esto:



Podríamos comprobar las colisiones desde los propios Items y, cuando ocurran, avisar al personaje para que lleve cuenta de los puntos y luego destruir el trigger.

Aun así, esta estructura se puede ir complicando cuando también haya que llevar la cuenta de los items restantes, de las vidas, del número de nivel actual, etc., por lo que puede ser más recomendable no hacer que sea el personaje el que lleve cuenta de todo eso, sino crear una clase que represente al juego (típicamente llamada "GameController" o "GameManager").

Así, en la clase Items crearíamos un script como éste:

```
using UnityEngine;

public class Diamante : MonoBehaviour
{
    private void OnTriggerEnter2D(Collider2D collision)
    {
        FindObjectOfType<GameController>().SendMessage("AnotarItemRecogido");
        Destroy(gameObject);
    }
}
```

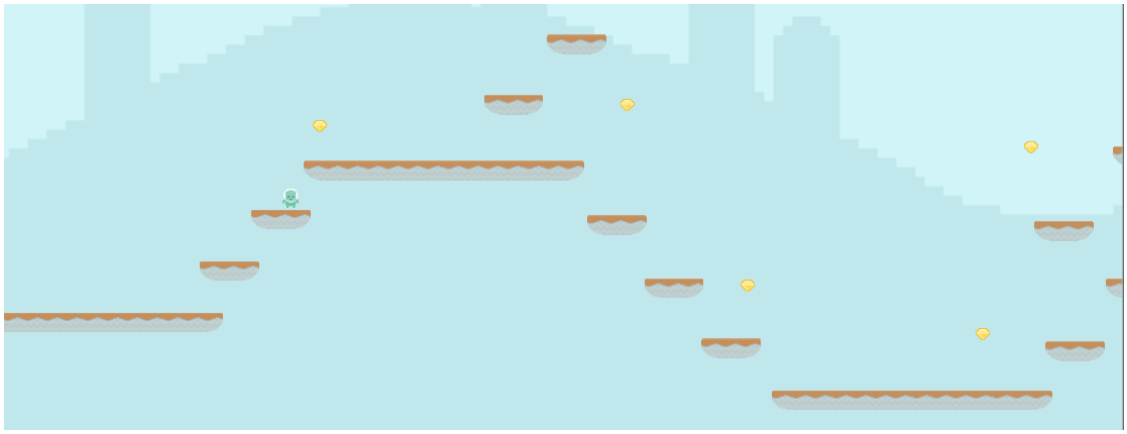
Y también prepararíamos una clase Juego, o Game Controller, que recibiera el aviso de que se ha recogido y llevara cuenta de los puntos:

```
using UnityEngine;

public class GameController : MonoBehaviour
{
    private int puntos;

    void Start()
    {
        puntos = 0;
    }

    public void AnotarItemRecogido()
    {
        puntos += 10;
        Debug.Log("puntos: " + puntos);
    }
}
```

Ejercicio propuesto 3.6.1: Crea ítems que se puedan recoger, así como un canvas con un texto que muestre la puntuación actual.

3.7. Un nivel más grande que la pantalla: Cinemachine y matriz de colisiones

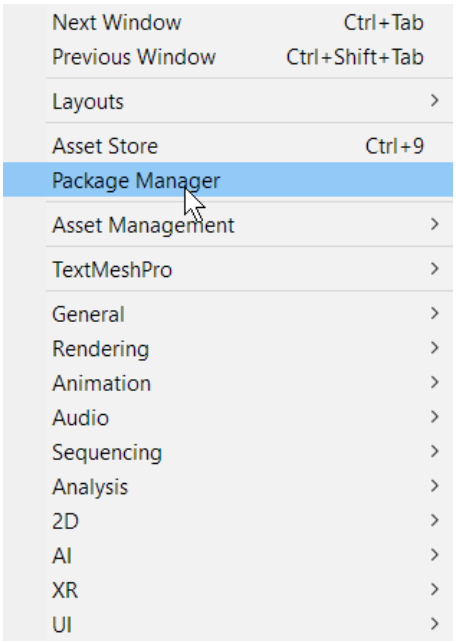
En el tema anterior vimos una primera forma de permitir que el nivel sea más grande que la pantalla: que la cámara sea "un hijo" del personaje, con lo que le irá siguiendo a medida que se mueva:



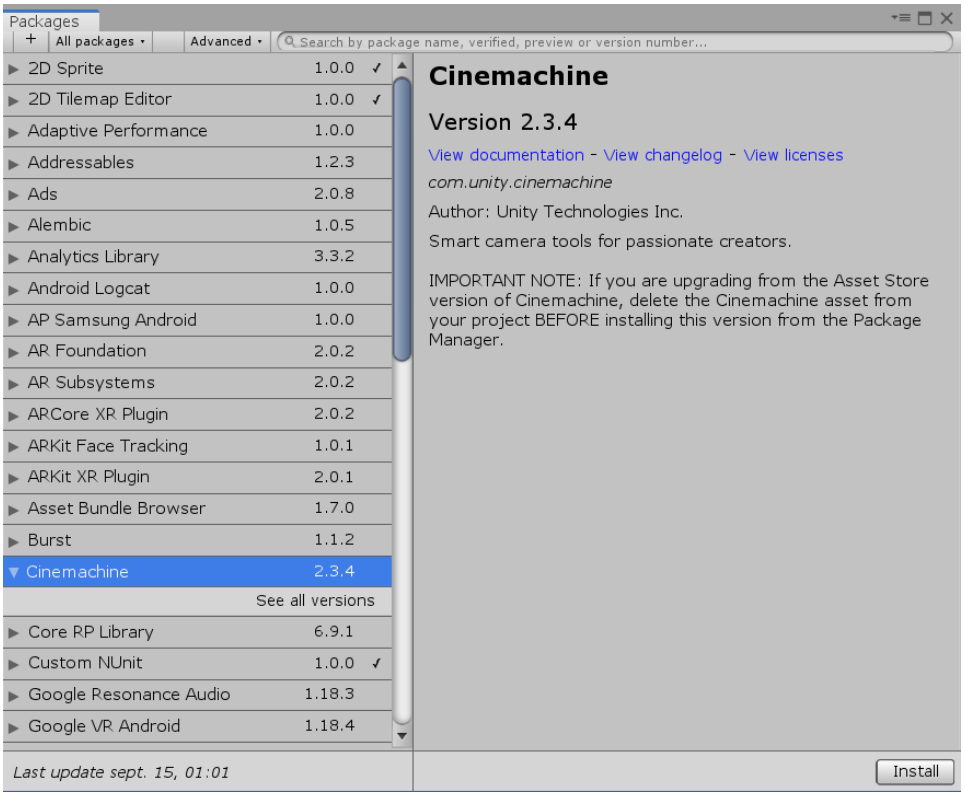
Aun así, este planteamiento tiene el inconveniente en que se puede ver el "color de fondo de la cámara" al acercarnos a las esquinas del mapa, y eso puede no quedar bien si el "fondo visible" del juego no es totalmente regular, como ocurre en éste.

Una alternativa más avanzada que nos ofrece Unity es el "Cinemachine", que permite indicar detalles como zonas a partir de las que la cámara no se debe mover. En Unity 2017 y anteriores

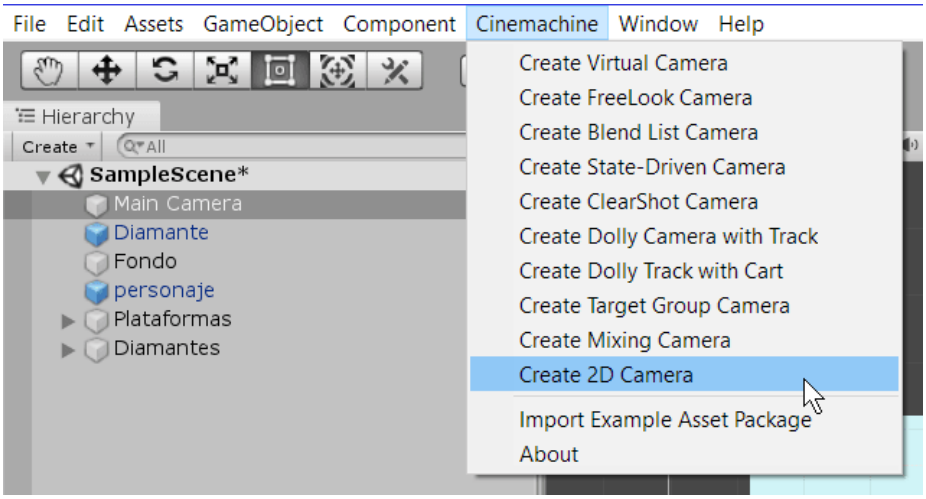
se trataba de un "plugin" que se debía descargar desde la tienda Unity ("Asset Store"), que usaremos más adelante. En 2018 y posteriores, es parte del entorno, que se debe habilitar desde el "Package Manager", al que se accede desde el menú Window:



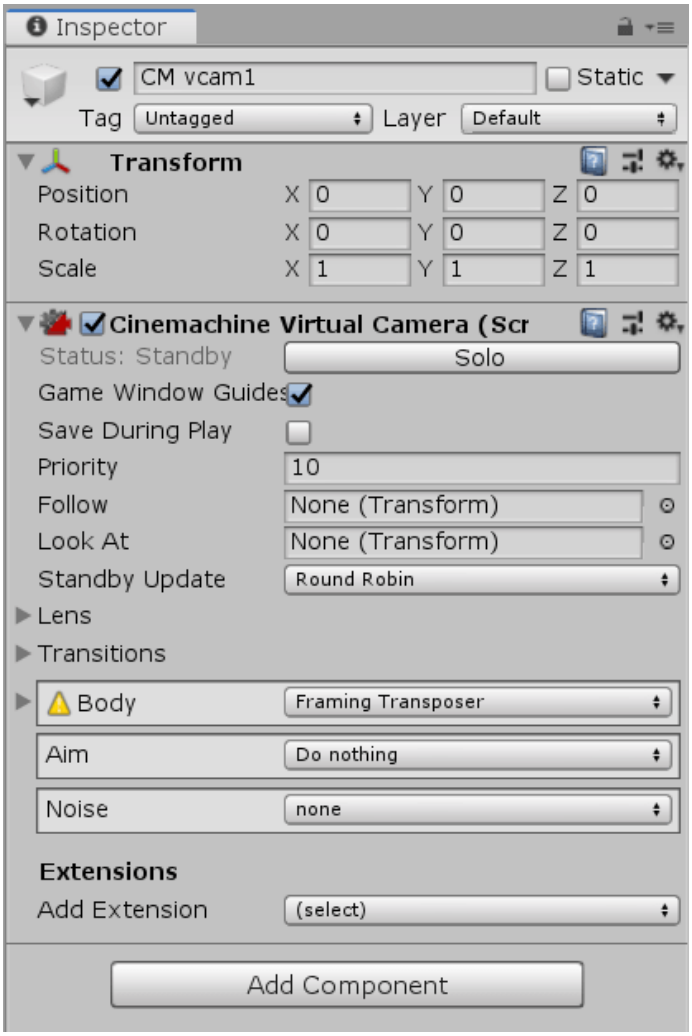
Y entre los paquetes disponibles para instalar, aparecerá "Cinemachine":



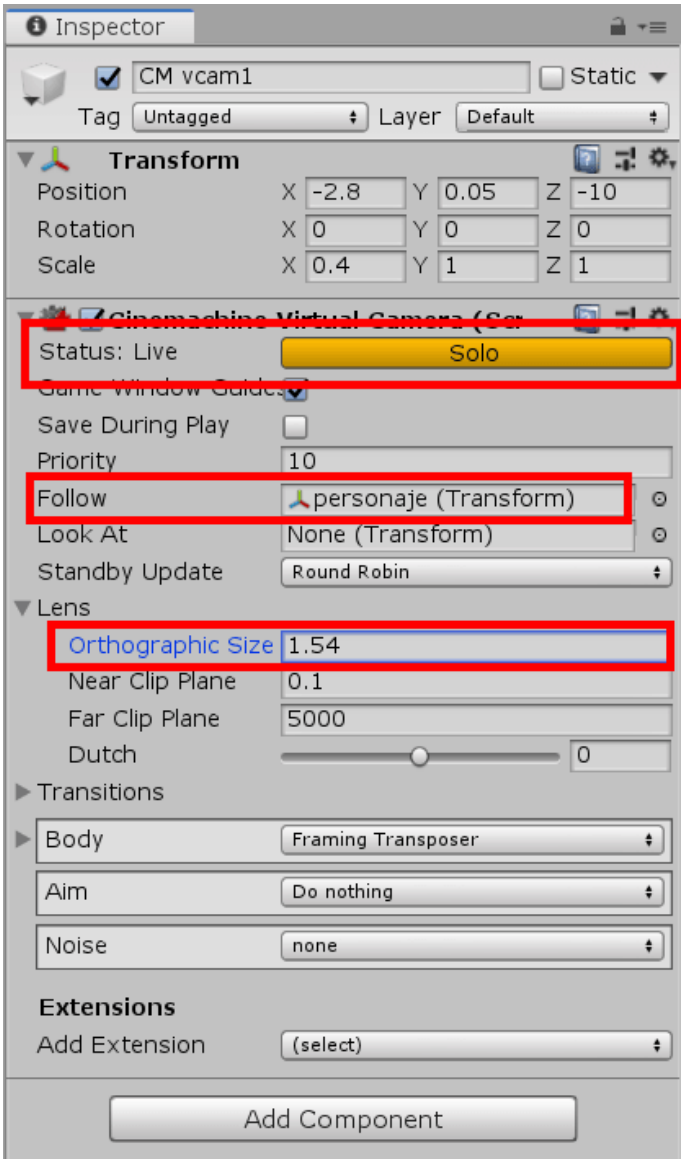
Tras instalarlo, aparecerá un nuevo menú llamado "Cinemachine", y en él tenemos una opción llamada "Create 2D Camera":



Aparecerá una "cámara virtual", cuyas propiedades podemos comprobar en el inspector:



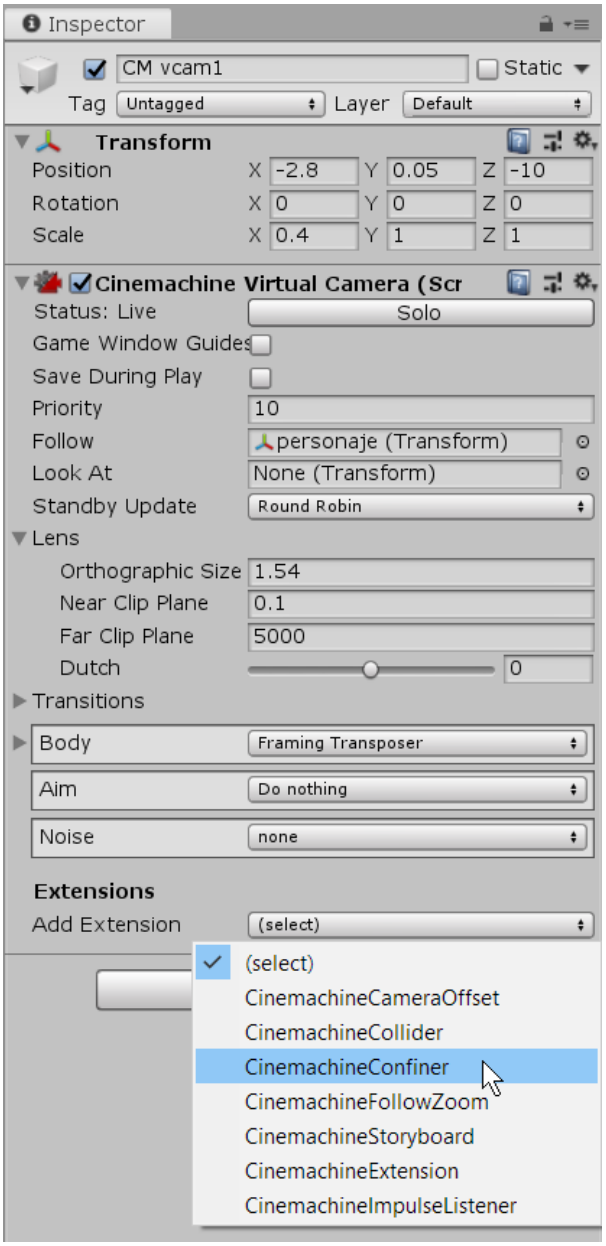
pero si desactivamos la cámara principal, veremos que Unity nos dice que no hay ninguna cámara activa. Deberemos hacer clic en el botón "Solo" de la nueva cámara virtual (que pasará a verse en color amarillo), para que su estado cambie a "Live":



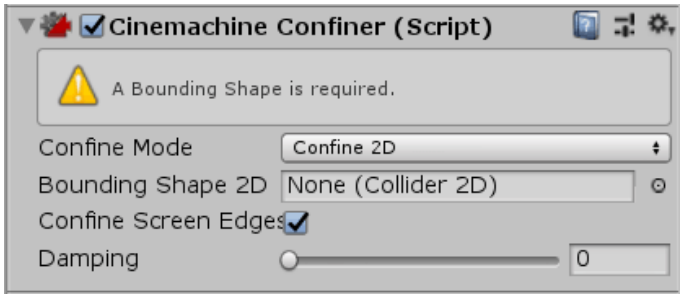
(Si no bastara con eso, deberás ir a la cámara principal y añadirle un nuevo componente: pulsa el botón "Add component" y teclea "brain" en la casilla de búsqueda, para escoger el componente "Cinemachine brain").

También deberemos ajustar el tamaño de la cámara ("ortographic size") e indicar que siga ("Follow") a nuestro personaje.

Así ya tenemos una cámara que sigue a nuestro personaje, y que hace un vistoso efecto de "retardo" cuando el personaje se mueve deprisa. Pero si nos acercamos mucho a un borde del mapa, se sigue viendo el fondo. Para corregirlo, podemos añadir una extensión llamada "Confiner":

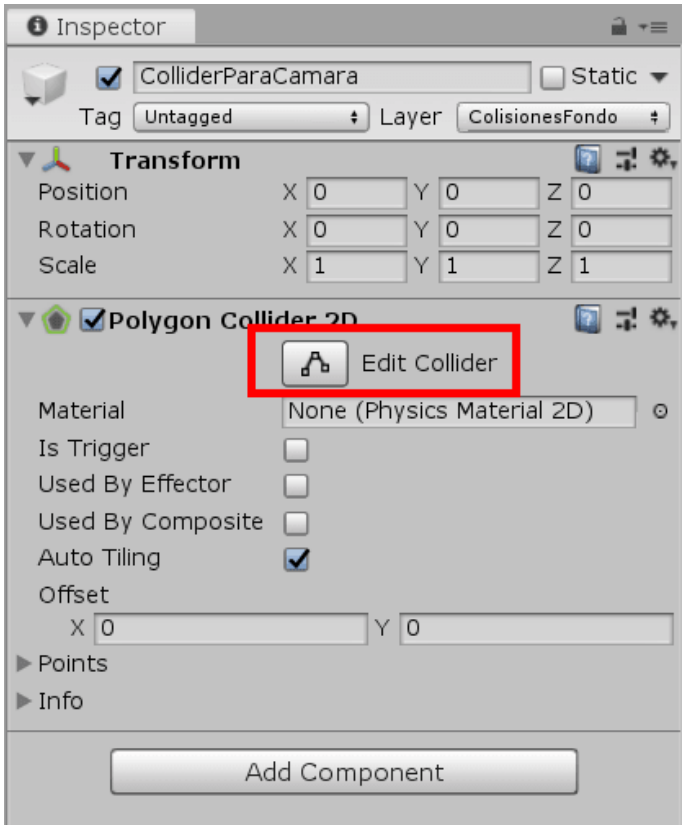


Y en el panel aparece un nuevo apartado para configurar ese "confiner", en el que se nos dirá que necesita un "collider2d":

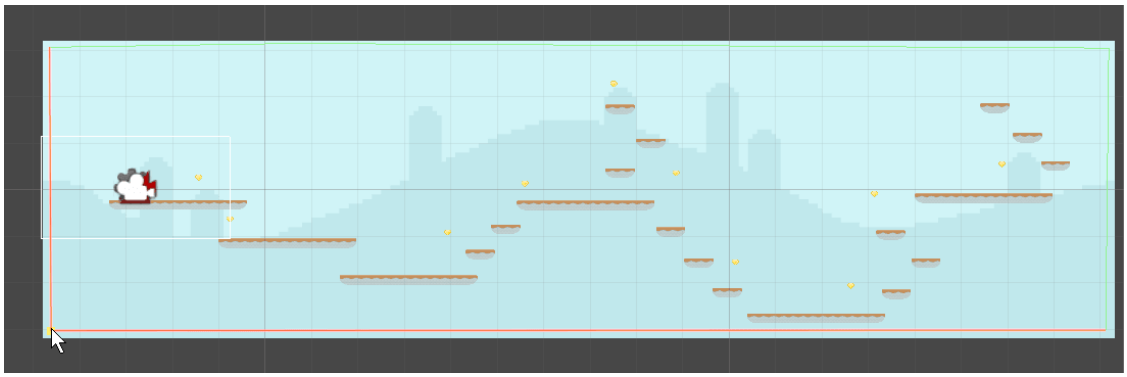


Además, ese Collider no puede ser un BoxCollider2D, que sería lo más sencillo, sino que debe ser un "PolygonCollider2D" o un "CompositeCollider2D". Aun así, se pedimos un Collider poligonal aparecerá un pentágono y podemos borrar su vértice superior.

Creamos un objeto vacío, le añadimos un PolygonCollider2D y pulsamos el botón "Edit Collider":



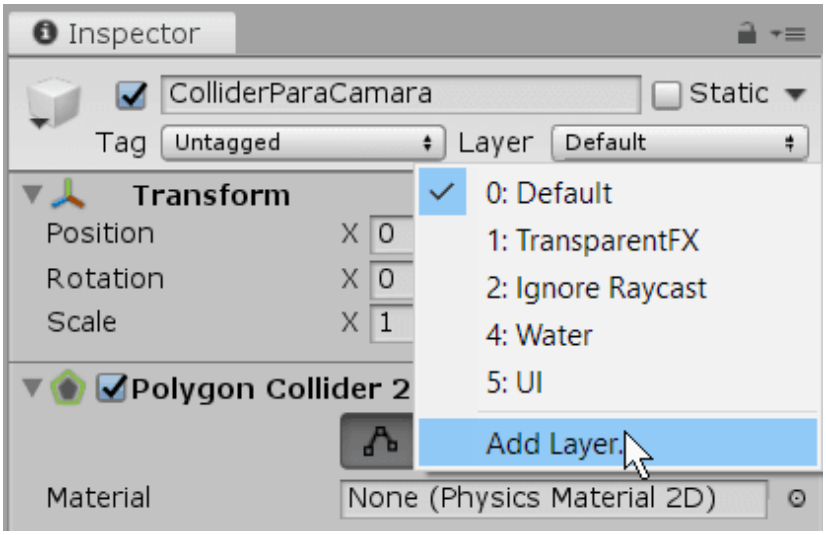
Podemos eliminar el vértice superior si hacemos Ctrl+clic sobre él, y entonces "estirar" los cuatro vértices restantes para crear un rectángulo que cubra casi todo el fondo visible del juego:



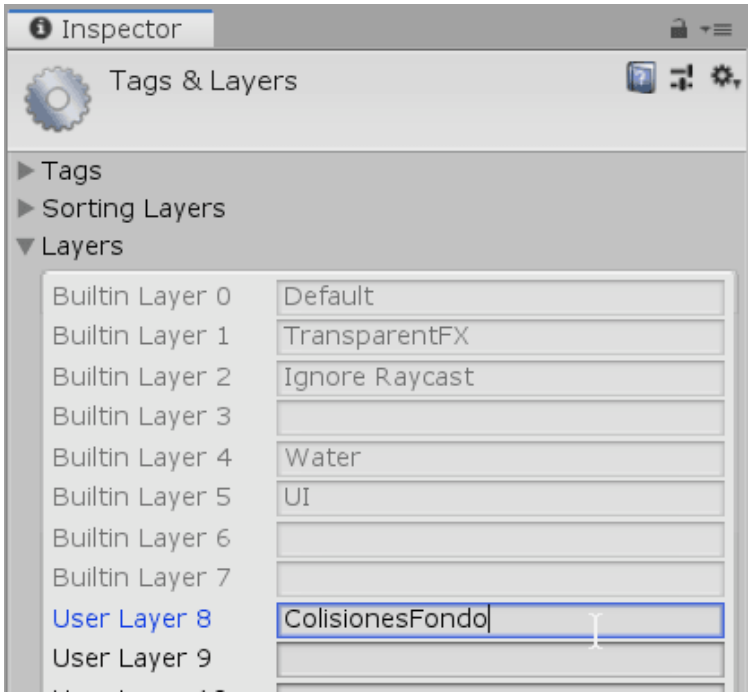
Deberemos "arrastrar" este "collider" a la correspondiente casilla del "confiner".

Pero hay un problema: si ahora lanzamos el juego, este "collider" choca con el personaje y le expulsa de la pantalla. La solución es mover este objeto a otra "capa" y usar la "matriz de

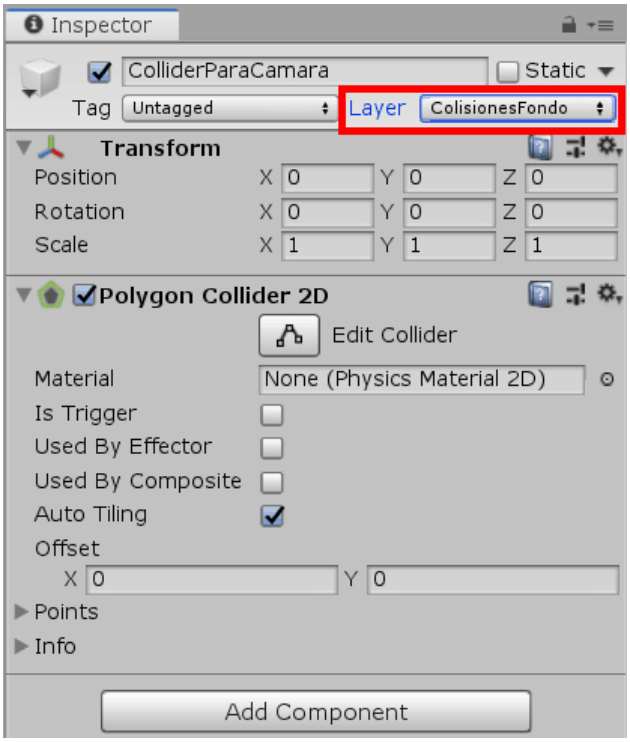
colisiones" para hacer que no se comprueben colisiones de esa capa con ninguna otra. Para añadir una capa, comenzamos por hacer clic en la pestaña de capas, en el inspector:



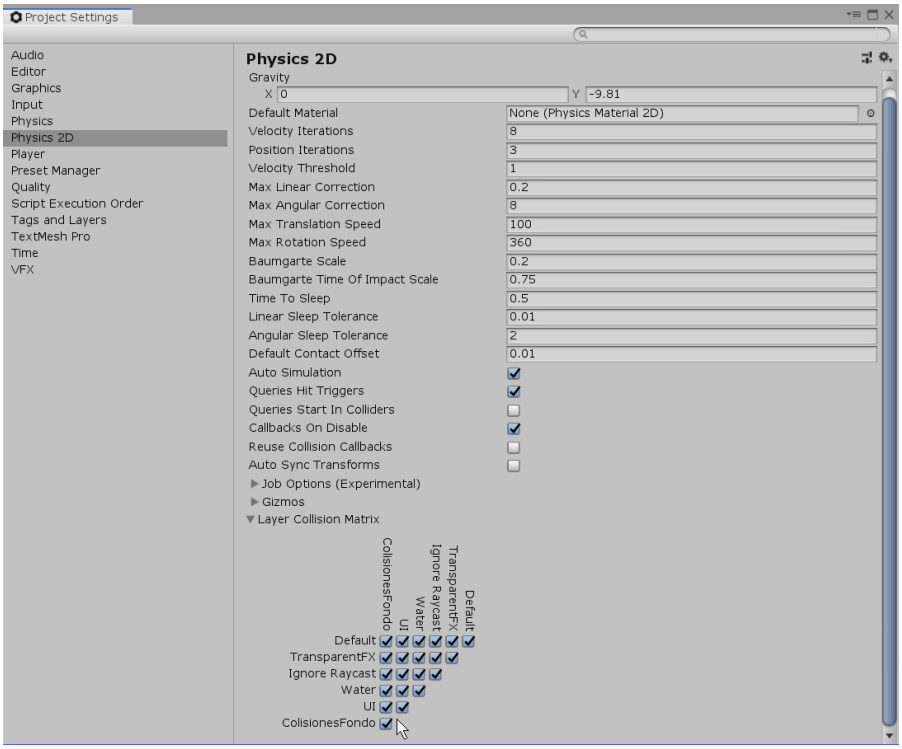
Dentro de las "capas de usuario", podemos dar nombre a la primera disponible:



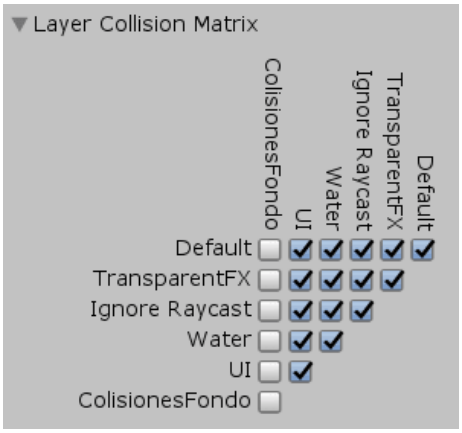
y entonces ya podemos escoger esa capa para nuestro nuevo objeto:



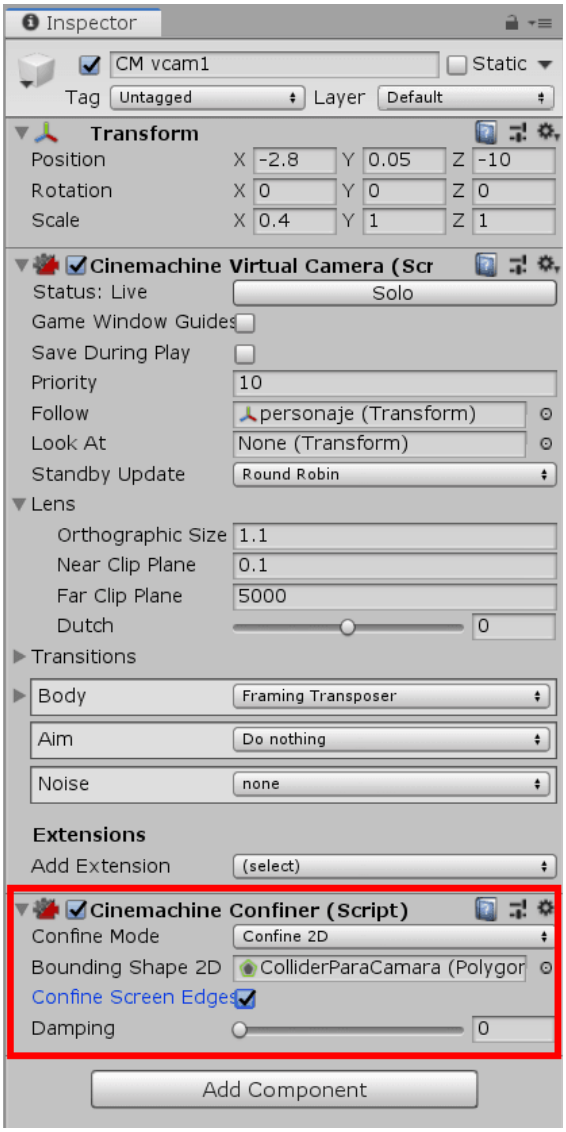
Ahora debemos ir a las preferencias del proyecto ("Project Settings"), en el menú Edit, y la matriz de colisiones aparece en la parte inferior:



Y en esa matriz, deberemos desactivar la comprobación de colisiones entre nuestra nueva capa y todas las demás:



Y con eso, el "confiner" ya estará listo para usar:



Ejercicio propuesto 3.7.1: Afina el comportamiento de la cámara, para que siga al personaje pero sin salirse de la zona visible.

3.8. Enemigos y vidas

Los pasos para crear "enemigos" son básicamente los mismos que para los "ítems" que se deben recoger, con un paso adicional de añadir una lógica para que no estén totalmente estáticos, ya sea cambiando sus coordenadas "de forma totalmente artesanal", como hicimos en el tema de contacto, o usando "waypoints", como en el tema anterior.

Aun así, en este apartado vamos a añadir dos enemigos que todavía serán estáticos, para centrarnos en la lógica de cómo perder vidas y terminar la partida.

Para ello, además de crear enemigos (imagen, posición, altura en la capa, collider, convertir en prefab), añadiremos en la clase Juego (que hemos llamado "GameController") el control de las vidas:

```
using UnityEngine;

public class GameController : MonoBehaviour
{
    private int puntos;
    private int vidas;

    void Start()
    {
        puntos = 0;
        vidas = 3;
    }

    public void AnotarItemRecogido()
    {
        puntos += 10;
        Debug.Log("puntos: " + puntos);
    }

    public void PerderVida()
    {
        vidas--;
        FindObjectOfType<Personaje>().SendMessage("Recolocar");
        Debug.Log("Una vida menos. Quedan: " + vidas);
        if (vidas <= 0)
        {
            // TO DO: Volver al menú principal
            Debug.Log("Partida terminada");
            Application.Quit();
        }
    }
}
```

(Recuerda que "Application.Quit()" no funcionará desde el editor, sólo si creas un ejecutable).

Y el script del enemigo se limitaría a avisar de que se debe perder una vida en caso de colisión:

```
using UnityEngine;

public class Enemigo : MonoBehaviour
{
    private void OnTriggerEnter2D(Collider2D collision)
    {
        FindObjectOfType<GameController>().SendMessage("PerderVida");
    }
}
```

(Exactamente esa misma sería la lógica del script asociado al fondo, cuando el personaje toca el "collider" inferior que avisa de una caída).

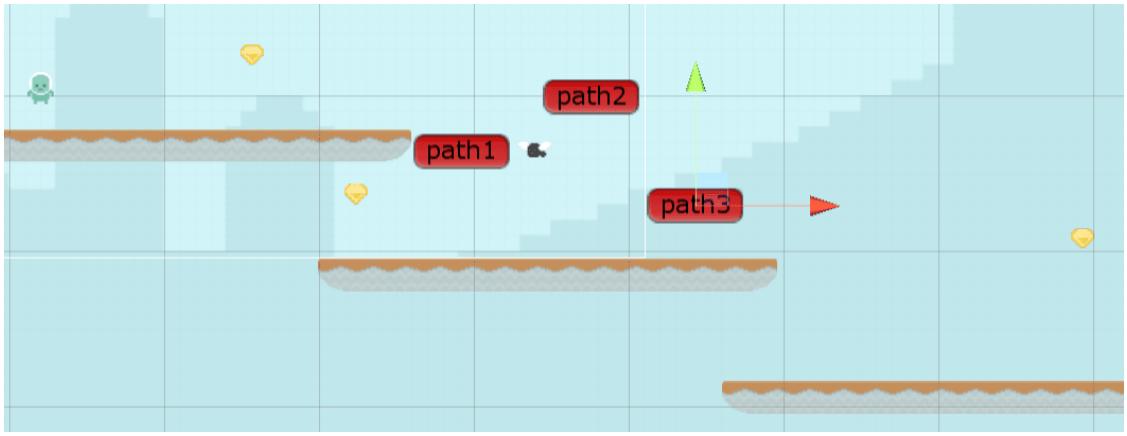
Ejercicio propuesto 3.8.1: Añade al menos 2 enemigos, comprueba colisiones con ellos, y haz que el personaje pierda una vida si los toca.

3.9. Movimiento de enemigos: Waypoints 2D

Si queremos que un enemigo se mueva de lado a lado o de arriba a abajo, podemos hacerlo "de forma manual", comprobando si sus coordenadas alcanzan un cierto límite, como hicimos en el primer acercamiento al "matamarcianos". Si su movimiento va a ser más complejo, podemos usar "waypoints", como habíamos hecho en el juego del laberinto 3D.

Los pasos serán:

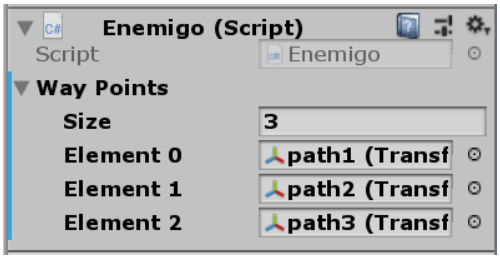
- Crear objetos vacíos
- Asignarles un color o icono
- Ajustar su posición
- Encadenarlos en un array o una lista
- Crear el fragmento de script que permita recorrer esa lista en orden



En esta ocasión, en vez de un array, vamos a usar una lista, que será un nuevo atributo del Enemigo:

```
[SerializeField] List<Transform> wayPoints;
```

pero el comportamiento será el mismo: aparecerá en el inspector, podremos indicar su tamaño y arrastrar a ella los elementos del path que habíamos colocado en pantalla:



Y el resto de pasos ya los conocemos:

Creamos un atributo para la velocidad, y (si queremos) lo dejamos accesible desde el editor:

```
[SerializeField] float velocidad = 2;
```

Creamos otro atributos para la siguiente posición (ya sea como coordenadas, o bien, por variar, como su número dentro de la lista de puntos a recorrer):

```
private byte siguientePosicion;
```

En "Start" le damos como valor la posición inicial:

```
private void Start()
{
    siguientePosicion = 0;
}
```

Y en "Update" usamos "Vector3.MoveTowards" para calcular una nueva posición que esté entre la actual ("transform.position") y la siguiente ("wayPoints[siguientePosicion]. transform. position"), y, cuando esté suficientemente cerca, apuntamos al siguiente destino:

```
private void Update()
{
    transform.position = Vector3.MoveTowards(
        transform.position,
        wayPoints[siguientePosicion].transform.position,
        velocidad * Time.deltaTime);

    if (Vector3.Distance(transform.position,
        wayPoints[siguientePosicion].transform.position) < distanciaCambio)
    {
        siguientePosicion++;
        if (siguientePosicion >= wayPoints.Count)
            siguientePosicion = 0;
    }
}
```

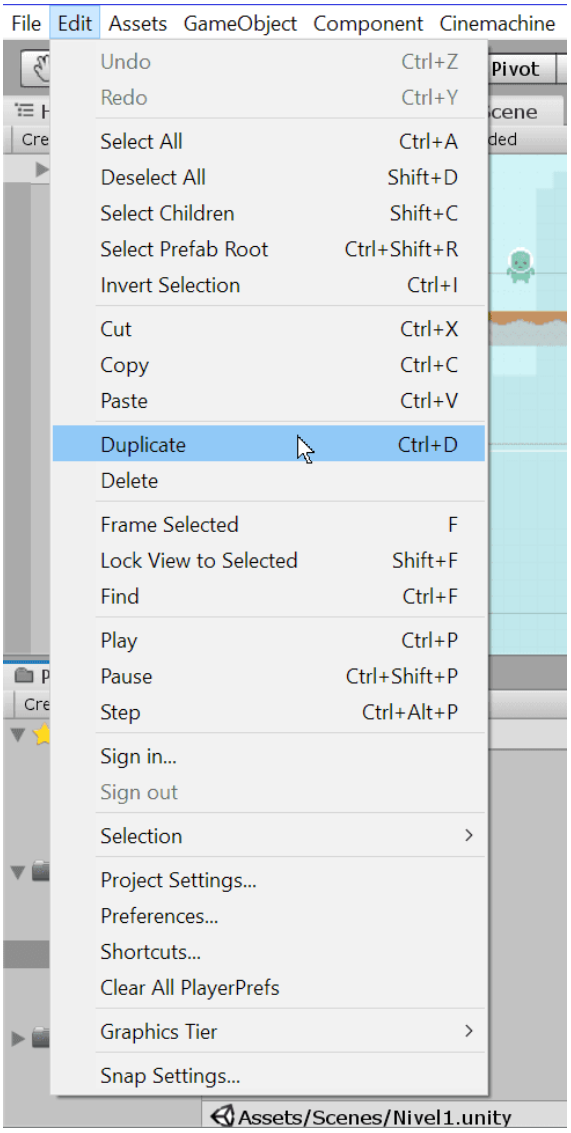
Nota: Si el recorrido del enemigo tocase algún "ítem", podríamos tener "falsos positivos" que nos hicieran creer que el personaje ha recogido algún elemento o que ha chocado con el enemigo. Para solucionarlo, podemos usar "tags" o bien la matriz de colisiones.

Ejercicio propuesto 3.9.1: Crea un recorrido para cada uno de tus dos enemigos.

3.10. Dos niveles consecutivos

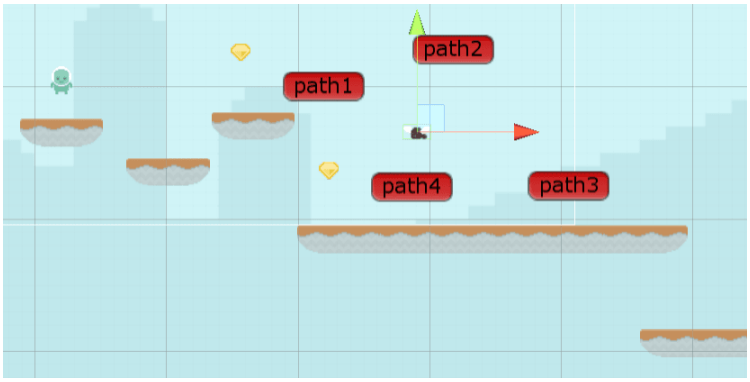
Cuando ya tenemos un primer nivel totalmente funcional y suficientemente probado, podemos usarlo como "plantilla" para un segundo nivel. Bastará con "duplicar" la escena actual y luego mover o añadir elementos, pero antes conviene que "casi todo" sean "prefabs", incluso elementos que hasta ahora no se repetían, como el personaje, porque ahora puede ocurrir que deseemos hacer cambios al personaje y que estos cambios se apliquen a todos los niveles existentes.

Para duplicar una escena basta con seleccionarla en el panel del proyecto, y, en el menú "Edit", escoger "Duplicate":



(Recuerda guardar antes los cambios en la escena actual)

Y en esa segunda escena podríamos cambiar detalles como la cantidad y situación de las plataformas, o el recorrido de los enemigos:



Además, deberíamos enriquecer el "GameController", con un atributo que lleve cuenta de la cantidad de ítems quedan por recoger.

```
private int itemsRestantes;
```

cuyo valor daríamos en "Start", buscando los objetos de tipo "Diamante" (o el que corresponda) y mirando su cantidad:

```
itemsRestantes = FindObjectsOfType<Diamante>().Length;
```

y "AnotarItemRecogido" no sólo sumaría puntos, sino que además descontaría un item restante:

```
public void AnotarItemRecogido()
{
    puntos += 10;
    Debug.Log("puntos: " + puntos);

    itemsRestantes--;
    Debug.Log("items restantes: " + itemsRestantes);
    if (itemsRestantes <= 0)
        AvanzarNivel();
}
```

Y ese "AvanzarNivel" de momento podría ser simplemente un saltar a la escena 2:

```
private void AvanzarNivel()
{
    SceneManager.LoadScene("Nivel2");
}
```

Aun así, esta lógica no es correcta: tras el nivel 2... seguiría en el nivel 2. Además, verás que se pierden los puntos al pasar al nuevo nivel. Solucionaremos ambas cosas en el siguiente apartado.

Ejercicio propuesto 3.10.1: Haz que tu juego permita pasar a un segundo nivel tras recoger todos los ítems. Como mejora opcional, puedes hacer que aparezca una llave al recoger todo, y sea la llave la que dé acceso al nuevo nivel.

3.11. Conservando los puntos: el patrón Singleton

El problema de tener varios niveles implementados como escenas distintas es que cuando se crea una escena nueva, se destruye la anterior, así como todos los objetos que contenía.

En nuestro caso, hay un objeto del que querríamos mantener siempre una y sólo una copia activa: el controlador del juego. Una alternativa sencilla, cuando sólo se trata de conservar datos simples (como vidas o puntos) es etiquetar esos atributos como "static". En casos más complejos, este planteamiento se puede quedar corto, pero existe una alternativa elegante...

Eso de mantener una copia y sólo una copia de un objeto es una necesidad frecuente en más de un programa, por lo que existe un "patrón de diseño" asociado: el patrón Singleton. Un "Singleton" es una clase de la que queremos que exista una única copia en nuestro programa, y existen distintas formas de implementar este patrón según la plataforma que se esté utilizando.

Aun así, para nuestro juego, crearemos una nueva clase antes de seguir, porque nuestro controlador del juego ("GameController") contiene detalles que no se deben reiniciar al cambiar de nivel, como los puntos o las vidas, pero también otros detalles que sí deberían reiniciarse, como la cantidad de ítems pendientes de recoger. Por eso, será mejor crear una clase adicional ("EstadoDelJuego" o "GameStatus"), que contenga sólo los datos que queremos que se conserven entre niveles.

Ahora sí, en Unity, una forma de crear un "singleton" es:

- Usar "DontDestroyOnLoad" para no destruir el "GameStatus" cuando se salta a otra escena.
- Al crear un nuevo objeto de la clase "GameStatus", comprobar si existe algún otro, y si es así, destruir el que se acaba de crear.
- Esto habrá que hacerlo antes incluso del método "Start", en el método "Awake".

Así, la clase GameStatus (asociada a un nuevo objeto vacío) podría ser:

```
using UnityEngine;

public class GameStatus : MonoBehaviour
{
    public int puntos = 0;
    public int vidas = 3;
    public int nivelActual = 1;
    public int nivelMasAlto = 2;

    private void Awake()
    {
        int gameStatusCount = FindObjectsOfType<GameStatus>().Length;

        if (gameStatusCount > 1)
        {
            Destroy(gameObject);
        }
        else
        {
            DontDestroyOnLoad(gameObject);
```

```
        }
    }
}
```

Y desde GameController se accedería a esos datos, tanto en "Start":

```
void Start()
{
    itemsRestantes = FindObjectsOfType<Diamante>().Length;
    puntos = FindObjectOfType<GameStatus>().puntos;
    vidas = FindObjectOfType<GameStatus>().vidas;
    nivelActual = FindObjectOfType<GameStatus>().nivelActual;
}
```

Como cuando cambien los puntos, las vidas o el nivel:

```
public void AnotarItemRecogido()
{
    puntos += 10;
    FindObjectOfType<GameStatus>().puntos = puntos;
    // ...
}

public void PerderVida()
{
    vidas--;
    FindObjectOfType<GameStatus>().vidas = vidas;
    //...
}

private void AvanzarNivel()
{
    nivelActual++;
    if (nivelActual > FindObjectOfType<GameStatus>().nivelMasAlto)
        nivelActual = 1;
    FindObjectOfType<GameStatus>().nivelActual = nivelActual;
    SceneManager.LoadScene("Nivel" + nivelActual);
}
```

Ejercicio propuesto 3.11.1: Haz que se guarden puntos y vidas al pasar de un nivel a otro.

3.12. Menú y pausa hasta volver a él

Ya sabemos crear un menú, y usar "LoadScene" para volver a él cuando termine la partida. Pero en general, sería deseable que el salto no fuera instantáneo, sino que diera tiempo al usuario a darse cuenta de por qué ha terminado la partida.

La forma de solucionarlo también la conocemos: usar una "corrutina" para saltar tras un cierto tiempo a la función que carga el menú:

En "PerderVida", de "GameController" haríamos

```
if (vidas <= 0)
{
    TerminarPartida();
}
```

donde esa función TerminarPartida sería simplemente:

```
private void TerminarPartida()
{
    Debug.Log("Partida terminada");
    StartCoroutine(VolverAlMenuPrincipal());
}
```

y la corrutina "VolverAlMenuPrincipal()" podría ser:

```
private IEnumerator VolverAlMenuPrincipal()
{
    yield return new WaitForSeconds(3);
    SceneManager.LoadScene("Menu");
}
```

Pero lo podemos mejorar un poco más:

- Podemos hacer que haya un texto de "Game Over" sobre el juego, inicialmente inactivo, que pase a estar activo en este momento.
- Podemos ralentizar el juego durante esos 3 segundos de espera, bajando el valor de "Time.timeScale", y restaurar la velocidad del juego justo antes de saltar al menú:

```
private IEnumerator VolverAlMenuPrincipal()
{
    textoGameOver.enabled = true;
    Time.timeScale = 0.1f;
    yield return new WaitForSeconds(0.3f); // 3 segundos de tiempo real
    Time.timeScale = 1;
```

```
    SceneManager.LoadScene("Menu");
}
```

Como hemos ralentizado el tiempo del juego a un 10% de su velocidad inicial, hemos empleado "WaitForSeconds(0.3f);" para esperar 3 segundos. Como alternativa, existe una variante que puede resultar más legible en este caso: "WaitForSecondsRealtime", con lo que el método anterior podría quedar:

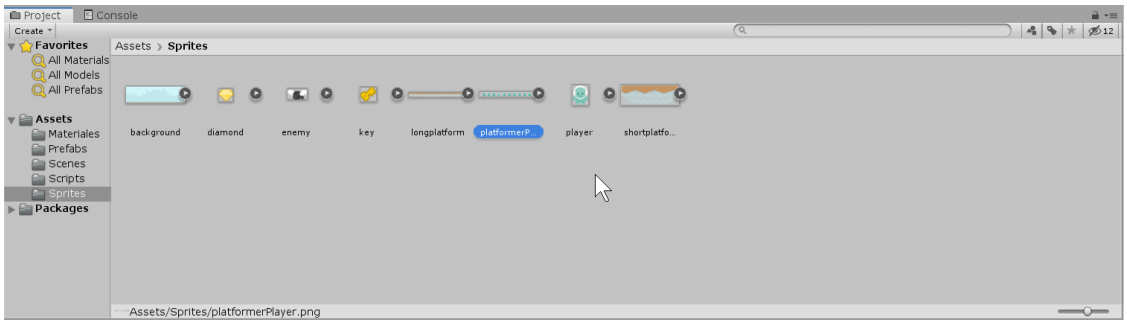
```
private IEnumerator VolverAlMenuPrincipal()
{
    textoGameOver.enabled = true;
    Time.timeScale = 0.1f;
    yield return new WaitForSecondsRealtime(0.3f);
    Time.timeScale = 1;
    SceneManager.LoadScene("Menu");
}
```

Ejercicio propuesto 3.12.1: Crea un menú principal, con un botón que permita entrar al juego. Haz que se vuelva a él al perder la partida, tras una pausa de 3 segundos.

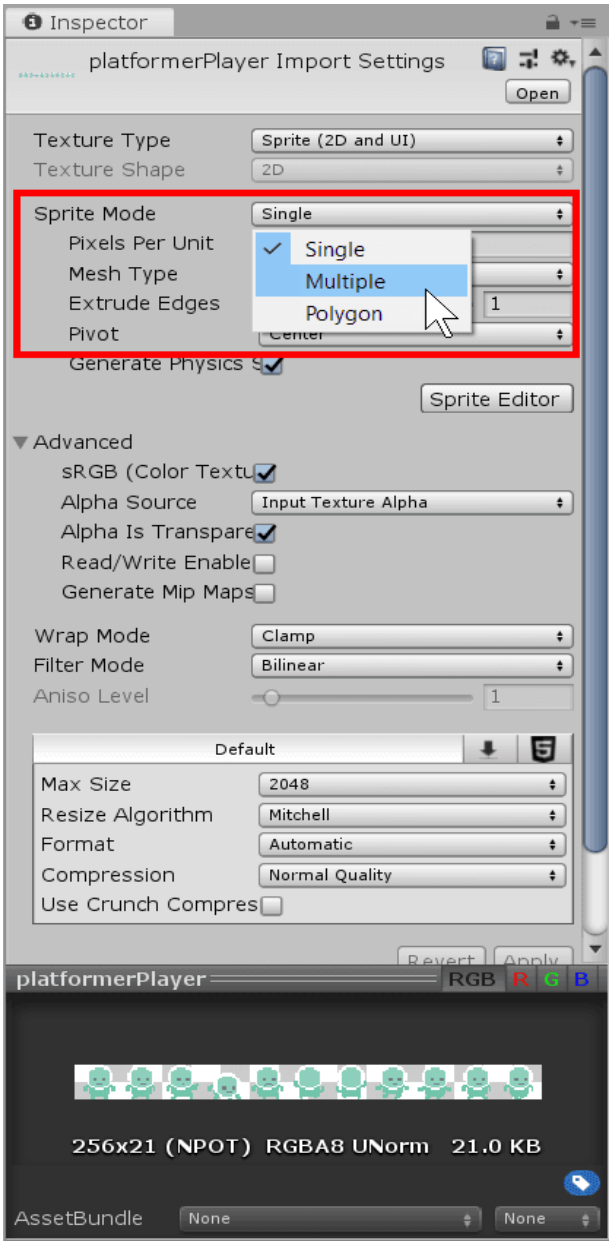
3.13. Spritesheets

Hasta ahora nuestros sprites estaban cada uno en una imagen distinta, pero, como ya habíamos anticipado, es frecuente encontrarlos agrupados en una única imagen, formando lo que se conoce como una "spritesheet".

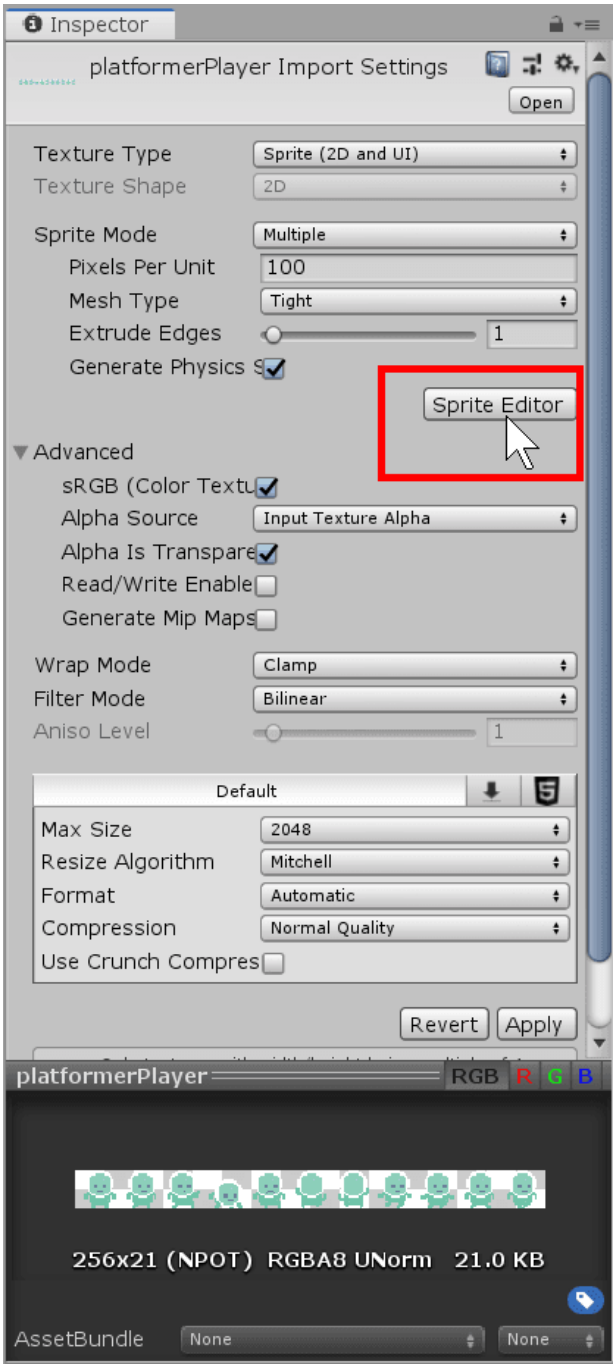
Para usar una "spritesheet", el primer paso será arrastrar la imagen a nuestra carpeta de sprites:



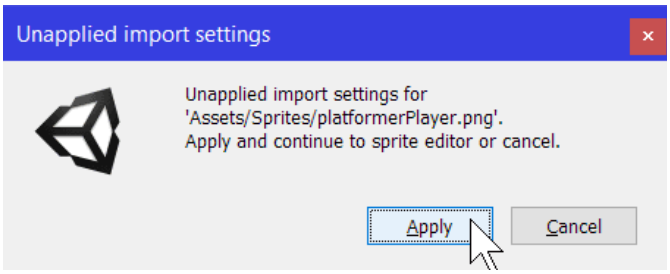
Y en el inspector cambiaríamos el "sprite mode" de "single" a "multiple":



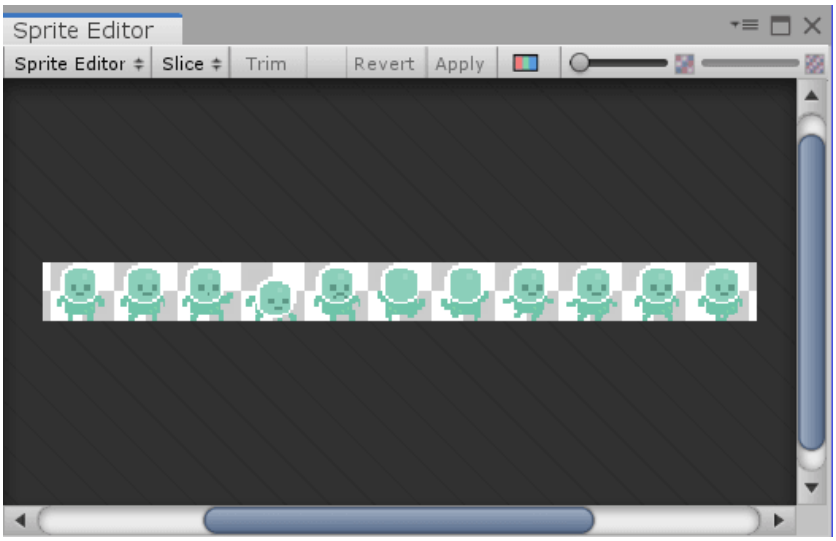
Y luego pulsaríamos el botón del "Sprite editor" para dividir esa imagen en trozos:



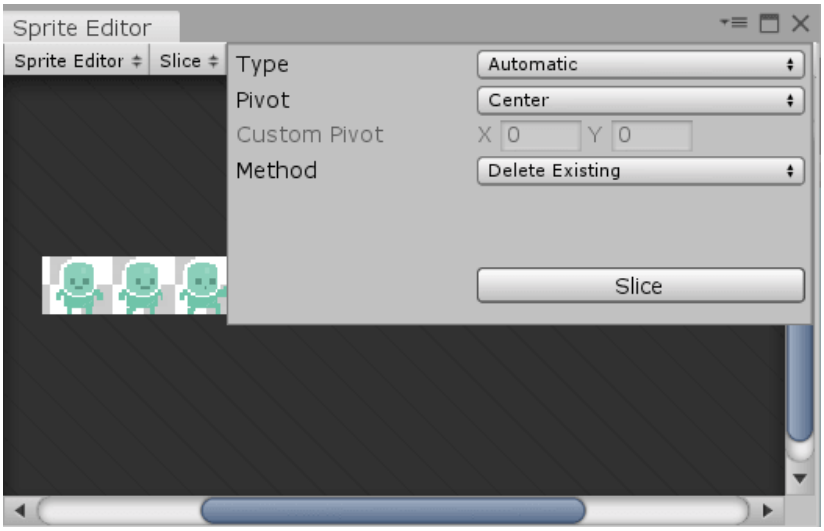
Si no hemos pulsado el botón de "aplicar cambios", se dará la oportunidad de guardarlos ahora:



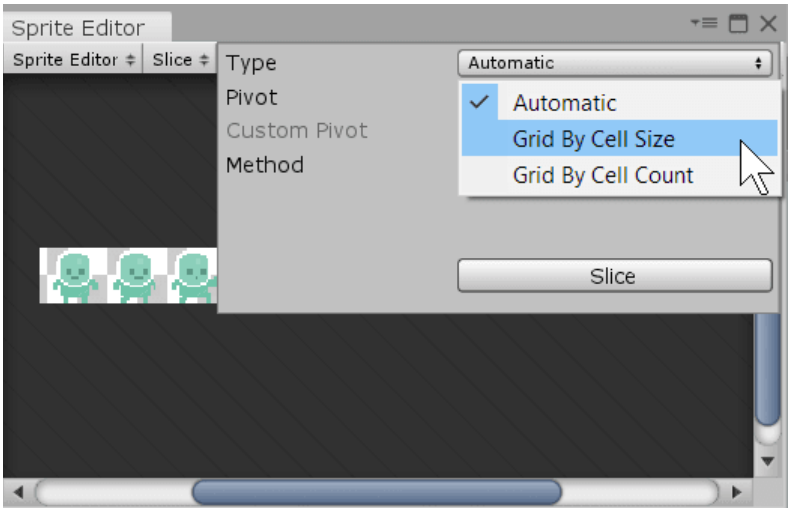
En el editor se nos mostrará nuestra imagen:



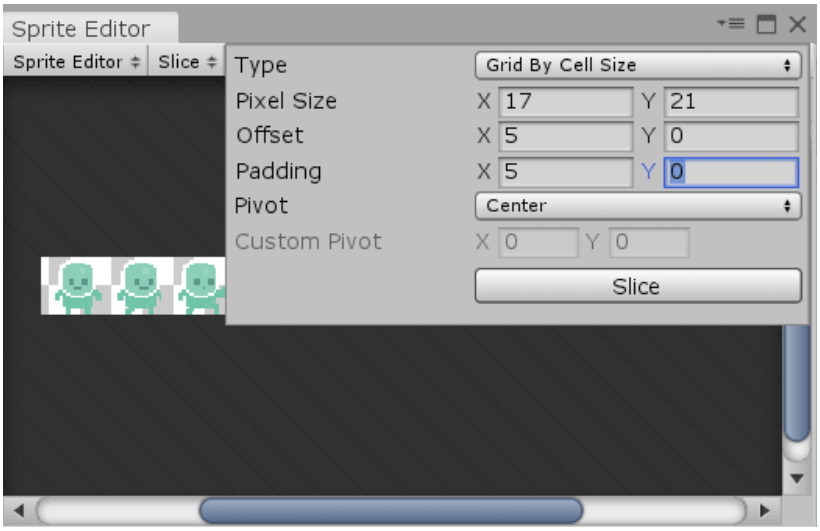
Y tendremos un botón "Slice" para indicar cómo queremos dividir la imagen:



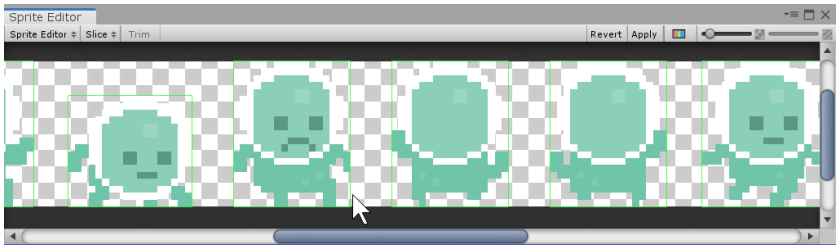
Además del método automático, podemos escoger indicar manualmente el tamaño de cada casilla o bien la cantidad de casillas:



Por ejemplo, en el modo de "tamaño de casilla" ("cell size"), indicaríamos el ancho y alto de cada imagen individual, la separación entre ellas ("padding") y, si no empieza en la esquina superior izquierda, el desplazamiento inicial ("offset"):

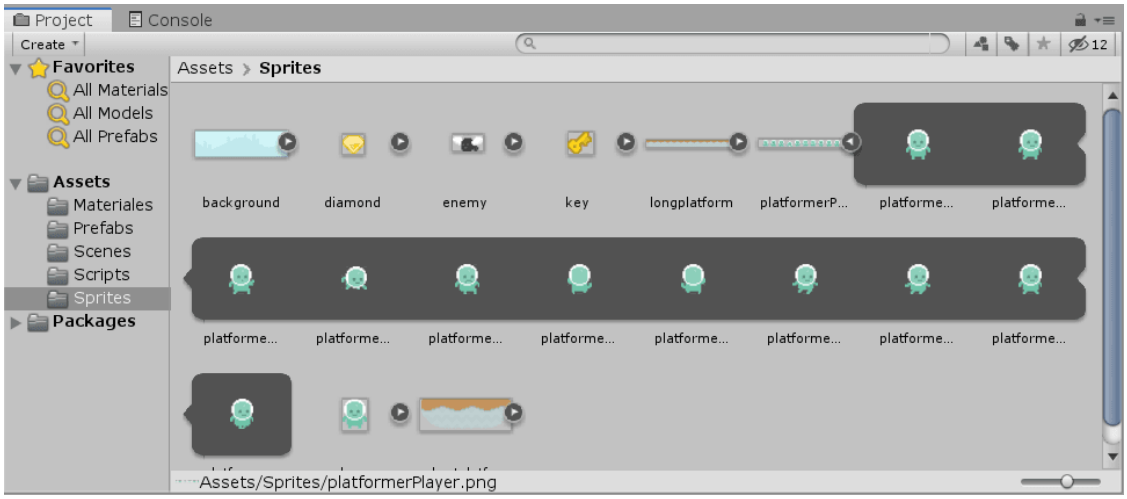


Aun así, el modo automático suele funcionar muy bien con imágenes sencillas. Tras pulsar el botón "Slice", si pulsamos la tecla "Ctrl", aparecerá un borde verde que nos permitirá ver qué se ha extraído como cada sprite individual.



Como alternativa, también podemos hacer clic en cualquiera de esos sprites.

Y en el panel inferior veremos una flecha que aparece junto a nuestra "spritesheet". Si pulsamos en esa flecha, veremos las imágenes individuales que se han extraído:

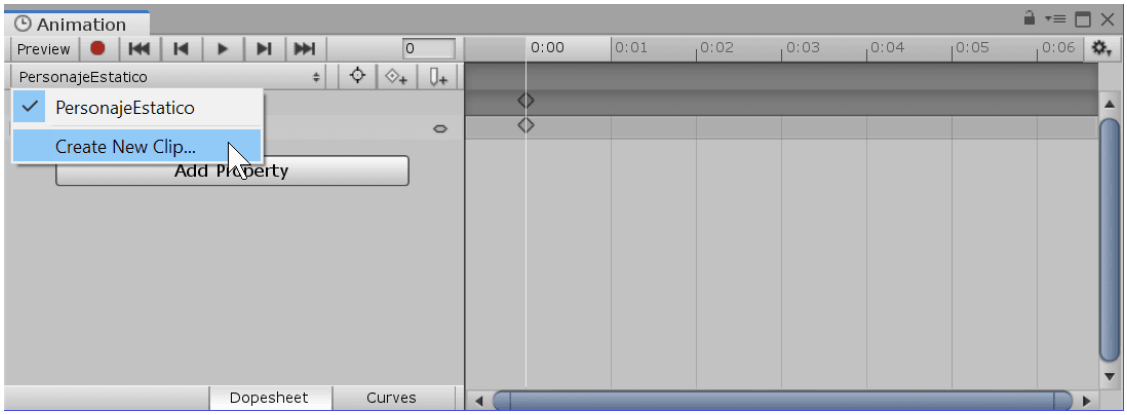


Ejercicio propuesto 3.13.1: Busca un "spritesheet" para el personaje de tu juego y extrae de él las imágenes individuales.

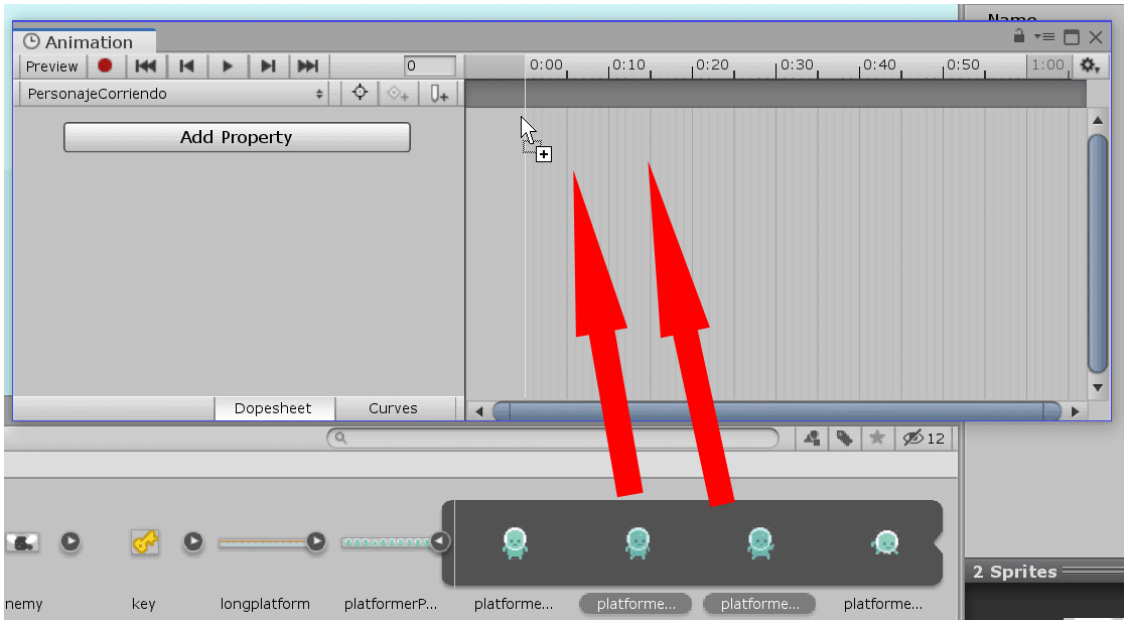
3.14. Distintas animaciones para un personaje

Ya sabemos cómo crear animaciones para un personaje. Ahora vamos a ver cómo encadenarlas. En primer lugar, partiendo de las imágenes que has extraído del "spritesheet", crea tres animaciones:

- Una (que puede estar formada por una única imagen) para cuando está estático. Su nombre puede ser "PersonajeEstatico". (Pasos: Menú Window / Animation / Animation, añadir, arrastrar el sprite a la línea de tiempo).
- Otra para cuando está avanzando hacia un lado, llamada "PersonajeCorriendo". (En la ventana Animation ya no aparecerá un botón para crear una nueva animación, pero se desplegará la opción "Create new clip" al pulsar en la lista de animaciones existentes:

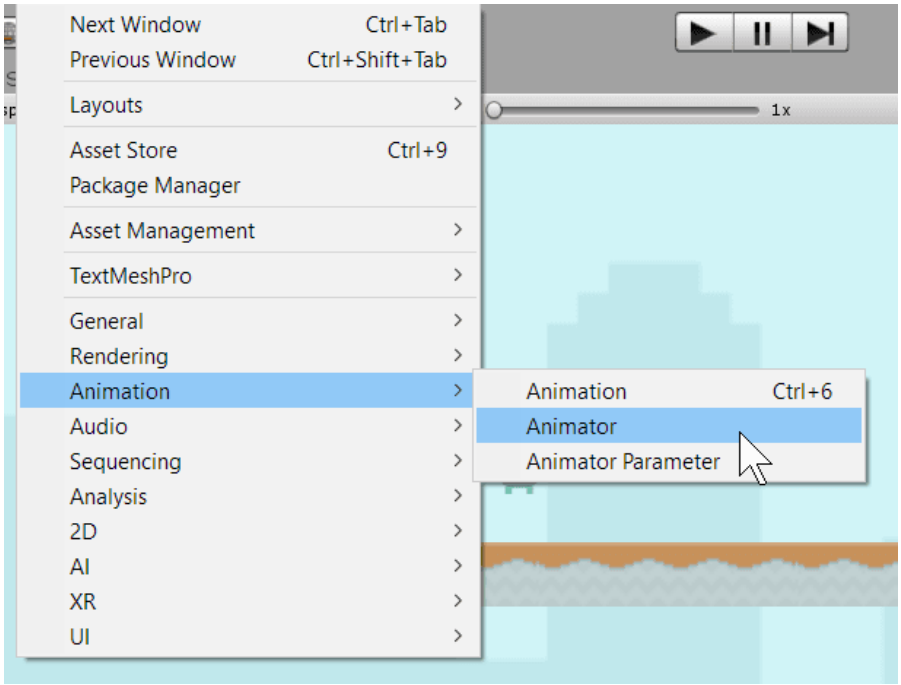


Y si queremos usar dos o más imágenes, podemos arrastrarlas a la vez a la línea de tiempo

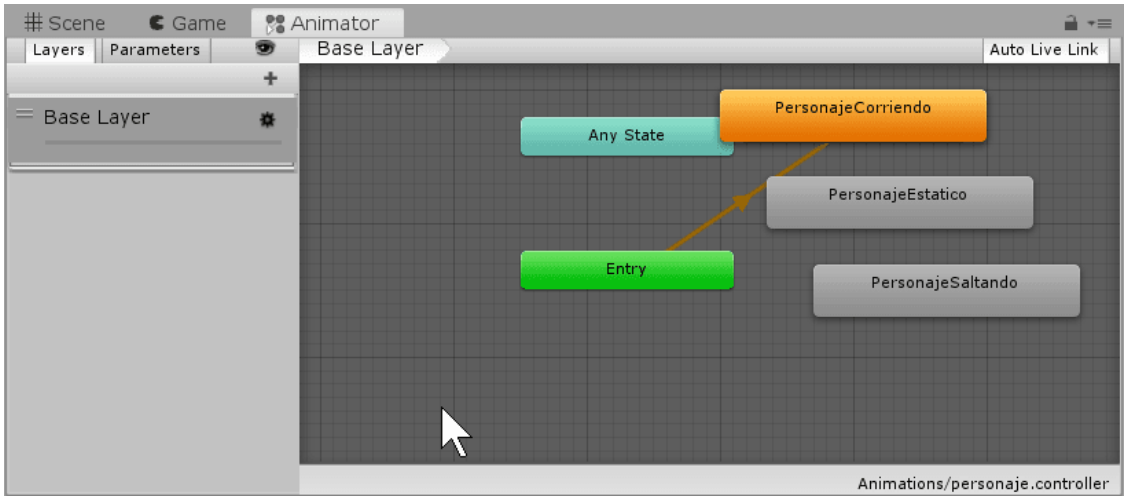


- Otra "PersonajeSaltando" para cuando se pulse el botón de salto.

Cuanto tenemos varias animaciones distintas, como en este caso, podemos indicar de cuál se debe volver a cuál (por ejemplo, tras saltar debe volver a estático). Para ello, abriremos la ventana del "Animator", en Windows / Animation:

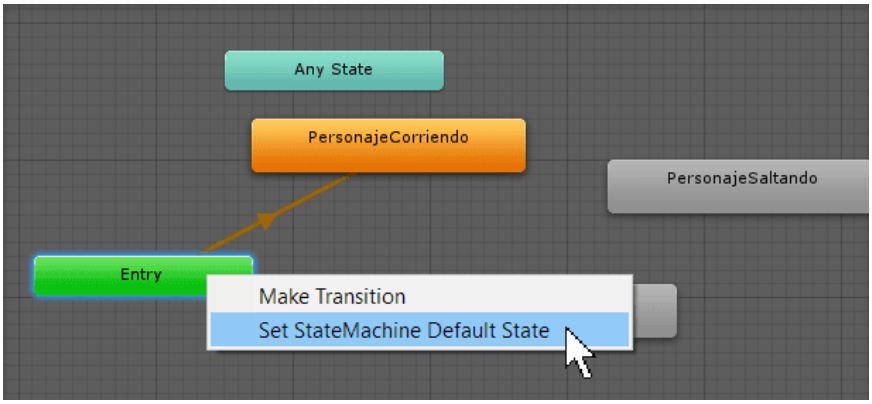


Y entonces aparecerán todas las animaciones que tenemos, y una de ellas estará conectada en color naranja a "Entry" para indicar que es la animación por defecto:

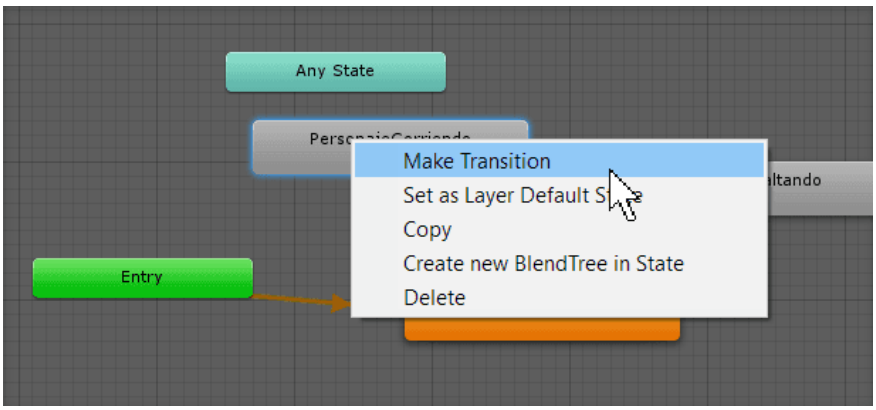


Es probable que esa no sea la que nosotros queramos que se tome como animación por defecto, sino que el sistema la haya tomado "al azar" (por ejemplo, por ser la primera en orden alfabético).

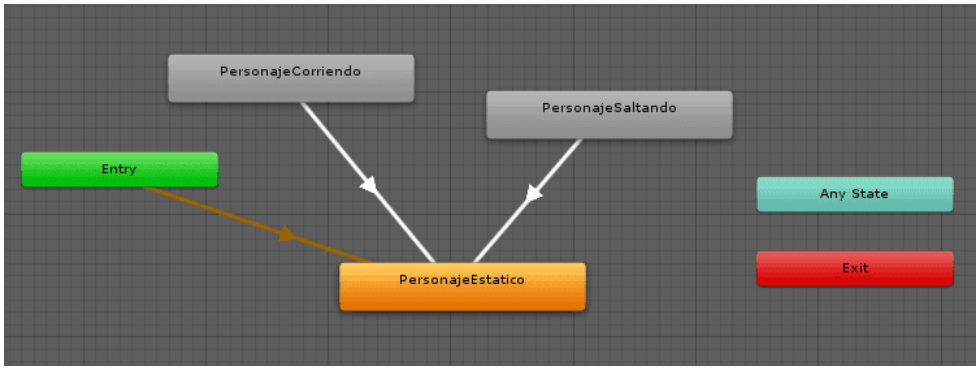
Si queremos cambiarla, deberemos pulsar el botón derecho en Entry y escoger "Set StateMachine Default State:



Y para indicar que de un estado se puede pasar a otro, usaremos el botón derecho sobre el estado inicial y escogeremos "Make transition", para dibujar una flecha hasta cualquier otro:



Por ejemplo, podríamos hacer que tras terminar la animación de correr se volviera a la "animación" de personaje estático:



Aun así, eso no es el caso exacto de nuestro juego: no querremos que se vuelva tras repetir la animación de correr una única vez, sino que la animación de correr se mantenga mientras el personaje esté corriendo. Para conseguirlo, añadiríamos un nuevo atributo:

```
private Animator anim;
```

En "Start()" le asignaríamos su valor:

```
anim = gameObject.GetComponent<Animator>();
```

Y en Update podríamos forzar el cambio de animación, por ejemplo así:

```
if ((horizontal > 0.1f) || (horizontal < -0.1f))
{
    anim.Play("PersonajeCorriendo");
}
```

Y veremos que al movernos hacia los lados, el personaje cambia de animación, y un breve instante después de dejar de movernos, vuelve a la animación de estático.

Aunque el efecto visual sería mejor aún si tuviéramos distintas animaciones para derecha e izquierda y reprodujéramos la una o la otra según hacia donde elija el jugador que se quiere desplazar:

```
if (horizontal > 0.1f)
{
    anim.Play("PersonajeCorriendoDerecha");
}
else if (horizontal < -0.1f)
{
    anim.Play("PersonajeCorriendoIzquierda");
}
```

Ejercicio propuesto 3.14.1: Crea una animación para cuando se mueva tu personaje y otra (quizá con un único fotograma) para cuando esté estático, y encadénalas.

3.15. Mapas de "tiles"

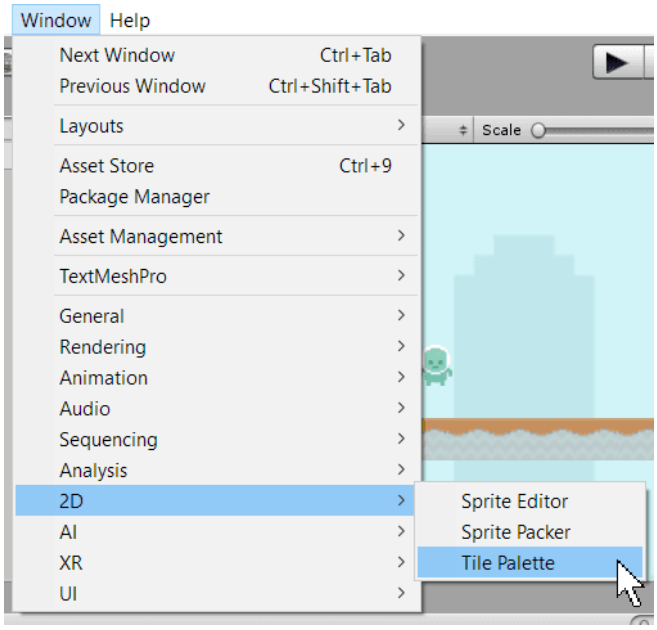
Las versiones más recientes de Unity incorporan un "editor de mapas de tiles", que permiten "dibujar" niveles a partir de "casillas repetitivas" ("tiles") individuales.

Vamos a comenzar por duplicar la escena "Nivel2" para crear una nueva escena "Nivel3".

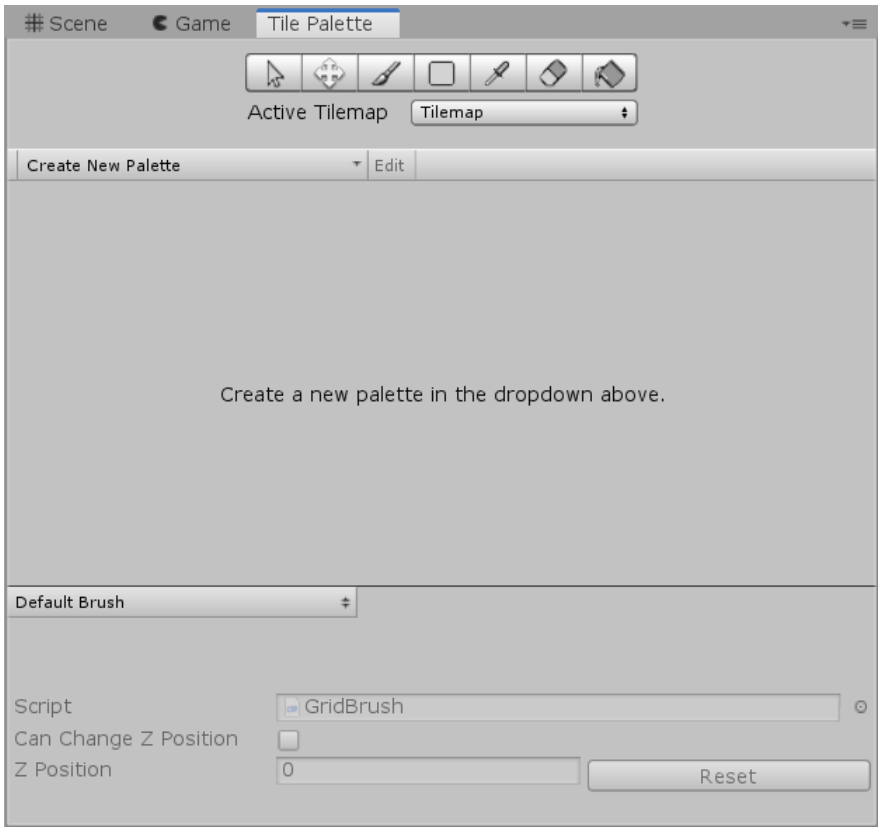
Los primeros pasos para crear el mapa de tiles, como ya hemos hecho con anterioridad, serían:

- Arrastrar la spritesheet a nuestra carpeta de sprites.
- En el "inspector", decir que se trata de un "sprite múltiple".
- Acceder al editor de sprites para partir ("slice"), de forma automática.
- Opcionalmente, hacer clic en los sprites individuales para asegurarnos de que se han recortado correctamente.

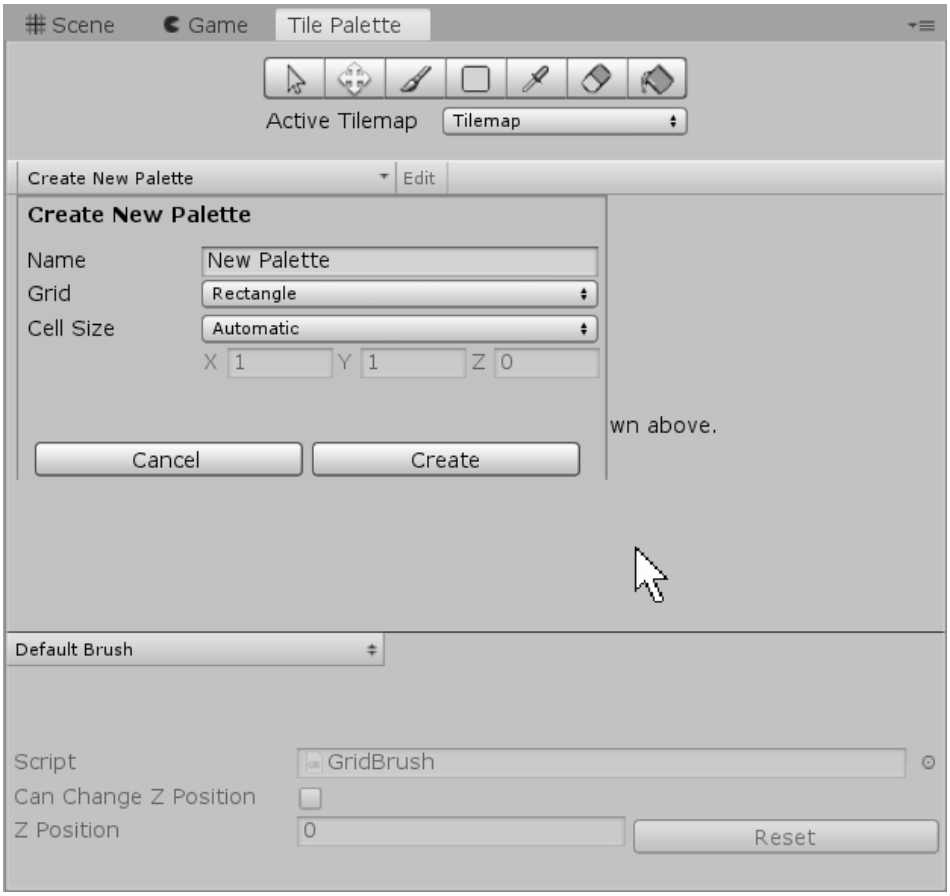
A continuación, desde crearemos una nueva "paleta de tiles", desde el menú "Window", opcion "2D / Tile palette":



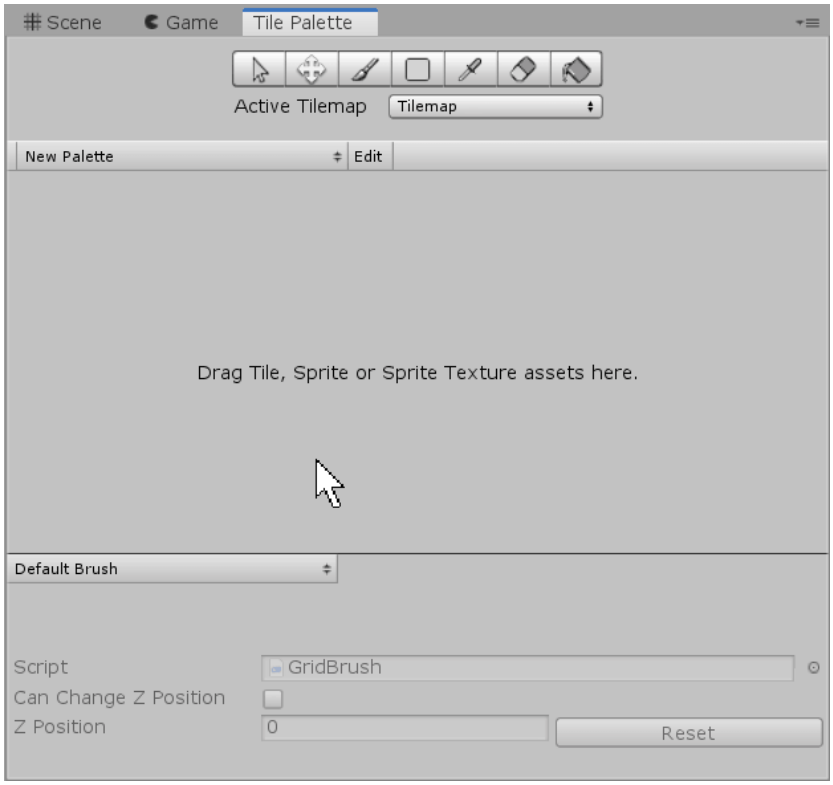
Y se nos dirá que usemos el desplegable superior para crear una paleta:



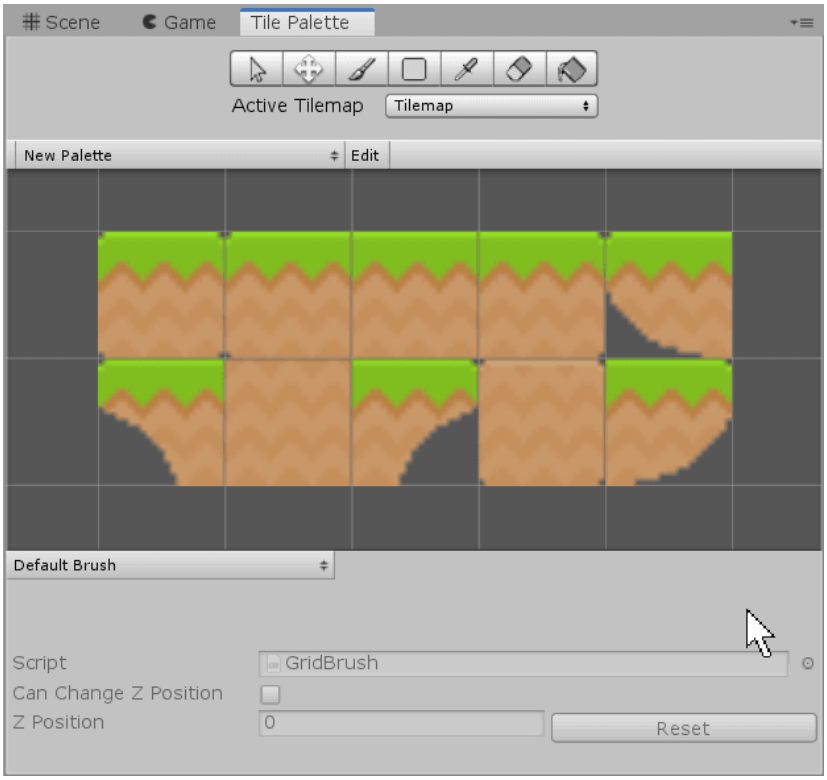
que puede ser rectangular y de tamaño automático:



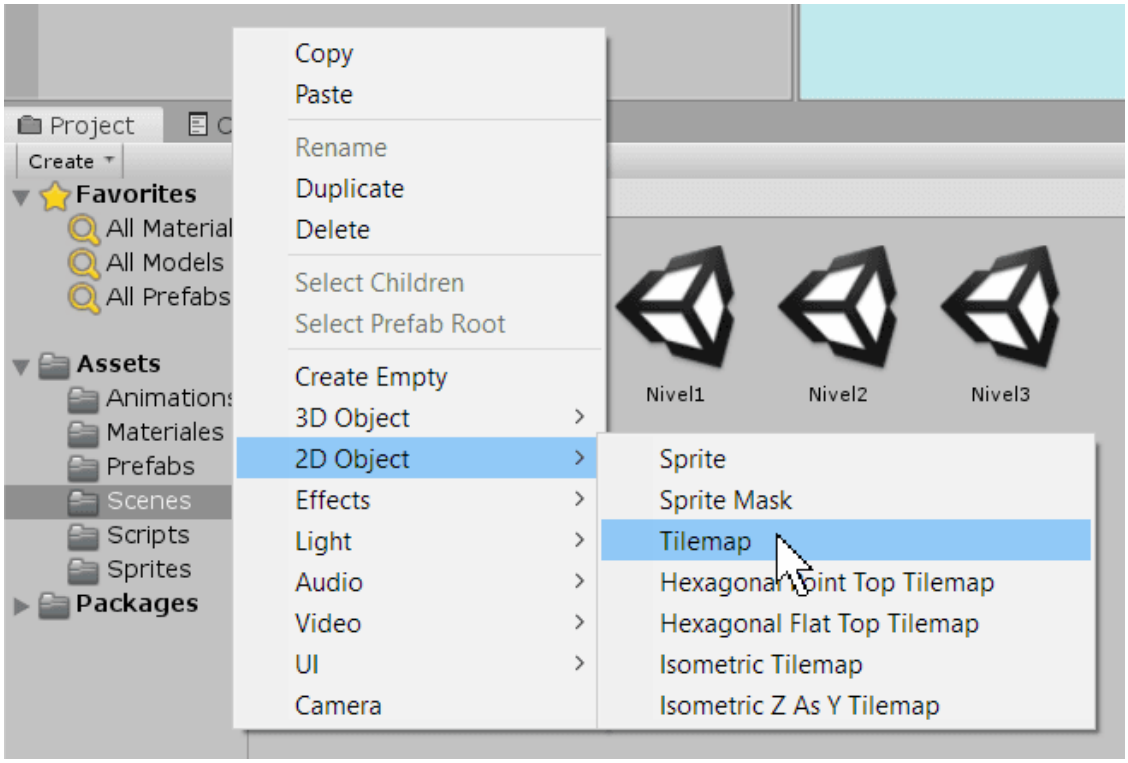
(Deberemos indicar también un nombre de fichero; por ejemplo, podríamos crear una subcarpeta dentro de "sprites"). Ahora se nos dirá que arrastremos un sprite o una textura:



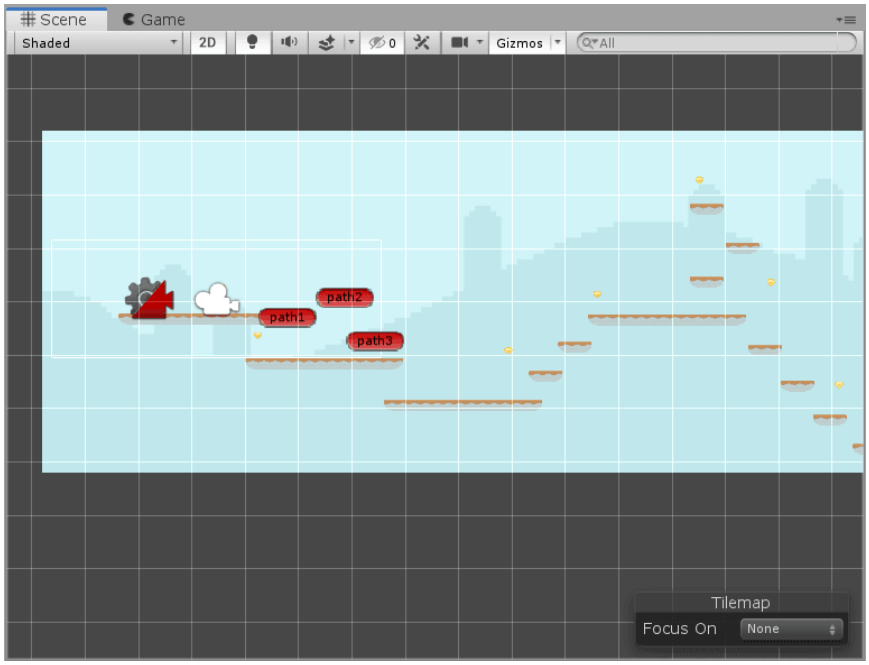
Y podemos arrastrar esa imagen múltiple que ya habíamos partido en fragmentos:



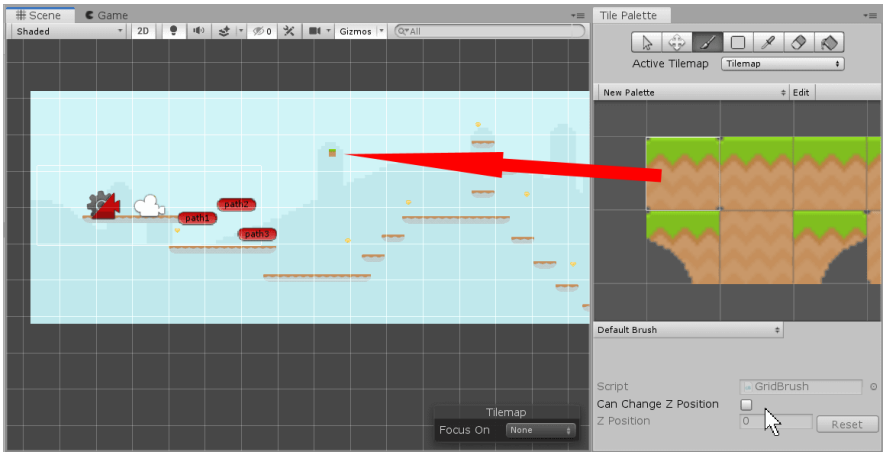
Ahora debemos superponer una rejilla por encima de nuestra escena. En la jerarquía, crearemos un "Tilemap" (de la categoría "2D Object"):



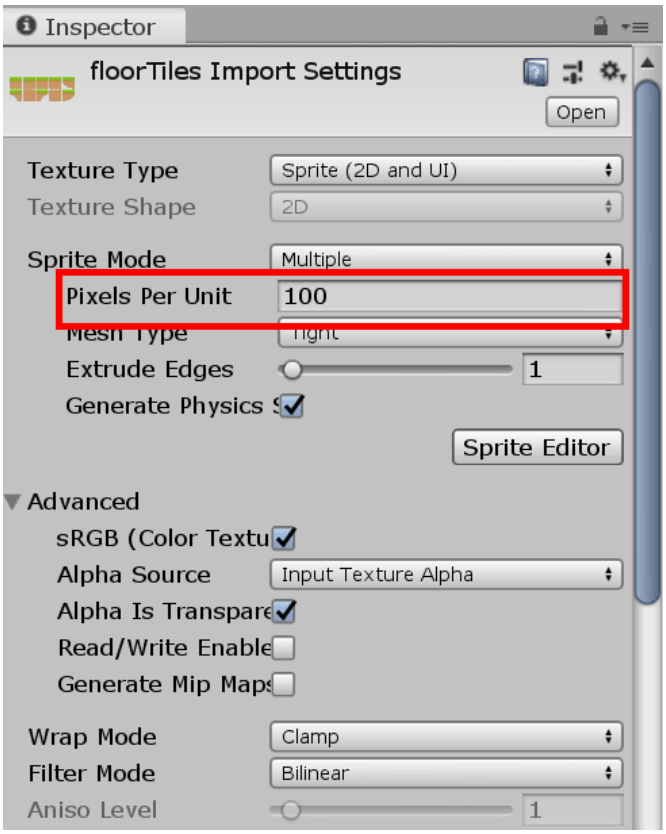
y esa rejilla aparecerá superpuesta:



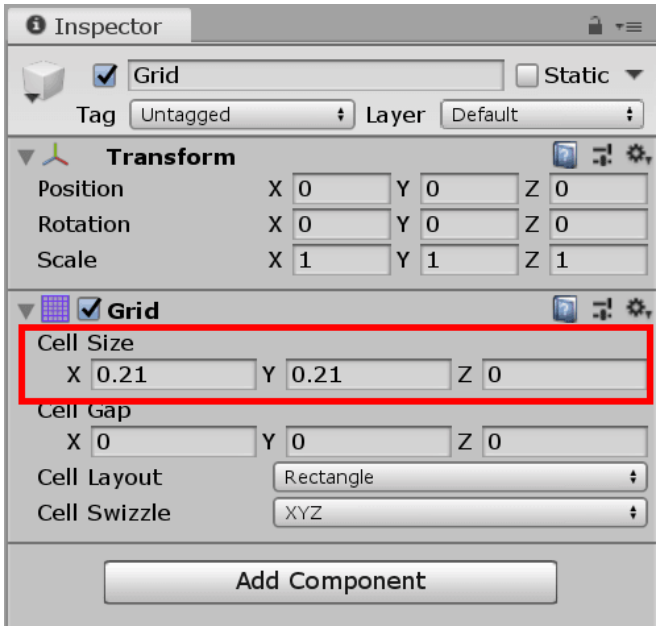
Ahora podemos colocar la ventana de la "tile palette" al lado de la escena, para poder "arrastrar" casillas de una a otra, pero es posible que nuestras "tiles" sean demasiado pequeñas para esa rejilla:



Para solucionarlo, podemos cambiar la resolución de la imagen ("pixels per unit"):

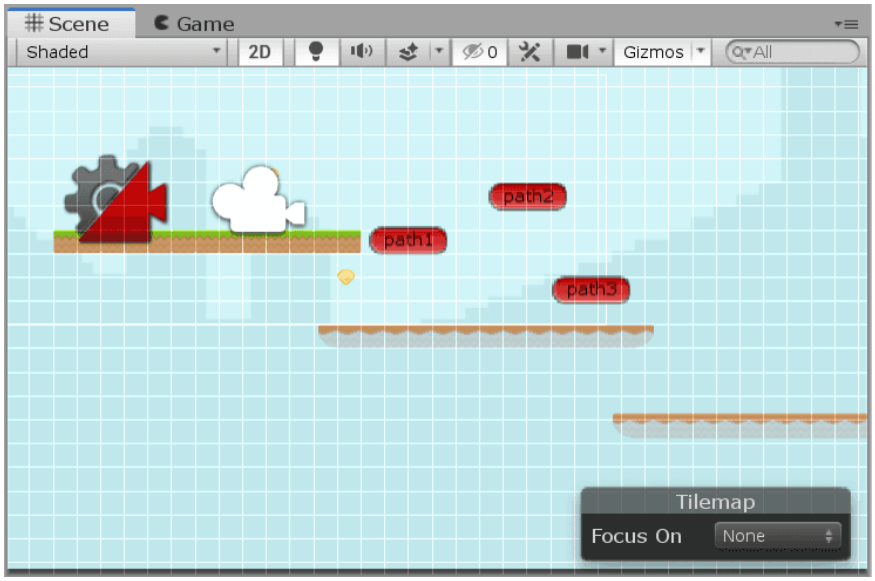


O bien el tamaño de la rejilla:

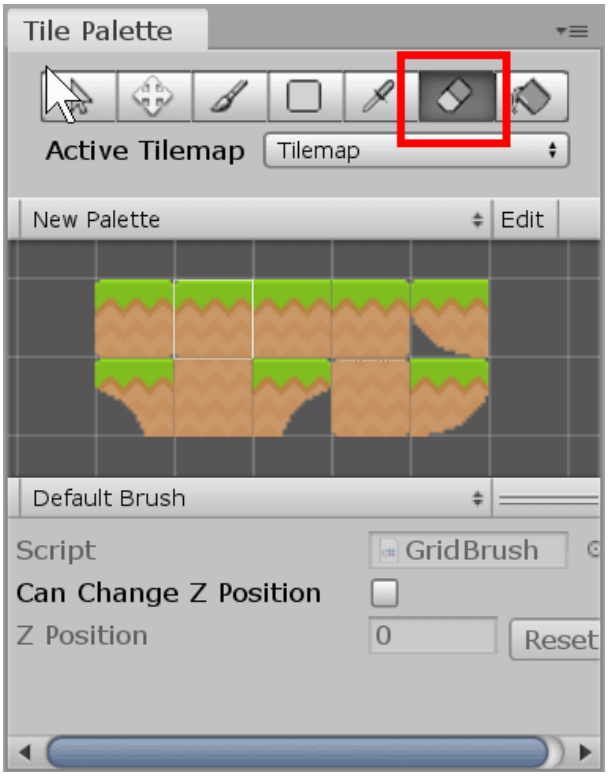


(o ambas cosas) para conseguir un tamaño cercano a la apariencia que deseamos:

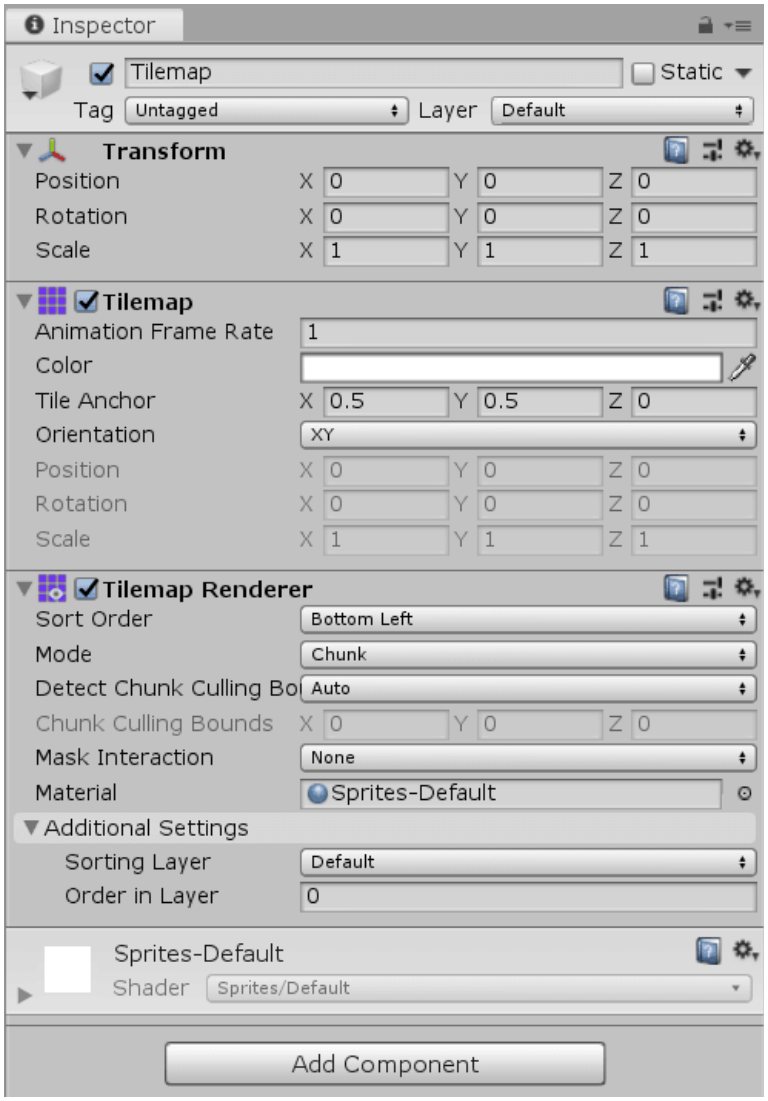
Para este juego, en que los "tiles" son de 21x21 píxeles, un tamaño de 0.21 unidades podría ser adecuado (porque la resolución



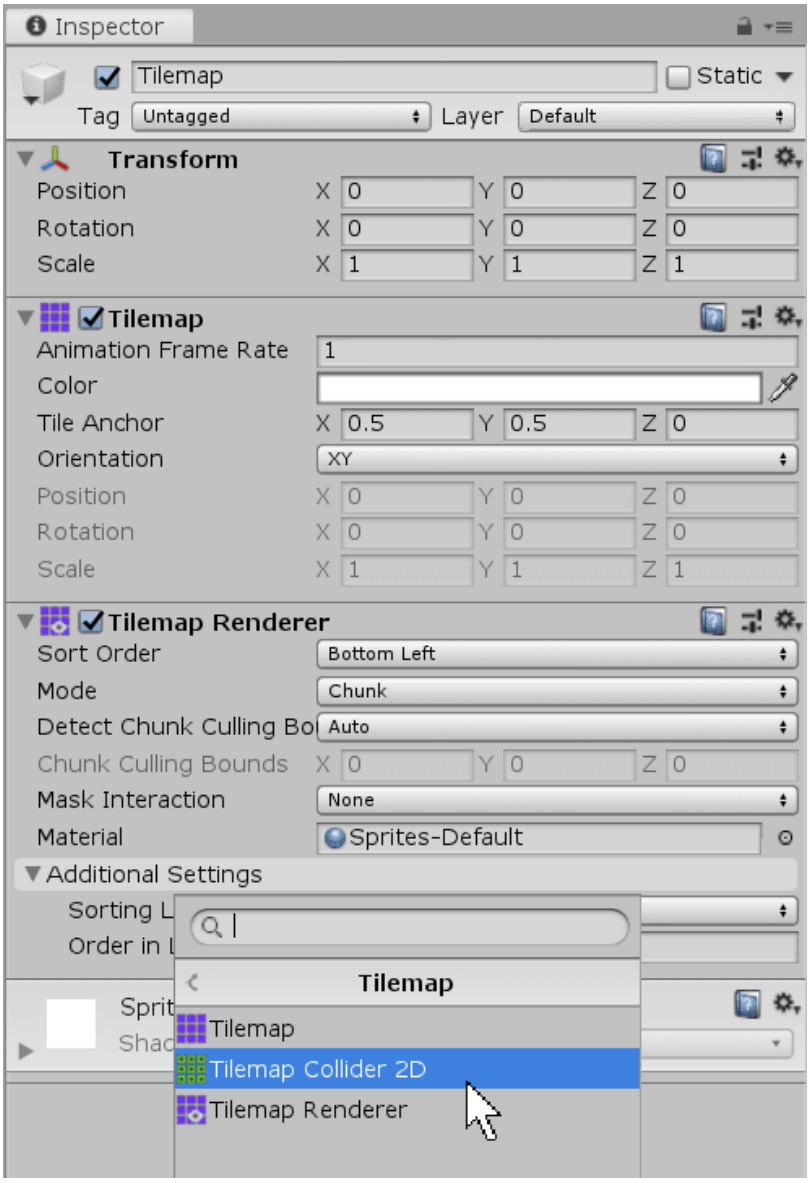
A partir de ahí, ya sólo es cuestión de escoger una casilla e ir "dibujando" sobre la rejilla. Si colocamos alguna en un sitio incorrecto, la podremos "borrar":



Sólo nos resta comprobar colisiones con esos "tiles", para que el personaje no caiga, sino que quede apoyado sobre ellos, pero eso es muy fácil: basta acudir al componente "TileMap" del "Grid" y pulsar el botón "Add component":



Para añadir un "tilemap Collider2D"



Y las casillas que hemos dibujado pasarán a ser casillas que no se podrán "pisar" y que bloquearán la caída del personaje.

Ejercicio propuesto 3.15.1: Crea un mapa repetitivo en una nueva escena de tu juego.

3.16. Sonidos en objetos que se destruyen

Si reproducimos un sonido asociado a un objeto y "destruimos" ese objeto, el sonido dejará de escucharse en ese momento. Eso puede que no se llegue a escuchar nada o que se interrumpa con un "chasquido".

Una alternativa para añadir sonidos a ciertos objetos, como explosiones, es usar "PlayClipAtPoint", al que se le indica en qué posición del espacio queremos que se reproduzca un cierto sonido:

```
AudioSource.PlayClipAtPoint(sonido, transform.position);
```

Eso reproduciría el clip asociado a la variable "sonido" en la posición actual del objeto que contiene ese script.

Aun así, como el componente "transform" se destruirá al eliminar el objeto, una alternativa más fiable es reproducir el sonido en la posición asociada a la cámara:

```
AudioSource.PlayClipAtPoint(sonido, Camera.main.transform.position);
```

Ejercicio propuesto 3.16.1: Añade algún sonido a tu juego, por ejemplo en el momento de recoger los ítems.