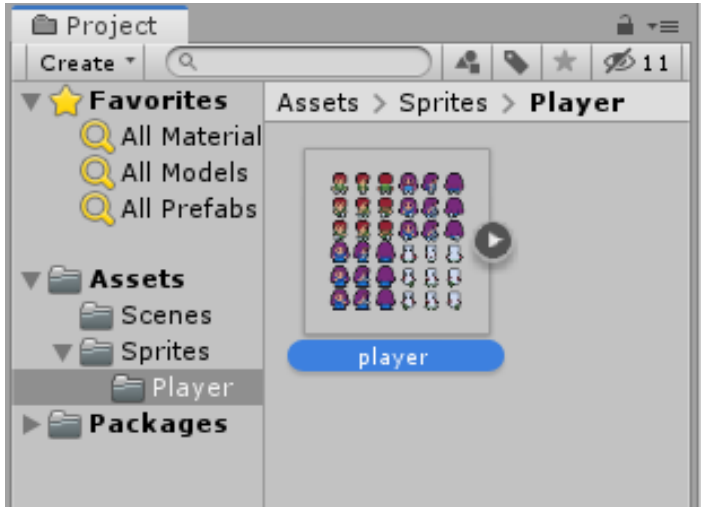
Construyendo un RPG

Índice

Implementando el personaje

- Vamos a empezar de cero
  - Creamos un empty (llámalo Player)
  - Le añadimos un componente Sprite Renderer
  - Crea el directorio Sprites\Player
  - Importa al proyecto la hoja de sprites correspondiente

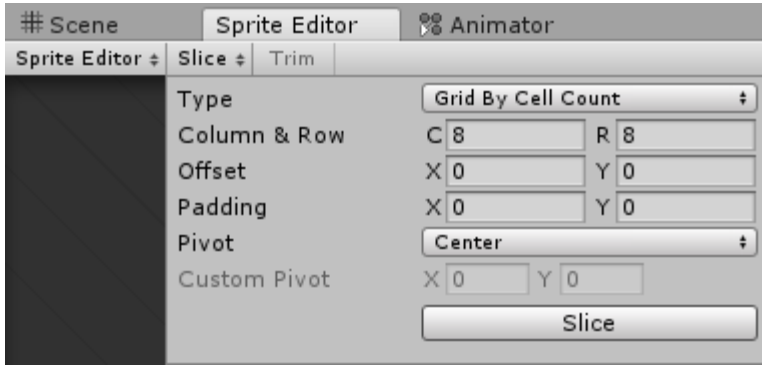


Extrayendo los sprites del personaje

- Vamos a extraer los sprites de la hoja usando el Sprite Editor:
  - Establece el tipo de la textura a Sprite (2D and UI)
  - Establece el modo de sprite a Multiple
  - Cambia el modo de filtrado de “Bilinear” a “Point (no filter)”
  - Establece los “Pixels per Unit” a 16
    - Indica que 16 píxeles del sprite equivalen a una unidad del sistema de coordenadas de Unity
  - Cambia el tipo de compresión a None
  - Pulsa Apply y abre el Sprite Editor

Extrayendo los sprites del personaje

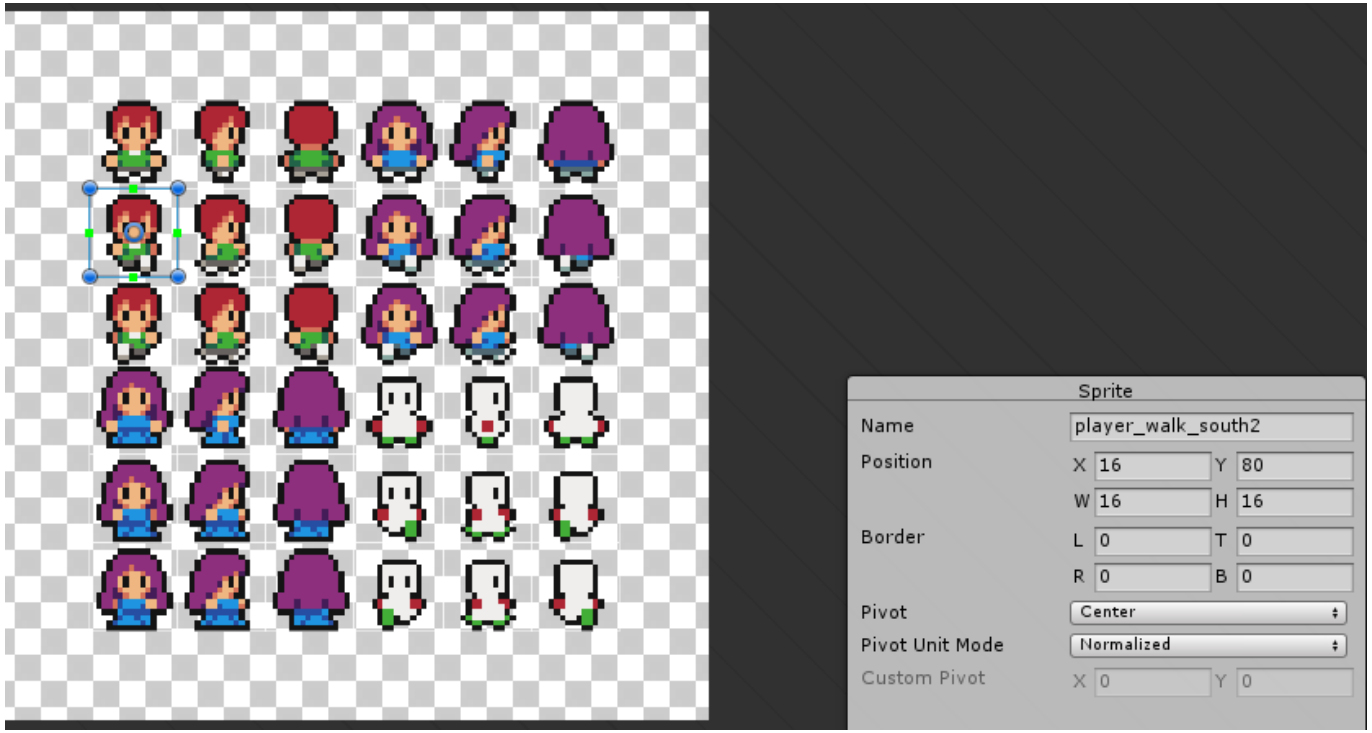
- El Sprite Editor ofrece varias herramientas para extraer los sprites de una imagen. Para nuestro ejemplo, vamos a seleccionar el tipo de Slice “Grid By Cell Count”, y le vamos a pedir que genere 8 x 8 celdas:



- Pulsa Apply y cierra el Sprite Editor

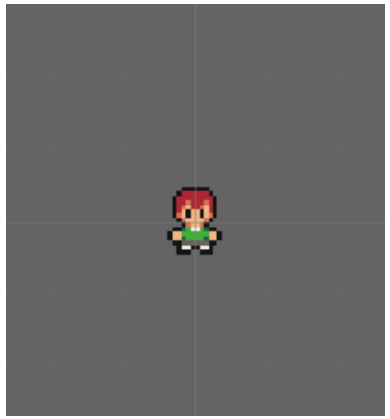
Extrayendo los sprites del personaje

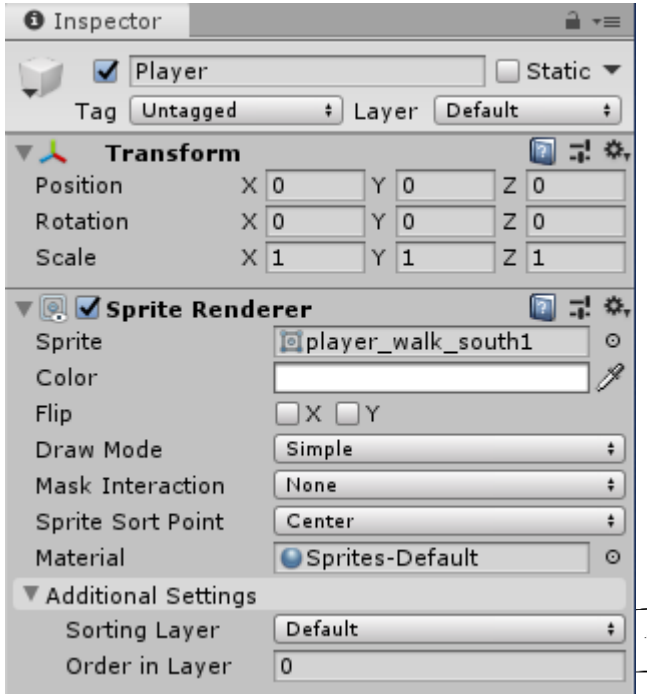
- Tras comprobar que el recortado se ha hecho correctamente, puedes dar nombre a cada sprite, para facilitar su uso más adelante



Extrayendo los sprites del personaje

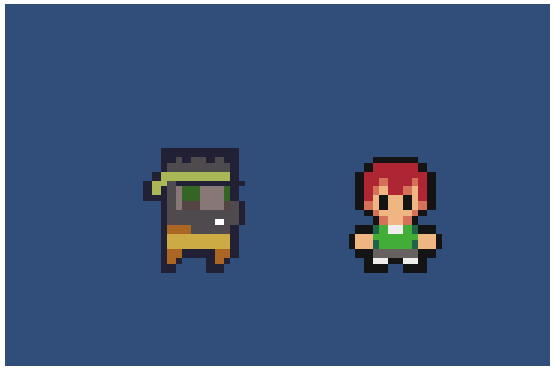
- En la ventana del proyecto ahora se pueden ver los sprites individuales del personaje
- Podemos seleccionar uno de los sprites para visualizar al jugador
  - Selecciónalo en la opción Sprite del Sprite Renderer





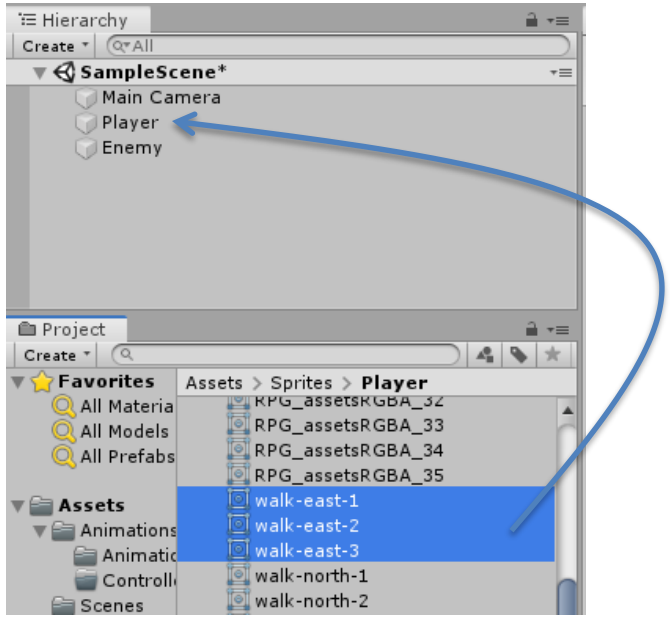
Preparando al enemigo

- Repite el proceso para el spritesheet del enemigo



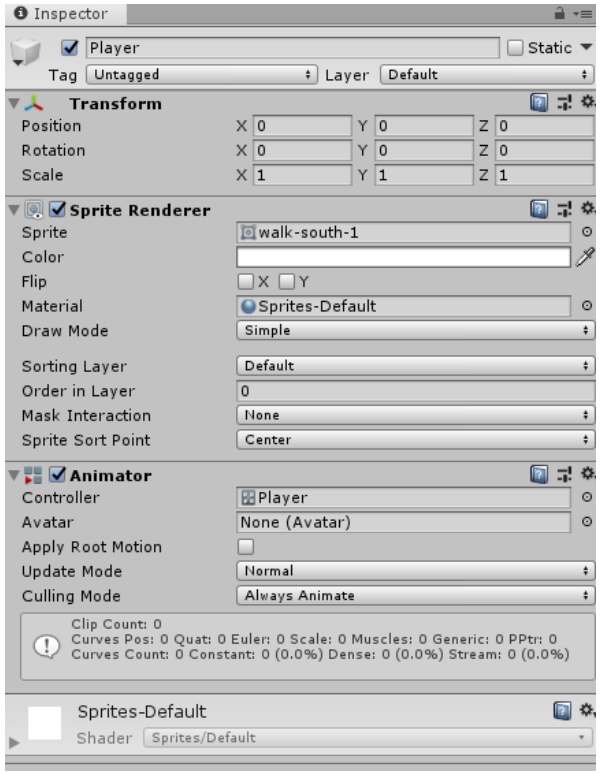
Animaciones

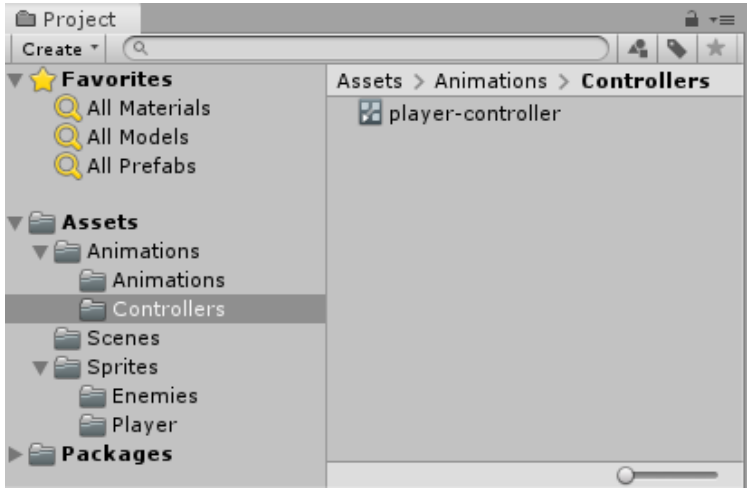
- Crea un directorio para guardar las animaciones
  - Y, dentro de él, dos directorios más: Animations y Controllers
  - Para crear la animación de andar hacia el este, seleccionando sus sprites individuales, y arrástralos sobre el jugador
  - Unity pedirá un nombre para el clip de animación (usa, p.e. player-walk-east.anim)



Animaciones

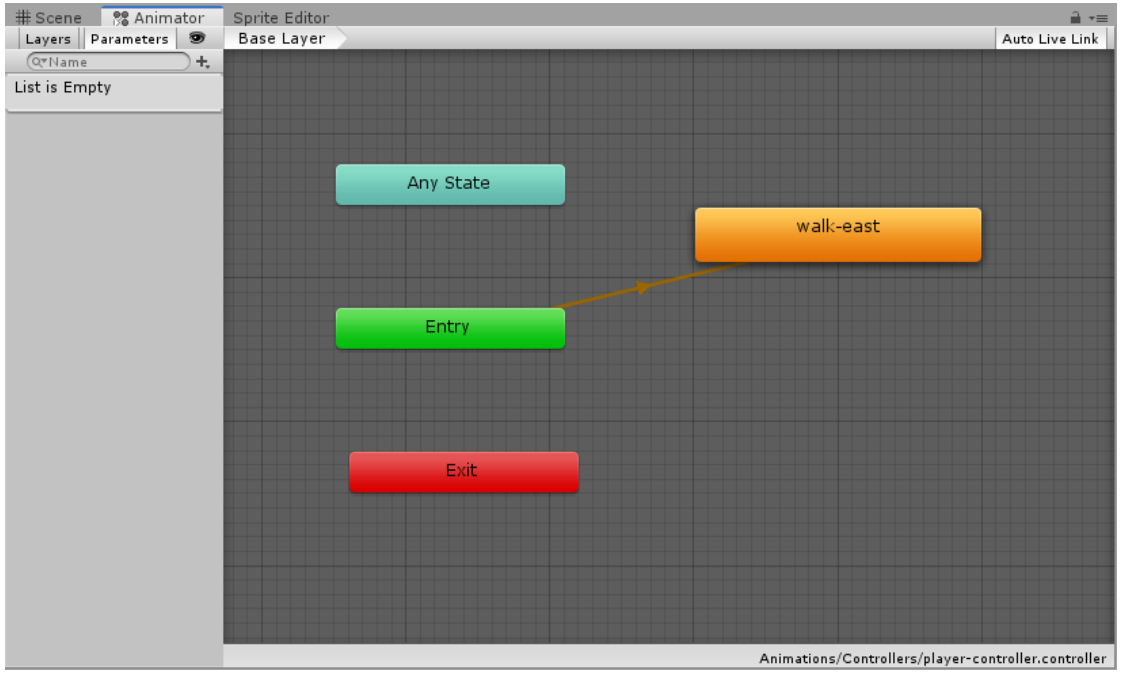
- Al haber creado la primera animación de un game object, Unity le añade un componente Animator, y también crea un nuevo Animator Controller
- Dale un nombre más descriptivo y mueve el controlador a su directorio





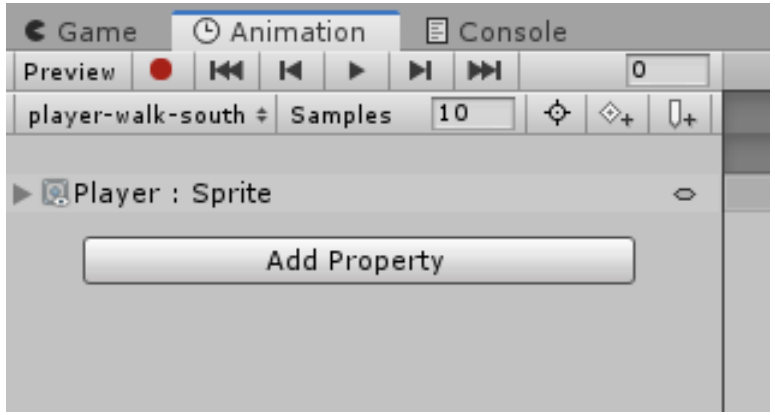
Configurando el controlador

- Haz doble clic sobre el Animation Controller:
- Estados por defecto:
  - Any State: se usa cuando queremos que una transición se pueda iniciar desde cualquier estado
  - Entry: estado inicial, enlazado con el estado por defecto
  - Exit: se usa cuando hay máquinas de estado anidadas



Configurando el controlador

- Repite el proceso para crear las animaciones de player-walk-north y player-walk-south
- Ajusta la velocidad de reproducción de los clips (es demasiado rápida)

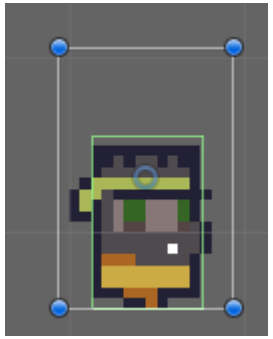


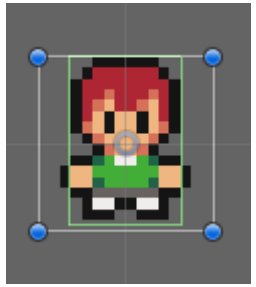
Configurando el controlador

- Repite el proceso para el enemigo

Colliders

- Los colliders se usan en la simulación física como sustitutos del objeto simulado (que puede tener una geometría compleja)
- Se usan principalmente para el cálculo de colisiones
- Unity tiene varios tipos de colliders 2D: cajas, círculos, polígonos, etc.
- Añade un box collider 2D tanto al jugador como al enemigo
  - Puedes ajustar los colliders con el botón “Edit Collider”, y previsualizar la animación en el panel Animation



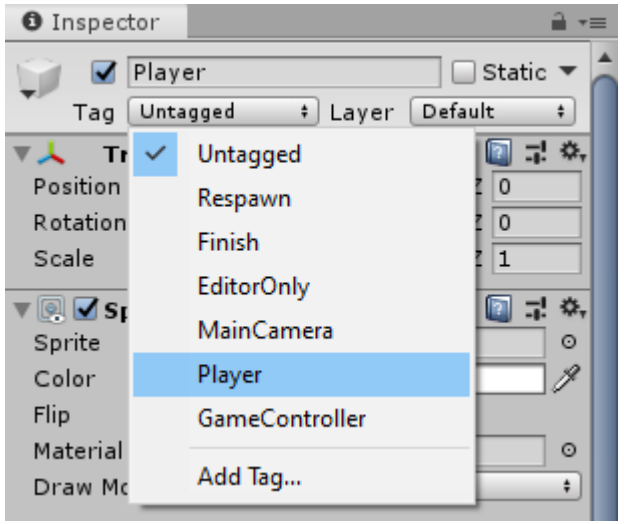


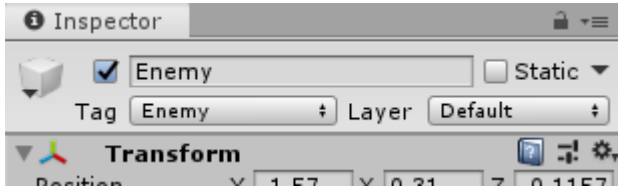
Física

- Para hacer que a un game object le afecten fuerzas, como la gravedad, debe tener un componente Rigidbody 2D
- Añádeselo al jugador
- Tipos de cuerpo:
  - Dynamic: participa en la simulación física, interacciona con otros objetos
  - Kinematic: no le afectan las fuerzas. Se mueven por script a través de su Transform
  - Static: el objeto no se mueve (pero se tiene en cuenta para el cálculo de colisiones)
- Establece las siguientes propiedades:
  - Mass: 1, Linear Drag: 0, Angular Drag: 0, Gravity Scale: 0,

Tags

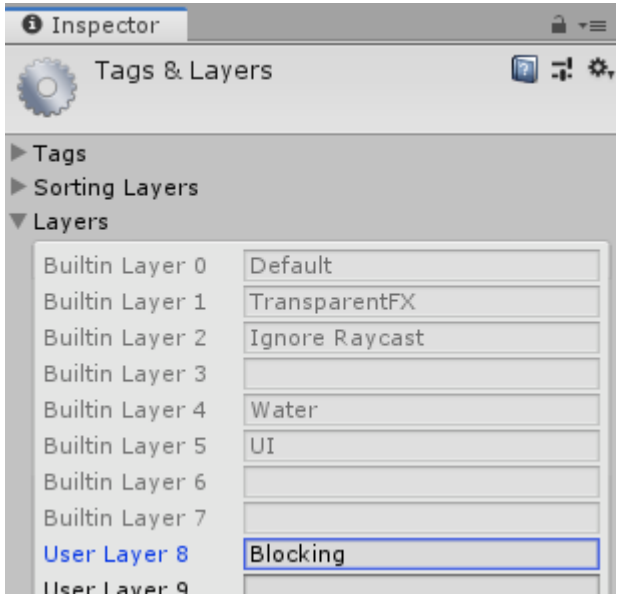
- En Unity, los tags nos permiten etiquetar los game objects para poder identificarlos por código en tiempo de ejecución
- La etiqueta “Player” está predefinida, por lo que se puede asignar directamente desde el Inspector
- Usa Add Tag… para asignarle la etiqueta “Enemy” al enemigo





Layers

- Las capas nos permiten organizar los game objects de una escena
  - Por ejemplo, podemos hacer que el cálculo de colisiones tenga en cuenta únicamente los game objects de una capa determinada
- En el desplegable “Layers” del Inspector, añade la capa “Blocking” como la capa de usuario 8
- Añade tanto al enemigo como al jugador a la nueva capa



Sorting Layers

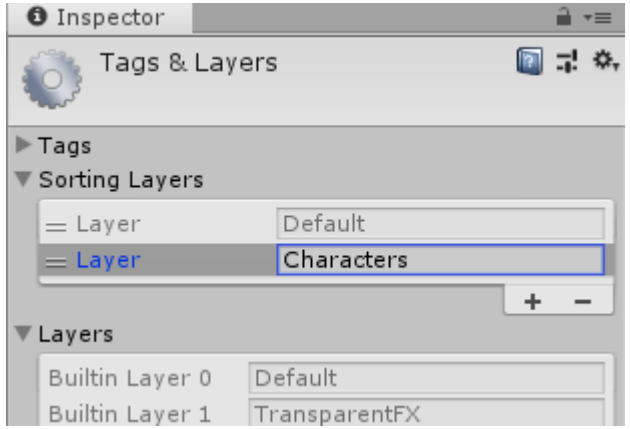
- Las capas de ordenación son distintas a las vistas antes
- Las capas de ordenación indican el orden en el que se deben dibujar los sprites

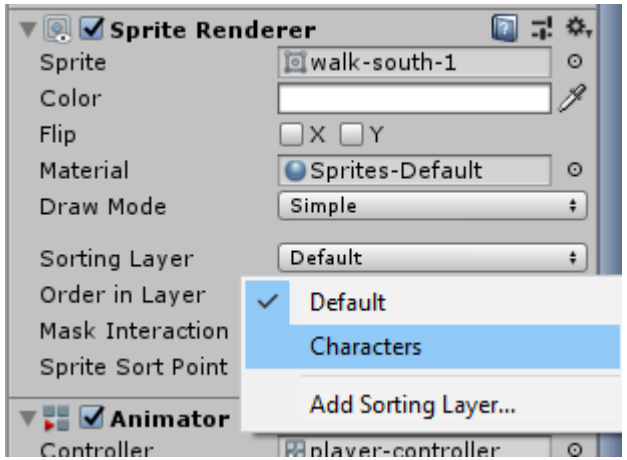
Final Fight, Capcom. 1989

Shadow of the Beast, Psygnosis. 1989

Sorting Layers

- Vamos a definir, en principio, dos capas: una para el suelo y otra para el jugador y el enemigo
  - Añade una capa de ordenación “Characters” desde el campo “Sorting Layer” del Sprite Renderer
  - Pon al jugador y al enemigo en la nueva capa de ordenación





Prefabs

- Haz sendos prefabs con el jugador y con el enemigo
- Esto nos permitirá:
  - Generar instancias de los objetos por código
  - Modificar todas las instancias del prefab sin tener hacerlo una por una
- Crea una carpeta para los prefabs

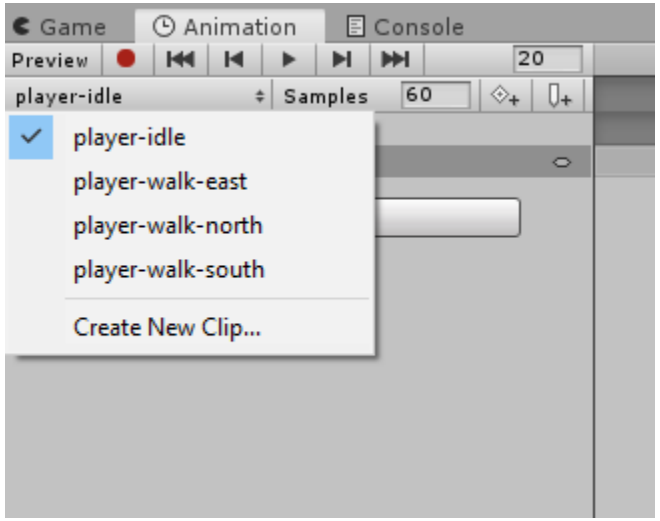
Código para el movimiento del personaje

- Vamos a añadir código para mover al personaje en la escena
- Crea un directorio Scripts
- Crea un script MovementController y asígnaselo al personaje

```
public class MovementController : MonoBehaviour {
    public float movementSpeed = 3.0f;
    Vector2 movement = new Vector2();
    Rigidbody2D rb2d;
    void Start() {
        rb2d = GetComponent<Rigidbody2D>();
    }
    private void FixedUpdate() {
        movement.x = Input.GetAxisRaw("Horizontal");
        movement.y = Input.GetAxisRaw("Vertical");
        movement.Normalize();
        rb2d.velocity = movement * movementSpeed;
    }
}
```

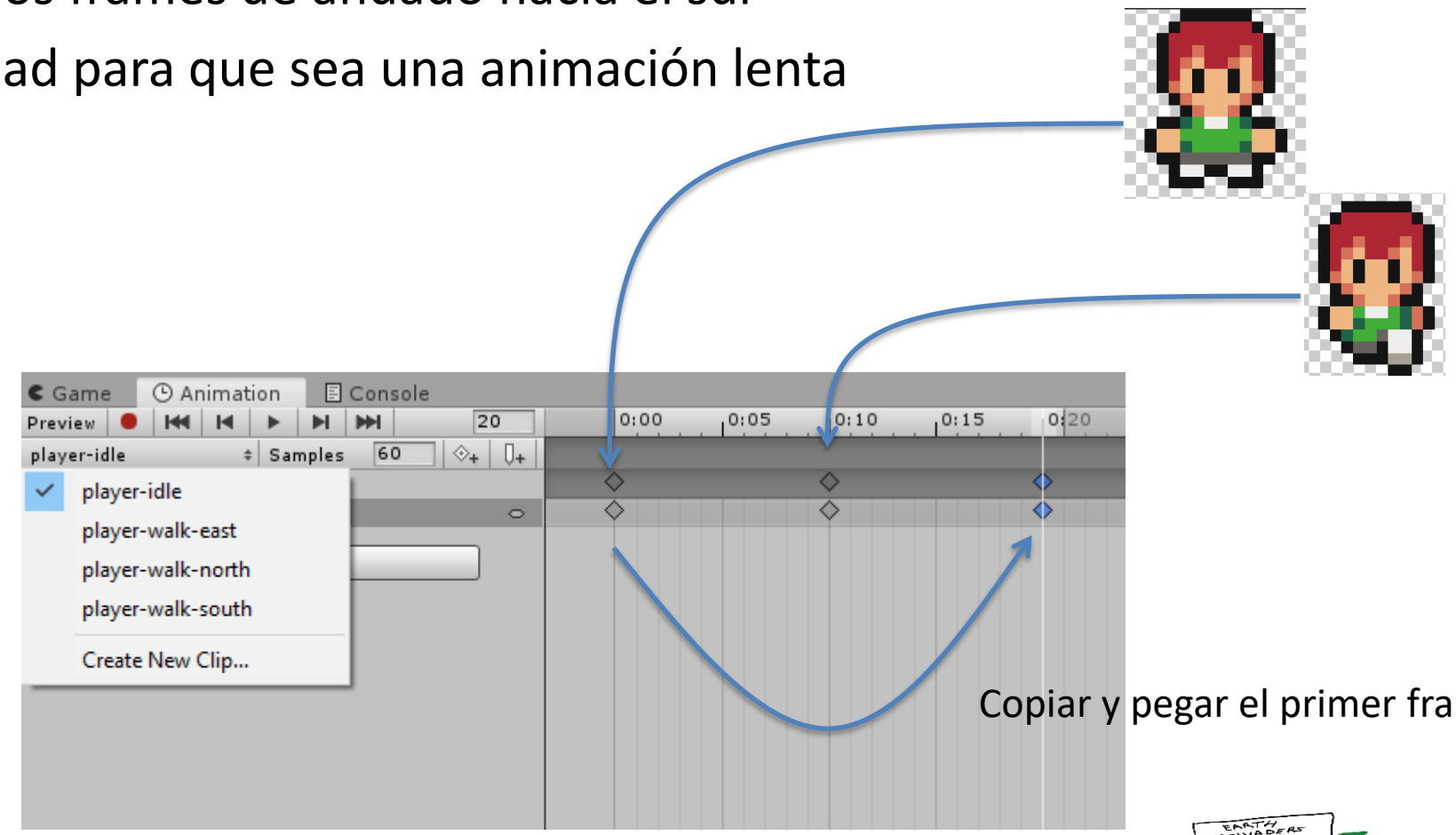

Completando las animaciones

- Vamos a añadir alguna animación más al personaje y preparar las transiciones entre los estados
- Primero, vamos a preparar una animación de espera (reutilizando dos sprites de andado, ya que no tenemos dicha animación)
- En primer lugar, vamos a crear un clip vacío
  - Selecciona el personaje
  - Abre el panel Animation
  - Despliega la lista de animaciones
  - Selecciona Create New Clip…
  - Crea el clip player-idle



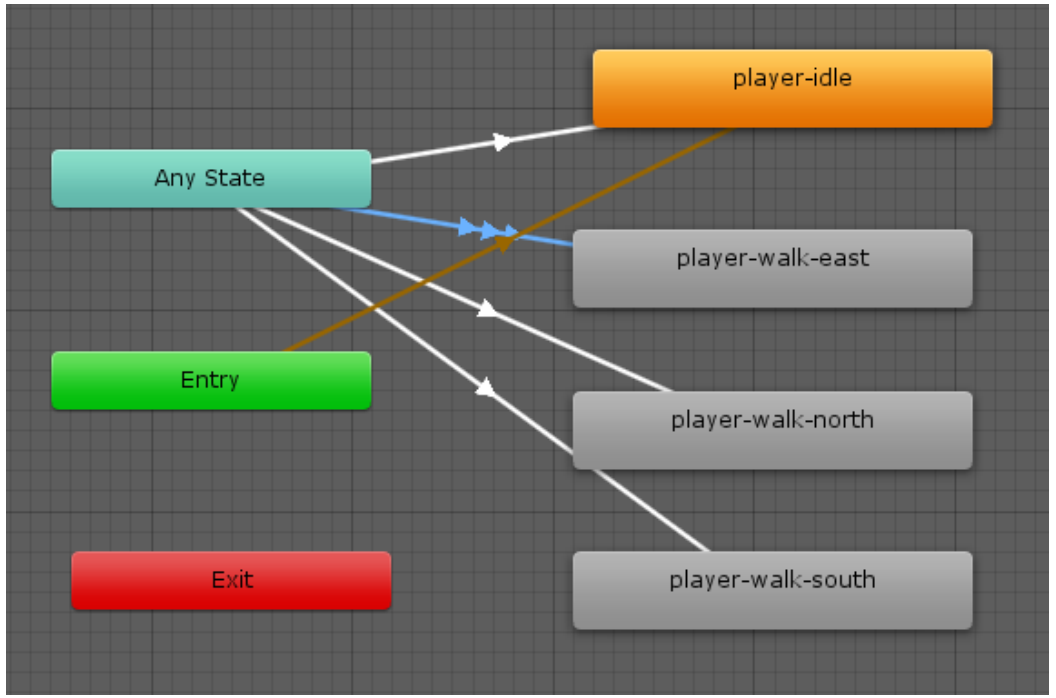
Completando las animaciones

- Ahora arrastra dos frames de andado hacia el sur
- Ajusta la velocidad para que sea una animación lenta



Completando las animaciones

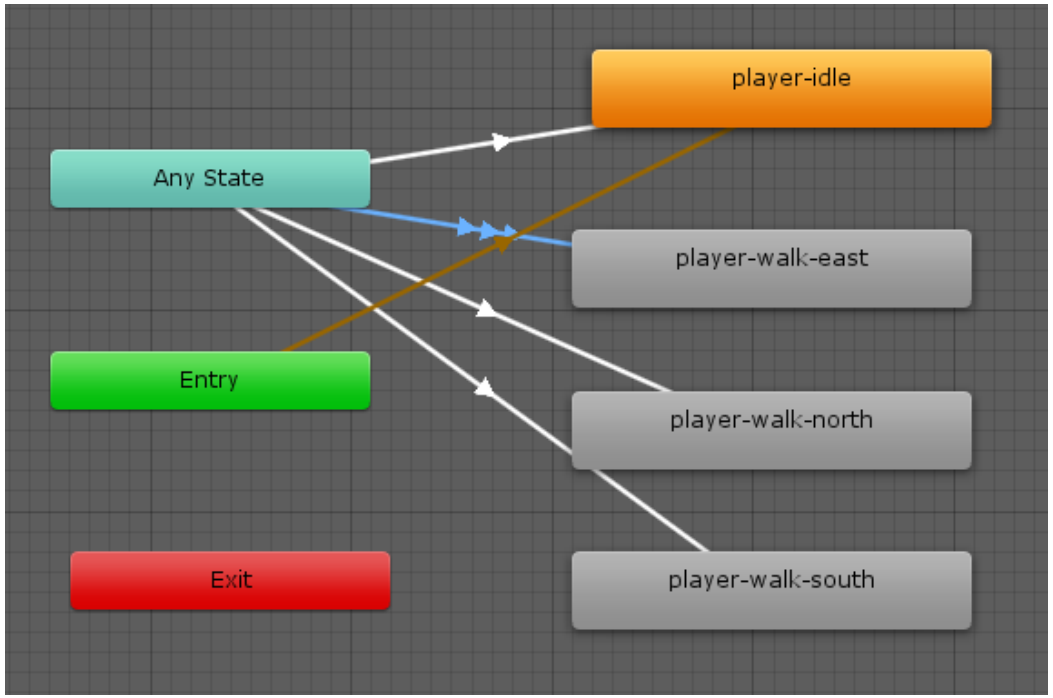
- Haz que la animación de espera sea el estado por defecto (clic derecho y “Set as Layer Default State”)
- Crea las siguientes transiciones:



Completando las animaciones

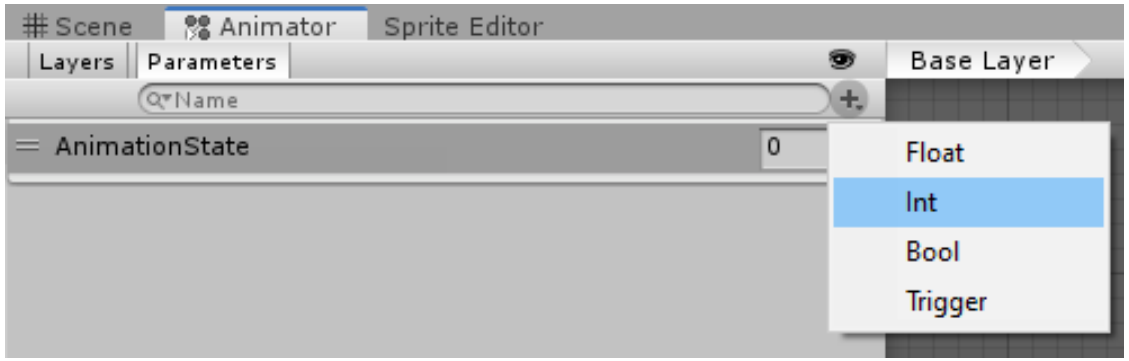
- Para controlar cuándo se dispara cada transición, vamos a definir un parámetro de tipo entero, que estableceremos desde el script según este patrón:

```
enum CharStates
    {
        walkEast = 1,
        walkSouth = 2,
        walkWest = 3,
        walkNorth = 4,

        idleSouth = 5
    }
```



Completando las animaciones

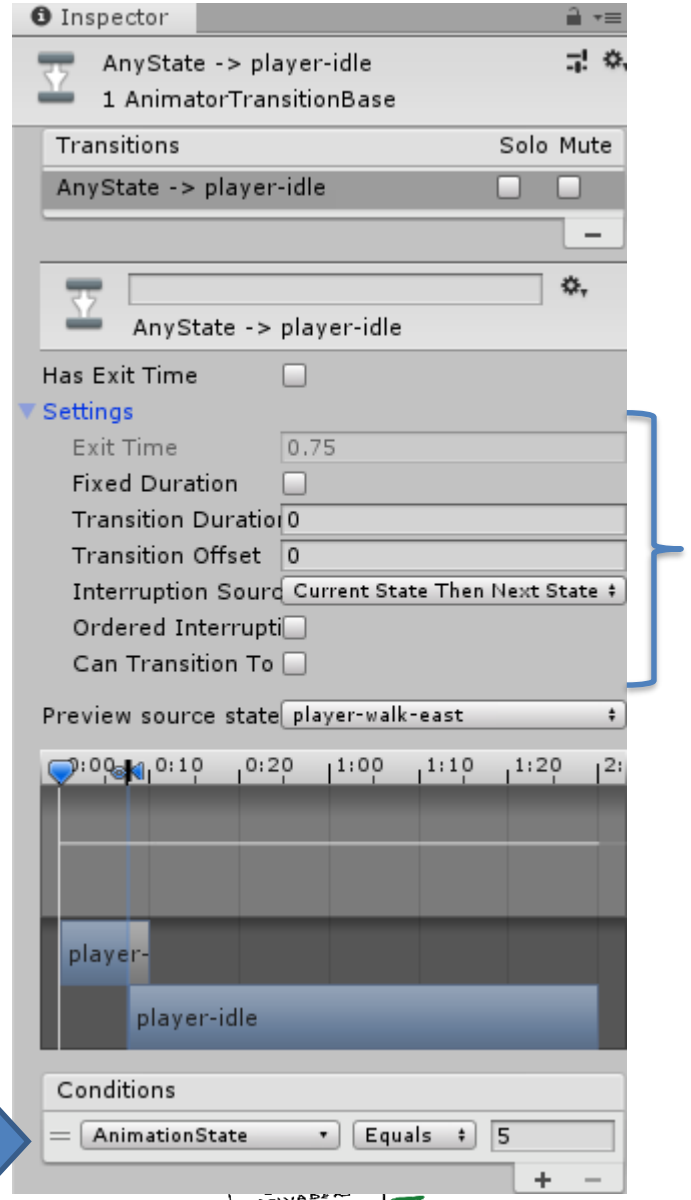
- Crea un parámetro “AnimationState” de tipo entero en el controlador:



Completando las animaciones

- Fíjate que no tenemos la animación de caminar hacia el oeste. Lo vamos a simular luego (por código) reflejando la animación de caminado hacia el este
- Selecciona todas las transiciones, estableciendo los parámetros de la figura

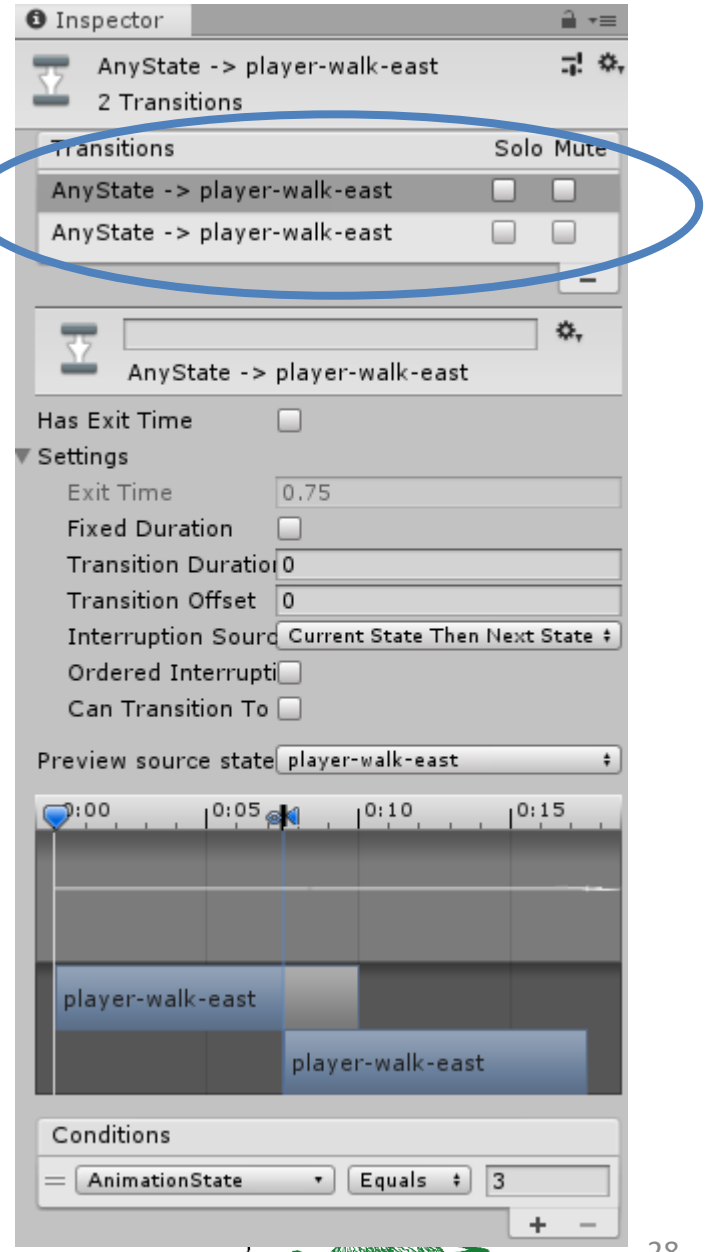
```
enum CharStates {
  walkEast = 1,
  walkSouth = 2,
  walkWest = 3,
  walkNorth = 4,

  idleSouth = 5
}
```



Establece la condición adecuada en cada transición

Completando las animaciones

- Como queremos que ambos movimientos laterales nos lleven al estado de “player-walk-east”, debemos establecer dos transiciones entre “Any State” y “player-walk-east”
  - Si, en vez de dos transiciones, pusiéramos dos condiciones en la lista de condiciones de una transición, la transición sólo se dispararía al hacerse ambas true

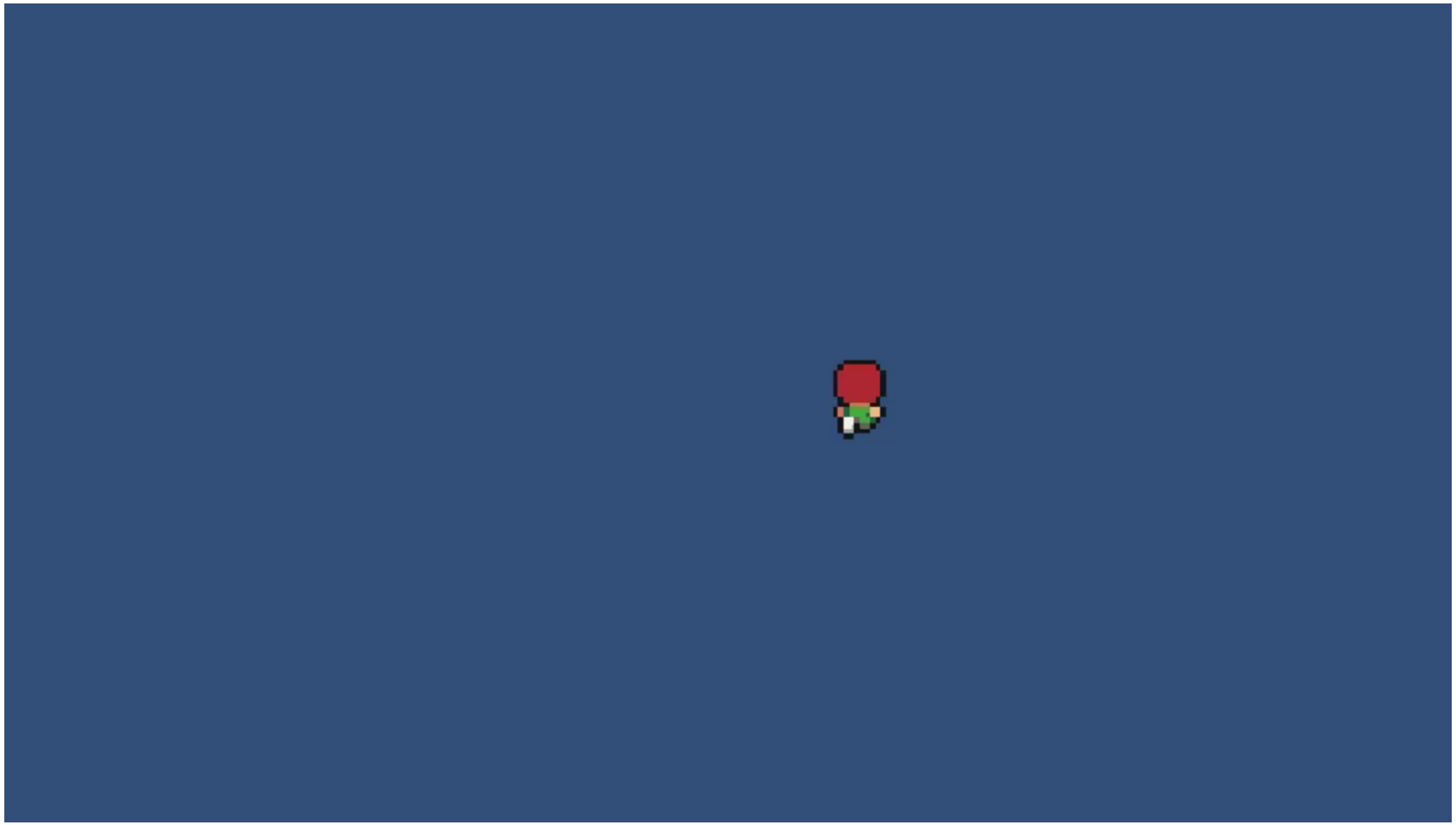


Completando las animaciones

```
public class MovementController : MonoBehaviour {
  public float movementSpeed = 3.0f;
  Vector2 movement = new Vector2();
  Rigidbody2D rb2d;
  Animator animator;
  SpriteRenderer spriteRenderer;

  enum CharStates {
    walkEast = 1,
    walkSouth = 2,
    walkWest = 3,
    walkNorth = 4,
    idleSouth = 5
  }

  void Start() {
    rb2d = GetComponent<Rigidbody2D>();
    animator = GetComponent<Animator>();
    spriteRenderer = GetComponent<SpriteRenderer>();
  }
```

```
  void Update() {
     UpdateState();
  }

  private void UpdateState() {
    CharStates state = CharStates.idleSouth;
    if (movement.x > 0) {
       state = CharStates.walkEast;
    } else if (movement.x < 0) {
        state = CharStates.walkWest;
    } else if (movement.y > 0) {
        state = CharStates.walkNorth;
    } else if (movement.y < 0) {
        state = CharStates.walkSouth;
    }
    animator.SetInteger("AnimationState", (int)state);
    spriteRenderer.flipX = (state == CharStates.walkWest);
  }
  private void FixedUpdate() {
    movement.x = Input.GetAxisRaw("Horizontal");
    movement.y = Input.GetAxisRaw("Vertical");
    movement.Normalize();
    rb2d.velocity = movement * movementSpeed;
  }
}
```


Completando las animaciones

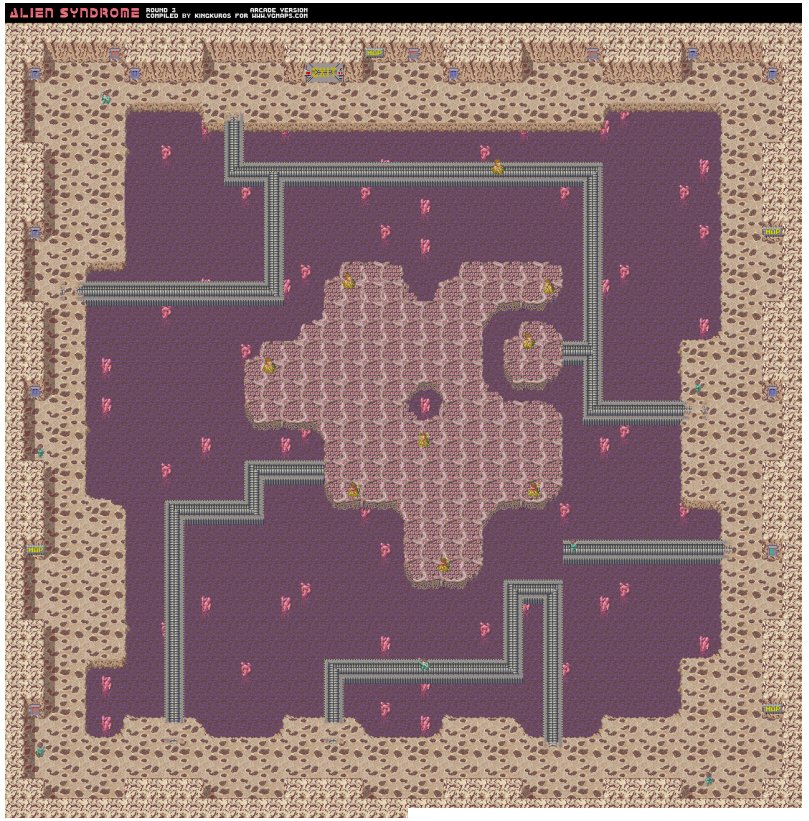


Construyendo el mundo

- Una vez que tenemos el control del personaje, vamos a darle un mundo en el que moverse

Ghosts 'n Goblins,Capcom. 1985

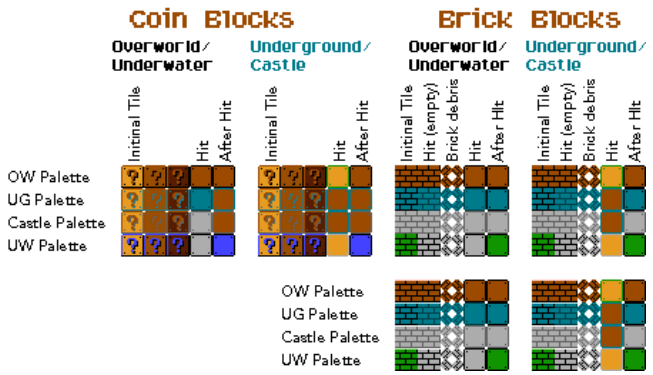
Pokémon Gold Version, Nintendo. 1999

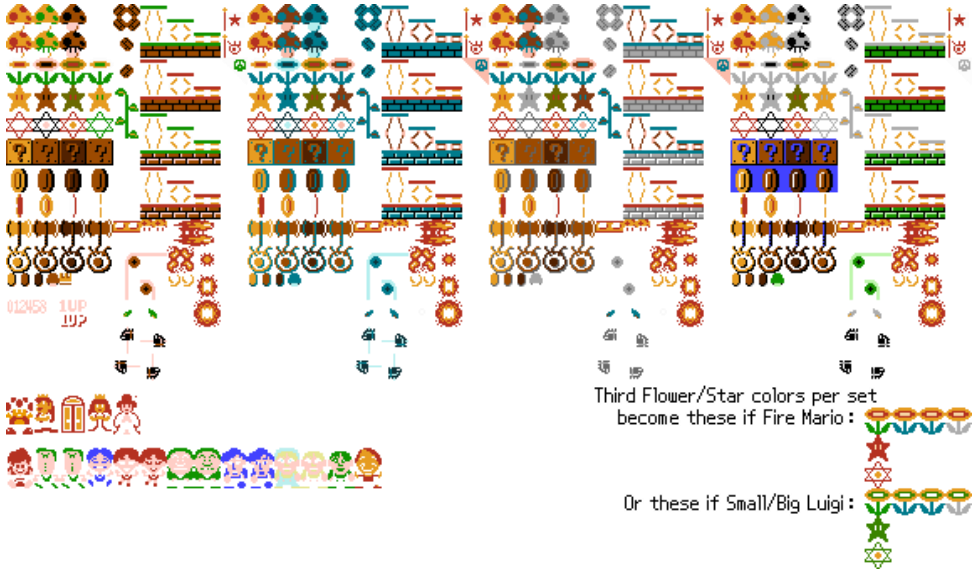


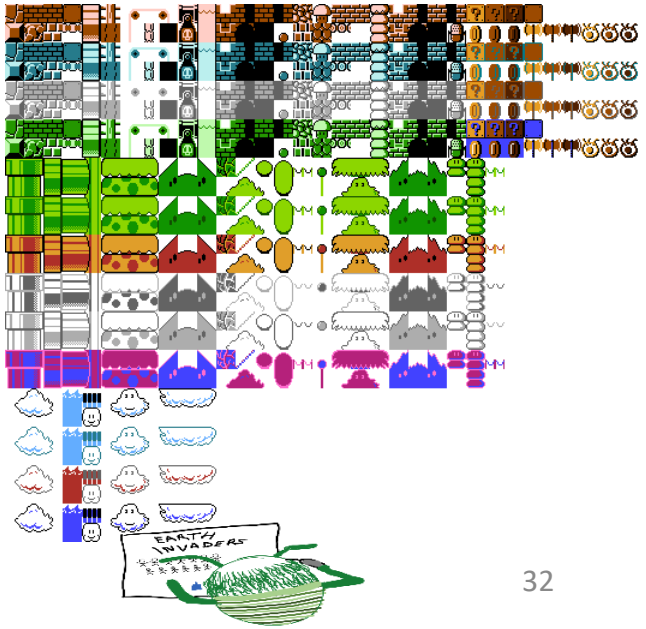
Alien Syndrome, Sega. 1987

Construyendo el mundo

- El mundo de los juegos 2D se construye a partir de una librería de tiles que se pueden combinar entre sí







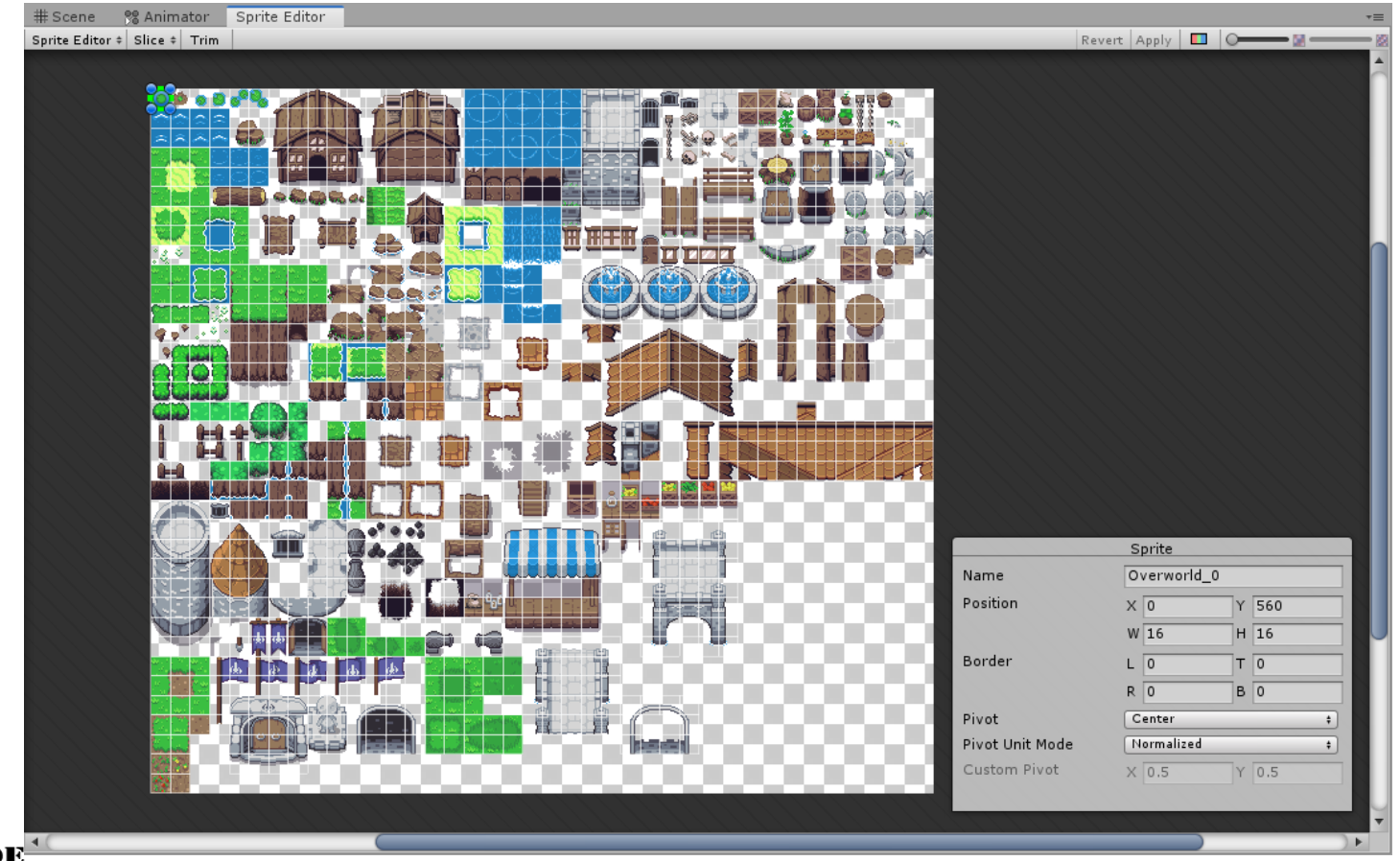
https://www.spriters-resource.com/nes/supermariobros/

Construyendo el mundo

- El mundo se construye situando tiles en una cuadrícula que cubra todo el fondo
- Se pueden apila unos tiles sobre otros para simular una ordenación con respecto a la cámara
- Unity ofrece la herramienta de Tilemaps para construir nuestro mapa a partir de los tiles
  - El Tilemap facilita la construcción de mundos 2D a partir de los tiles
  - Es una cuadrícula sobre la que se “pintará” el mapa usando los distintos tipos de tile

Construyendo el mundo

- Crea una carpeta dentro de Sprites llamada “Outdoors” e importa la hoja de sprites correspondiente
  - Usa la misma configuración que antes

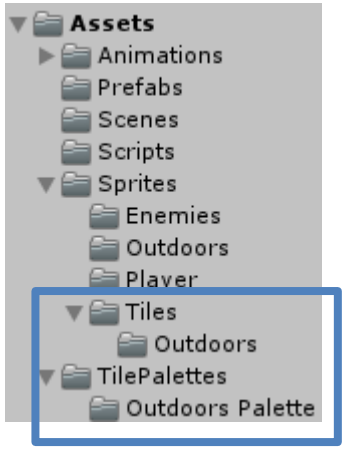


Construyendo el mundo: Tilemaps

- Vamos a crear el primer Tilemap:
  - En la jerarquía Create\2D Object\Tilemap
- Se crea un objeto Grid y el Tilemap:
  - El Grid es el padre de todos los Tilemaps
  - Cada Tilemap es una capa sobre la que podemos organizar los objetos del mundo
  - El orden en el que se dibujan los Tilemaps se establece con las capas de ordenación (Sorting Layer)
    - Cada Tilemap está asociado a una capa de ordenación
  - Llama a la nueva capa “Tilemap_base”

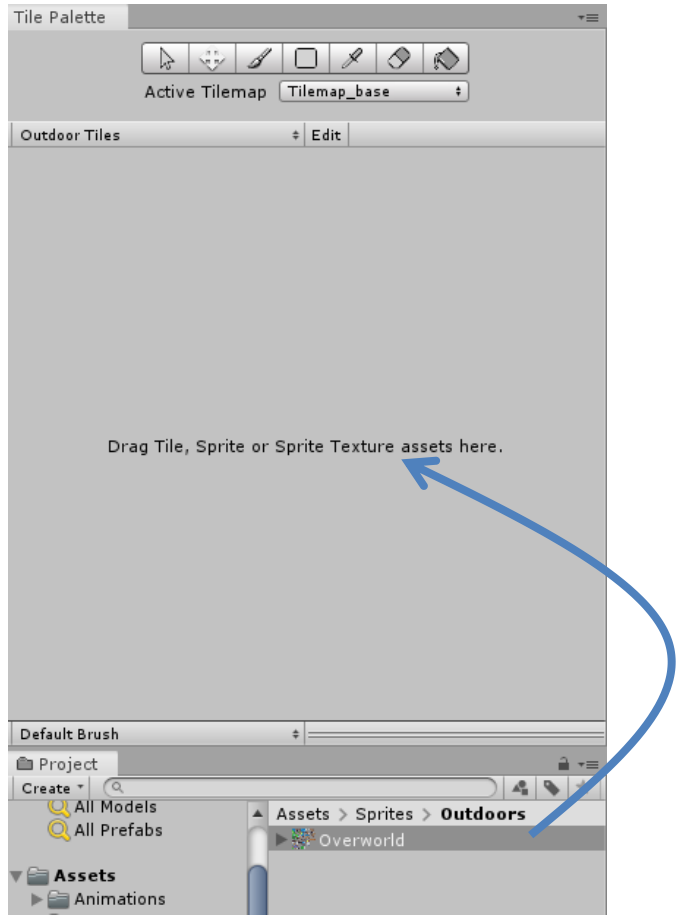
Construyendo el mundo: paletas de tiles

- La paleta de tiles (Tiles palettes) nos muestra los tiles disponibles para “pintar” nuestro mapa
- Abre el panel correspondiente en Window\2D\Tile Palette
- Vamos a crear directorios para las paletas y para los tiles individuales:



Construyendo el mundo: paletas de tiles

- En la paleta de tiles, crea una nueva paleta y llámala “Outdoors Tiles”
- Selecciona la hoja de sprites y arrástrala a la paleta de tiles
  - Pedirá el directorio donde generar los tiles individuales. Selecciona el directorio Assets\Sprites\Tiles\Outdoors



Construyendo el mundo: paletas de tiles

- Herramientas de la paleta de tiles:


Seleccionar tiles de la paleta (un tile o un bloque de tiles)


Mover selección en el mapa

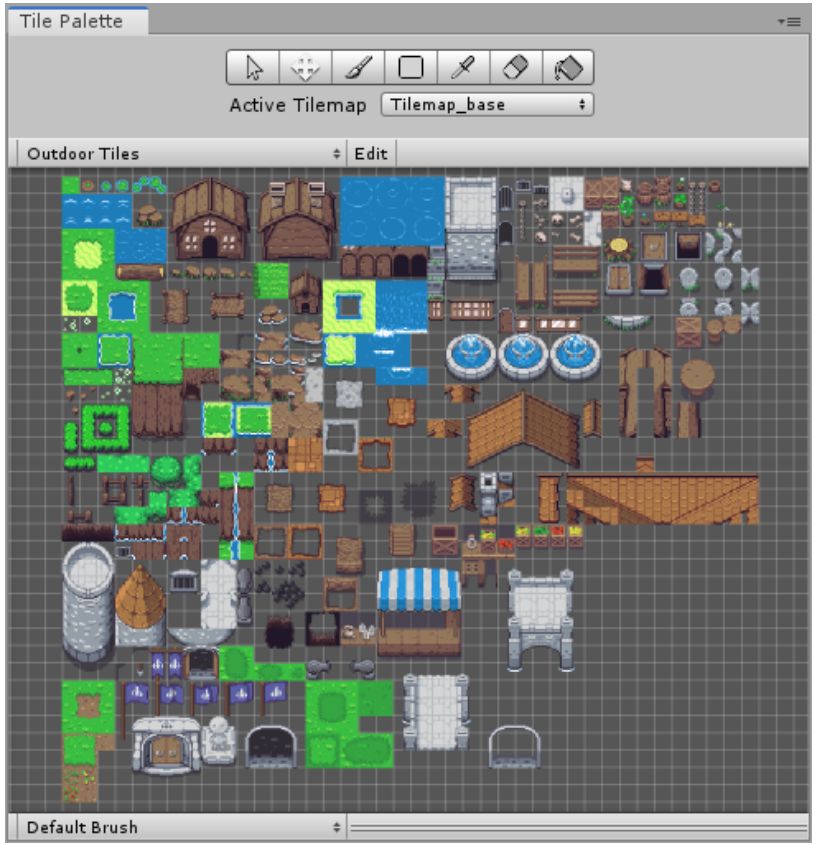

“Pintar” los tiles seleccionados sobre el mapa (pueden ser varios)


Rellenar con los tiles seleccionados una zona rectangular del mapa


Seleccionar un tile directamente en el mapa


Borrar tiles del mapa


Rellenar un área del mapa con los tiles seleccionados



Construyendo el mundo: paletas de tiles

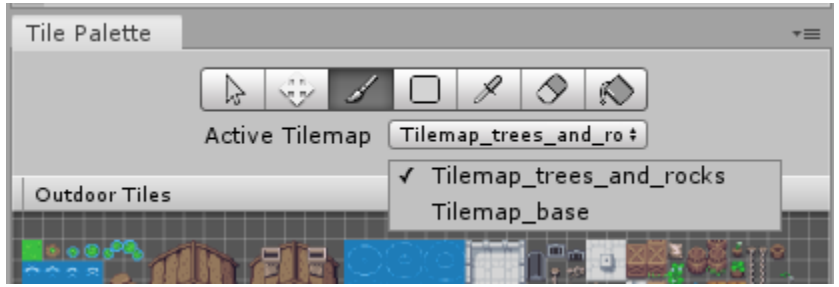
- Atajos en la paleta de tiles:
  - Ruleta para hacer zoom
  - Clic en la ruleta para desplazarse por la paleta (y por el mapa)
  - Al tener seleccionada la brocha:
    - Mayús + clic: borrar el tile
    - Ctrl+clic: seleccionar el tile para usarlo a continuación
  - ‘ y ¡ para rotar el tile

¡Construye tu mundo!

- Diseña la base de tu mundo (el suelo). De momento no añadas detalles como piedras, casas u otros objetos

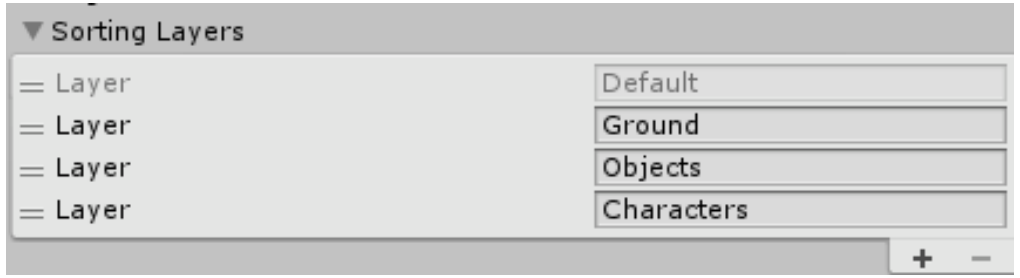
Construyendo el mundo: varias capas

- Para añadir el siguiente nivel de objetos a la escena, vamos a crear un nuevo Tilemap:
  - Botón derecho sobre Grid\2D Object\Tilemap
  - Llama a la nueva capa: “Tilemap_tres_and_rocks”
  - Ahora podemos añadir tiles a la nueva capa
    - Selecciónala en la paleta de tiles:



Construyendo el mundo: varias capas

- Ahora hay que definir el orden de dibujado de los distintos tilemaps
  - Selecciona el Tilemap_base en la jerarquía, y asígnale una nueva capa de ordenación llamada “Ground” en el campo correspondiente del componente Tilemap Renderer
  - Repite el proceso para Tilemap_trees_and_rocks
  - Al final tendremos las siguientes capas de ordenación:

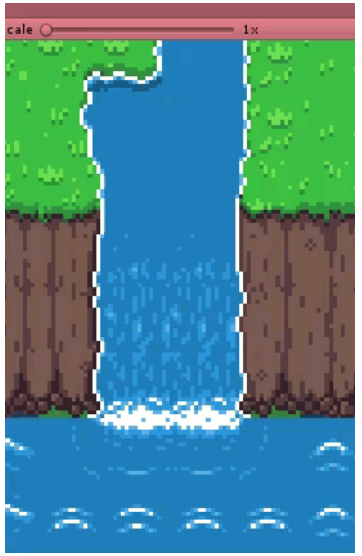


Los objetos de la capa de ordenación Objects se dibujan sobre los de Ground, y los de Characters sobre los demás.

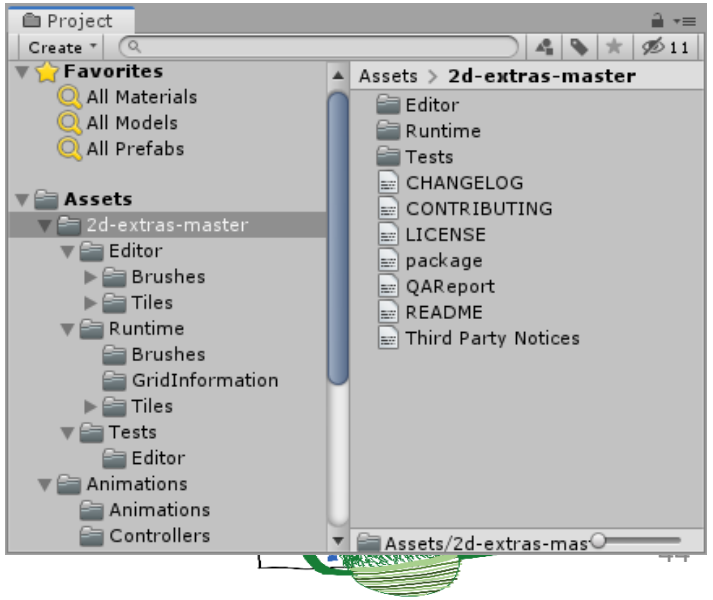
Construyendo el mundo: varias capas

- Añade elementos a la nueva capa para hacer más interesante el mapa

Añadiendo animaciones al mundo: tiles animados

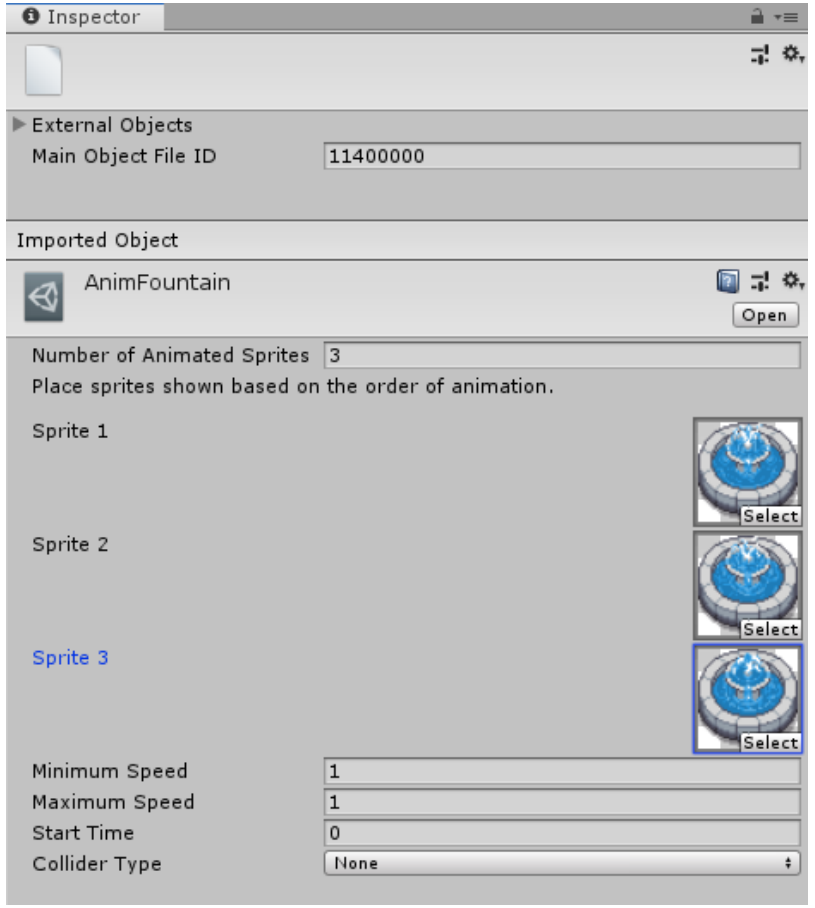


- Es común tener tiles animados, que definen varios frames
- Unity ofrece los scriptable tiles, que añaden funcionalidad a los tiles estáticos
- El código que soporta los tiles animados no está integrado en Unity, hay que añadirlo desde Github:
  - Para Unity 2019.2:
    - https://github.com/Unity-Technologies/2d-extras
  - Para otras versiones, consultar el fichero README.md
  - Descomprimir el zip dentro del directorio Assets del proyecto



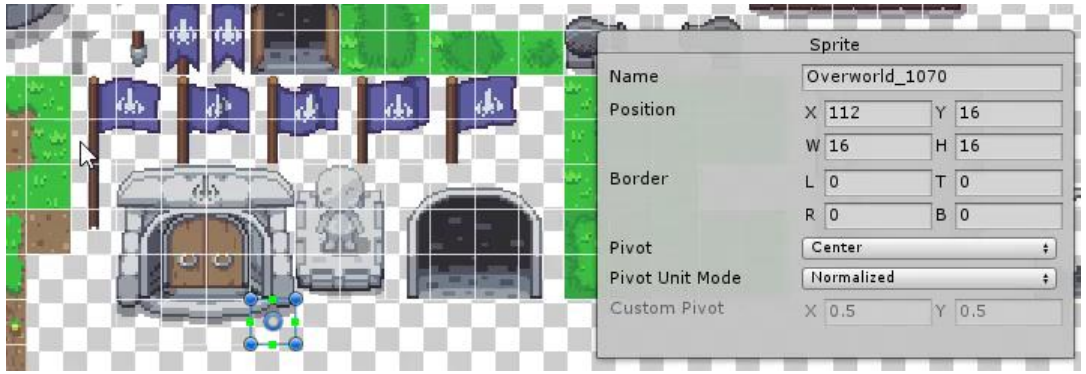
Añadiendo animaciones al mundo: tiles animados

- A continuación, hay que crear un Tile animado:
  - [Opcional] Crear un directorio para los Tiles animados
  - Create\Tiles\Animated Tiles
  - En el Inspector, indicar el número de Sprites (frames de animación)
  - Arrastrar desde el proyecto o seleccionar cada sprite
  - [Opcional] Crear una paleta para los sprites animados
  - Arrastrar el tile animado a la paleta
  - Arrastrar el tile animado desde la paleta a la escena
  - Ajustar la velocidad de reproducción (aleatoria entre min y max)



Añadiendo animaciones al mundo: tiles animados

- Puede ser que tengas sprites de distintos tamaños, y que al hacer el recortado automático, estén en distintos tiles
- Puedes definir nuevos tiles de otros tamaños para recortar sprites más grandes desde el Sprite Editor (no borres los anteriores si ya has creado una paleta):
  - Apunta la posición del primer tile de cada frame
  - Añade un sprite nuevo arrastrando sobre una zona libre del Sprite editor
  - Establece la posición y el tamaño del frame
  - Repite para cada frame
  - Pulsa Apply en el Sprite Editor
- Mejor: separa todos los sprites antes

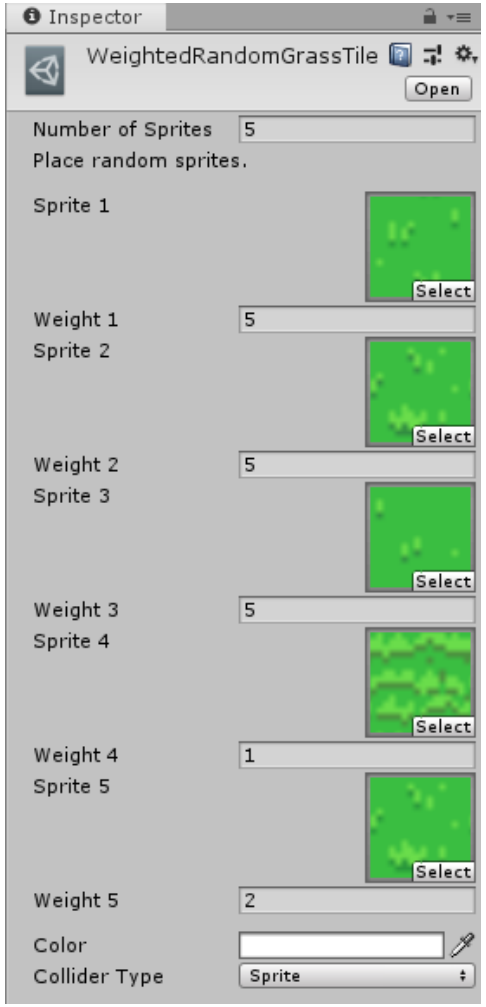


Otras ayudas para la creación de niveles: tiles aleatorios

- Otro tile muy útil para construir el nivel de nuestro juego es el tile aleatorio
- Cada vez que pintamos una celda del nivel, elige un tile aleatorio entre un conjunto predefinido:
  - Create\Tiles\Random Tile
  - Define el número de sprites
  - Selecciona los sprites
  - Añade el Random Tile a una paleta y úsalo para rellenar el nivel

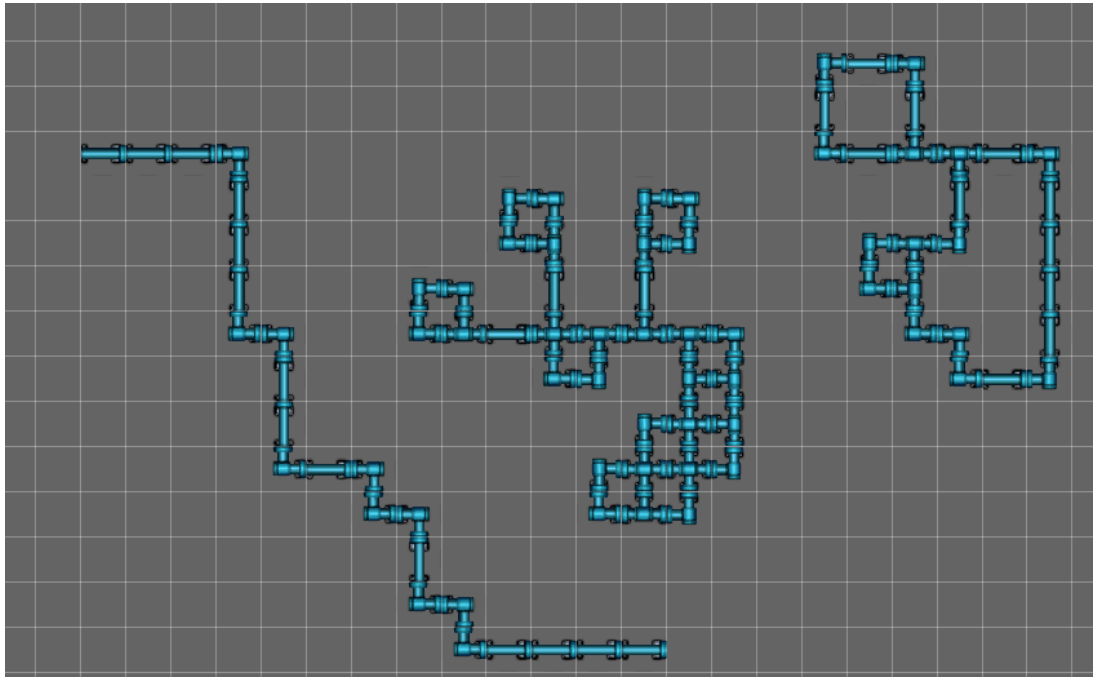
Otras ayudas para la creación de niveles: tiles aleatorios con peso

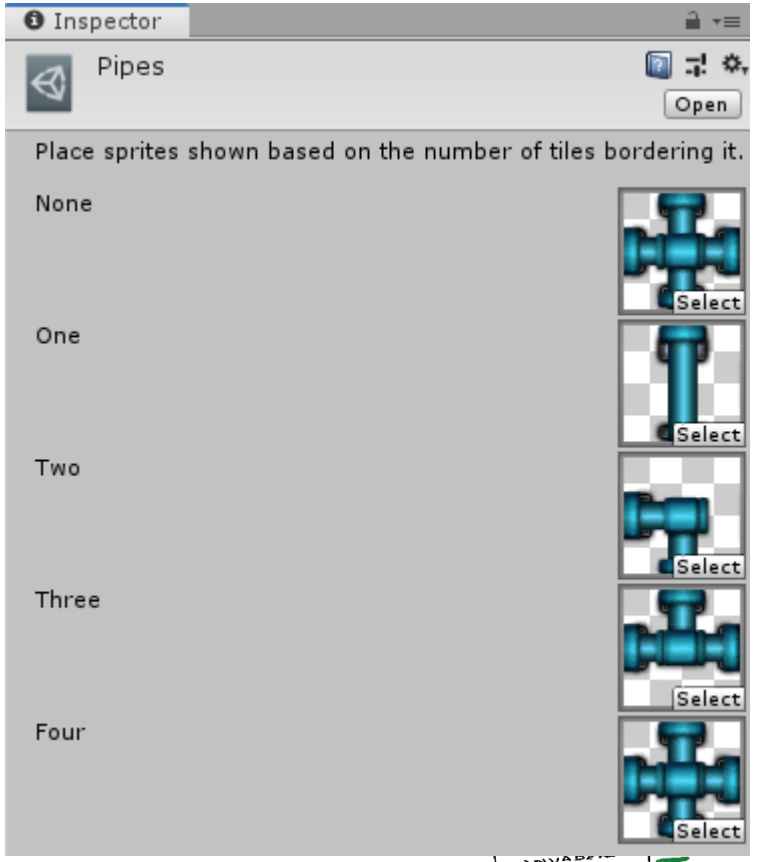
- Una alternativa del anterior nos permite dar más peso a ciertos tiles
- *Weighted Random Tile*, además de cada sprite, espera un número que será proporcional a su probabilidad de aparición



Otras ayudas para la creación de niveles: tuberías

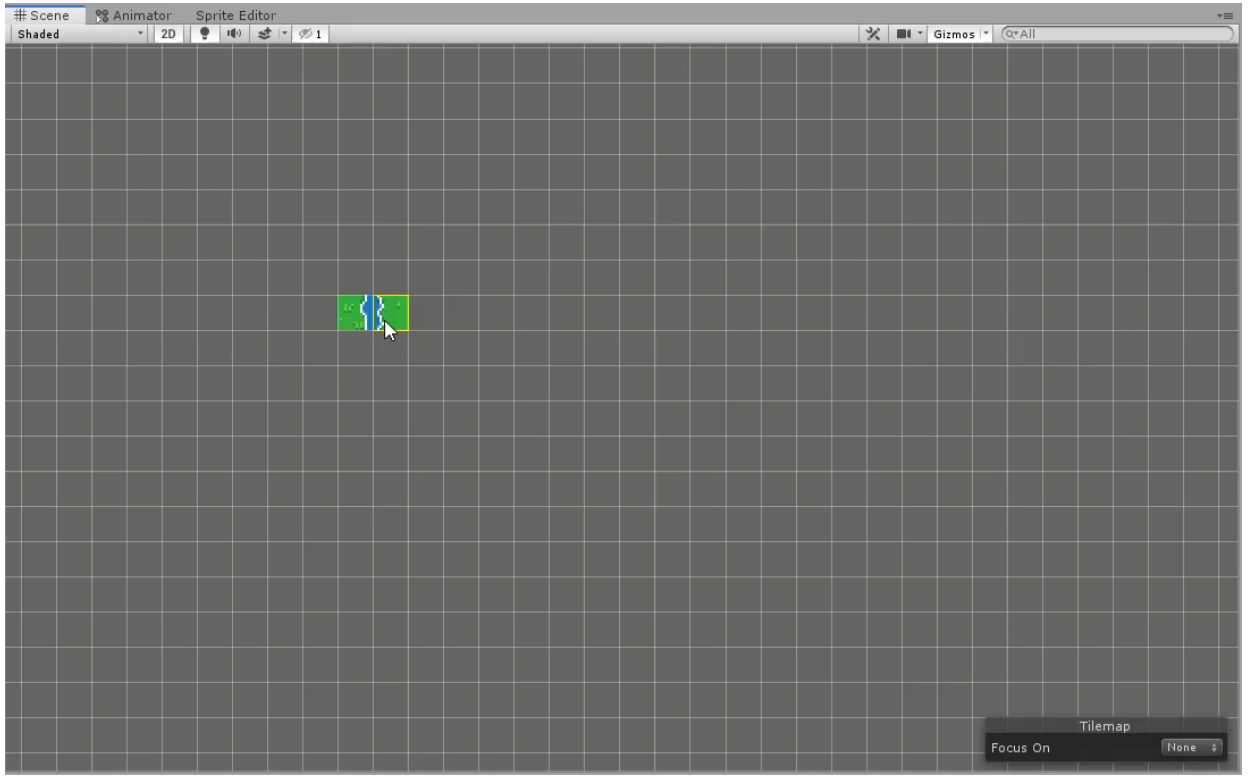
- Este tipo de tile (Pipeline Tile) permite dibujar tuberías (o elementos similares), a partir de tan sólo 4 piezas:





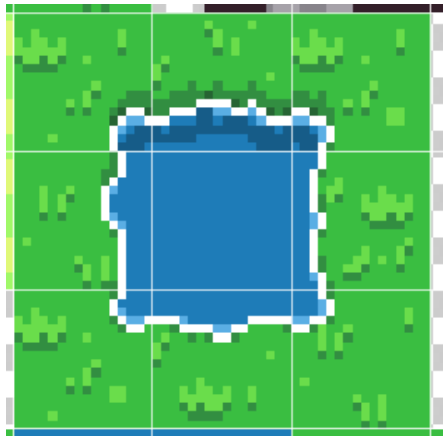
Otras ayudas para la creación de niveles: Rule Tile

- Este tipo de tile es el más flexible
- Permite definir la forma de construir plataformas o estructuras 2D

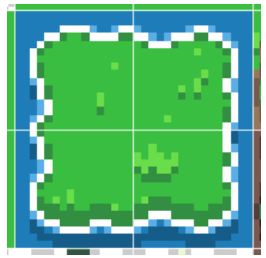


Otras ayudas para la creación de niveles: Rule Tile

- Las Rule Tiles funcionan definiendo una serie de sprites, y las condiciones que deben cumplir (como, por ejemplo, que tengan un tile arriba, pero no tengan uno a la derecha)
- Unity recorrerá de arriba abajo las reglas, y seleccionará el sprite de la primera que se cumpla
- Por ejemplo:

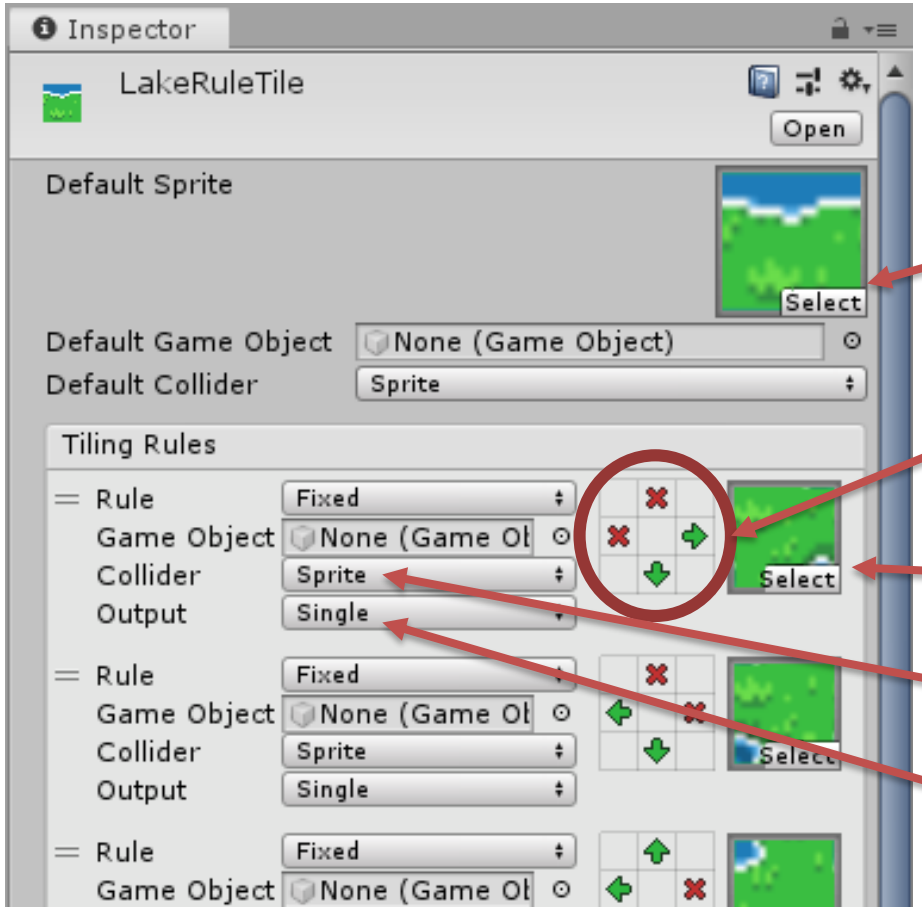


Esquinas exteriores, laterales y centro



Esquinas interiores

Otras ayudas para la creación de niveles: Rule Tile



Sprite por defecto (se mostrará también en la paleta)

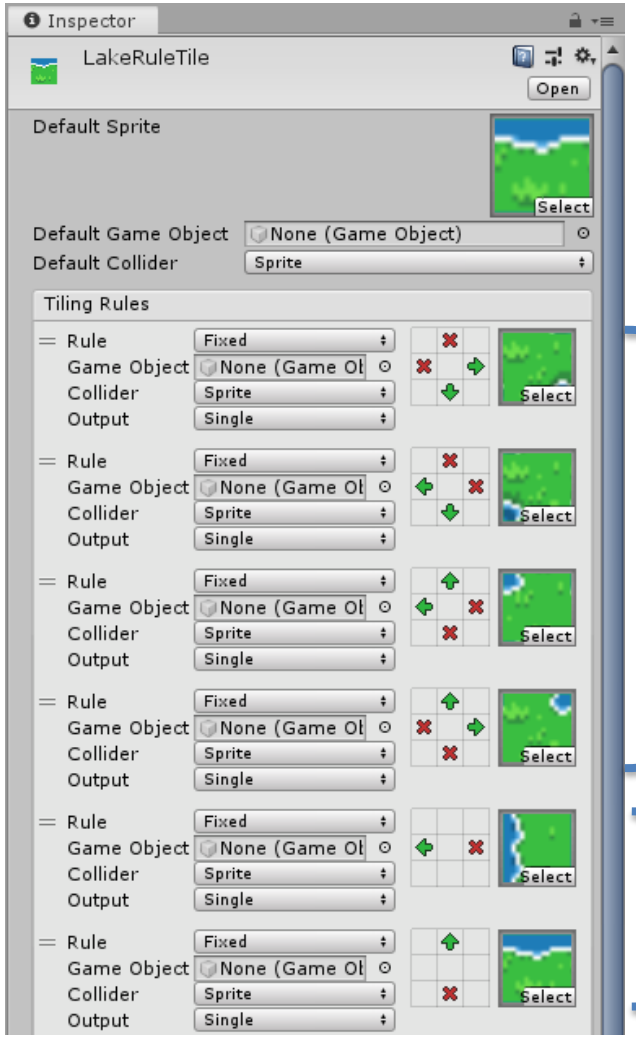
DEBE tener un tile a la derecha y otro abajo
NO DEBE tener un tile arriba ni a la izquierda
DA IGUAL si tiene tiles o no en diagonal

Sprite de la regla

Collider del script: Ninguno, Sprite o Grid

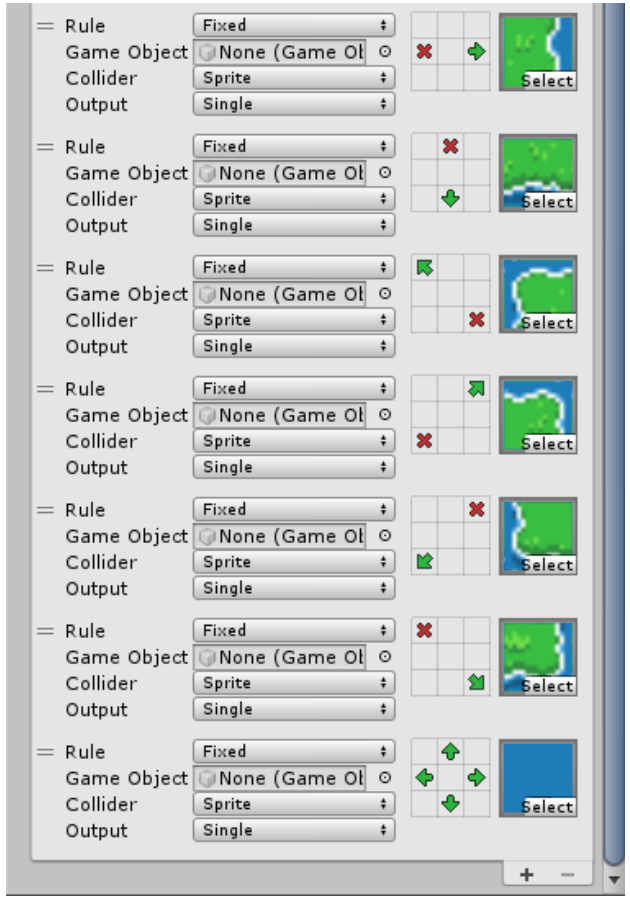
Tipo de sprite: sencillo, aleatorio o animado

Otras ayudas para la creación de niveles: Rule Tile



Esquinas exteriores

Laterales (1)



Laterales (2)

Esquinas interiores

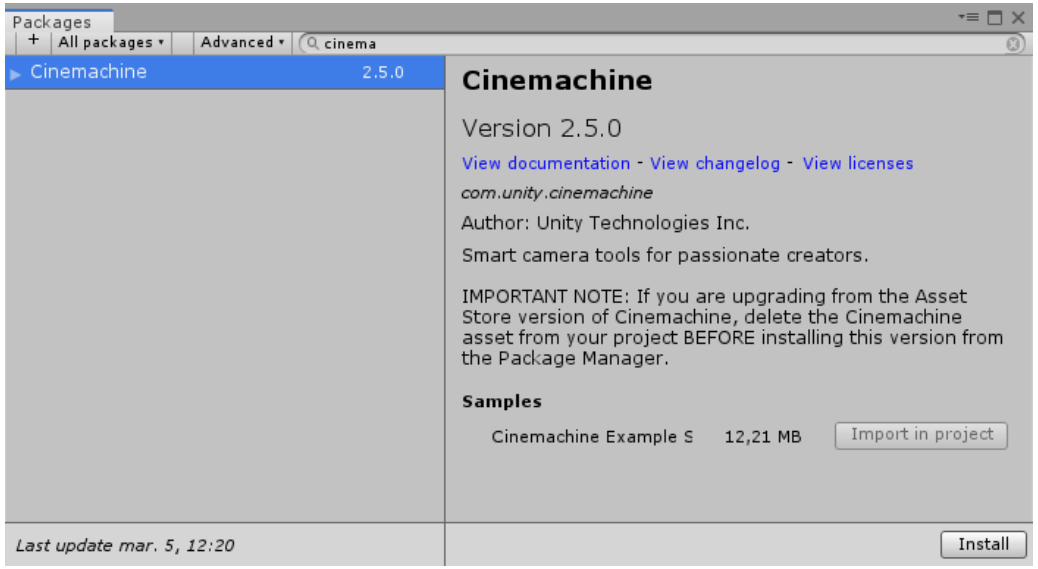
Interior

Ajustando la cámara y el antialiasing

- Selecciona la cámara y ajusta el parámetro Size para definir el rectángulo del mundo que puede ver el jugador en cada momento
- Ejecuta el juego
- Si aparecen líneas en la vista del juego
  - desconecta el antialising: Edit\Project Settings\Quality\Anti Aliasing y Anisotropic Textures
  - Crea un material usando el shader Sprites/Default y activa la opción Pixel snap
    - Asigna el nuevo material al campo Material del Tilemap Renderer de todos los tilemaps

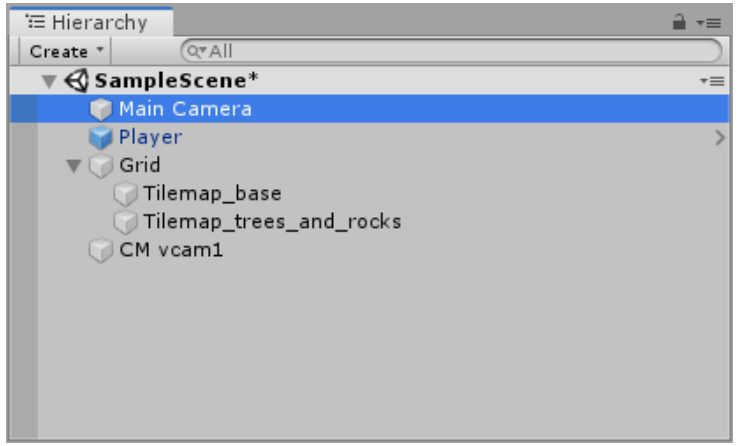
Animando la cámara

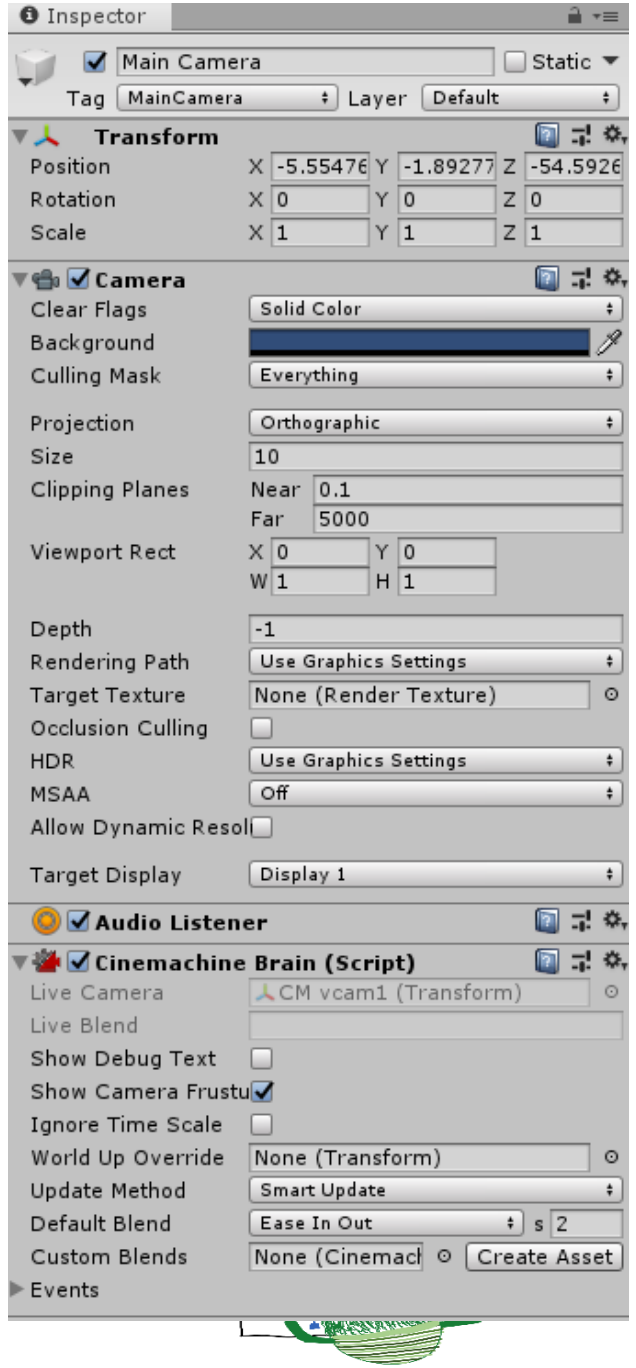
- Hasta ahora la cámara es estática. Vamos a hacer que persiga al personaje
- En vez de hacerlo por código, vamos a utilizar un paquete llamado Cinemachine, que ofrece muchas opciones
  - Window\Package Manager\
  - Clic en *Install*, y luego en *Import in project*.
  - Después de instalarlo, aparecerá un nuevo menú



Animando la cámara

- Añade una nueva cámara gestionada por Cinemachine:
  - Cinemachine\Create 2D Camera
  - Unity añade una nueva cámara virtual a la jerarquía, y un componente Cinemachine Brain a la cámara principal
- La cámara virtual se encarga de mover y configurar la cámara principal, a través del Brain
  - Puede haber varias cámaras virtuales para hacer transiciones



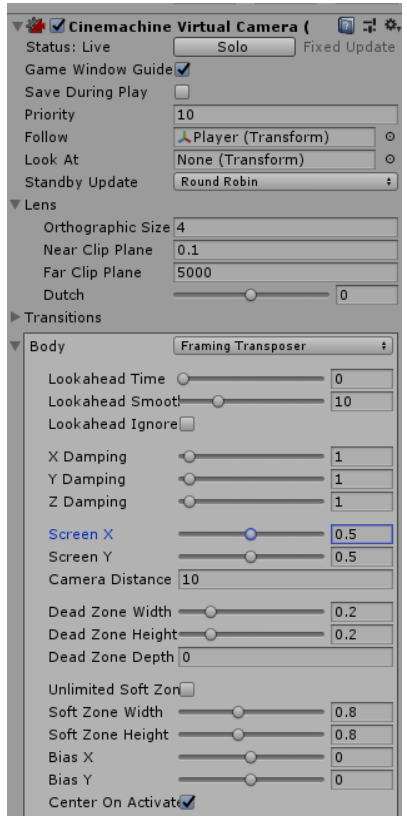


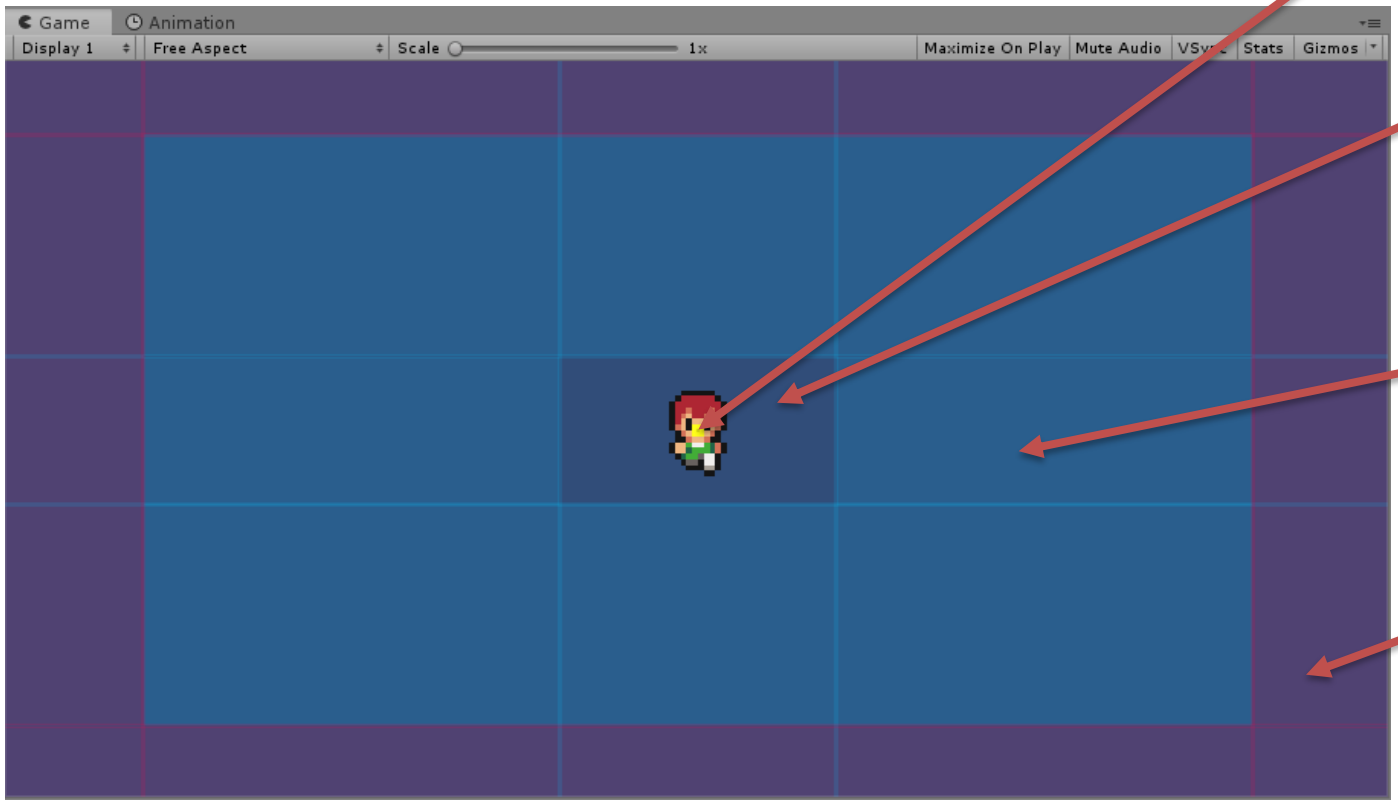
Animando la cámara

- Ahora los parámetros de la cámara principal se gestionan a partir de las cámaras virtuales
  - Configura el tamaño de la cámara en el apartado Lens de la cámara virtual
  - Arrastra al personaje al campo “Follow” para que la cámara lo siga
  - Ahora la cámara sigue al personaje por el mapa, con una animación suave

Animando la cámara

- Parámetros de control de la cámara:





Tracking point: el punto que persigue la cámara

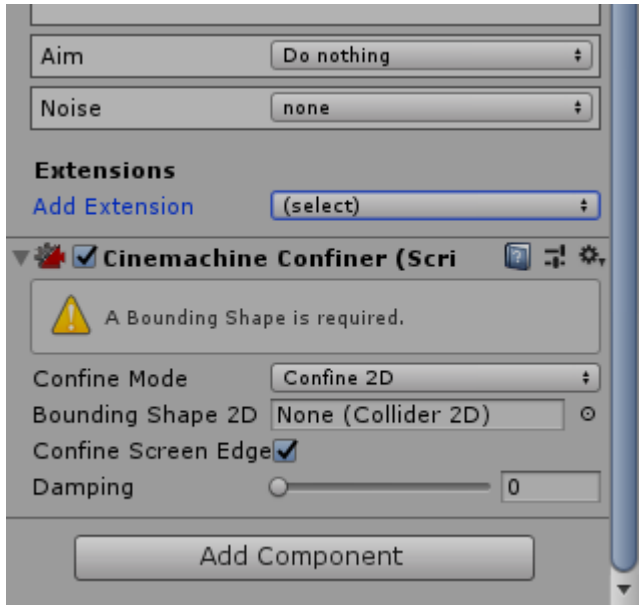
Dead-zone: mientras que el tracking point esté dentro de esta zona, la cámara no se moverá

Soft-zone: si el tracking point llega a esta zona, la cámara se mueve suavemente hasta devolverlo a la dead-zone

Si el tracking point llega a esta zona, la cámara se moverá a su misma velocidad

Animando la cámara

- Limitando el movimiento de la cámara
  - Podemos hacer que la cámara no pueda salir del terreno de juego
  - Selecciona la cámara virtual
  - Selecciona la extensión Cinemachine Confiner en la lista Extensions
  - Añade un Polygon Collider a Tilemap_base. Este collider definirá por dónde puede moverse la cámara
    - Pulsa Edit Collider y ajústalo a la capa
    - Atajos: Mayúsculas + clic para añadir un vértice, Ctrl+clic para borrar un vértice
    - Márcalo como Is Trigger
  - Arrastra a Tilemap_base al campo Bounding Shape 2D del confiner de la cámara virtual
    - Asegúrate de que Confine Screen Edges está activado



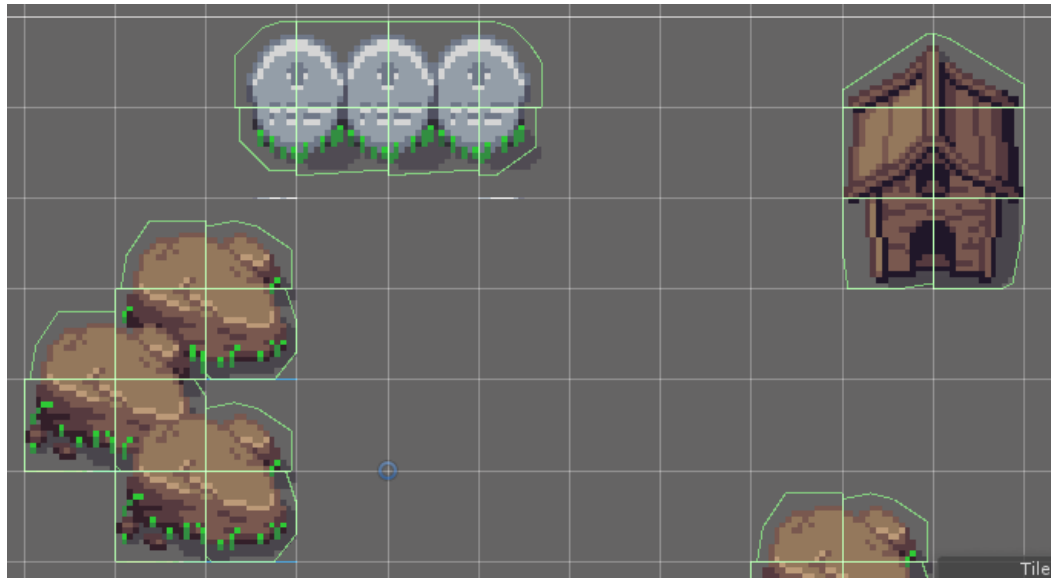
Animando la cámara

- En la animación de la cámara se producen pequeños temblores, sobre todo al frenar
- Añade el siguiente script a la cámara virtual

```
using Cinemachine;
public class RoundCameraPos : CinemachineExtension {
    public float PixelsPerUnit = 16;
    protected override void PostPipelineStageCallback(CinemachineVirtualCameraBase vcam,
        CinemachineCore.Stage stage, ref CameraState state, float deltaTime) {
      if (stage == CinemachineCore.Stage.Body) {
            Vector3 pos = state.FinalPosition;
            Vector3 pos2 = new Vector3(RoundPPU(pos.x), RoundPPU(pos.y), pos.z);
            state.PositionCorrection += pos2 - pos;
      }
    }
    float RoundPPU(float x) {
        return Mathf.Round(x * PixelsPerUnit) / PixelsPerUnit;
    }
}
```

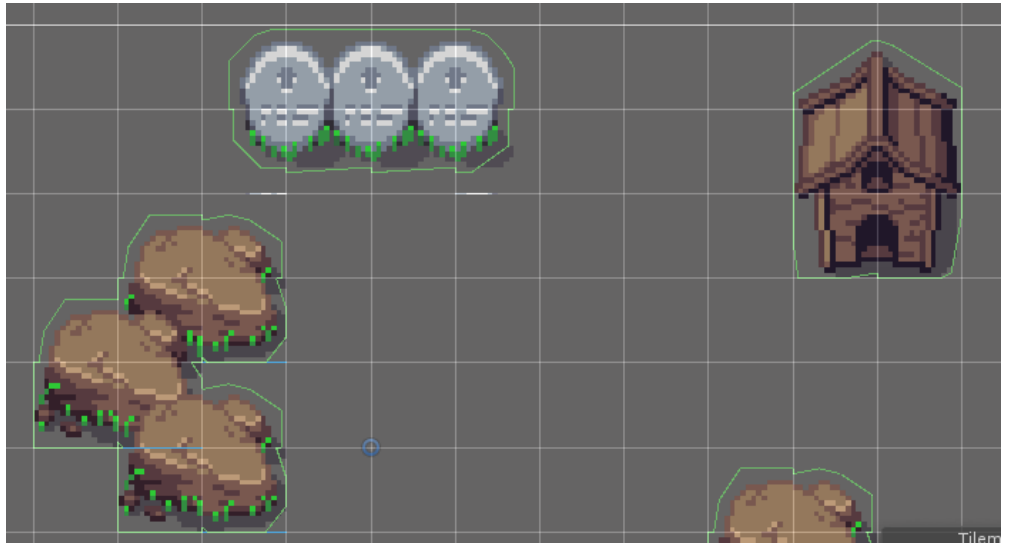

Añadiendo colliders a los elementos de la escena

- Hasta ahora el jugador puede atravesar los elementos del mapa (p.e., rocas, árboles, vallas, etc)
- Unity ofrece un componente que añade un collider a todos los sprites de una capa
  - Selecciona la capa “Tilemap_trees_and_rocks” y añade un componente Tilemap Collider 2D



Añadiendo colliders a los elementos de la escena

- Tener un collider para cada elemento de la capa es ineficiente. Unity permite combinar muchos collider en uno:
  - Añade a la capa “Tilemap_trees_and_rocks” un componente “Composite Collider 2D”
  - En el componente “Tilemap Collider 2D”, activa la opción “Used by composite”
  - En el Rigidbody 2D que ha aparecido al añadir el Composite, cambia el tipo de cuerpo a Static (los elementos no se moverán)

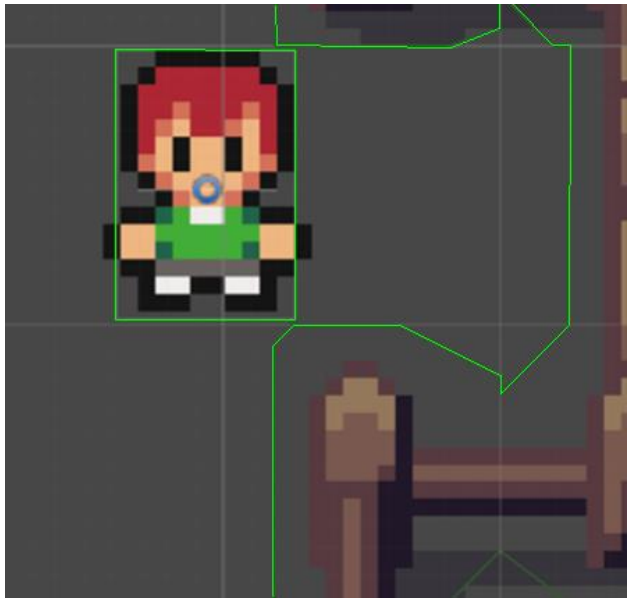


Fijando la rotación del personaje

- Como el personaje está gobernado por el motor de física, al tocar algún elemento del entorno gira
  - Activa la opción Freeze Rotation Z en el Rigidbody 2D del personaje

Ajustando las colisiones

- Unity es bastante conservador a la hora de calcular los collider, por lo que puede pasar que sean demasiado grandes y hayan colisiones cuando los objetos están aún lejos

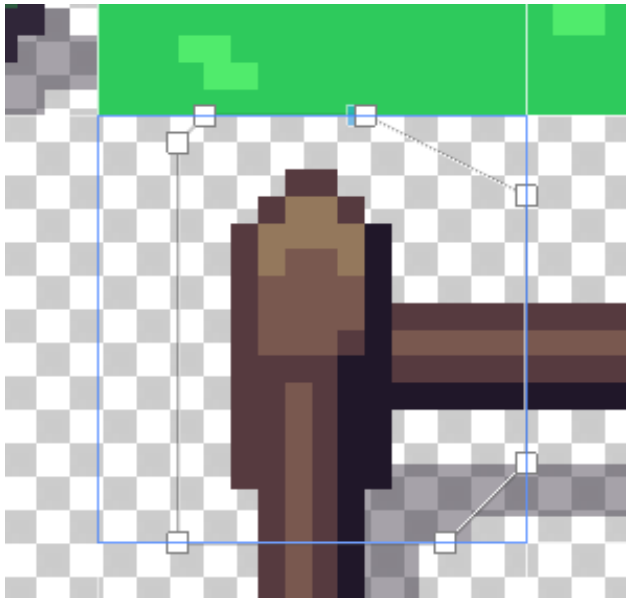


Ajustando las colisiones

- Se puede ajustar a mano la forma del collider desde el Sprite Editor:
  - Selecciona la hoja de sprites en la ventana del proyecto
  - Abre el Sprite Editor
  - Selecciona Custom Physics Shape en el primer desplegable del Sprite Editor

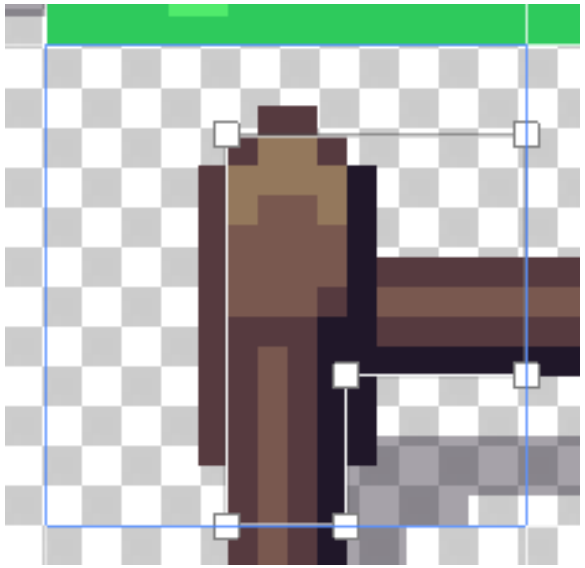
Ajustando las colisiones

- Se puede ajustar a mano la forma del collider desde el Sprite Editor:
  - Selecciona el sprite al que quieres ajustar el collider y pulsa el botón Generate
  - Unity calculará el collider a partir de los píxeles del tile y del valor establecido en “Outline Tolerance”



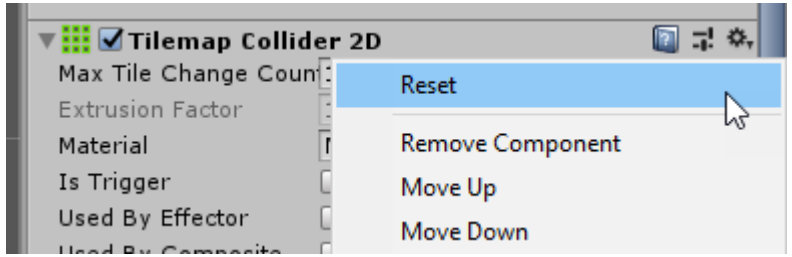
Ajustando las colisiones

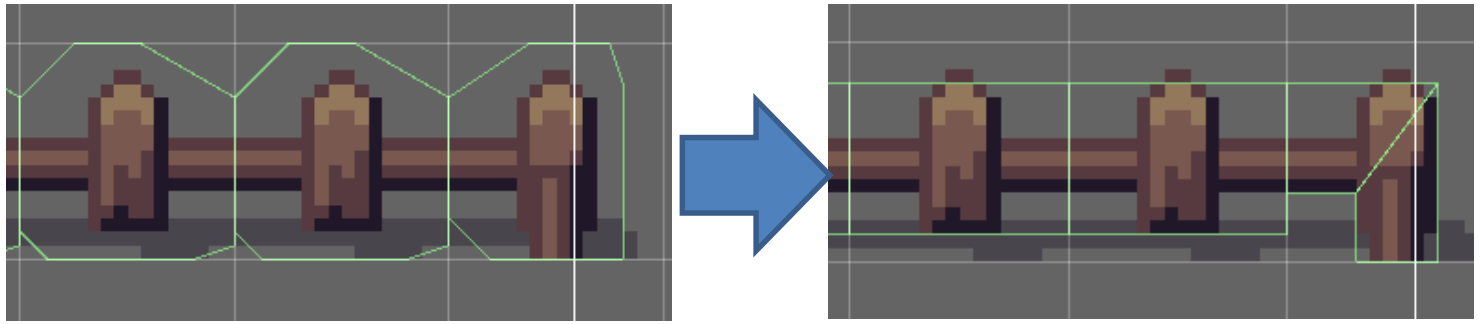
- Se puede ajustar a mano la forma del collider desde el Sprite Editor:
  - Mueve los manejadores para ajustar los vértices del collider
  - Puedes añadir nuevos manejadores haciendo clic en una arista
  - Puedes borrar el manejador seleccionado con la tecla Supr (Delete)



Ajustando las colisiones

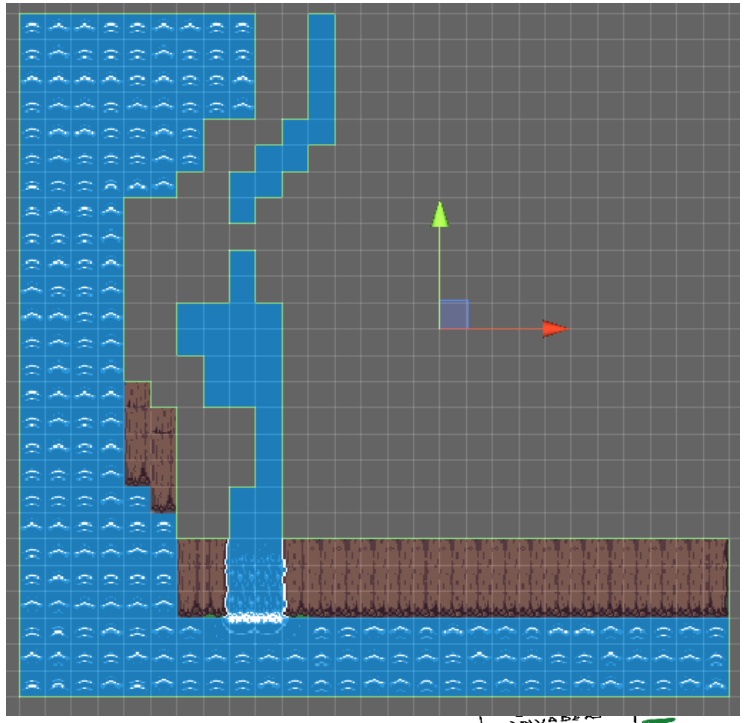
- Se puede ajustar a mano la forma del collider desde el Sprite Editor:
  - Cuando hayas acabado, pulsa Apply
  - En el botón de la rueda dentada del Tilemap Collider 2D del tilemap correspondiente, selecciona la opción Reset para que use los nuevos colliders





Ajustando las colisiones

- Usando la misma técnica, podemos definir zonas donde no puede entrar el personaje (p.e., agua)
- Añadimos un nuevo Tilemap y movemos todos los tiles del nivel donde no se pueda entrar (ver la siguiente traspa)
- Añadir un componente Tilemap Collider 2D
- Añadir un Composite Collider 2D
- Marcar la opción Used By Composite
- Hacer que el Rigidbody 2D sea estático

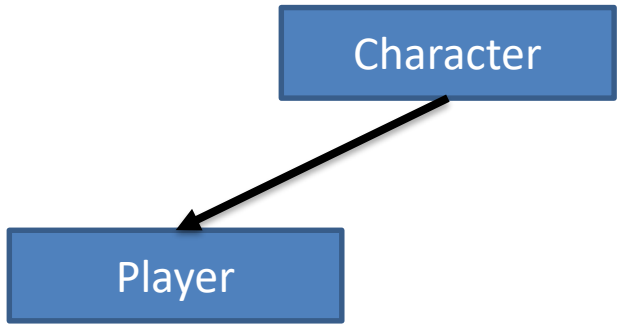


Mover tiles entre tilemaps

- El editor de Unity no permite “cortar y pegar” tiles entre tilemaps, pero los pasos siguientes se aproximan:
  1. Selecciona la capa origen en la paleta
  2. Selecciona el pincel (con el icono o con B)
  3. Pulsando Ctrl, selecciona el rectángulo de tiles a mover
  4. Pulsando Mayúsculas, haz clic en la misma zona para borrar los tiles
  5. Selecciona la capa destino
  6. Haz clic en la zona donde estaba el bloque para dibujarlo en la capa seleccionada

Implementando el código del jugador

- A continuación vamos a implementar el código con la funcionalidad de los personajes del juego
- Vamos a definir una estructura de clases y un código sencillo
- Añade el script Player al personaje y dale 5 puntos de vida, y 10 de máximo

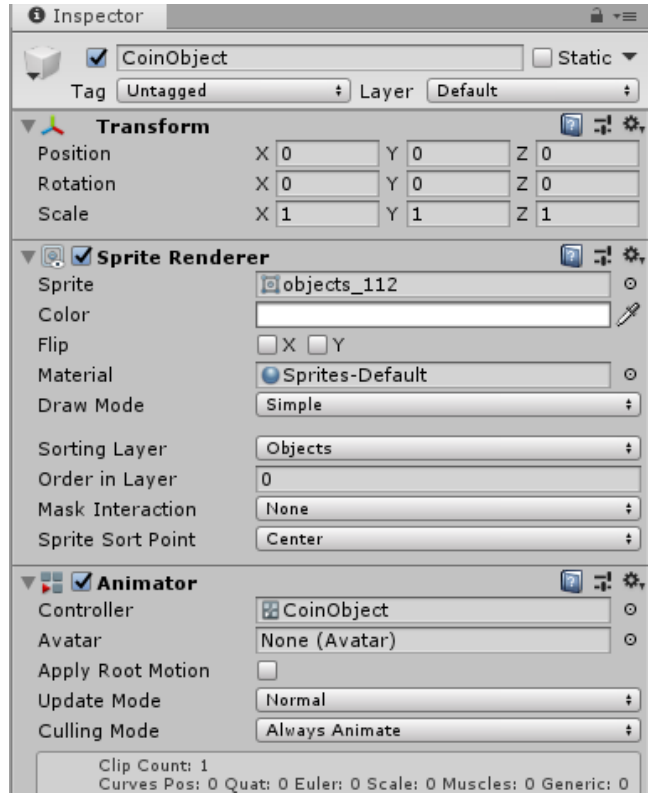



```
public abstract class Character : MonoBehaviour
{
    public int health;
    public int maxHitPoints;
}
```

```
public class Player : Character
{
}
```


Creando recolectables: una moneda

- Vamos a crear un prefab que contendrá la animación de una moneda rotando y el mecanismo para detectar que el personaje la ha recogido
  - Crea un Empty llamado CoinObject
  - Selecciona los cuatro frames de la animación de la moneda que se encuentra en la hoja de sprites Objects
  - Arrástralos sobre el Empty
  - Llama a la nueva animación “coin-spin”, y guarda la animación y el controlador en sus respectivos directorios
  - Selecciona el sprite que representará a este objeto en la escena en el campo “Sprite” del Sprite Renderer



Creando recolectables: una moneda

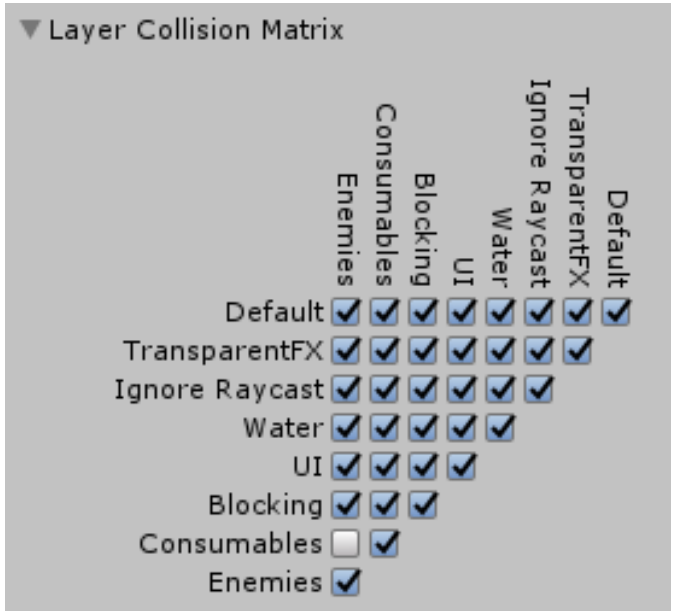
- Crea un nueva capa de ordenación para los objetos (llamada Objects), situada entre las capas del fondo y la de los personajes
- Lleva la moneda a esa capa de ordenación
- Para detectar que el jugador recoge la moneda:
  - Añade un Circle Collider 2D y ajústalo a la moneda
  - Activa la opción “Is Trigger” del collider
  - Crea un Tag para todos los objetos recolectables (“CanBePickedUp”) y asígnaselo a CoinObject
  - Crea el prefab CoinObject y borra la instancia de la escena

Optimizando el cálculo de colisiones

- Unity permite definir entre qué objetos hay que calcular colisiones y entre qué objetos no (p.e., queremos calcular cuando el personaje toca una moneda, pero no cuando lo hace un enemigo)
- Por defecto, todos los GameObjects (con colliders) calculan colisiones con los demás, independientemente de la capa donde se encuentren
  - Pero podemos especificar que no se calculen colisiones entre GameObjects de ciertas capas
- Vamos a crear dos nuevas capas:
  - “Consumables”: para todos los objetos que podrá recoger el jugador (incluyendo las monedas)
  - “Enemies”: para los enemigos
- Asigna los prefab CoinObject y Enemy a sus respectivas capas

Optimizando el cálculo de colisiones

- Para definir entre qué capas se calculan colisiones: Edit\Project Settings\Physics 2D
- En el apartado Layer Collision Matrix, desactivar las interacciones no deseadas:



Creando recolectables: una moneda

- Como se calculará la colisión entre la moneda y el personaje, y el collider de la moneda tiene activada la opción Is Trigger, Unity llamará al método OnTriggerEnter2D en ambos objetos cuando se toquen
- Añade el siguiente código a Player.cs, para desactivar los recolectables al tocarlos:

```
void OnTriggerEnter2D(Collider2D other)
{
  if (other.CompareTag("CanBePickedUp")) {
    other.gameObject.SetActive(false);
  }
}
```

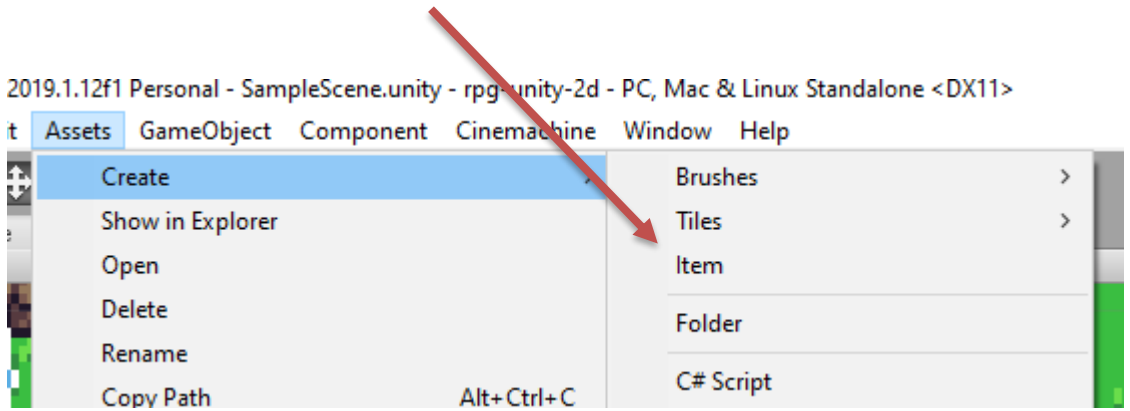
Scriptable Objects

- Es una funcionalidad de Unity para almacenar información desde el editor, y utilizarla en tiempo de ejecución
- Son clases de C#, pero no derivan de MonoBehaviour, por lo que no se pueden añadir a los GameObjects (pero sí dentro de otros scripts)
- Facilitan la creación de objetos similares con la misma información, y optimizan el uso de memoria, ya que sólo hay una copia en memoria
  - P.e: si creamos un componente MonoBehaviour dentro de un prefab, los datos dentro del mismo se copiarán para cada instancia. Si todas las instancias de un prefab hacen referencia a un ScriptableObject, en memoria sólo hay una copia de dichos datos
- Se almacenan como un asset, por lo que la información que contiene se mantiene entre distintas ejecuciones del editor de Unity

Scriptable Objects

- En nuestro ejemplo usaremos los Scriptable Objects para almacenar la información asociada a cada recolectable
- Crea el script “Item.cs” en el directorio Assets/Scripts/Scriptable Objects:

```
[CreateAssetMenu(fileName = "New item", menuName = "Item")]
public class Item : ScriptableObject
{
    public string objectName;
    public Sprite sprite;
    public int quantity;
    public bool stackable;
    public enum ItemType { COIN, HEALTH};
    public ItemType itemType;
}
```



Scriptable Objects

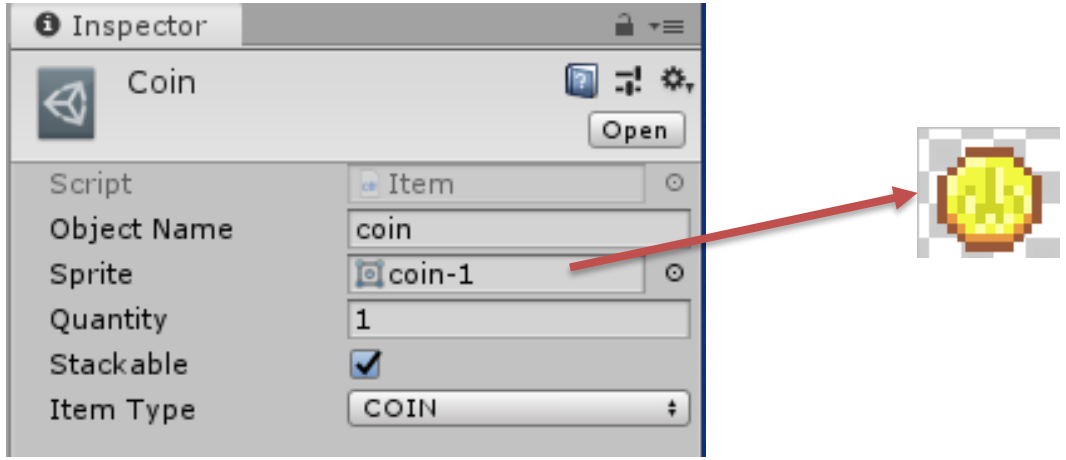
- Para poder incluir el script anterior en un GameObject, tenemos que crear un MonoBehaviour que lo incluya.
- Añade el siguiente script, llamado “Consumable”:

```
public class Consumable : MonoBehaviour
{
    public Item item;
}
```

  - Sólo se encarga de almacenar una referencia a un objeto Item (le asignaremos el asset desde el editor)
- Añade este script al prefab de CoinObject

Scriptable Objects

- Vamos a crear el Scriptable Object que describe a las monedas, y asignárselo al prefab:
  - En el directorio de Scriptable Objects, crea un nuevo Item, llamado Coin
  - Selecciona el nuevo objeto y, en el inspector, cambia las propiedades
  - Asigna el nuevo objeto al campo correspondiente del prefab



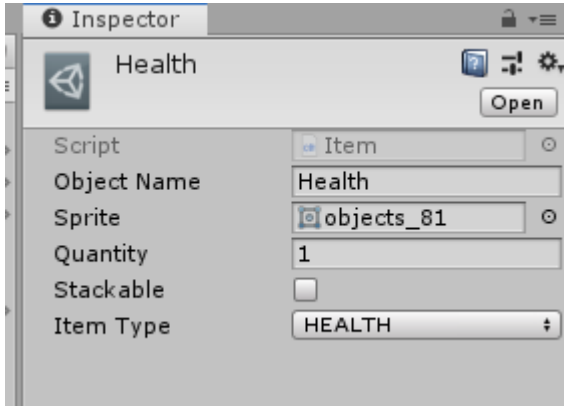
Scriptable Objects

- Ahora podemos saber el objeto con el que ha colisionado el personaje:

```
public class Player : Character {
  private void OnTriggerEnter2D(Collider2D other) {
    if (other.CompareTag("CanBePickedUp"))
    {
      Item item = other.gameObject.GetComponent<Consumable>().item;
      if (item != null) {
        Debug.Log("Item: " + item.objectName);
        other.gameObject.SetActive(false);
      }
    }
  }
}
```

Ejercicio

- Repite los pasos anteriores para crear un recolectable de vida



Aumentando la vida del personaje

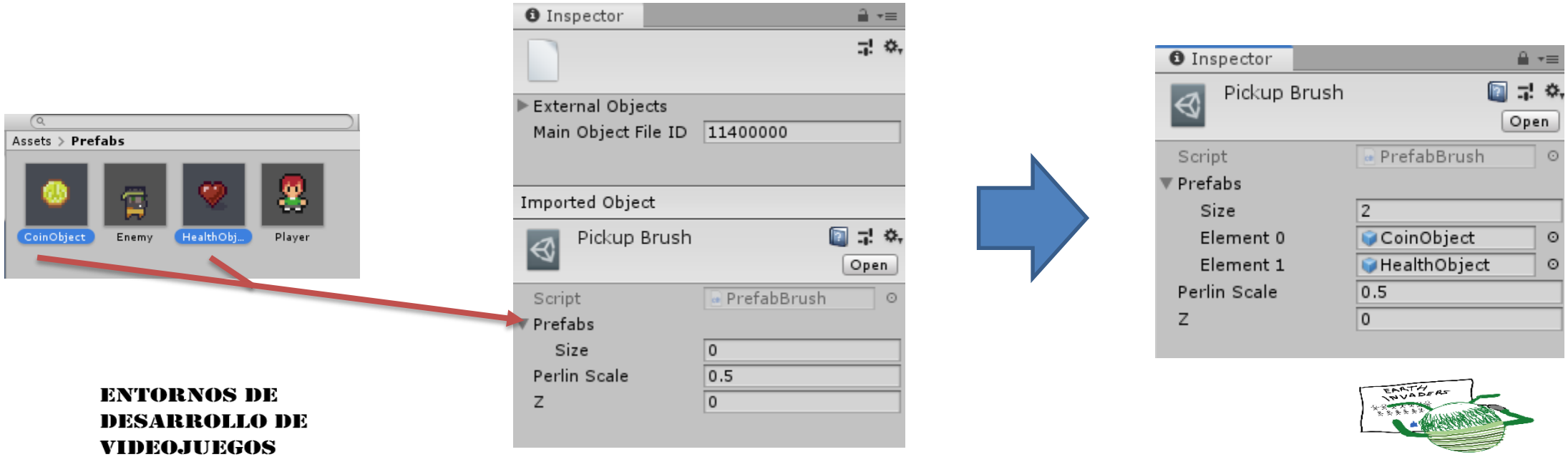
- Ahora podemos incrementar la vida del personaje al recoger un corazón:

```
public class Player : Character {
  private void OnTriggerEnter2D(Collider2D other) {
    if (other.CompareTag("CanBePickedUp")) {
      Item item = other.gameObject.GetComponent<Consumable>().item;
      if (item != null) {
        Debug.Log("Item: " + item.objectName);
        switch(item.itemType) {
          case Item.ItemType.COIN:
             break;
          case Item.ItemType.HEALTH:
            AdjustHitPoints(item.quantity);
            break;
        }
        other.gameObject.SetActive(false);
      }
    }
  }
```

```
  public void AdjustHitPoints(int amount) {
    health += amount;
    Debug.Log("Health: " + health);
  }
}
```

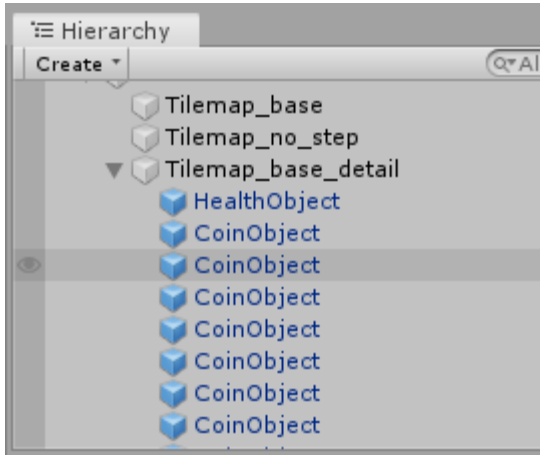

Posicionando prefabs alineados a los tiles

- El paquete unity-extras-2d también ofrece una herramienta para colocar prefabs alineados a la cuadrícula:
  - Por ejemplo, en TilePalettes, crea un Prefab brush (Create\Brushes\Prefab Bush)
  - Llámalo Pickup Brush
  - Arrastra los prefab de la moneda y la vida al campo Prefabs



Posicionando prefabs alineados a los tiles

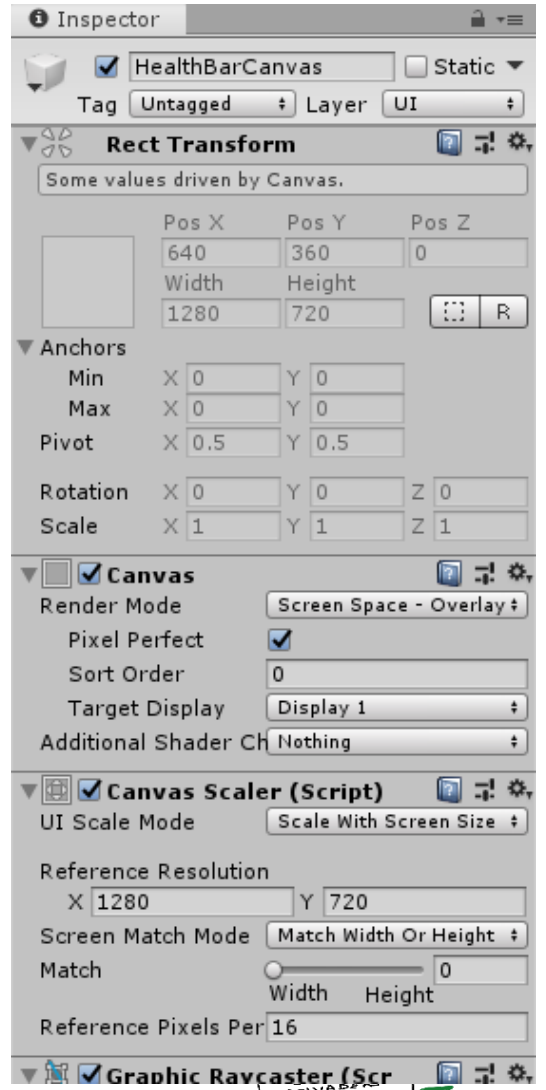
- En el panel Tile Palette, seleccionar la opción “Pickup Brush”, seleccionar el tilemap donde añadirlos (puedes crear uno nuevo), y sitúa tantos pickups como desees
  - Así se dibujan recolectables aleatoriamente. También se puede crear un pincel por cada tipo de recolectable



Las instancias aparecen como hijas del tilemap seleccionado

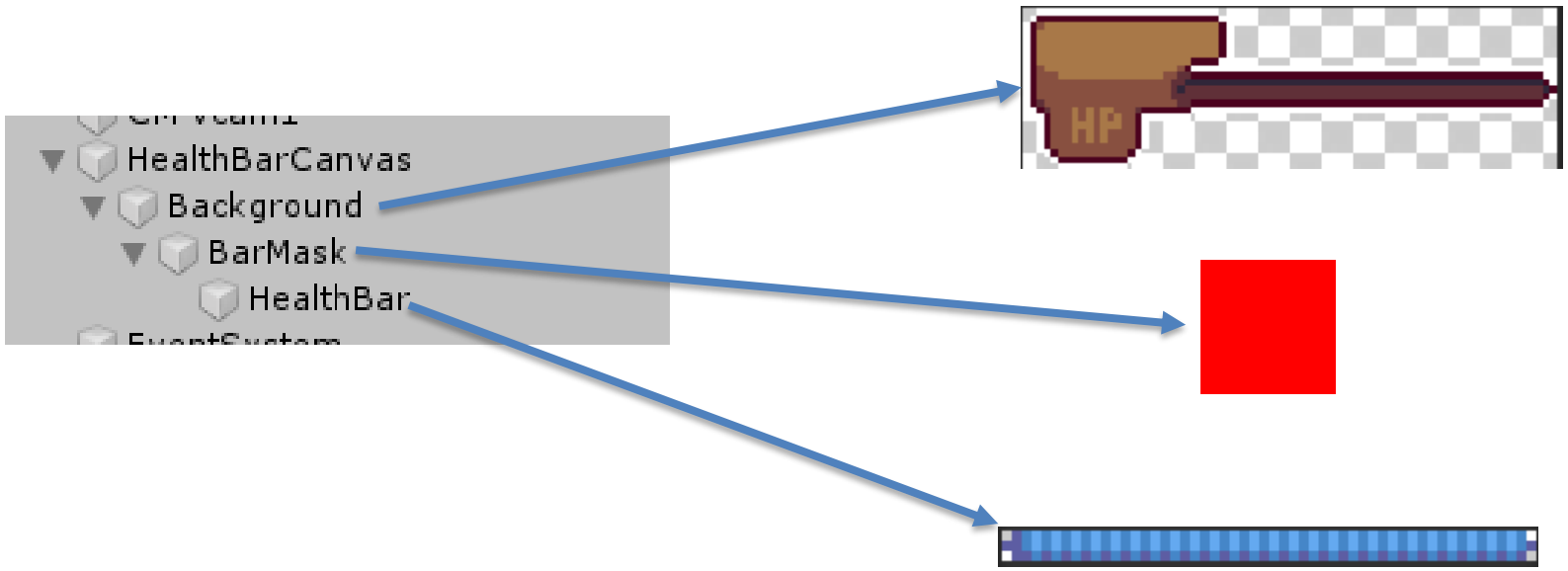
Barras de vida

- Vamos a implementar una interfaz de usuario para mostrar la barra de vida
  - Crea un Canvas (llamado HealthBarCanvas)
    - Asegúrate de que estás en modo Screen Space-Overlay
    - Activa la opción Pixel Perfect
    - En el Canvas Scaler, selecciona el modo de escalado “Scale With Screen Size”, establece la resolución de referencia (1280x720) y el número de pixeles por unidad a 16
  - Como el usuario no interactúa con la barra, puedes borrar el objeto EventSystem



Barras de vida

- La barra esta compuesta por 3 imágenes anidadas, sirviendo la intermedia como máscara:



Barras de vida

- Imagen de fondo:
  - Ánclala a la esquina superior derecha, dejando un margen con respecto a los bordes
- Imagen de máscara
  - Sitúala en el espacio donde aparecerá la barra de vida
  - Cambia el tipo de imagen a “Filled”, Fill Method a “Horizontal” y Fill Origin a “Left”.
  - Añade un componente Mask a la imagen BarMask
- Imagen de la barra de vida
  - Haz que ocupe todo el espacio del padre
- Si cambias el valor de Fill Amount en la imagen de la máscara, se debería animar la barra de vida

Barras de vida

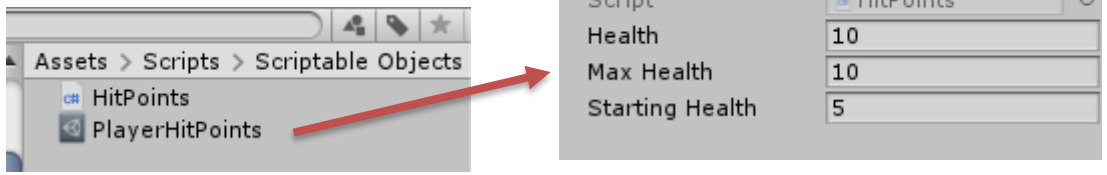
- Vamos a añadir también una etiqueta con la cantidad de vida restante
  - Añade un Text como hijo a la imagen de fondo (HPText)
  - Ajusta la posición con respecto a la esquina superior izquierda del padre
  - Selecciona la fuente slkscr.ttf y ajusta las propiedades a tu gusto
  - Ajusta el tamaño y alineamiento para que la etiqueta quede siempre centrada
  - Prueba con números de distintas cifras para comprobar que la etiqueta siempre queda bien

Barras de vida

- Vamos a conectar la barra de vida con las variables correspondientes.
- Una forma de compartir el contador de vida entre la barra de salud y el objeto Player sin acoplarlos es usando un Scriptable Object

```
[CreateAssetMenu(fileName ="HitPoints", menuName = "HitPoints")]
public class HitPoints : ScriptableObject {
    public int health;
    public int maxHealth;
    public int startingHealth;
}
```

- Create\HitPoints, y llámalo PlayerHitPoints



Barras de vida

- El código en Character.cs se simplifica:

```
public abstract class Character : MonoBehaviour {
    public HitPoints hitPoints;
}
```

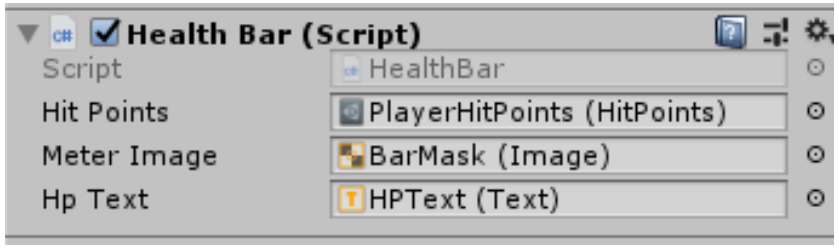
Barras de vida

```
public class Player : Character {
  private void Start() {
    hitPoints.health = hitPoints.startingHealth;
  }
  private void OnTriggerEnter2D(Collider2D other) {
    if (other.CompareTag("CanBePickedUp")) {
      Item item = other.gameObject.GetComponent<Consumable>().item;
      if (item != null) {
        Debug.Log("Item: " + item.objectName);
        bool shouldDisappear = false;
        switch(item.itemType) {
          case Item.ItemType.COIN:
            shouldDisappear = true;
            break;
          case Item.ItemType.HEALTH:
            if (hitPoints.health < hitPoints.maxHealth) {
              AdjustHitPoints(item.quantity);
              shouldDisappear = true;
            }
            break;
        }
        if (shouldDisappear)
          other.gameObject.SetActive(false);
      }
    }
  }

  public void AdjustHitPoints(int amount) {
    hitPoints.health += amount;
    Debug.Log("Health: " + hitPoints.health);
  }
}
```

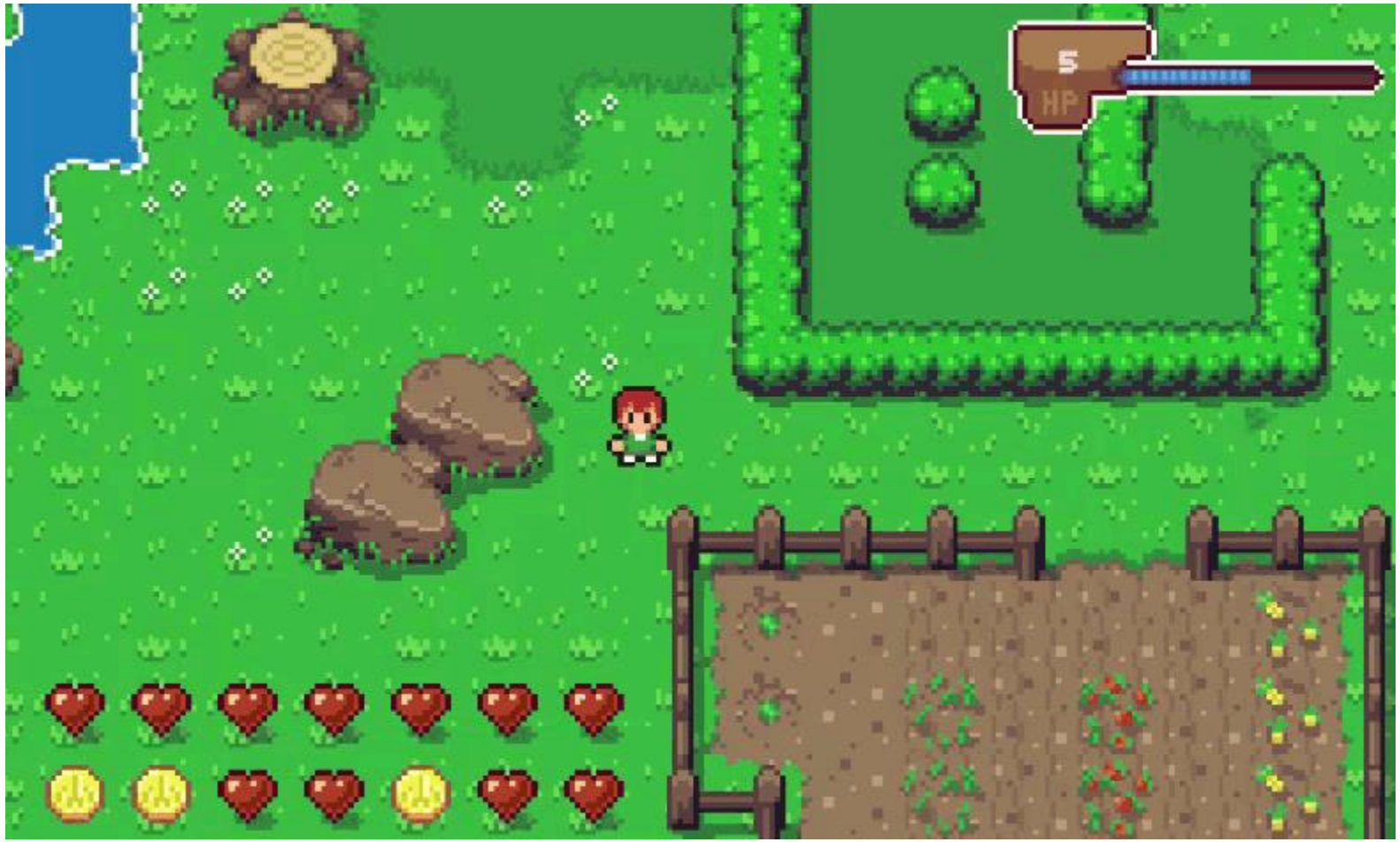
Barras de vida

- Añade el siguiente script al objeto HealthBarCanvas y haz las conexiones correspondientes en el editor:

```
using UnityEngine.UI;

public class HealthBar : MonoBehaviour {
  public HitPoints hitPoints;
  public Image meterImage;
  public Text hpText;

  void Update() {
    meterImage.fillAmount = (float)hitPoints.health / hitPoints.maxHealth;
    hpText.text = hitPoints.health.ToString();
  }
}
```

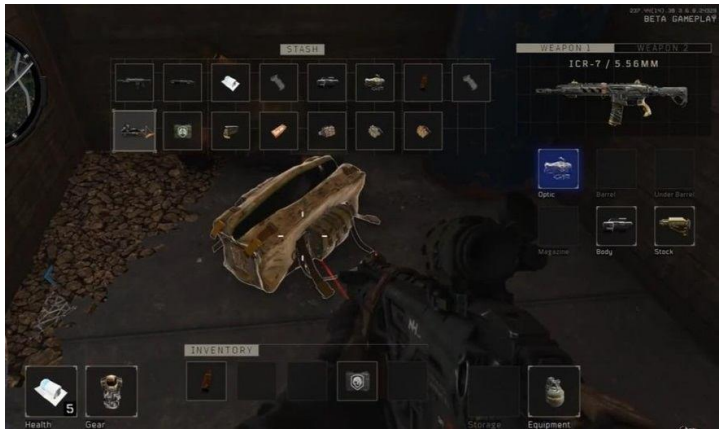


Barras de vida



Inventario

- El inventario de un juego mantiene los objetos que ha ido recogiendo el jugador y que más adelante podrá usar



Call of duty 4. Black Ops

Mario Kart

Might and Magic III

Diablo III

Inventario

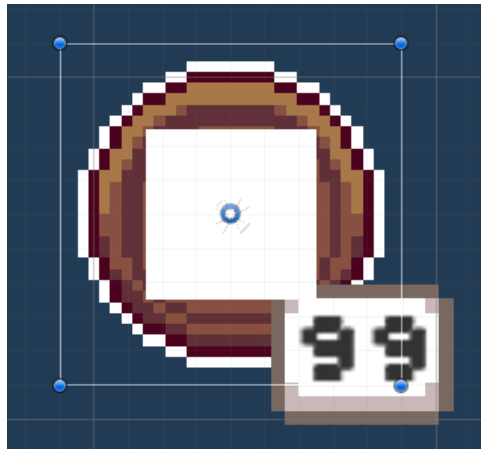
- Vamos a crear otro canvas para la barra de inventario (InventoryCanvas)
  - Puedes borrar el EventSystem
  - Ajusta las propiedades del canvas igual que hiciste con del de la barra de vida
- Los objetos en el inventario aparecerán desde la esquina superior izquierda, ordenados horizontalmente
  - Añade un empty al canvas que servirá de padre de todos los elementos visuales del inventario
  - Añádele un componente Horizontal Layout Group, para que organice a sus hijos en fila

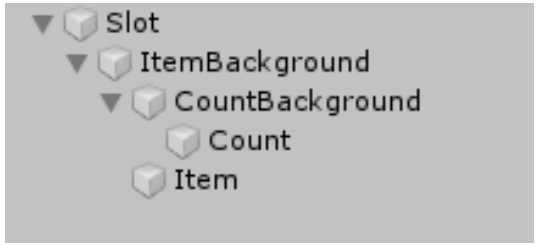
Inventario

- En el inventario de nuestro juego:
  - Tendrá 5 “slots”
  - Habrá dos tipos de elementos:
    - Los que se pueden acumular sin límite en un mismo slot (cuando el ítem correspondiente tiene marcada la opción Stackable). En este caso aparecerá un contador
    - Los que ocupan un slot (no tienen contador asociado)

Inventario

- Vamos a crear la jerarquía para un slot:
  - Crea un empty (hijo del contenedor de slots) llamado Slot
  - Ajusta su tamaño
  - Crea la jerarquía indicada con imágenes y Texts
  - Ajusta los tamaños, posiciones y alineamientos de los elementos para que al añadir la imagen de un ítem aparezca correctamente:

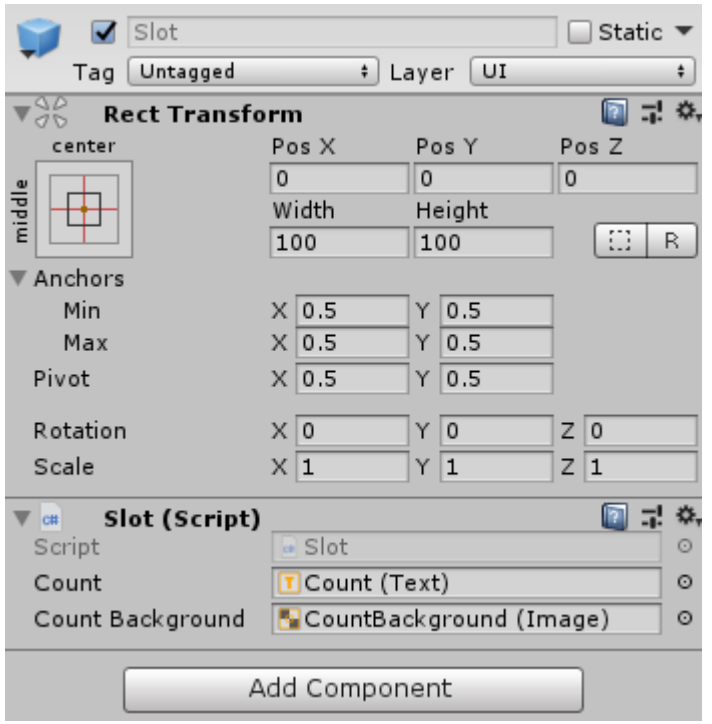




Inventario

- Vamos a crear la jerarquía para un slot:
  - Añade el siguiente script a Slot y haz las conexiones correspondientes en el Inspector:

```
using UnityEngine.UI;
public class Slot : MonoBehaviour {
    [SerializeField]
    private Text count;
    [SerializeField]
    private Image countBackground;

    public void DisableCounter() {
        count.enabled = false;
        countBackground.enabled = false;
    }

    public void SetCount(int n) {
        count.text = n.ToString();
    }
}
```



Inventario

- Crea un prefab con el Slot y elimínalo de la escena
  - Debería quedar en la escena el canvas y el contenedor de slots
- Vamos a definir una clase Managers, que mantendrá una referencia al objeto inventario
  - La clase Inventory se encargará de gestionar la lista de ítems recogidos por el jugador y de mostrarlos en pantalla
  - Crea un empty llamado Managers en la escena, y añádele el script de la siguiente transparencia

Inventario

```
using UnityEngine.UI;
public class Inventory : MonoBehaviour {
  public static Inventory instance = null;

  private void Awake() {
    if (instance != null  && instance != this) { // Definiendo un singleton en Unity
      Destroy(gameObject);
    } else {
      instance = this;
    }
  }

  public const int numSlots = 5;
  public Slot slotPrefab;
  public GameObject slotsParent;

  Item[] items = new Item[numSlots];
  Slot [] slots = new Slot[numSlots];
```

```
Slot CreateNewSlotForItem(string name, Item item) {
  Slot newSlot = Instantiate<Slot>(slotPrefab);
  newSlot.name = name;
  newSlot.transform.SetParent(slotsParent.transform);
  Transform imageTransform = newSlot.transform.Find("ItemBackground/Item");
  Image itemImage = imageTransform.gameObject.GetComponent<Image>();
  itemImage.sprite = item.sprite;
  if (item.stackable) {
    newSlot.SetCount(item.quantity);
  } else {
    newSlot.DisableCounter();
  }
  return newSlot;
}
```

```
public bool AddItem(Item itemToAdd) {
  for (int i = 0; i < numSlots; i++) {
    if (items[i] != null &&
        items[i].itemType == itemToAdd.itemType &&
        itemToAdd.stackable) {
      items[i].quantity++;
      slots[i].SetCount(items[i].quantity);
      return true;
    } else if (items[i] == null) {
      items[i] = Instantiate(itemToAdd);
      slots[i] = CreateNewSlotForItem("ItemSlot" + i, items[i]);
      return true;
    }
  }
  return false;
 }
}
```


Inventario

- El siguiente paso es actualizar el script de Player.cs:

```
private void OnTriggerEnter2D(Collider2D other) {
  if (other.CompareTag("CanBePickedUp")) {
    Item item = other.gameObject.GetComponent<Consumable>().item;
    if (item != null) {
      bool shouldDisappear = false;
      switch(item.itemType) {
        case Item.ItemType.COIN:
          shouldDisappear = Inventory.instance.AddItem(item);
          break;
        case Item.ItemType.HEALTH:
          if (hitPoints.health < hitPoints.maxHealth) {
            AdjustHitPoints(item.quantity);
            shouldDisappear = true;
          }
          break;
       }
       if (shouldDisappear) other.gameObject.SetActive(false);
    }
  }}
```

Gestor del juego

- Vamos a preparar otro gestor que se encargue de instanciar al personaje y a los enemigos

```
public class GameManager : MonoBehaviour {
  public static GameManager instance = null;
  private void Awake() {
    if (instance != null && instance != this) {
      Destroy(gameObject);
    } else {
      instance = this;
    }
  }
  void Start() {
    SetupScene();
  }
  public void SetupScene() {
  }
}
```


Gestor del juego. Puntos de inicio/regeneración

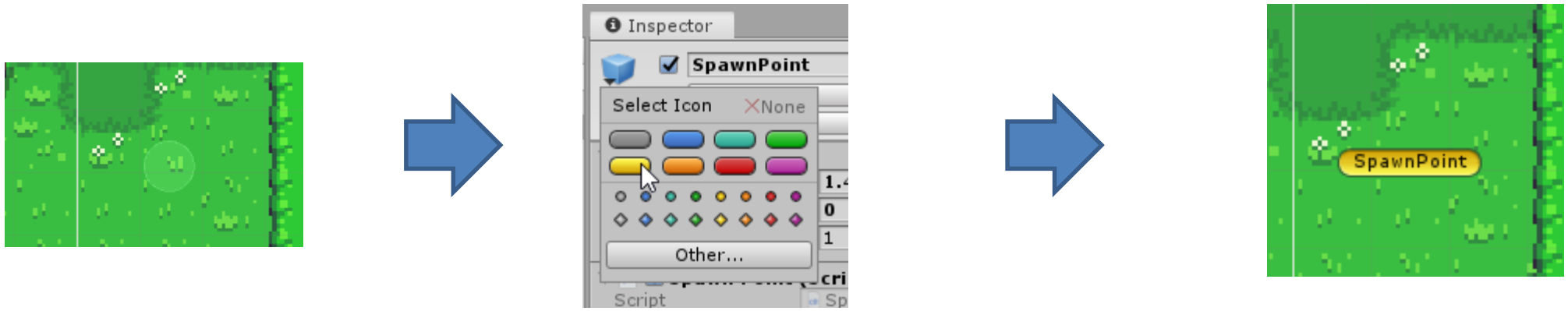
- Los *spawn points* son lo lugares del nivel donde aparecen los personajes, los enemigos o los recolectables
- Es común que los enemigos y los recolectables se generen a intervalos regulares de tiempo
- Vamos a preparar un prefab genérico que sirva para generar al personaje y a los enemigos
  - Añade un GameObject llamado “SpawnPoint” a la escena y añádele el siguiente script

Gestor del juego. Puntos de inicio/regeneración

```
public class SpawnPoint : MonoBehaviour {
  public GameObject prefabToSpawn;
  public float repeatInterval;

  void Start() {
    if (repeatInterval > 0) {
      InvokeRepeating("SpawnObject", 0.0f, repeatInterval);
    }
  }

  public GameObject SpawnObject() {
    if (prefabToSpawn != null) {
      return Instantiate(prefabToSpawn, transform.position, Quaternion.identity);
    }
    return null;
  }
}
```

Gestor del juego. Puntos de inicio/regeneración

- Este objeto servirá para generar instancias de un prefab a intervalos definidos desde el editor
- Como no tiene icono asociado, es difícil de ver, pero Unity permite asociarle uno para facilitar su visualización en la escena:



Gestor del juego. Puntos de inicio/regeneración

- Crea un prefab con el SpawnPoint y quítalo de la escena
- Arrastra una instancia del prefab hasta el punto de inicio del jugador
  - Cambia el nombre a PlayerSpawnPoint
  - Configura las propiedades del script (al prefab del personaje e intervalo cero, ya que no queremos generar más de un jugador)
- Modifica y configura el gestor del juego para que instancie al jugador al arrancar

```
public class GameManager : MonoBehaviour {
  public SpawnPoint playerSpawnPoint;
  […]
  public void SpawnPlayer() {
    if (playerSpawnPoint != null) {
      GameObject player = playerSpawnPoint.SpawnObject();
    }
  }
```

```
public void SetupScene() {
  SpawnPlayer();
}
```

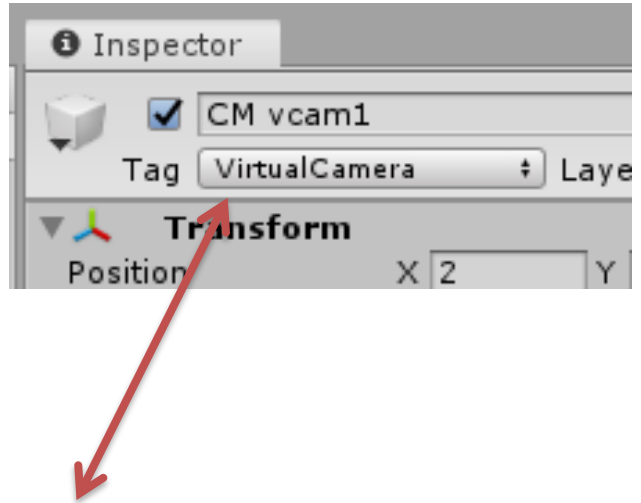

Gestor del juego. Puntos de inicio/regeneración

- Ejercicio
  - Crea un punto de regeneración de enemigos para que genere un enemigo cada 10 segundos

Gestor de cámara

- Te habrás fijado que la cámara ya no sigue al personaje
  - Al generar el personaje por código, la cámara virtual no sabe a quién seguir
- Añade a Managers un nuevo script llamado CameraManager

```
using Cinemachine;
public class CameraManager : MonoBehaviour {
  public static CameraManager instance;
  [HideInInspector]
  public CinemachineVirtualCamera virtualCamera;
  void Awake() {
    if (instance != null && instance != this) {
      Destroy(gameObject);
    } else {
      instance = this;
      GameObject vCamGO = GameObject.FindWithTag("VirtualCamera");
      virtualCamera = vCamGO.GetComponent<CinemachineVirtualCamera>();
    }
  }
}
```



Gestor de cámara

- Modifica el gestor del juego para que conecte la cámara con el jugador recién creado:

```
public class GameManager : MonoBehaviour
{
  […]
  public void SpawnPlayer() {
    if (playerSpawnPoint != null) {
      GameObject player = playerSpawnPoint.SpawnObject();
      CameraManager.instance.virtualCamera.Follow = player.transform;
    }
  }
 […]
}
```

Diseño de la clase Character

- Implementar la funcionalidad común a jugador y enemigos:

```
public abstract class Character : MonoBehaviour {
  public HitPoints hitPoints;
  public virtual IEnumerator DamageCharacter(int damage, float interval) {
    while (true) {
      hitPoints.health -= damage;
      hitPoints.health = hitPoints.health < 0 ? 0 : hitPoints.health;
      if (hitPoints.health <= 0) {
        KillCharacter();
        break;
      }
      if (interval > 0.0f) {
        yield return new WaitForSeconds(interval);
      } else {
        break;
      }
    }
  }
```

```
  public virtual void KillCharacter() {
     Destroy(gameObject);
   }
   public virtual void ResetCharacter() {
     hitPoints.health =
          hitPoints.startingHealth;
   }
}
```

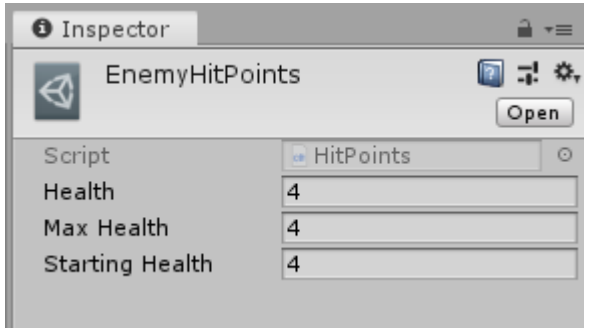
Dando funcionalidad al enemigo

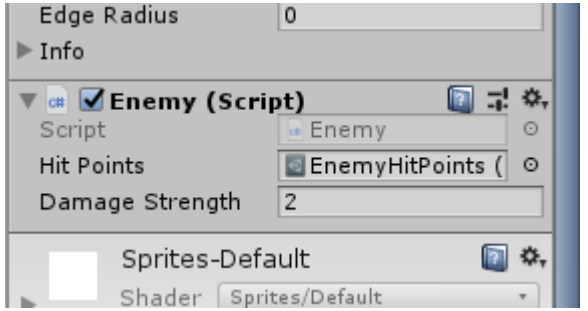
Enemy.cs

```
public class Enemy : Character {
  public int damageStrength;
  Coroutine damageCoroutine;
  private void OnEnable() {// Llamada cada vez que se habilita o activa el gameobject
    hitPoints = Instantiate(hitPoints);
    ResetCharacter();
  }
  private void OnCollisionEnter2D(Collision2D other) {
    if (other.gameObject.CompareTag("Player")) {
      Player player = other.gameObject.GetComponent<Player>();
      if (damageCoroutine == null){
        damageCoroutine = StartCoroutine(player.DamageCharacter(damageStrength, 1.0f));
      }
    }
  }
  private void OnCollisionExit2D(Collision2D other) {
    if (other.gameObject.CompareTag("Player")) {
      if (damageCoroutine != null) {
        StopCoroutine(damageCoroutine);
        damageCoroutine = null;
      }
}}}
```

Dando funcionalidad al enemigo

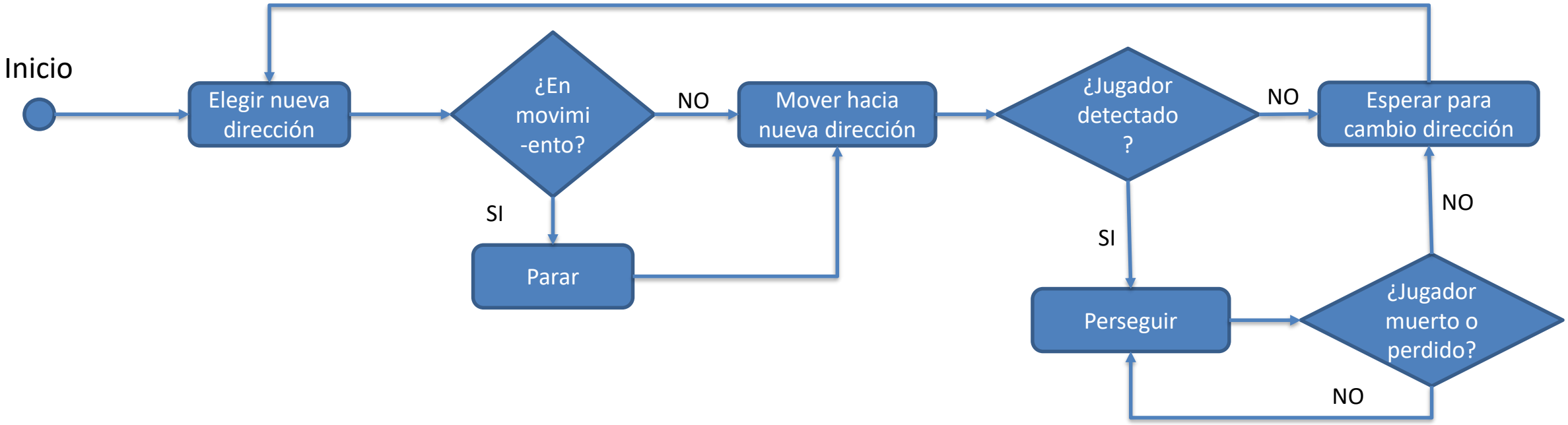
- Añade el script al prefab del enemigo
- Crea un scriptable object de tipo HitPoint para el enemigo, con sus parámetros de vida
- Configura los puntos de daño en el prefab
- Prueba a acercarte al enemigo. El personaje perderá vida cada segundo hasta que su salud llegue a cero





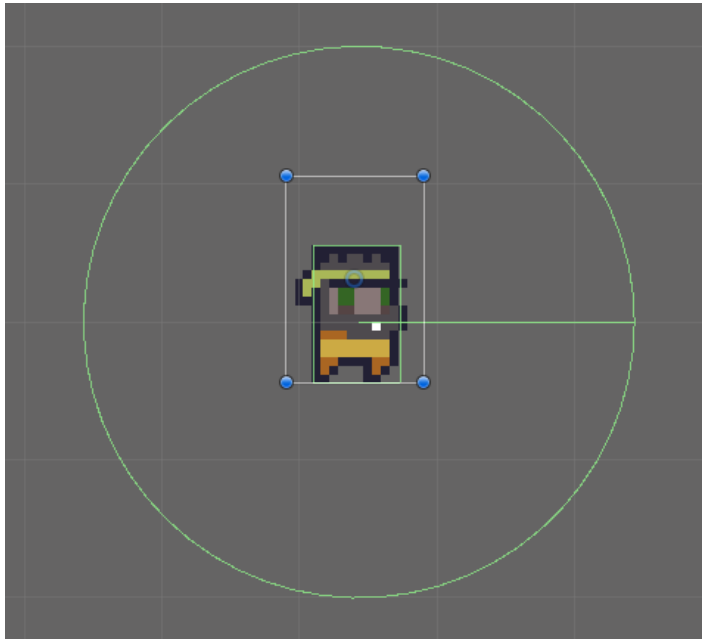
Dando inteligencia al enemigo

- El siguiente paso consiste en hacer que el enemigo pueda atacar al jugador
  - Estrategia: el enemigo deambula aleatoriamente por el nivel. Si detecta al jugador cerca, entonces lo perseguirá hasta que se vuelva a escapar, el jugador lo mate, o el jugador muera



Dando inteligencia al enemigo

- Para detectar al jugador usaremos un Circle Collider 2D
  - El radio especifica la distancia de detección
  - Activa la opción Is Trigger



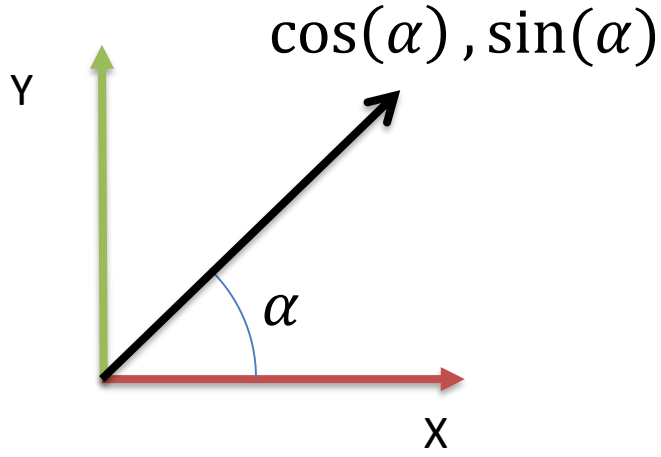
Dando inteligencia al enemigo

```
using Random = UnityEngine.Random;

[RequireComponent(typeof(Rigidbody2D))]
[RequireComponent(typeof(CircleCollider2D))]
[RequireComponent(typeof(Animator))]
public class Wander : MonoBehaviour {
  public float pursuitSpeed;
  public float wanderSpeed;
  float currentSpeed;

  public float directionChangeInterval;
  public bool followPlayer;
  Coroutine moveCoroutine;

  Rigidbody2D rb2d;
  Animator animator;

  Transform targetTransform = null;

  Vector3 endPosition;
  float currentAngle = 0;
```

```
void Start() {
  animator = GetComponent<Animator>();
  currentSpeed = wanderSpeed;
  rb2d = GetComponent<Rigidbody2D>();
  StartCoroutine(WanderRoutine());
}

IEnumerator WanderRoutine() {
  while (true) {
    ChooseNewEndpoint();
    if (moveCoroutine != null) {
      StopCoroutine(moveCoroutine);
    }
    moveCoroutine = StartCoroutine(Move());
    yield return new
          WaitForSeconds(directionChangeInterval);
  }
}
```

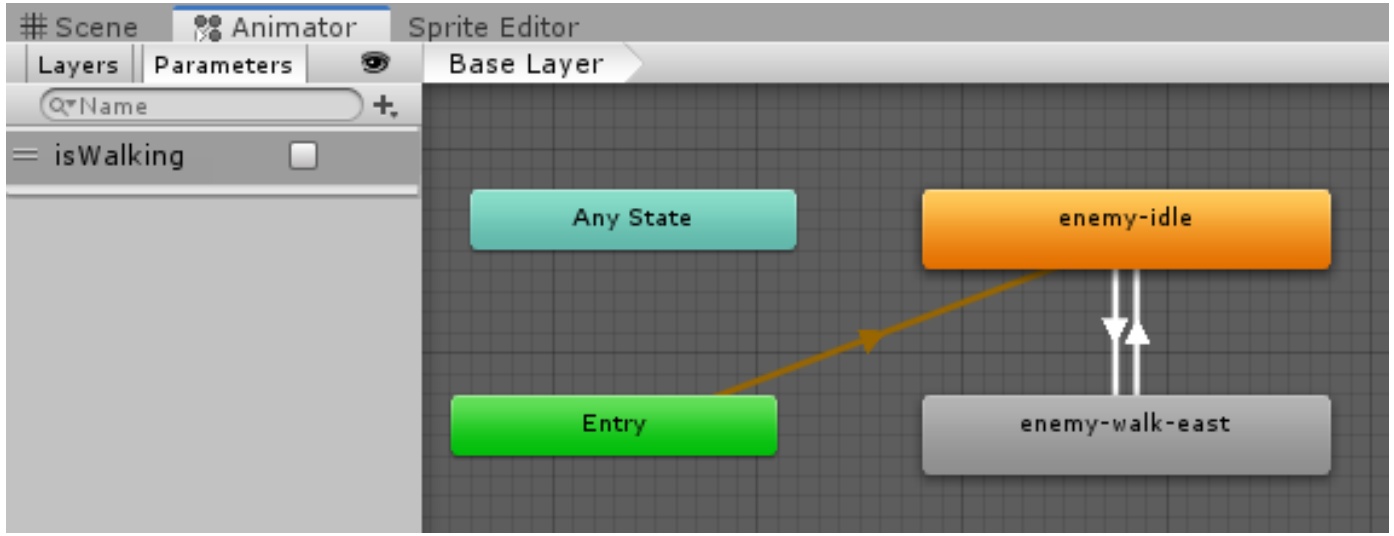
Dando inteligencia al enemigo

```
  void ChooseNewEndpoint() {
    currentAngle += Random.Range(0, 360);
    currentAngle = Mathf.Repeat(currentAngle, 360);
    endPosition += Vector3FromAngle(currentAngle);
  }

  Vector3 Vector3FromAngle(float inputAngleDegrees) {
    float radians = inputAngleDegrees * Mathf.Deg2Rad;
    return new Vector3(Mathf.Cos(radians), Mathf.Sin(radians), 0);
  }

  IEnumerator Move(Rigidbody2D rb2d, float currentSpeed) {
    return null; // TODO
  }
}
```



Dando inteligencia al enemigo

- Vamos a completar las animaciones del enemigo
  - Configura las animaciones y las transiciones
  - Crea un parámetro booleano para cambiar de estado



Dando inteligencia al enemigo

- Completa el código de Wander.cs:

```
IEnumerator Move() {
  float remainingDistance = (transform.position - endPosition).sqrMagnitude;
  while (remainingDistance > float.Epsilon) {
    if (targetTransform != null) {
      endPosition = targetTransform.position;
    }
    animator.SetBool("isWalking", true);
    Vector3 newPosition =
            Vector3.MoveTowards(rb2d.position, endPosition, currentSpeed * Time.deltaTime);
    rb2d.MovePosition(newPosition);
    remainingDistance = (transform.position - endPosition).sqrMagnitude;
    yield return new WaitForFixedUpdate();
  }
  animator.SetBool("isWalking", false);
}
```

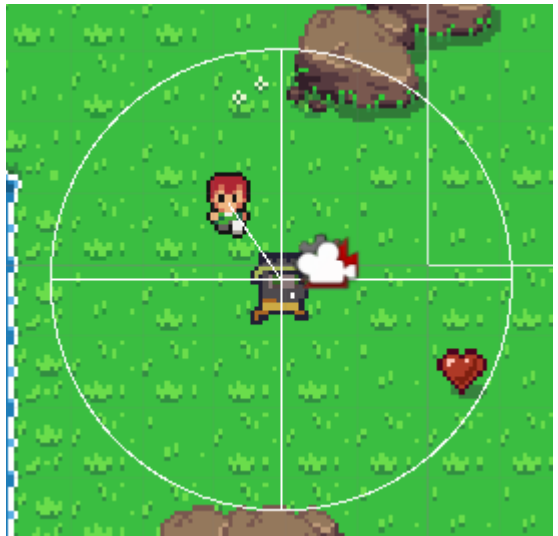
Dando inteligencia al enemigo

```
private void OnTriggerEnter2D(Collider2D other) {
  if (other.CompareTag("Player") && followPlayer) {
    currentSpeed = pursuitSpeed;
    targetTransform = other.transform;
    if (moveCoroutine != null) {
      StopCoroutine(moveCoroutine);
    }
    moveCoroutine = StartCoroutine(Move());
  }
}

private void OnTriggerExit2D(Collider2D other) {
  if (other.CompareTag("Player")) {
    animator.SetBool("isWalking", false);
    currentSpeed = wanderSpeed;
    if (moveCoroutine != null) {
      StopCoroutine(moveCoroutine);
    }
    targetTransform = null;
 }
}
```

Depurando el movimiento del enemigo

- Añadir el siguiente código a Wander.cs para mostrar el área de detección del enemigo y su objetivo actual:

```
CircleCollider2D circleCollider;

private void OnDrawGizmos() {
  if (circleCollider != null) {
    Gizmos.DrawWireSphere(circleCollider.transform.position, circleCollider.radius);
    Gizmos.DrawLine(rb2d.position, endPosition);
  }
}
```

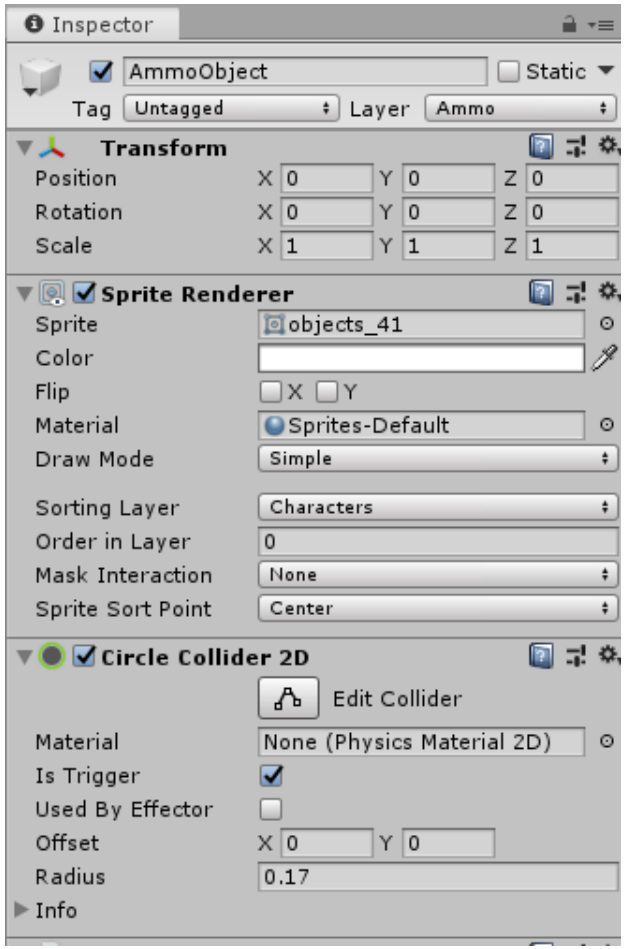


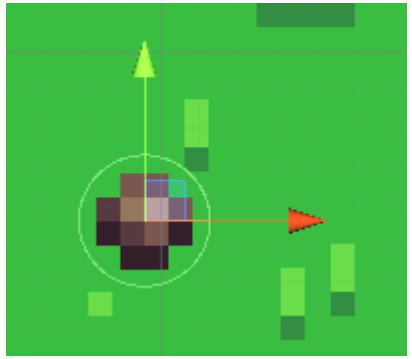
Implementando el arma del personaje

- Nuestro personaje está equipado con un tirachinas para defenderse
  - El botón izquierdo del ratón lanzará un proyectil en un arco en la dirección del puntero
  - Clases:
    - Weapon: asociada a Player, detectará si el jugador quiere disparar, cambiará la animación y creará el proyectil
    - Ammo: calcula si el proyectil ha tocado a un enemigo, y aplica el daño correspondiente
    - Arc: clase para mover el proyectil en un arco, en vez de en línea recta

Implementando el arma del personaje

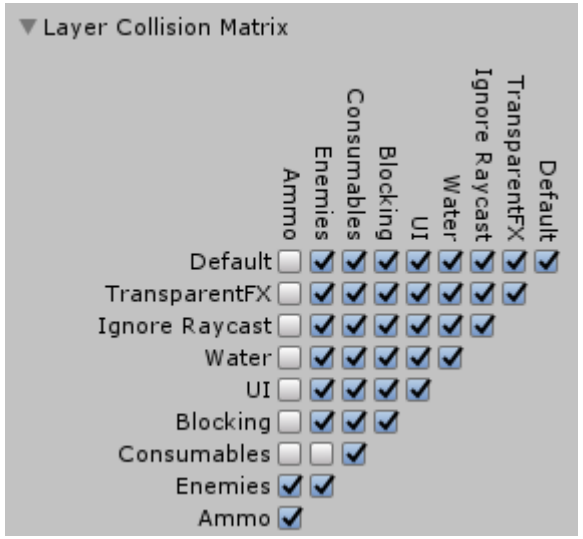
- Proyectil:
  - Crea un Empty llamado AmmoObject
  - Añádele un Sprite Renderer
    - Sorting Layer: Characters
    - La imagen correspondiente
  - Un Circle Collider 2D ajustado al sprite
    - Activa la opción Is Trigger
  - Crea una nueva capa llamada Ammo y asígnala al GameObject





Implementando el arma del personaje

- Proyectil
  - Vamos a hacer que los proyectiles sólo puedan colisionar con los enemigos y con otros proyectiles
  - Edit\Project Settings\Physics 2D



Implementando el arma del personaje

- Proyectil
  - Añade el siguiente script y crea un prefab

```
public class Ammo : MonoBehaviour {
  public int damageInflicted;

  private void OnTriggerEnter2D(Collider2D other) {
    if (other is BoxCollider2D) {
      Enemy enemy = other.gameObject.GetComponent<Enemy>();
      StartCoroutine(enemy.DamageCharacter(damageInflicted, 0.0f));
      gameObject.SetActive(false);
    }
  }
}
```

Object pooling

- Si tu juego crea y destruye muchos game objects (p.e., proyectiles), puede pasar que el juego tenga paradas o bajadas de velocidad de refresco
  - No controlamos cuándo trabaja el recolector de basura
- La técnica de object pooling minimiza la creación/destrucción de objetos activando y desactivando un pool de instancias preconstruido
  - Cuando hace falta una nueva instancia, se busca y se activa un objeto del pool
  - Cuando ya no hace falta un objeto, se desactiva y se devuelve al pool

Implementando el arma del personaje

```
public class Weapon : MonoBehaviour {
  public GameObject ammoPrefab;
  static List<GameObject> ammoPool;
  public int poolSize;

  void Awake() {
    if (ammoPool == null) {
      ammoPool = new List<GameObject>();
    }
    for (int i = 0; i < poolSize; i++) {
      GameObject ammoObject = Instantiate(ammoPrefab);
      ammoObject.SetActive(false);
      ammoPool.Add(ammoObject);
    }
  }
  void Update() {
    if (Input.GetMouseButtonDown(0)) {
      FireAmmo();
    }
  }
```

```
  GameObject SpawnAmmo(Vector3 location) {
    foreach (GameObject ammo in ammoPool) {
      if (ammo.activeSelf == false) {
        ammo.SetActive(true);
        ammo.transform.position = location;
        return ammo;
      }
    }
    return null;
  }
  void FireAmmo() {
     // TODO
  }

  private void OnDestroy() {
    ammoPool = null;
  }
}
```

Añadir a Player

Implementando el arma del personaje

- Añade el siguiente script a AmmoObject:

```
public class Arc : MonoBehaviour {
  public float maxHeight = 1.5f;
  public IEnumerator TravelArc(Vector3 destination, float duration) {
    Vector3 startPosition = transform.position;
    float percentComplete = 0.0f;
    while (percentComplete < 1.0f) {
      percentComplete += Time.deltaTime / duration;
      float currentHeight = Mathf.Sin(Mathf.PI * percentComplete) * maxHeight;
      transform.position = Vector3.Lerp(startPosition, destination, percentComplete) +
                           Vector3.up * currentHeight;
      yield return null;
    }
    gameObject.SetActive(false);
  }
}
```

Implementando el arma del personaje

- Ahora completamos el método FireAmmo de Weapon.cs:

```
public float weaponVelocity;
void FireAmmo() {
  Vector3 mousePosition = Camera.main.ScreenToWorldPoint(Input.mousePosition);
  GameObject ammo = SpawnAmmo(transform.position);
  if (ammo != null)
  {
    Arc arcScript = ammo.GetComponent<Arc>();
    float travelDuration = 1.0f / weaponVelocity;
    StartCoroutine(arcScript.TravelArc(mousePosition, travelDuration));
  }
}
```

Implementando el arma del personaje

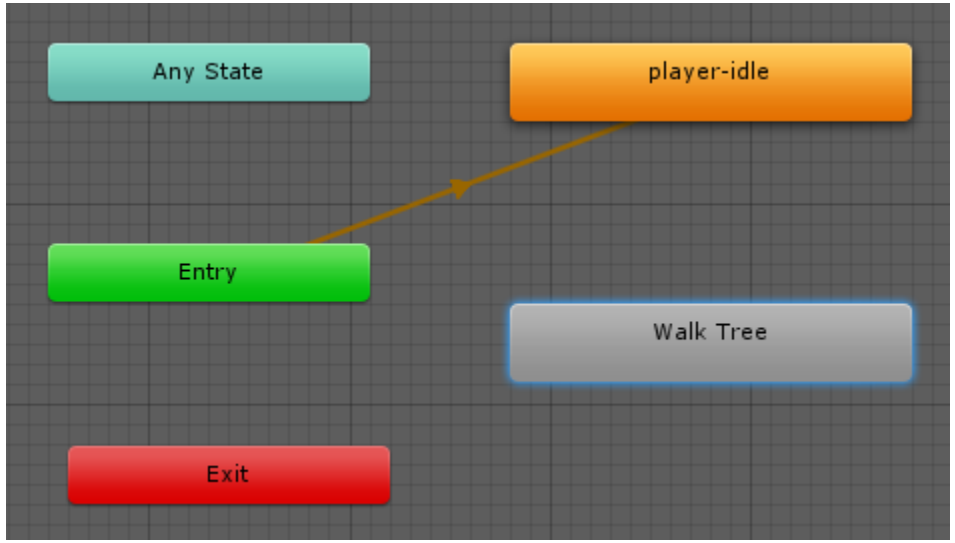
- Configura las propiedades de Weapon para que tu personaje dispare un proyectil al hacer clic con el ratón
  - Decide el número de proyectiles simultáneos (pool size) y su velocidad

Organizando la animación del personaje

- Como vamos a añadir las animaciones de ataque del personaje, primero organizaremos la máquina de estados para hacerla más legible
- La herramienta Blend Tree de Unity nos permite encapsular una serie de estados en un nodo Blend Tree, que gestiona la animación a utilizar dependiendo de uno o más parámetros
  - Así podemos reducir el número de estados mostrados en el animador
- Vamos a rehacer el animator controller del personaje para que utilice Blend Trees

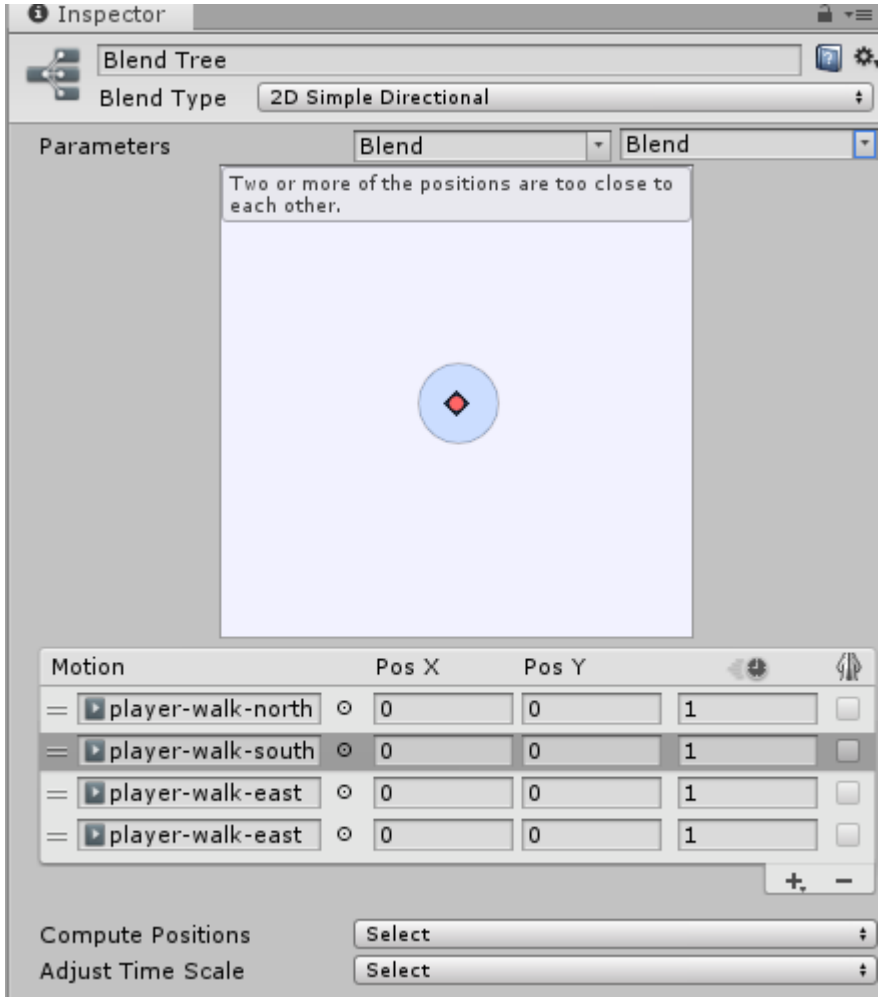
Organizando la animación del personaje

- Quita todas las animaciones excepto player-idle del controller.
- Quita la transición desde Any State a player-idle
- En el Animator, botón derecho\Create State\From New Blend Tree
  - Llámalo “Walk Tree”



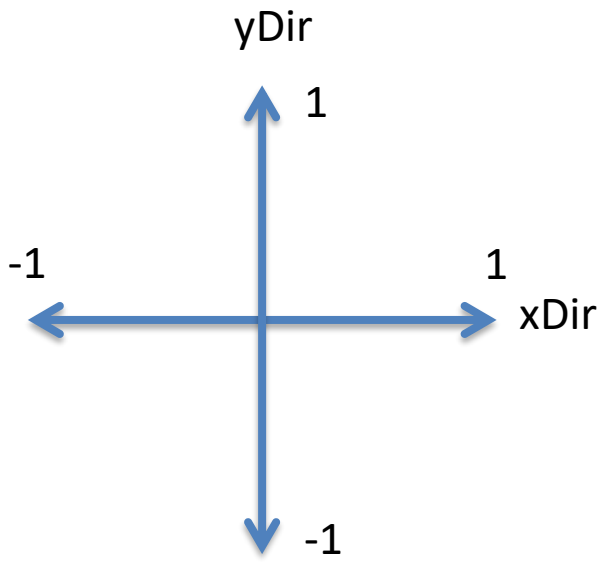
Organizando la animación del personaje

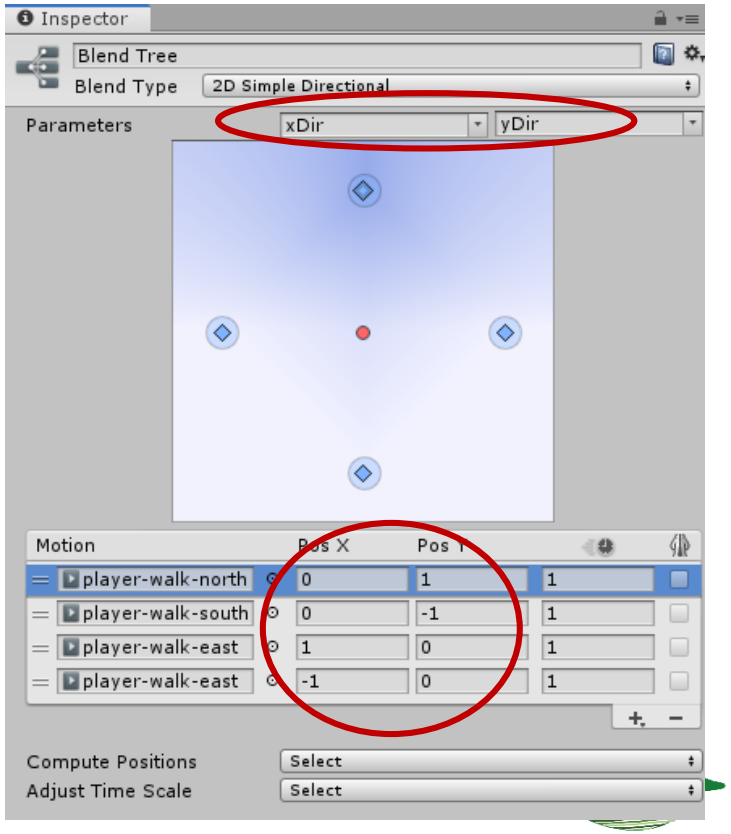
- Haz doble clic sobre el Blend Tree para abrirlo, selecciona el nodo y cambia su tipo en el Inspector de “1D” a “2D Simple Directional”
- Clic derecho\Add Motion
- Añade una animación en el campo “Motion” y repite para cada animación:



Organizando la animación del personaje

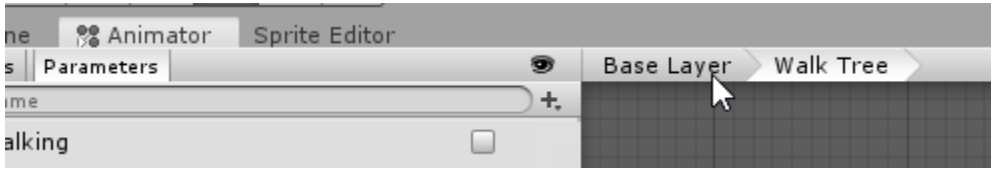
- Para controlar qué animación reproducir se usarán dos parámetros
  - Añade al Animator Controller dos parámetros de tipo float “xDdir”, “yDir” y otro de tipo bool “isWalking”
  - Dichos parámetros dividen el espacio entre -1 y 1
  - Configura el Blend Tree

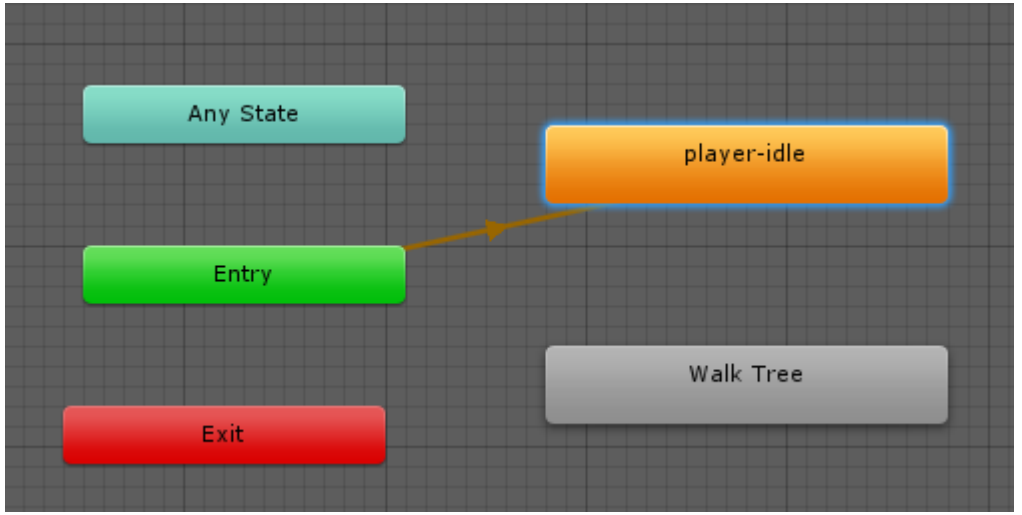




Organizando la animación del personaje

- Si en la ventana Animator volvemos a la capa base (el Blend Tree está en una capa separada), podemos ver cómo ha quedado la máquina de estados principal





Organizando la animación del personaje

- Crea las transiciones entre player-idle y el Walk Tree dependiendo del valor del parámetro “isWalking”
- Elimina el siguiente código de MovementController.cs:

```
public class MovementController : MonoBehaviour
{
    enum CharStates
    {
        walkEast = 1,
        walkSouth = 2,
        walkWest = 3,
        walkNorth = 4,

        idleSouth = 5
    }
```

Organizando la animación del personaje

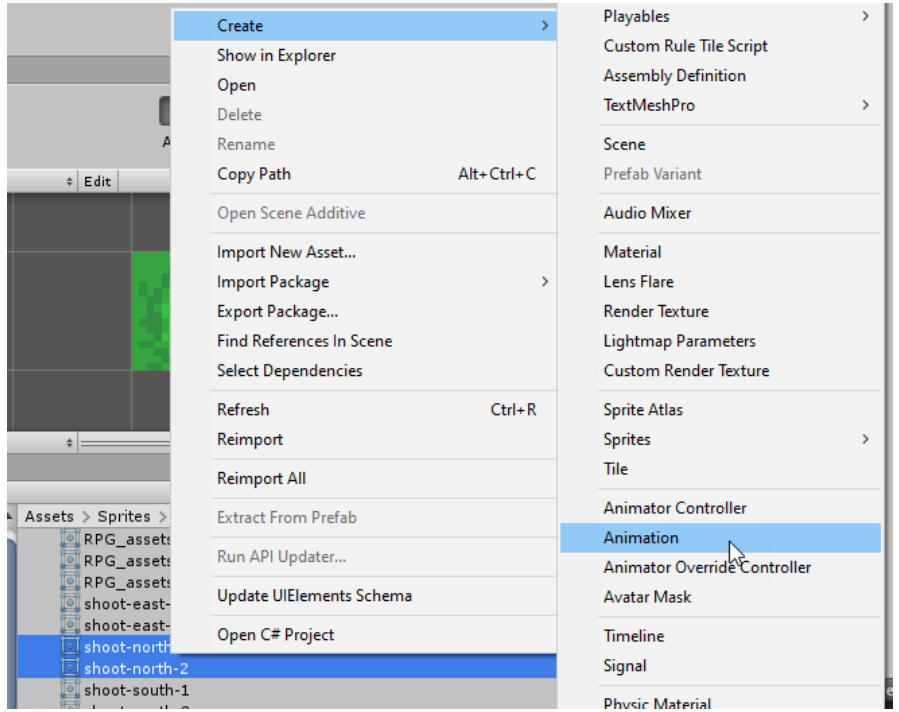
- Sustituye la función UpdateState por:

```
private void UpdateState()
    {
        bool stopped = Mathf.Approximately(movement.x, 0) && Mathf.Approximately(movement.y, 0);
        animator.SetBool("isWalking", !stopped);
        animator.SetFloat("xDir", movement.x);
        animator.SetFloat("yDir", movement.y);
        spriteRenderer.flipX = (movement.x < 0);
    }
```

- Prueba que el personaje se sigue moviendo correctamente

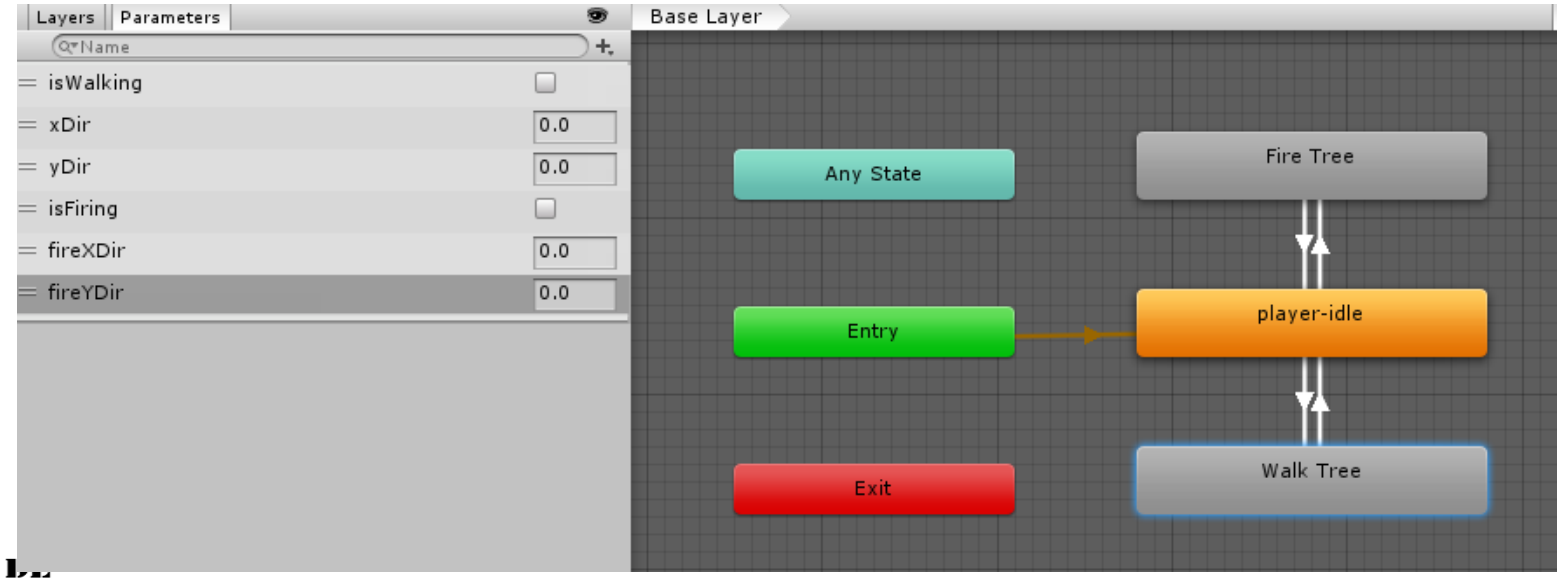
Ampliando la animación del personaje

- Vamos a crear las animaciones de disparo
  - Selecciona los sprites de una animación, botón derecho\Create\Animation:
    - player-shoot-east
    - player-shoot-west
    - player-shoot-north
    - player-shoot-south
  - Lleva los clips a su directorio



Ampliando la animación del personaje

- Repite los pasos anteriores para crear un nuevo Blend Tree (“Fire Tree”)
  - Nuevos parámetros: “isFiring”, “fireXDir”, “fireYDir”
- Crea las transiciones entre player-idle y Fire Tree
  - Condición: isFiring
  - De Fire Tree a player-idle, Exit Time: 1



Ampliando la animación del personaje

- Añade el siguiente código a Weapon.cs:

```
[RequireComponent(typeof(Animator))]
public class Weapon : MonoBehaviour {
  […]
  bool isFiring;
  [HideInInspector]
  public Animator animator;
  Camera localCamera;

  enum Quadrant {
    East, South, West, North
  }
  void Start() {
    animator = GetComponent<Animator>();
    isFiring = false;
    localCamera = Camera.main;
  }
```

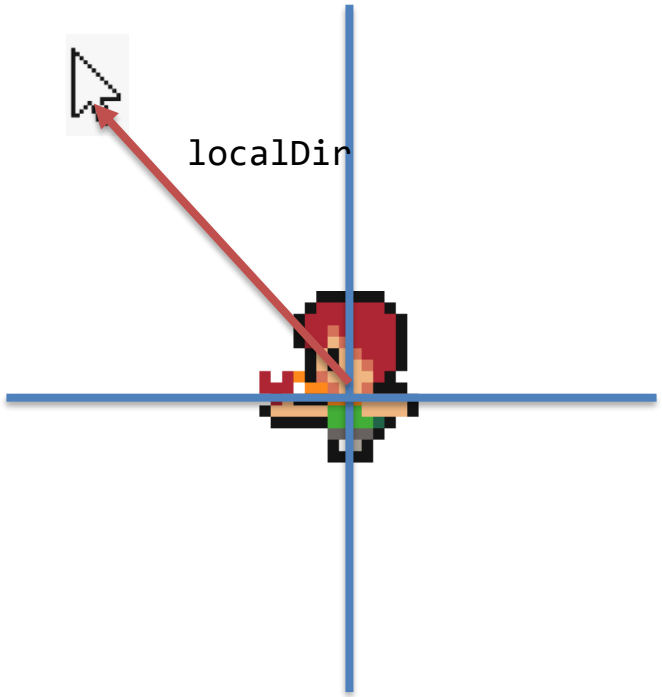
Implementando el arma del personaje

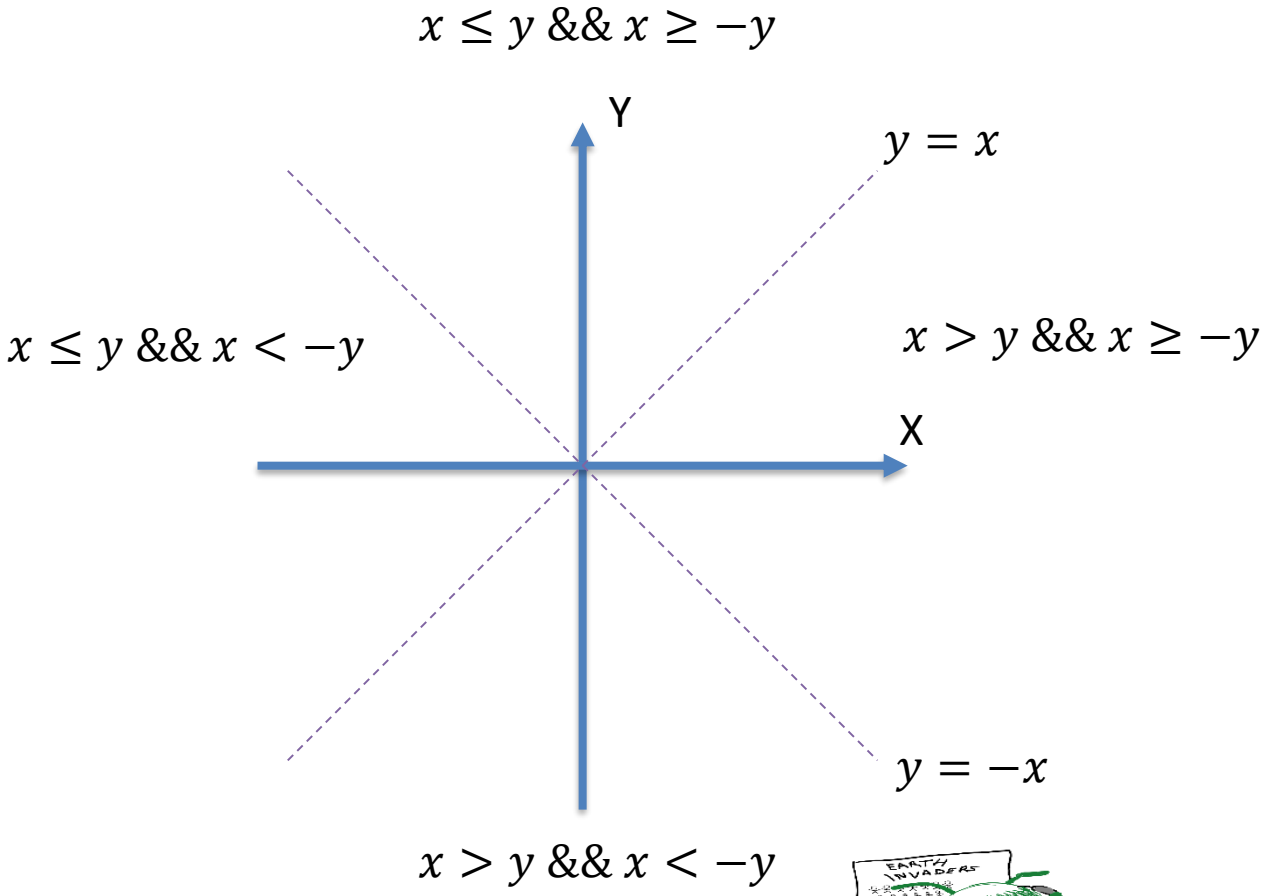
```
void Update() {
  if (Input.GetMouseButtonDown(0)) {
    isFiring = true;
    FireAmmo();
  } else {
    isFiring = false;
  }
  UpdateState();
}
```

```
private void UpdateState() {
  if (isFiring) {
    Vector2 quadrantVector = Vector2.zero;
    switch (GetQuadrant()) {
      case Quadrant.East:
        quadrantVector.x = 1.0f;
        break;
      case Quadrant.West:
        quadrantVector.x = -1.0f;
        break;
      case Quadrant.North:
        quadrantVector.y = 1.0f;
        break;
      case Quadrant.South:
        quadrantVector.y = -1.0f;
        break;
    }
    animator.SetFloat("fireXDir", quadrantVector.x);
    animator.SetFloat("fireYDir", quadrantVector.y);
  }
  animator.SetBool("isFiring", isFiring);
}
```

Implementando el arma del personaje

- Para calcular el cuadrante hacia donde se dispara:





Implementando el arma del personaje

```
Quadrant GetQuadrant() {
  Vector2 mousePosition = localCamera.ScreenToWorldPoint(Input.mousePosition);
  Vector2 playerPosition = transform.position;
  Vector2 localDir = mousePosition - playerPosition;
  if (localDir.x > localDir.y)
    if (localDir.x > -localDir.y)
      return Quadrant.East;
    else
      return Quadrant.South;
  else
    if (localDir.x < -localDir.y)
      return Quadrant.West;
    else
      return Quadrant.North;
}
```

Recibiendo daño

- Es común dar feedback al jugador cuando el personaje (o un enemigo) recibe daño:

```
public abstract class Character : MonoBehaviour {
  […]
  public virtual IEnumerator DamageCharacter(int damage, float interval) {
    while (true) {
      StartCoroutine(FlickerCharacter());
      hitPoints.health -= damage;
      […]
  }
  public virtual IEnumerator FlickerCharacter() {
    SpriteRenderer sr = GetComponent<SpriteRenderer>();
    sr.color = Color.red;
    yield return new WaitForSeconds(0.1f);
    sr.color = Color.white;
  }
}
```

Bibliografía

- Jared Halpern. Developing 2D Games with Unity. Apress, 2019
- Francis Vace. Tilemaps and Rule Tiles In Unity.
  - https://www.youtube.com/watch?v=KGrjZQ2qUoo
- Recursos:
  - Tileset mundo:
    - https://opengameart.org/content/zelda-like-tilesets-and-sprites
  - Personajes:
    - https://0x72.itch.io/pixeldudesmaker
    - https://opengameart.org/content/rpg-character-sprites