Programación de Juegos 2D en Unity

STEEL JUSTICE

Juan Alberto Domínguez Vázquez

Saúl Rodríguez Naranjo

Universidad de Huelva | Ingeniería Informática | Programación de Juegos (2024)

ÍNDICE

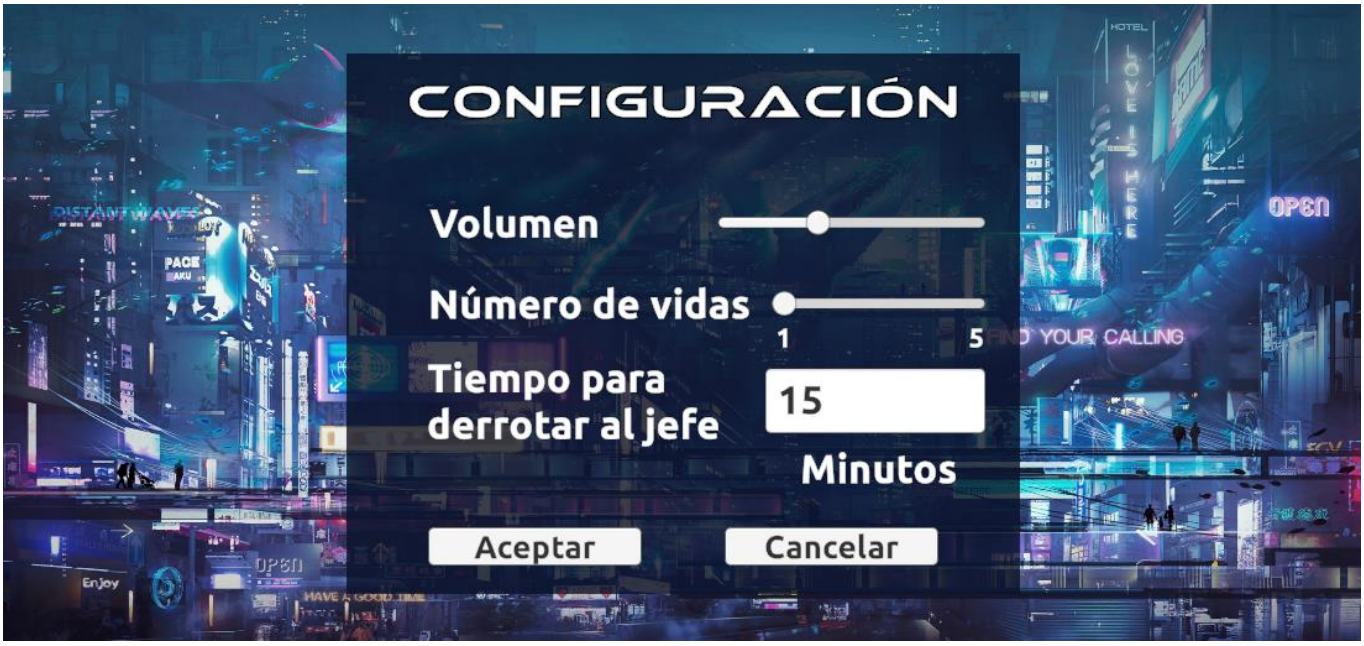
INTRODUCCIÓN

Steel Justice es un videojuego en 2D basado en la jugabilidad de juegos retro como pueden ser Megaman, Metal Slug o Contra. El contexto en el que se desarrolla es aquel del de una sociedad futurista con tintes cyberpunk, en el que encarnamos a un robot de combate llamado Legión que ha de luchar contra los criminales de la ciudad.

PANTALLA DE TÍTULO

La pantalla de título muestra el nombre del juego junto con dos botones, “COMENZAR”, que cargará el primer nivel del juego y “CONFIGURACIÓN” que abrirá una pequeña ventana de configuración donde se podrán modificar varios parámetros del juego.

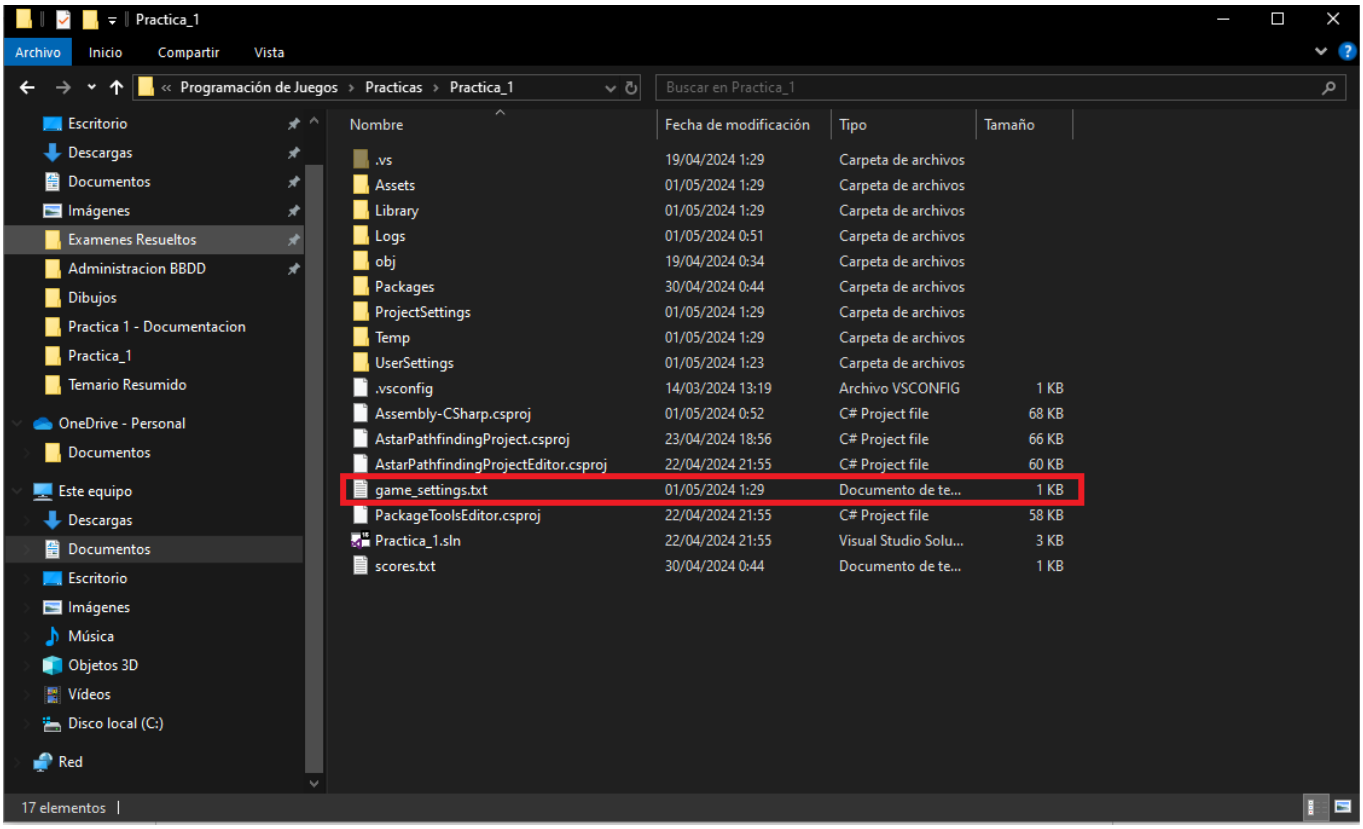
En el menú de configuración se encuentran tres parámetros configurables, el número de vidas, el volumen del juego y el tiempo para derrotar al jefe que se encuentra en el nivel 2. Por defecto los valores son de 0.5 para el volumen, una sola vida, y 15 minutos como tiempo límite.

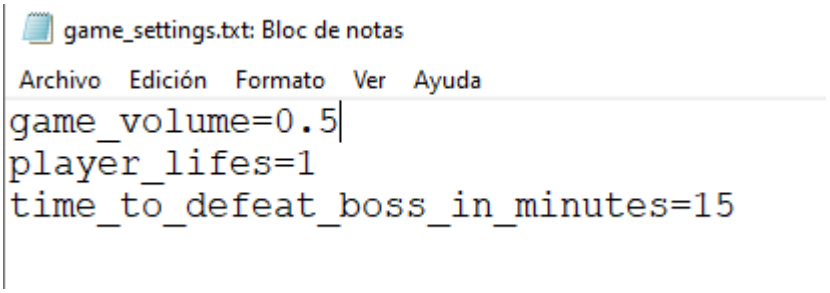


Esta configuración se almacena en un fichero de texto, que es cargado de forma automática en memoria mediante la clase GameSettings, a la que se accederá a lo largo de todas las escenas para controlar los parámetros

correspondientes al tiempo de juego, número de vidas del jugador y el volumen tanto de sonidos como de la música de fondo.

El fichero de texto que almacena dicha configuración se encuentra en la carpeta raíz del proyecto de Unity, cuya ruta es calculada de forma relativa a través del directorio de ejecución.






```
private void readGameSettingsFile()
{
    StreamReader sr = null;

    try
    {
        string currentPath = Path.GetDirectoryName(Assembly.GetExecutingAssembly().Location);

        Debug.Log("Current path: " + currentPath);

        string projectFolder = Directory.GetParent(Directory.GetParent(currentPath).FullName).FullName;

        Debug.Log("Project folder: " + projectFolder);

        sr = new StreamReader(projectFolder + "\\game_settings.txt");
```

Toda la interfaz tanto de la pantalla de título como de la ventana de configuración se adaptan a las dimensiones de la pantalla utilizando la herramienta de “Anchor” provista por Unity para la construcción de interfaces gráficas.

PRIMER NIVEL

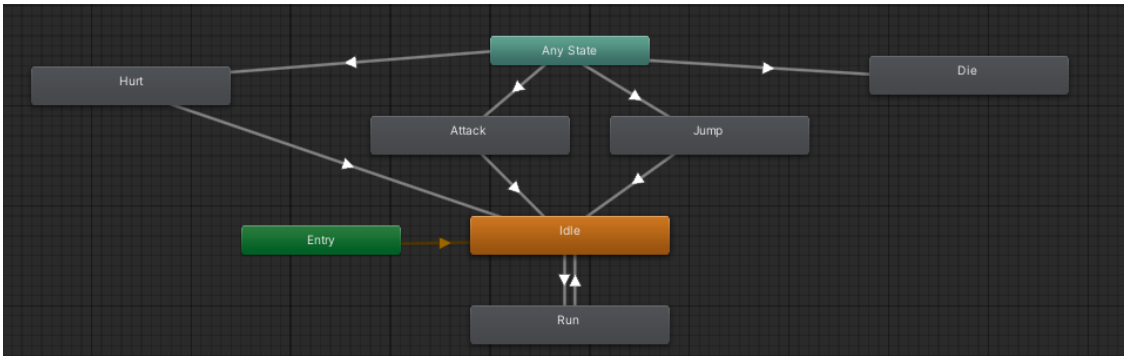
En este primer nivel nos encontramos en un laboratorio secreto subterráneo donde deberemos matar a todos los enemigos presentes.

Primero nos centraremos en el diseño del personaje principal y es que nuestro robot Legión, es representado dentro de Unity con el objeto Jugador, el cual usa para poder moverse, el script MovimientoJugador, que cuando pulsamos las teclas A o D, se moverá a izquierda o derecha respectivamente, así como cuando pulsemos la barra espaciadora, se producirá la acción de salto (y se comprobará cuando el personaje vuelve a tocar el suelo mediante el uso de un RayCast2D) y cuando pulsemos el click izquierdo del ratón comenzará a disparar bolas de plasma. Cabe añadir que, para dotar de física al personaje, se le han añadido los componentes de Unity BoxCollider2D y RigidBody2D y que también el script posee varias variables serializadas como el poder de salto la velocidad y la capa en la que se muestra al personaje, que podrán ser modificadas en caso de necesidad.

Si volvemos a la acción de disparo, esta es gestionada por el script AtaqueJugador, el cual tiene varias variables serializadas como el tiempo que debe de pasar entre cada disparo, el punto desde el cual se produce el disparo, un array de objetos de tipo Proyectil, que veremos más adelante y, por último, el sonido de disparo. Este script para la creación de los disparos hace uso de la técnica denominada Object Pooling, la cual consiste en crear un array de objetos para nuestros disparos, en este caso del tipo Proyectil, a continuación, se crea un punto desde el cuál se va a disparar, que ha de coincidir con la animación de disparo de nuestro personaje, y vamos recorriendo dicho array para en cada ejecución “devolver” un proyectil que se encuentre disponible dentro de los tiempos de enfriamiento entre cada acción de disparo. Con esta técnica nos evitamos tener que estar creando y destruyendo instancias del objeto proyectil cada vez que el jugador quiera disparar y, por tanto, disminuimos significativamente el uso de recursos del juego y aumentamos su eficiencia al ejecutarse.

El script Proyectil, se encarga de definir el comportamiento de cada disparo cuando colisiona con otro objeto gracias al método, OnCollisionEnter2D que se llama cuando el proyectil entra en contacto con otro Collider2D. Establece la variable golpeado en true para evitar que el proyectil cause más daño, desactiva el collider del proyectil y reproduce una animación de explosión a través del Animator. Si el objeto con el que colisiona tiene la etiqueta "Enemy", reduce su salud llamando al método TakeDamage() del componente Health del objeto colisionado. También dentro del método update, si el proyectil ha golpeado algo, simplemente retorna y no hace nada más. Si no ha golpeado nada, mueve el proyectil en la dirección establecida multiplicando la velocidad por el tiempo transcurrido desde el último fotograma. Por último, lleva un seguimiento del tiempo de vida del proyectil y si supera cierto límite (5 segundos en este caso), desactiva el GameObject del proyectil.

Por otra parte, para la gestión de las animaciones del personaje, se ha decidido utilizar una máquina de estados finitos de la siguiente forma:



Observamos entonces varios estados:

- **Entry**: el estado por donde comenzara la ejecución de máquina.
- **Idle**: estado inactivo, donde nuestro personaje se mantendrá a la espera de órdenes por parte del jugador.

- **Run**: estado de movimiento al cual pasaremos siempre y cuando el jugador pulse la tecla A o D y nos mantendremos en él, mientras sigamos pulsando dichas teclas, gestionado en la máquina mediante un parámetro booleano llamado run.
- **Attack**: estado que representa la acción de disparo, al cuál podremos acceder desde cualquier estado, siempre y cuando pulsemos el click izquierdo, gestionado en la máquina mediante un trigger llamado attack.
- **Jump**: estado que representa la acción de salto, al cual podremos acceder desde cualquier estado, siempre y cuando pulsemos la barra espaciadora, gestionado en la máquina mediante un parámetro trigger llamado jump y otro parámetro booleano llamado grounded para controlar cuando el personaje se encuentra en el suelo.
- **Hurt**: estado que representa el daño sobre el personaje, se pasará a el mediante un trigger llamado hurt, que se activará cuando un enemigo dañe al jugador.
- **Die**: estado que representa la muerte del personaje, se pasará a el siempre y cuando el jugador pase a tener 0 vidas, situación controlada mediante un trigger llamado die.

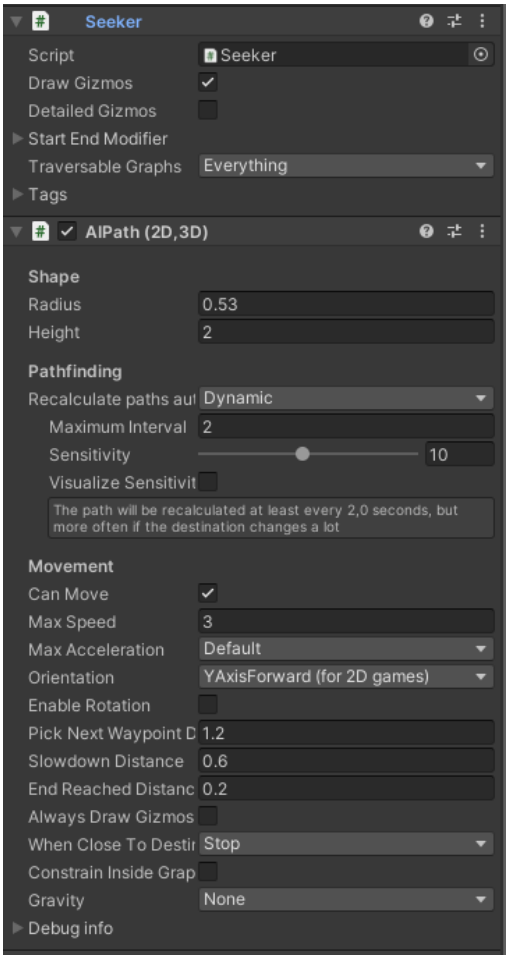
En cuanto al sistema de vida del jugador, se gestiona mediante el script Health, el cual maneja la salud de un objeto en el juego, así como la lógica asociada con los daños recibidos y la invulnerabilidad temporal después de ser golpeado:

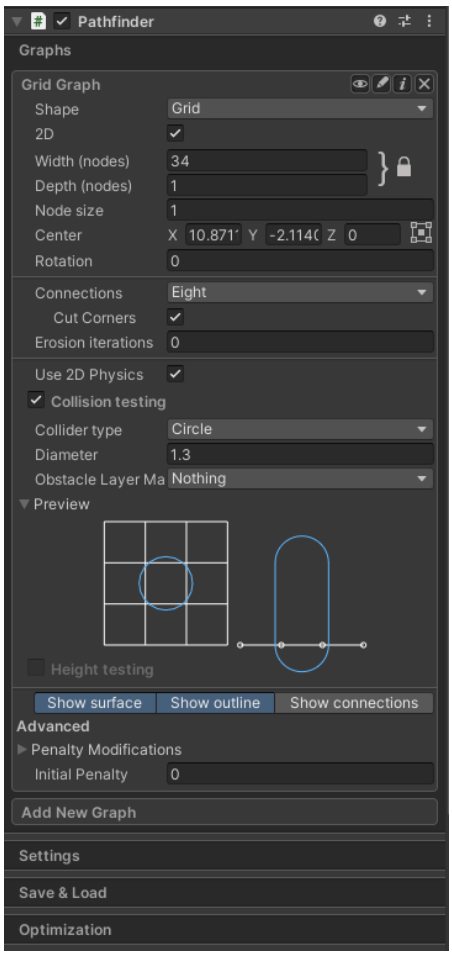
- **Variables SerializeField**: Define varias variables que se pueden ajustar desde el editor de Unity. Estas incluyen la salud inicial (startingHealth), la duración de los frames de invulnerabilidad (iFramesDuration), el número de destellos durante la invulnerabilidad (numberOffFlashes), un array de componentes (components) que se desactivarán al morir y sonidos asociados con la muerte y el daño.
- **Variables Privadas**: Almacena la salud actual del objeto, una referencia al componente Animator y un indicador booleano para saber si el objeto está muerto.
- **Awake()**: Se ejecuta al inicio para inicializar la salud actual y obtener referencias al Animator y al componente SpriteRenderer del objeto.
- **TakeDamage(float _damage)**: Esta función se llama cuando el objeto recibe daño. Reduce la salud actual en la cantidad de daño especificada (_damage). Si la salud actual es mayor que cero, activa una animación de daño en el Animator, inicia una corutina para hacer al objeto invulnerable temporalmente, y reproduce un sonido de daño. Si la salud cae a cero o menos, activa una animación de muerte en el Animator, desactiva los componentes especificados en el array components, marca al objeto como muerto, reproduce un sonido de muerte y aumenta la puntuación del jugador.
- **addHealth(float _value)**: Esta función aumenta la salud actual del objeto en la cantidad especificada (_value), asegurándose de que no exceda la salud inicial.
- **Invulnerability()**: Una corutina que hace al objeto invulnerable durante un período de tiempo específico (iFramesDuration). Durante este tiempo, el objeto parpadea alternando entre el color rojo y su color original. También ignora las colisiones con las capas 10 y 11 durante este tiempo.

Dicho script es usado también por los enemigos para recibir daño y representar su vida.

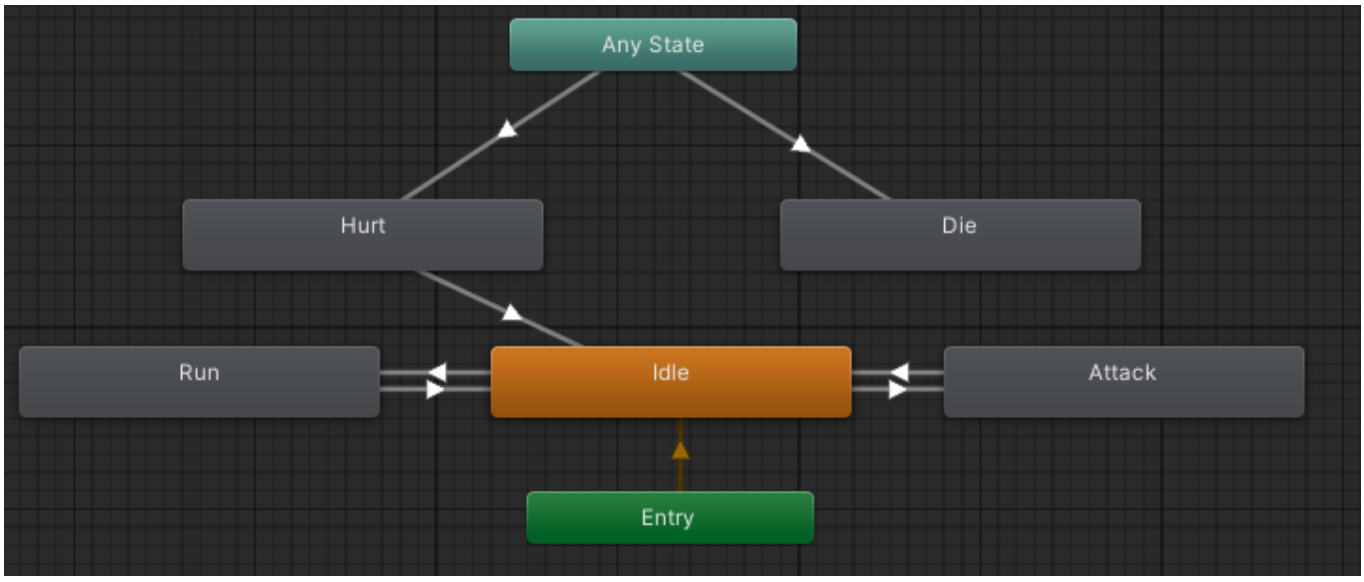
Para la representación de las vidas del jugador se usa el objeto UICanvas y sus objetos HealthBar y HealthBarCurrent que dibujan en la pantalla arriba a la izquierda los corazones que le quedan al jugador. Por otra parte, cabe añadir que el objeto main camera del nivel posee un script llamado ControlCamara, el cual se encarga de seguir al jugador allá por donde vaya.

En cuanto a la estructura del nivel, tenemos dos habitaciones, en la primera se nos presentan trampas que deberemos de sortear y el primer enemigo a batir, llamado espectro, el cual tiene aplicado un algoritmo A* para el seguimiento de caminos, que en este caso su objetivo es perseguir al jugador hasta acabar con él. Dicho algoritmo se ha implementado con la ayuda de la librería astarpathfindingproyect, la cual nos ofrece la posibilidad de dar esta cualidad a nuestros objetos del juego de manera sencilla, simplemente añadiendo varios scripts al mismo y configurando el comportamiento, alcance y velocidad de seguimiento.





También para la gestión de las animaciones se ha usado una máquina de estados finitos de la siguiente forma:

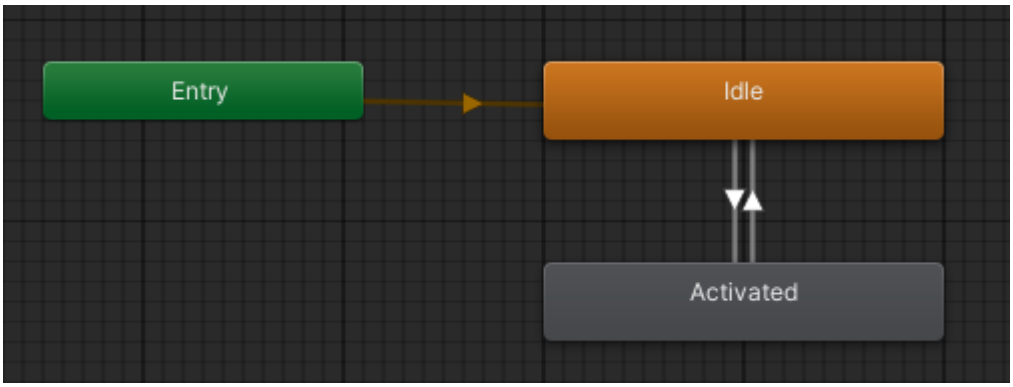


Observamos entonces varios estados:

- **Entry**: el estado por donde comenzara la ejecución de máquina.
- **Idle**: estado inactivo, donde nuestro enemigo se mantendrá a la espera de que el jugador entre en su zona habilitada para seguirlo.
- **Run**: estado de movimiento al cual pasaremos siempre que detectemos al jugador y vaya a perseguirlo, gestionado en la máquina mediante un parámetro booleano llamado run.
- **Attack**: estado que representa la acción de disparo, al cual podremos acceder desde cualquier estado, siempre y cuando detectemos al jugador en nuestro alcance, gestionado en la máquina mediante un trigger llamado attack, saldremos de él cuándo no estemos dentro de nuestro alcance.
- **Hurt**: estado que representa el daño sobre el enemigo, se pasará a él mediante un trigger llamado hurt, que se activará cuando el jugador dañe al enemigo.

- **Die**: estado que representa la muerte del enemigo, se pasará a él siempre y cuando pase a tener 0 vidas, situación controlada mediante un trigger llamado die.

En cuanto a las trampas todas del nivel funcionan de manera similar, gestionadas sus animaciones también mediante una máquina de estados de la siguiente forma:

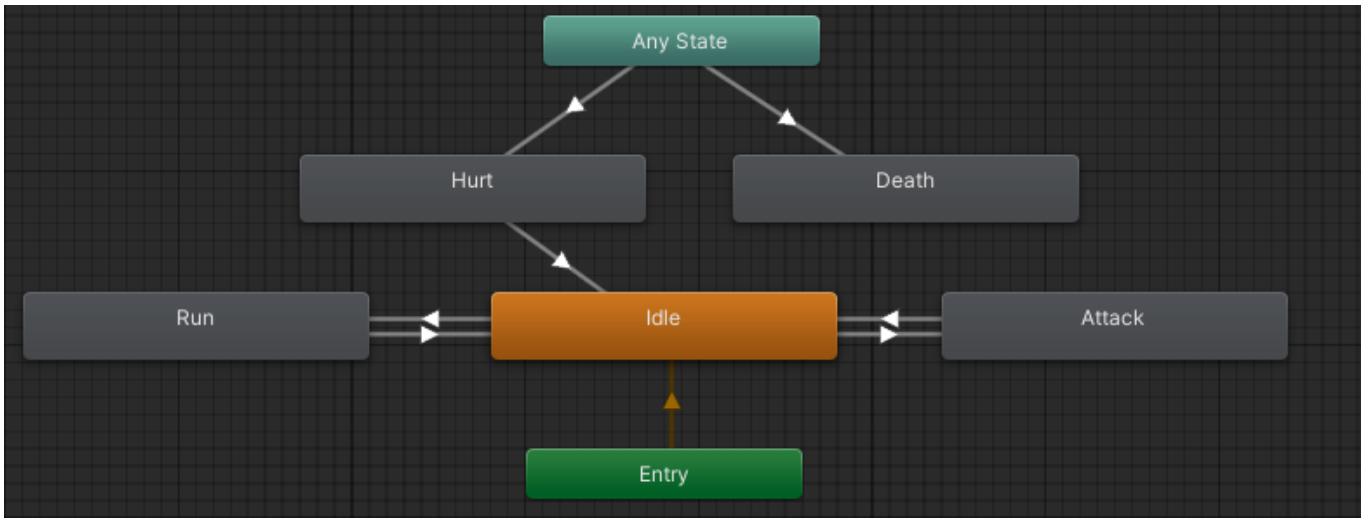


Observamos dos estados:

- **Idle**: estado inicial donde la trampa no estará activa y esperará a que el jugador entre dentro de su coño de acción.
- **Activated**: estado donde la trampa pasara a activarse y a hacer daño al jugador para después pasar de nuevo al estado Idle, cuando se acabe el tiempo de activación.

También cada una posee un script propio para su comportamiento ElectricTower, para la torreta eléctrica, LaserTower, para la torreta laser y enemy_sideaways para la sierra. Los dos primeros usan una corutina que se activara dentro del método OnTriggerEnter2D cuando colisione con el jugador para establecer su Sprite a rojo e indicar que se ha activado para después de un tiempo de activación, si el jugador se encuentra dentro de su cono de acción, proceder a dañar al jugador. La sierra en cambio se mueve libremente de izquierda a derecha dentro de unos límites izquierdo y derecho respectivamente y cuando colisione con el jugador procederá a hacerle daño.

En cuanto a la segunda habitación, encontramos al segundo enemigo del juego, llamado bringer of death, el cual se encuentra patrullando esta segunda sala entre dos puntos fijos y en cuanto el jugador entre en su cono de acción, controlado mediante un raycast2D, comenzará a atacarle. Sus animaciones también han sido gestionadas mediante una máquina de estados de la siguiente forma:



Que como vemos es muy similar a la del primer enemigo, con lo cual no procedemos a explicarla de nuevo.

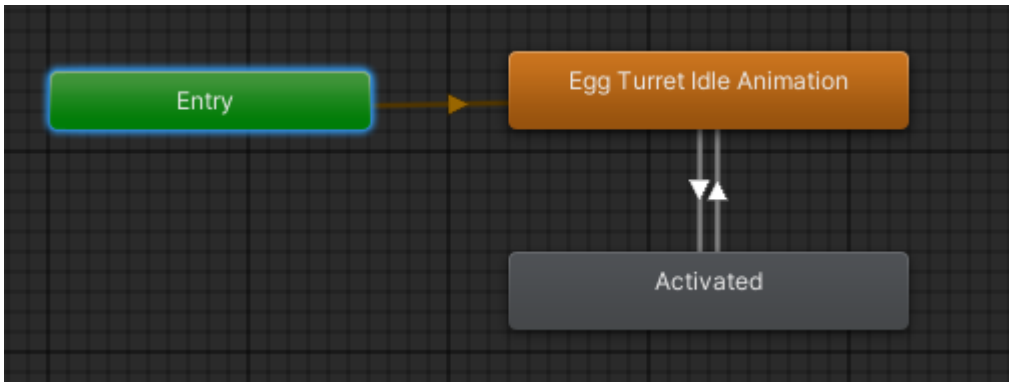
También en este nivel tenemos el objeto coleccionable de vida, el cuál nos proporcionará un corazón que recuperar en caso de que nos falte vida. Su comportamiento esta definido en el script HealthCollectable.

Para finalizar en la parte final de este nivel, nos encontramos con el objeto puerta, el cual mediante el script CambioEscena pasa de escena al nivel 2, cuando el jugador colisione con él.

SEGUNDO NIVEL

En este segundo nivel, pasamos al exterior, y de fondo veremos la ciudad que juramos defender.

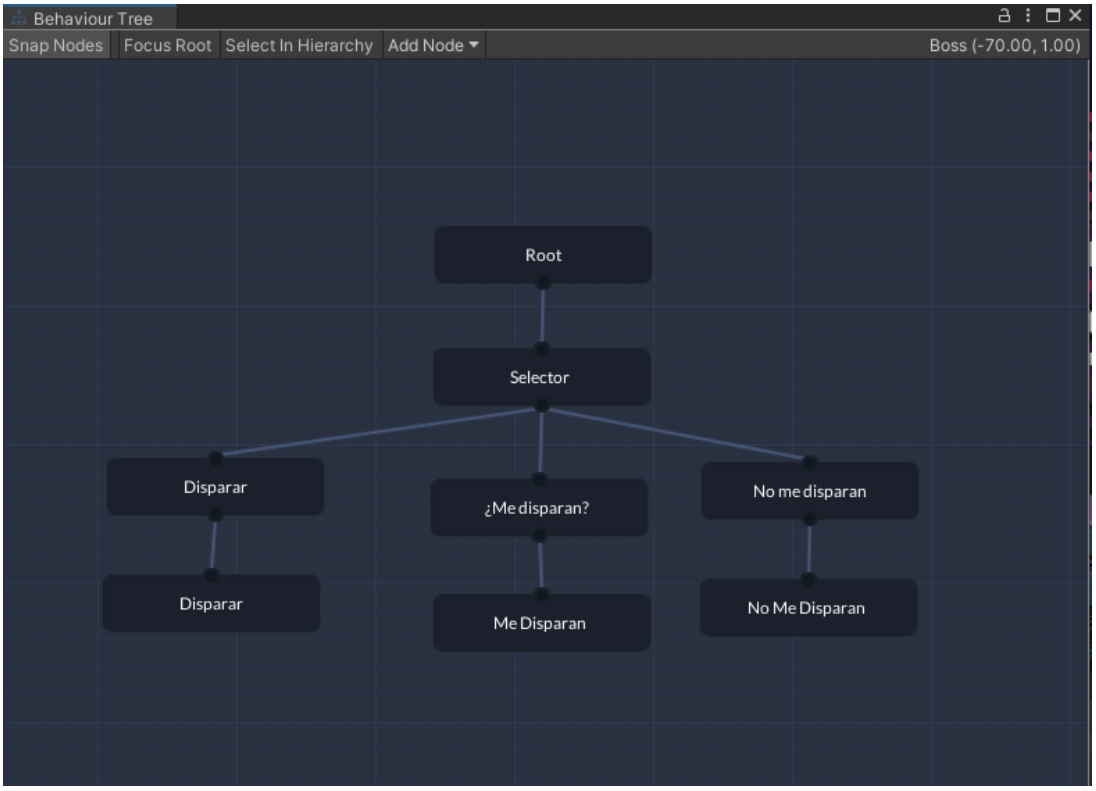
Al principio del nivel, nos toparemos con la primera trampa del mismo, la Egg-Turret, la cual, posee un comportamiento especial, ya que en cuanto el jugador entre en su cono de acción, esta se activará y comenzará a disparar proyectiles. Este comportamiento hace uso de varias técnicas explicadas anteriormente, como object pooling para los proyectiles, iFrames para la activación de la torreta y poner su Sprite en rojo una vez activada, uso de RayCast2D, para detectar al jugador y uso de una máquina de estados para gestionar sus animaciones de la siguiente forma:



Donde vemos dos estados:

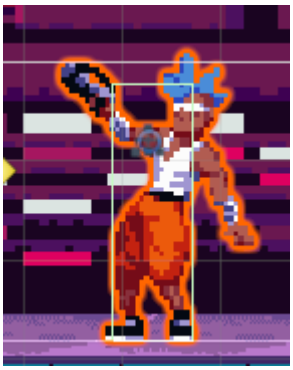
- **Egg turret idle animation**: que representa el estado el cual la torreta está inactiva, esperando a que el enemigo entre en su cono de acción.
- **Activated**: estado el cual representa que la torreta ha detectado al jugador y comenzara a dispararle. Está gestionado mediante un trigger llamado “activated”.

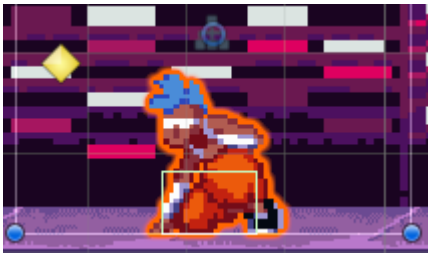
Por último, el jefe final funciona mediante un árbol de comportamiento. Se ha utilizado una librería abierta que aporta las utilidades y facilidades necesarias para que la implementación de esta técnica sea lo mas óptima posible.



El árbol consta de una raíz, la cual pasa por un selector que comprueba los diferentes casos de izquierda a derecha, cuando recibe una respuesta positiva de cualquiera de las ramas, no comprueba las siguientes. En el caso del primer nodo “Disparar” cuando el jefe final observa al jugador dentro de un rango, comienza a disparar, pero no comprueba si le disparan o no, dichas ramas únicamente se evalúan cuando el jefe no está disparando.

Cuando el jefe detecta disparos, se agacha, reduciendo su box collider, esquivando así las balas del jugador.



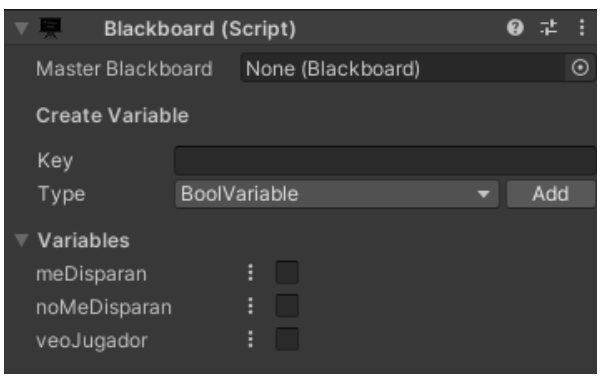


Todas las ramas están sujetas a una probabilidad, no siempre el jefe dispara, ni comprueba si le están disparando, lo hace tan solo un porcentaje de las ocasiones para dar oportunidad al jugador de poder derrotarle.

Estas acciones se realizan en los nodos más profundos del árbol (2º nodo “Disparar”, “Me Disparan”, “No Me Disparan”) que heredan de la clase “Leaf” de la librería utilizada. Dichos nodos ejecutan un simple script, en el caso de “Disparar” simplemente se acciona la animación de ataque del jefe:

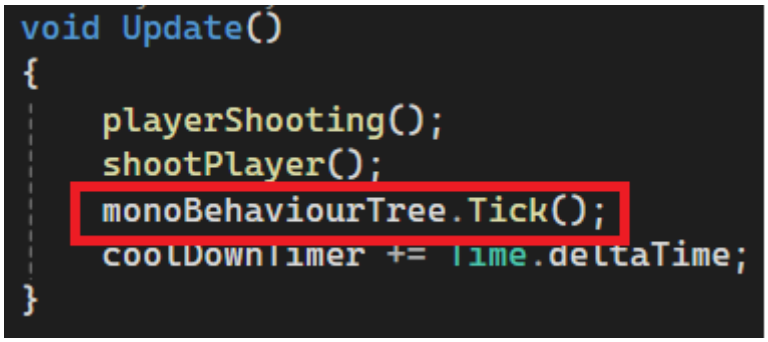
```
[AddComponentMenu("")]
[MBTNode(name = "Examples/Disparar")]
Script de Unity (1 referencia de recurso) | 0 referencias
public class Disparar : Leaf
{
    [SerializeField] private Animator animator;

    0 referencias
    public override NodeResult Execute()
    {
        animator.SetTrigger("attack");
        return NodeResult.success;
    }
}
```

Para comprobar cuando se debe entrar a una de las ramas del árbol, se utiliza otro componente provisto por la librería, llamado Blackboard, una simple utilidad para compartir variables entre todos los nodos del árbol. En esta Blackboard se han creado tres variables de tipo booleano, meDisparan, noMeDisparan y veoJugador, inicialmente todas a falso.



Los primeros nodos del árbol son del tipo “Is Set Condition” que comprueban alguna de las variables dentro de la Blackboard, si una de ellas es verdadera, entra por la rama, ignorando el resto, ya que el nodo padre de la primera fila de nodos es un selector como se explicó anteriormente.

El árbol se recorre desde su raíz hasta los niveles más profundos cada vez que se realiza un Update dentro del script BossControl asignado al jefe.



PANTALLA DE PUNTUACIÓN

En esta pantalla, se mostrarán las 10 primeras puntuaciones de los jugadores basadas en el tiempo que han tardado en completar el juego y la puntuación conseguida por matar enemigos. Si el jugador es capaz de conseguir una puntuación mayor que alguna de ellas, podrá introducir su nombre y aparecerá en dicha tabla.

Su implementación se basa en el uso de objetos UI de Unity, text objects e images objects.

RECURSOS EXTERNOS

- **Imagen de fondo en pantalla de título y puntuación:** https://wall.alphacoders.com/big.php?i=1283852
- **Asset para configuración gráfica:** https://docs.unity3d.com/Packages/com.unity.render-pipelines.universal@17.0/manual/Setup.html
- **Fuente de texto para los títulos:** https://www.1001fonts.com/ethnocentric-font.html
- **Fuente de texto para el resto de elementos:** https://www.1001fonts.com/ubuntu-font.html
- **Sprites Nivel 1:** https://foozlecc.itch.io/sci-fi-lab-tileset-decor-traps/download/eyJleHBpcmVzIjoxNzEyODM2MDU2LCJpZCI6OTE1MzA5fQ%3D%3D.FB1dXAzmMkvJm1hZsBClPMp7v8M%3D
- **Animaciones nivel 1:** https://creativekind.itch.io/nightborne-warrior, https://clembod.itch.io/bringer-of-death-free, https://wuhuli.itch.io/robot-sprite-brawlbot
- **Música nivel 1:** https://pixabay.com/es/music/construir-escenas-dark-sci-fi-cyberpunk-112399/
- **Tema principal:** https://pixabay.com/es/music/optimista-cyborg-in-me-background-music-for-video-blog-promo-stories-188531/
- **Sprites, música y animaciones Nivel 2:** https://assetstore.unity.com/packages/2d/environments/warped-city-2-200208
- **Efectos de sonido nivel 1 y 2:** https://pixabay.com/es/sound-effects/, https://www.mediafire.com/file/wszdta62jwdgyru/Audio.zip/file
- **Librería A*:** https://arongranberg.com/astar/
- **Librería árboles de comportamiento:** https://github.com/Qriva/MonoBehaviourTree
- **Pantalla de Game Over (Exclusiva para este proyecto por Álvaro Javier Rodríguez Naranjo):** https://www.linkedin.com/in/%C3%A1lvaro-javier-rodriguez-naranjo-a68bb513b/?originalSubdomain=es