Álvaro Benítez

ÍNDICE

*IMPORTANTE PROBAR EL JUEGO CON EL ASPECTO DE PANTALLA EN 1920*1080 Y NO EN FREE ASPECT*

1. Historia

El reino digital de DAM ha experimentado un colapso en su estructura de datos, y Skippy, un hábil programador, es el héroe convocado para liberar al reino.

La corrupción en forma de código basura y errores se ha extendido por todo el sistema, afectando a los habitantes digitales de DAM.

El objetivo de Skippy es recoger trozos de código XML esenciales que han sido dispersados por diferentes rincones.

A medida que avanza, se encontrará muchos enemigos en forma de bugs maliciosos que intentan obstaculizar su misión y corromper aún más el código.

El juego termina con todos los fragmentos de código recolectados y la puerta del reino abierta para permitir la salida.

2. Clases implementadas

CamaraScript

- Este script se encuentra asociado al GameObject de MainCamera y define el comportamiento de la cámara con respecto al personaje jugable.

- Cuenta con las siguientes variables:

  -Pública de tipo Transform (**personaje**) que será utilizada para vincular mediante el inspector a nuestro protagonista.

- Código:

  -En el método *Update()*, ejecutado cada frame irá actualizando la posición de la cámara. Para ello utiliza la posición del personaje y resta un vector con un valor de (0,0,1) para que la cámara vaya una unidad atrasada con respecto al sprite del caballero.

CampanaManager

- Asociamos este script a cada uno de nuestros checkpoints.

  El checkpoint está formado por un poste y una campana, posee animación y sonido para cuando el personaje contacta.

- El uso dado es ir actualizando las coordenadas en las que reaparecerá el personaje si sufre alguna caída durante su recorrido por el nivel.

- Cuenta con las siguientes variables:

-Pública de tipo Animator para hacer referencia a la animación (**anim**).

-Dos variables públicas y estáticas para almacenar las coordenadas x e y que serán accesibles desde fuera del script (**coordenadaX**,**coordenadaY**).

-Pública de tipo AudioClip para almacenar el efecto de sonido que sonará (**audioClip**).

-Privada de tipo AudioSource que servirá como referencia al componente del objeto (**audioSource**).

- Código:

  - En el método *Start()* guardaremos dentro de la variable audioSource el componente AudioSource.

  -Utilizará el método *OnTriggerEnter2D()* para detectar las diferentes colisiones, dentro compara si la etiqueta de la colisión es o no 'Player'.

  En caso afirmativo activa la animación de movimiento de la campana, ejecuta la corrutina *EsperarSegundos()*, y hace que suene el AudioClip asociado. Por último guarda las coordenadas en las que se encuentra el checkpoint actualizando las variables **coordenadaX** y **coordenadaY**.

  -El método *rellenarVida()* activa todas las imágenes en las listas 'vidas' y 'contenedores', desactiva las imágenes de los contenedores y vidas que representan vidas adicionales más allá de las vidas iniciales y restablece la cantidad de vidas actuales a la cantidad inicial.

  -En la corrutina *EsperarSegundos()* esperará unos segundos antes de desactivar la animación de la campana.

CrearBola

- Encontraremos este script asociado a nuestro Personaje.

- Lo usaremos para crear las instancias de los disparos que ejecuta el personaje, estos disparos los ganará al recoger monedas por el mapa.

El HUD cuenta con la cantidad de disparos disponibles.

- Cuenta con las siguientes variables:

  -Pública de tipo Transform para asociar desde el inspector el puntero que se encuentra dentro del GameObject del personaje (**bola**).

  -Pública de tipo GameObject para asociar el objeto bolaBuena (**lanzamientoBola**) .

  -Pública de tipo entero para llevar el control de la munición disponible (**municion**).

  -Pública de tipo Text que hace referencia al texto mostrado en el HUD (**balas**).

  -Pública de tipo AudioClip para almacenar el efecto de sonido que sonará (**audioClip**).

  -Privada de tipo AudioSource que servirá como referencia al componente del objeto (**audioSource**).

- Código:

  -En el método *Start()* guardaremos dentro de la variable audioSource el componente AudioSource. Además inicializamos la munición en 0.

  -En el método *Update()*, se ejecutará cada frame el método *dispararBola()*.

  -*dispararBola()* actualiza las balas disponibles en **municion**, y comprueba que en el caso de que el usuario pulse la tecla asignada al disparo y cuente con munición disponible se lanzará el proyectil.
  Además descontará una unidad a la munición y reproducirá el efecto de sonido asociado.

LanzarBola

- Encontraremos este script asociado al objeto bolaBuena

- Lo usaremos para definir el comportamiento que tendrá el proyectil.

- Cuenta con las siguientes variables:

-Privada de tipo RigidBody2D para hacer referencia al RigidBody del objeto (**bolaCirculo**).

-Privada de tipo float para controlar la velocidad a la que desplaza el proyectil con valor inicial de 7f (**velocidad**).

-Privada de tipo float para controlar el tiempo de vida que tendrá el proyectil con valor inicial de 2f (**tiempoActiva**).

-Pública de tipo Transform para asociar por el inspector el transform del Personaje (**transPersonaje**).

-Pública de tipo GameObject para guardar la referencia del caballero (**personaje**).

-Privada de tipo float para guardar el localScale.x (**localScaleX**).

-Privada de tipo float para guardar el localScale.y (**localScaleY**).

-Pública de tipo Animator para hacer referencia a la animación (**anim**).

-Pública de tipo AudioClip para almacenar el efecto de sonido que sonará (**audioClip**).

-Privada de tipo AudioSource que servirá como referencia al componente del objeto (**audioSource**).

- Código:

-En el método *Start()* guardaremos **localScaleX** y **localScaleY.** Además obtenemos y guardamos los componentes de RigidBody, el objeto del personaje y su transformada.

Por último ejecuta el método *dispararBola()*, que se encargará de instanciar el proyectil tras comprobar hacia donde mira el personaje.
Le dará velocidad y la dirección correspondiente.

-En el método *Update()*, se ejecutará cada frame el método *Destroy()* que gracias a **tiempoActiva**, podremos destruir el objeto tras el paso del tiempo.

-Utilizará el método *OnCollisionrEnter2D()* para detectar si el proyectil colisionó con un elemento que tenga el layer de ‘Enemigo’, si es el caso,

sacará el componente de su animación de muerte para reproducirla tras ejecutar el efecto especial de muerte.
Tras todo esto se destruirá la bola.

MenuInicio

- Encontraremos este script asociado al Canvas de la escena Inicio.
- Lo usaremos para configurar la partida y gestionar los diferentes menús antes de empezar la partida.
- Cuenta con las siguientes variables:

  -Pública de tipo GameObject para asociar por el inspector el GameObject de Menu (**menu**).

  -Pública de tipo GameObject para asociar por el inspector el GameObject de Settings (**settings**).

  -Pública de tipo GameObject para asociar por el inspector el GameObject de ElegirDificultad (**elegirDificultad**).

  -Pública de tipo enum para almacenar las diferentes dificultades (**Dificultad**).

  -Pública estática de tipo Dificultad que almacenará la dificultad, inicialmente Media (**dificultadActual**).

- Código:

  -En el método *Start()* ejecutaremos el método *mostrarMenuInicio()* que hace visible el menú principal, y oculta los otros submenús.

  -Con el método mostrarSettings() ocultamos todos los submenús excepto el de la configuración. Se llamará desde un botón.

-Con el método mostrarElegirDificultad() ocultamos todos los submenús excepto el de elegir la dificultad. Se llamará desde un botón.

-Con el método salir() ejecutaremos el método *Application.Quit()* para cerrar el juego.
Se llamará desde un botón.

-Métodos *seleccionarDificultadX*: Establecemos la dificultad con el método *establecerDificultad()* pasándole como parámetro la dificultad del enum, y ajustando la cantidad de segundos en el script tiempo.
Por último carga la escena del nivel.

MenuPausa

- Encontraremos este script asociado al Canvas de la escena PrimerNivel.

- Lo usaremos para poder pausar el juego, y mostrar un menú de pausa que nos permitirá salir al menú, salir del juego y cambiar el volumen o mutear el sonido.

- Cuenta con las siguientes variables:

  -Pública de tipo GameObject para asociar por el inspector el GameObject de MenuPausa (**menuPausa**).

  -Privada de tipo booleana para controlar si se está mostrando el menú de pausa o no.

- Código:

  -En el método *Start()* haremos que el menú de pausa empiece desactivado.

  -En el método *Update()*, comprobará en cada frame si el usuario está pulsando la tecla Escape, si es el caso y el menú está ocultado lo mostrará (usando el método *Reanudar()*); de lo contrario lo ocultará (usando el método *Pausar()*).

  -Con el método *Pausar()* pondremos visible el menú, pararemos el tiempo y activaremos la booleana de **esPausado.**

-Con el método *Reanudar()* pondremos oculto el menú, reanudamos el tiempo y desactivaremos la booleana de **esPausado.**
Se llamará desde un botón.

-Con el método *Menu()* cargaremos la escena de Inicio, reanudamos el tiempo y desactivaremos la booleana de **esPausado.**
Además reiniciamos los pergaminos, las monedas y las vidas
Se llamará desde un botón.

-Con el método *Salir()* ejecutaremos el método *Application.Quit()* para cerrar el juego, reanudamos el tiempo y desactivaremos la booleana.
Se llamará desde un botón.

MoverPersonaje

- Encontraremos este script asociado al Personaje.

- Lo usaremos para poder pausar el juego, y mostrar un menú de pausa que nos permitirá salir al menú, salir del juego y cambiar el volumen o mutear el sonido.

- Cuenta con las siguientes variables:

  -Privada de tipo Rigibody2D para guardar una referencia al Rigibody en Personaje (**rb2D**).

  -Privada de tipo float para almacenar la velocidad del jugador en Personaje (**velocidad**).

  -Privada de tipo float para almacenar la escala en el eje X del jugador en Personaje (**localScaleX**).

  -Privada de tipo float para almacenar la escala en el eje Y del jugador en Personaje (**localScaleY**).

  -Privada de tipo float para almacenar la fuerza implicada al salto del jugador cuando este se realiza desde el suelo en Personaje (**fuerzaSalto**).

  -Privada de tipo float para almacenar la fuerza que se le aplicará al jugador en dirección contraria hacia donde mira cuando este salte mientras se desliza por una pared en Personaje (**fuerzaSaltoParedX**).

-Privada de tipo float para almacenar la fuerza que se le aplicará al jugador hacia arriba cuando este salte mientras se desliza por una pared en Personaje (**fuerzaSaltoParedY**).

-Privada de tipo float para almacenar el tiempo durante el cual el jugador no podrá cambiar la dirección del personaje cuando salte mientras se desliza por una pared en Personaje (**tiempoSaltoPared**).

-Privada de tipo booleana para controlar cuando salta mientras se desliza en Personaje (**saltandoDePared**).

-Privada de tipo Vector2 para almacenar la velocidad a la que saldrá rebotado el jugador al ser dañado en Personaje (**velocidadRetroceso**).

-Privada de tipo float para almacenar la velocidad a la que saldrá rebotado hacia arriba el jugador al eliminar a un enemigo en Personaje (**velocidadRebote**).

-Privada de tipo float para almacenar la coordenada que representa el límite del mapa por la parte izquierda en Personaje (**xMinima**).

-Pública de tipo Transform para hacer referencia al GameObject gracias al cual el personaje detecta que está en el suelo en Personaje (**pie**).

-Pública de tipo Transform para hacer referencia al GameObject gracias al cual el personaje detecta que está en una pared en Personaje (**costado**).

-Pública de tipo LayerMask para guardar la máscara donde detectaremos que nuestro personaje está en el suelo en Personaje (**suelo**).

-Pública de tipo LayerMask para guardar la máscara donde detectaremos que nuestro personaje está en la pared en Personaje (**pared**).

-Pública de tipo Vector3 para modificar el tamaño del GameObject “costado” en Personaje (**anchoPersonaje**).

-Pública de tipo float para modificar el tamaño del GameObject “pie” en Personaje (**pie**).

-Pública de tipo int para asignar el número de saltos que puede realizar el jugador cuando no toca el suelo en Personaje (**numeroSaltos**).

-Pública de tipo booleana para bloquear el movimiento del jugador cuando salta mientras se desliza en Personaje (**pierdeControl**).

-Pública de tipo booleana para controlar cuando el jugador está deslizándose Personaje (**deslizando**).

-Privada de tipo float para almacenar la velocidad al que el jugador se desliza por una pared en Personaje (**velocidadDeslizar**).

-Privada de tipo Animator que hace referencia al Animator del jugador en Personaje (**anim**).

-Pública de tipo ParticleSystem que hace referencia al GameObject que contiene el sistema de partículas del jugador en Personaje (**partículas**).

- Código:

-En el método *Start()* estableceremos las coordenadas de reaparición al comenzar el juego, da valor a las fuerzas de salto mientras se desliza. Inicializa las máscaras, el GameObject “pie”, la escala inicial del personaje, el animator y asigna a “true” el método “freezeRotation” del RigidBody del personaje.

-En el método *Update()*,comprobará en cada frame si la booleana de bloquear el control del personaje está desactivada, en ese caso, llamará a todos los métodos que permiten al jugador moverse. Además, comprueba que si el jugador toca el suelo o una pared, este obtiene un salto adicional en el aire.

-Con el método *listenerAvance()* controlamos el desplazamiento lateral del personaje**.** Guarda la velocidad y del RigidBody en **miVelocidadY**.

Según el sentido del personaje se moverá a una dirección u otra, activando la animación de ‘andando’, además de establecer el localScale correcto.

Seguidamente comprueba que el personaje no supera los límites establecidos en el mapa, cambiando la velocidad a 0 si esto ocurre.

-En el método *listenerSalto()* comprueba que el usuario esté pulsando la tecla Espacio mientras tenga **numeroSaltos** y además la booleana **deslizando** esté en false. Provocará una fuerza con valor **fuerzaSalto** y restará un salto a **numeroSaltos.**

Si **deslizando** es true, termina la animación de deslizar y coloca la booleana **deslizando** a false. Nuevamente añadirá una fuerza hacia el sentido contrario de la pared con un valor **fuerzaSaltoParedY.**

Ejecutará para finalizar la corrutina *CambioSaltoPared()*.

-La corrutina *CambioSaltoPared()* activa la booleana **pierdeControl** bloqueando los controles del personaje durante el tiempo determinado en **tiempoSaltoPared**. Tras esto, vuelve a activar los controles del personaje.

-El método *listenerDeslizamiento()* controla el movimiento del personaje mientras se desplaza hacia abajo por la pared.
Comprobará que el personaje no está en el suelo, pero sí en la pared para activar la booleana **deslizando** y su correspondiente animación.
Además ralentiza la caída del personaje.

Cuando el personaje deja de mirar a una pared, termina la animación y cancela la booleana.

Mientras se encuentre deslizándose reproducirá las partículas y cambiará la escala del sprite según el sentido.

-El método *estaEnElSuelo()* devuelve true o false tras comprobar el método *Physics2D.OverlapCircle()* entre la posición del collider del pie, **radioPie** y el layer ‘suelo’.

-El método *estaEnPared()* devuelve true o false tras comprobar el método *Physics2D.OverlapBox()* entre la posición del collider del costado, **anchoPersonaje** y el layer ‘pared’.

- El método *OnDrawGizmos()* nos proporciona la opción de ver los contornos de un GameObject sin collider asignándole un color a la figura con el método *Gizmos.*color, el cuál requiere ser igualado a un color.
Y con el método *Gizmos.DrawWireCube*, que recibe la posición y el tamaño del GameObject, se dibuja el contorno.

-El método *rebote()* cambia la velocidad en la y de nuestro personaje al matar enemigos, el valor de este nuevo vector en la coordenada y depende de **velocidadRebote.**

-El método *retroceso()* recibe un Vector2 golpe correspondiente a una colisión, y aplicará una velocidad en nuestro personaje hacia el lado contrario del golpe. Este Vector2 depende de ***velocidadRetroceso***

FondoScript

- Este script está asociado al GameObject del fondo y controla el movimiento de este mismo con respecto a la posición del personaje en la escena.

- Se declara una variable pública de tipo Transform (**personaje**) que permite asignar el transform del personaje al que seguirá el fondo.

- Código:

    - El método *Update()* se llama en cada frame del juego. La lógica principal de este script se encuentra en este método. El propósito es ajustar la posición del objeto 'transform' basándose en la posición del personaje. Para ello utiliza la posición del personaje y resta un vector con un valor de (0,0,-1).

Tiempo

- Este script está asociado al GameObject tiempo y se encarga de gestionar el tiempo y controlar si ha llegado a cero.

- Cuenta con las siguientes variables:
    - Una variable pública de tipo Text donde le asignaremos un objeto de tipo 'Text' desde el Editor. Este objeto Text se usará para mostrar el tiempo en la interfaz de usuario.
    - Una variable estática y pública de tipo entera que representa la cantidad de tiempo inicial.
    - Una variable estática y privada de tipo float que mantiene el tiempo actual restante.

- Código:

    - En el método *Start()* llamado cuando se inicia el objeto se establece el tiempo actual **tiempo** en el valor inicial definido por la variable **TIEMPO_NORMAL**.

- En el método Update() que se llama en cada frame, se llama al método *cambiarTexto()* para actualizar el texto que se muestra y se verifica si el tiempo ha terminado llamando al método *comporobarTiempoTranscurrido()*, en caso afirmativo se carga la escena (‘NivelFinalizado’).

- El método *cambiarTexto()* se encarga de actualizar el tiempo restante. Resta el tiempo transcurrido desde el último frame (**‘**Time.deltaTime**’**) y actualiza el texto en el objeto Text asociado.

- El método *comprobarTiempoTerminado()* este método devuelve true si el tiempo restante es menor o igual a cero, indicando así que el tiempo ha terminado.

RecogerPergamino

- Este script está asociado al GameObject XMLs y está diseñado para gestionar la recogida de XML.

- Cuenta con las siguientes variables:

    - Una variable pública y estática de tipo entera **pergaminos** que almacena la cantidad total de pergaminos recogidos.

    - Una variable pública de tipo Text **cantidadPergaminos**. Se usará para mostrar la cantidad actual de XMLs en la interfaz de juego.

    - Una variable pública de tipo AudioClip llamada **audioClip** que establecerá el sonido que se reproduce cuando se recoge un xml.

    - Una variable privada de tipo AudioSource **audioSource** que determinará la fuente de audio que reproduce el sonido.

    - Una variable de tipo ParticleSystem **particula** que hace referencia a un sistema de partículas, que se activará al recoger un xml.

- Código:

    - En el método Start() se asigna la componente AudioSource del GameObject donde está asignado el script.

- El método *OnTriggerEnter2D(Collider2D collision)* se ejecuta cuando el GameObject colisiona con otro Collider que tenga el tag “Player”. Cambia temporalmente el tag del GameObject a “Untagged” para evitar que otros objetos lo recojan repetidamente.
  Posteriormente se incrementa el contador de pergaminos, actualiza el texto en la interfaz de juego y reproduce un sonido y se desactiva la representación visual del GameObject y de su sistema de partículas.

  Finalmente, se destruye el GameObject después de un tiempo, en este caso después de 0.08f.

- El método estático *reiniciarPergaminos()* se encarga de reiniciar el contador de pergaminos a cero.

RecogerMonedas

- Este script está asociado al GameObject moneda y está diseñado para gestionar la recogida de monedas siguiendo el mismo modelo que el anterior script ‘RecogerPergamino’.

- Código:

  - Lo único que se añade diferente al código del script ‘RecogerPergamino’ es que en el método *OnTriggerEnter2D(Collider2D collision)* se incrementa la munición de una “Bola”.

Puntuacion

- Este script está asociado al GameObject Canvas de la escena Resultado. Desde el Inspector le damos valor a las variables públicas de tipo Text de la puntuación y del tiempo.

- Cuenta con las siguientes variables:

  - Una variable pública de tipo Text **puntuacionText** que hace referencia al objeto de texto que mostrará la puntuación actual en la interfaz.

  - Una variable pública de tipo Text **tiempoTxt** que hace referencia al objeto de texto que mostrará el mensaje de tiempo en la interfaz.

- Una variable privada de tipo entera ‘recordScore’ para almacenar el récord actual del jugador.

- Una variable privada y estática de tipo Dificultad **dificultadActual** que almacena la dificultad actual del juego.

- Código:

  - El método *Start()* obtiene la dificultad actual del menú de inicio y si la escena activa es “Resultado”, carga el récord almacenado, muestra el mensaje de tiempo y actualiza el texto con el record actual.
  - El método *mostrarMensajeTiempo()* calcula la puntuación del jugador basándose en el tiempo final, la cantidad de monedas recogidas y un multiplicador de puntuación del menú de inicio. Después muestra la puntuación en el ‘tiempoTxt’ y llama a *UpdateScore()* para actualizar el récord si es necesario.
  - El método *UpdateScore()* actualiza el récord si la nueva puntuación es mayor al récord actual. Después llama a *SaveRecord()* para guardar el nuevo récord y a *UpdateRecordText()* para actualizar el texto con el nuevo récord.
  - El método *SaveRecord()* guarda el récord en PlayerPrefs (almacenamiento persistente en Unity).
  - El método *LoadRecord()* carga el récord almacenado desde PlayerPrefs.
  - El método *UpdateRecordText()* actualiza el texto en ‘puntuacionTxt’ con el récord actual.
  - El método *restart()* restablece la dificultad, reinicia la puntuación de pergaminos y monedas, reinicia las vidas y carga la escena del juego llamada ‘PrimerNivel’. Este método está asignado al botón restart por lo que se ejecuta cuando se pulsa dicho botón.
  - El método *menu()* reinicia la puntuación de pergaminos y monedas, reinicia las vidas y carga la escena ‘Inicio’. Este método está asignado al botón menú por lo que se ejecuta cuando se pulsa dicho botón.

PerderVida

- Este script está asociado al GameObject Personaje y está diseñado para gestionar la vida del personaje del jeugo.

- Cuenta con las siguientes variables:

- Una variable pública y estática de tipo entera **vidasMaximas** que hace referencia al número de vidas que puede tener el personaje.

- Una variable pública y estática de tipo entera **vidaInicial** que indica la cantidad inicial de vidas del personaje.

- Una variable pública y estática de tipo entera **vidaActual** para la cantidad actual de vidas que posee el personaje.

- Una variable pública de tipo List<Image> **vidas**, una lista de imágenes que representan las vidas individuales del personaje.

- Una variable pública de tipo List<Image> **contenedores**, una lista de imágenes que actúan como contenedores de las vidas.

- Una variable privada de tipo MoverPersonaje ‘movimiento’ para controlar el movimiento del personaje.

- Una variable privada de tipo float **tiempoPerdidaControl**, hace referencia a la duración en segundos durante la cual el personaje pierde el control después de recibir el daño.

- Una variable pública de tipo AudioClip **audioClip** para el sonido que se reproduce al recibir daño.

- Una variable privada AudioSource **audioSource** para reproducir sonidos.

- Una variable privada de tipo Animator **anim** para referenciar al componente Animator (para gestionar las animaciones del personaje).

● Código:

- Método *Start()* se llama al método *inicializarVida()*, y obtiene las referencias a los componentes necesarios (movimiento, anim, audioSource).

- El método *recibirDaño(Vector2 posicion)* se llama cuando el personaje recibe daño. Inicia una animación de golpe, reproduce un sonido, pierde temporalmente el control y aplica un retroceso.

- El método *perderControl()* hace que el personaje pierda el control temporalmente durante un tiempo específico.

- El método *inicializarVida()* configura la vida inicial del personaje según la dificultad seleccionada. Itera sobre la lista de **vidas** y 'contenedores' y establece la propiedad enabled en true para todas las imágenes, esto garantiza que todas las imágenes asociadas a vidas estén visibles al inicio.
  Posteriormente desactiva las vidas adicionales, iterando desde **vidaInicial** + 1 hasta **vidasMaximas** establece enabled a false de los contenedores y las imágenes de vidas correspondientes más allá de la cantidad inicial. Esto asegura que no se muestren vidas adicionales al principio del juego.

- El método *OnCollisionEneter2D(Collision2D collision)* se activa cuando hay una colisión entre el personaje y otro objeto en el escenario. Usa los tags 'damage' y 'Muerte' para identificar los objetos con los que ha colisionado.

    - Lógica de colisiones:

        - 'damage' Tag:
            - *recibirDaño(collision.GetContact(0).normal)* llama al método *recibirDaño()* y le pasa la normal de la colisión para determinar la dirección del retroceso y activar la animación.
            - 'vidas[vidaActual].enabled = false', desactiva una imágen de vida.
            - 'vidaActual--;' disminuye la vida actual en 1.
            - Si la vida llega a cero o menos, se carga la escena 'NivelFinalizado'.
        - 'Muerte' Tag:
            - *recibirDaño(collision.GetContact(0).normal)* realiza las mismas acciones que en el caso de damage.
            - Después de disminuir la vida actual, llama al método *revivir()*.

- El método *comprobarCheckPoints()* usa las coordenadas **CampanaManager.coordenadaX** y **CampanaManager.coordenadaY** que son las coordenadas del último checkpoint activo. Después con el 'transform.position' cambia la posición del personaje al vector2 con las coordenadas del último checkpoint cogido.

- El método *revivir()* se encarga de gestionar la lógica cuando el jugador necesita ser revivido. Si la **vidaActual** es menor que 0 significa que el

jugador ha perdido todas las vidas por lo que se carga la escena 'NivelFinalizado'.
En caso de que le queden vidas todavía, se llama al método *comprobarCheckPoints()* para que el personaje se dirija a la posición del último checkpoint accedido.

- El método *reiniciar()* se encarga de reiniciar la vida del jugador asignando al valor de la variable **vidaActual** el valor de **vidaInicial**.

Video

- Este script está asignado a un GameObject vacío dentro de la escena Tutorial y se encarga de controlar el video del tutorial y permitir al jugador saltarlo al presionar cualquier tecla.

- Cuenta con la siguiente variable:
  - Una variable pública y estática de tipo booleano 'saltar' que indica si el jugador quiere saltar el tutorial o no (se inicializa como false).

- Código:
  - El método *Start()* usa el método Invoke() para llamar al método *cambioEscena()* después de 49 seg.
  - El método *cambioEscena()* verifica si la variable saltar es false y si lo es, carga la escena "PrimerNivel".
  - El método *listenerSkip()* verifica si se ha presionado alguna tecla y en caso afirmativo, llama al método *cambioEscena()* y establece la variable saltar en true.
  - El método *Update()* llama al método *listenerSkip()* en cada frame para verificar si se ha presionado alguna tecla.

AbrirPuerta

- Este script está asignado al GameObject PuertaFinal y está diseñado para abrir una puerta cuando se recogen al menos 5 XMLs.

- Cuenta con las siguientes variables:
  - Una variable privada de tipo booleana 'abrir' indica si la puerta debe abrirse o no (se inicializa en false).
  - Una variable pública y estática de tipo float 'tiempoFinal' almacena el tiempo final cuando el jugador entra en la zona específica.

- Código:
  - El método *Update()* llama al método *abrirPuerta()* en cada frame.
  - El método *abrirPuerta()* verifica si la cantidad de pergaminos recogidos es mayor o igual a 5 y si la puerta aún no se ha abierto. Si se cumple la

condición, cambia la propiedad ‘flipC’ del componente SpriteRenderer simulando la apertura de la puerta. Una vex que la puerta se ha abierto, ‘abrir’ se establece en true asegurando que la puerta no se abra repetidamente.

- El método *OnTriggerEnter2D()* verifica si el objeto que entra tiene el tag ‘Player’ y si la cantidad de pergaminos recogidos es mayor o igual a 5 y si se cumple la condición, asigna el valor scr.tiempo a tiempoFinal (almacenando el tiempo cuando se abrió la puerta) y carga la escena “Resultado”.

PatrullaSeguimientoScript

- Este script se le asigna al prefab de uno de los enemigos y se encarga de que el enemigo siga al jugador.

- Cuenta con las siguientes variables:

    - Una variable [SerializeField] (este tipo de variables que independientemente de la privacidad se puede acceder a ella desde el inspector) de tipo Transform **jugador** que representa la posición del jugador.

    - Una variable [SerializeField] de tipo float **rangoDeVision** que determina la distancia máxima a la cual el enemigo puede detectar al jugador.

    - Una variable [SerializeField] de tipo float **velocidad** para determinar la velocidad a la que se mueve el enemigo.

    - Una variable [SerializeField] de tipo SpriteRendered **sprite** que se usa para controlar la orientación visual del enemigo.

    - Una variable privada de tipo float **distancia** para saber la distancia actual entre el enemigo y el jugador.

    - Una variable privada de tipo Animator **anim** para gestionar las animaciones del enemigo.

    - Una variable pública de tipo AudioClip **audioClip** y otra privada de tipo AudioSource **audioSource** para reproducir un sonido cuando el jugador colisiona con el enemigo.

- Código:

- El método *Start()* calcula la distancia inicial entre el enemigo y el jugador y obtiene las referencias a los componentes SpriteRenderer, Animator y AudioSource.

- El método *Update()* actualiza la distancia actual entre el enemigo y el jugador y llama al método girar() para ajustar la orientación del enemigo.

- El método *girar(float distancia)* comprueba si el jugador está dentro del rango de visión, después calcula la dirección hacia la posición del jugador.
  Posteriormente mueve al enemigo en esa dirección con una velocidad determinada, voltea el sprite dependiendo de la posición del jugador y activa o desactiva la animación de seguimiento según si el jugador está dentro o fuera del rango.

- El método *OnCollisionEnter2D(Collision2D collision)* se ejecuta cuando el enemigo colisiona con otro objeto (especialmente el jugador). Después verifica si la colisión es con el jugador (‘Player’) y si la colisión es desde arriba desetiqueta al enemigo para evitar más interacciones con el jugador, aplica un rebote al jugador usando *rebote()* del script MoverPersonaje, reproduce un sonido de muerte y activa la animación de muerte.

- El método *muerteHuskHornhead()* destruye el objeto del enemigo.

PatrullaVolando

- Este script está asignado al prefab del de enemigo volador y sirve para manejar su comportamiento.

- Cuenta con las siguientes variables:

  - Una variable privada y [SerializeField] de tipo float ‘vel’ para determinar la velocidad de movimiento del objeto volador.

  - Una variable privada y [SerializeField] de tipo Transform[] **puntosVuelo** lista de puntos a los cuales el objeto se moverá.

  - Una variable privada y [SerializeField] de tipo float **distanciaMin** distancia mínima para cambiar al siguiente punto de vuelo.

- Una variable privada de tipo Animator ‘anim’ se usa para gestionar las animaciones.

- Una variable privada de tipo entera **siguientePaso.**

- Una variable privada de tipo SpriteRenderer **sprite.**
- Una variable pública de tipo AudioClip y otra privada de tipo AudioSource usados para reproducir un sonido cuando el objeto volador colisiona con el jugador.

- Código:

    - El método *start()* inicia las referencias a los componentes SpriteRenderer, Animator y AudioSource.

    - El método *Update()* mueve el enemigo hacia el siguiente punto de vuelo a una velocidad determinada y verifica si el objeto ha llegado al punto de vuelo actual y, en caso afirmativo, pasa al siguiente punto y llama al método *girar()*.

    - El método *girar()* se encarga de voltear el sprite del objeto dependiendo de su posición en comparación con el siguiente punto de vuelo.

    - El método *OnCollisionEnter2D(Collision2D collision)* se ejecuta cuando el objeto volador colisiona con otro objeto (especialmente el jugador). Después verifica si la colisión es con el jugador (‘Player’) y si la colisión es desde arriba desetiqueta al objeto volador para evitar más interacciones con el jugador, aplica un rebote al jugador utilizando el método *rebote()* del script MoverPersonaje, reproduce un sonido de muerte y finalmente activa la animación de muerte.

    - El método *muerteArmouredSquit()* destruye el objeto del enemigo.

Patrullar

- Este script está asignado al prefab del enemigo que patrulla y está diseñado para controlar el comportamiento de un enemigo que patrulla horizontalmente y reacciona cuando colisiona con una plataforma o con el jugador.

- Cuenta con las siguientes variables:

- Una variable privada de tipo Rigibody2D **rb** para controlar el movimiento del enemigo.

- Una variable pública de tipo float **vel** para la velocidad horizontal del enemigo.
- Una variable privada de tipo Animator **anim** se usa para gestionar las animaciones.

- Una variable pública de tipo AudioClip y otra privada de tipo AudioSource para reproducir un sonido cuando el enemigo colisiona con el jugador.

- Código:

    - El método *Start()* inicializa las referencias a los componentes Rigibody2D, Animator y AudioSource.

    - El método *Update()* asigna una velocidad horizontal constante al Rigibody2D para que el enemigo se mueva en una dirección específica.

    - El método *OnTriggerExit2D(Collider2D collision)* se ejecuta cuando otro collider deja de estar en contacto con el collider de este objeto, verifica si el objeto es una plataforma mediante su tag (“plataforma”). Después invierte la dirección **vel** *= -1 del movimiento del enemigo y voltea el enemigo horizontalmente para que apunte en la dirección correcta.

    - El método *OnCollisionEnter2D(Collision2D collision)* se ejecuta cuando este collider entra en contacto con otro collider. Después verifica si la colisión es con el jugador (“Player”) y si la colisión es desde arriba desetiqueta el objeto para evitar que haya más interacciones con el jugador, aplica un rebote al jugador usando *rebote()*, reproduce un sonido de muerte y finalmente activa la animación de muerte.

    - El método *muerteCrawlid()* destruye el objeto del enemigo.

3. Dinámica de juego

El juego se centra en la exploración, la resolución de combates entre los 3 tipos de enemigos y la búsqueda de todos los trozos de código XML para abrir la puerta final y ganar la partida.

Aspectos de la dinámica de juego:

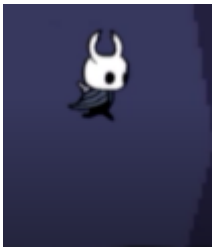
- **Exploración**: El juego se desarrolla en un reino, donde el personaje explora diferentes zonas del escenario con diversos obstáculos y plataformas, adquiriendo la posibilidad de lanzar hechizos al recolectar las monedas del juego.

- **Combate**: El personaje se enfrenta a diversos enemigos que le obstaculizan el paso, cada uno cuenta con movimientos diferentes (patrullaje de una zona terrestre, patrullaje de una zona aérea y perseguir al personaje).

- **Plataformas**: Existen diferentes plataformas que requieren de pensamiento estratégico y habilidades de salto precisas para poder avanzar con éxito. Además de esquivar obstáculos (pinchos) que le restan vida a nuestro personaje.

4. Manual

- **Controles Del Juego:**
  - **Salto**: Pulsando la tecla “Space” (Espacio).

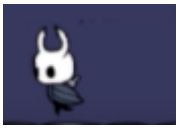


  - **Doble Salto**: Pulsando dos veces la tecla “Space” (Espacio).

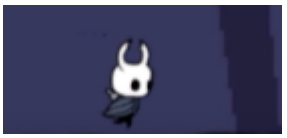
  - **Lanzar Hechizo**: Pulsando la tecla E.

  - **Menú Pausa**: Pulsando la tecla ESC (Escape).

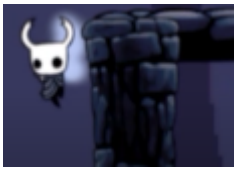
➢ **Mover Izquierda**: Pulsando la tecla A o flecha izquierda.



➢ **Mover Derecha**: Pulsando la tecla D o flecha derecha.



➢ **Deslizarse Pared**: Manteniendo la tecla A o flecha izquierda, en el caso de que la pared se ubique en la izquierda. En el caso de que la pared se encuentre en la parte derecha manteniendo la tecla D o flecha derecha.

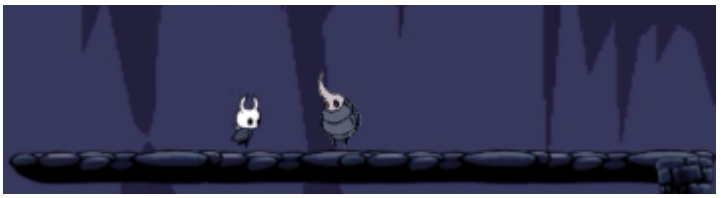


❖ **Enemigos Y Combate:**

Disponemos de 3 tipos de enemigos:

➢ **Enemigo Patrullador**: este enemigo patrulla una zona determinada del escenario.

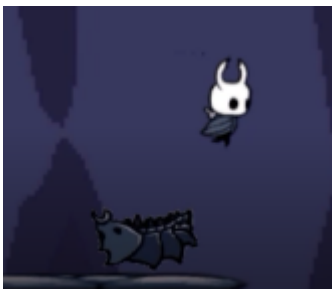
➢ **Enemigo Perseguidor**: este enemigo nos persigue durante el juego.



➢ **Enemigo Volador**: este enemigo vuela desde un punto a otro de la zona del mapa.

Disponemos de 2 tipos de ataque:

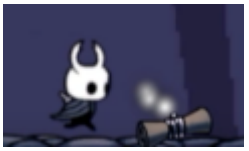
➢ **Ataque pisotón**: el personaje puede matar a los diferentes enemigos saltando por encima de ellos y pisarlos.



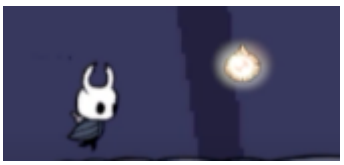
➢ **Ataque hechizo**: el personaje puede lanzar un hechizo, en el que lanza una bola negra, por la que cuando colisione algún enemigo con ella lo mata.

❖ **Objetos:** existen dos tipos de objetos en el juego.

➢ **Trozo XML:** es el objeto más importante en el juego. Necesitamos recoger todos los Trozos XML para abrir la puerta final y ganar la partida.



➢ **Moneda:** es un objeto, el cúal nos permite incrementar nuestros hechizos y la puntuación final de la partida.



- **Punto De Reaparición:** existen puntos de reaparición en forma de campana repartidos en diferentes zonas del mapa, donde al atravesarlo, en el caso de que el personaje muera, reaparece en el punto de reaparición atravesado y te rellena la vida al máximo.

- **Exploración:** el personaje tendrá que recorrer todo el mapa, donde se encontrará diferentes plataformas, obstáculos (pinchos).

- **Objetivos:** el objetivo principal es recolectar todos los trozos de XML para abrir la puerta final. Además, hay que llegar a dicha puerta en el menor tiempo posible y recogiendo el máximo número de monedas para conseguir la máxima puntuación.