|  |  |  |
| --- | --- | --- |
|  | Universidad de Burgos  Escuela Politécnica Superior  Gºen Ingeniería en Informática |  |

D. <nombre\_tutor>, profesor del departamento de Ingeniería <...>, área de <...>

Expone:

Que el alumno D. C P.A., con DNI <...>, ha realizado el titulado: .

y que dicho trabajo ha sido realizado por el alumno bajo la dirección del que suscribe, en virtud de lo cual, Se autoriza su presentación y defensa.

En Burgos a de <junio de 2016>

<nombre\_tutor>

**Resumen**

La intención de este trabajo va a ser el desarrollo de un videojuego de plataformas 2D con el motor gráfico Unity y el lenguaje de programación C#.

Las principales mecánicas del juego consistirán en influir sobre la física y el tiempo para ayudar o dificultar que el jugador se pase los niveles del juego.

**Descriptores**

Videojuegos, plataformas 2D, Unity, C#

**Abstract**

The intention of this work will be to develop a 2D platformer video game with the graphics engine Unity and the programming languaje C#.

The main mechanics of the game will consist of manipulating physics and time to help or obstruct the player to pass the levels of the game.

**Keywords**

Video game, Platformer 2D, Unity, C#

# **Índice general**

[**Índice general** 5](#_Toc67762321)

[**Índice de figuras** 7](#_Toc67762322)

[**Introducción** 8](#_Toc67762323)

[**Objetivos del proyecto** 9](#_Toc67762324)

[**Conceptos teóricos** 10](#_Toc67762325)

[***Unity*** 10](#_Toc67762326)

[**MonoBehaviour(2)** 10](#_Toc67762327)

[**GameObject(3)** 11](#_Toc67762328)

[**Escena** 12](#_Toc67762329)

[**Físicas en Unity** 12](#_Toc67762330)

[**Colisión entre objetos** 13](#_Toc67762331)

[**Técnicas y herramientas** 14](#_Toc67762332)

[***Motor de juego*** 14](#_Toc67762333)

[**Unity** 14](#_Toc67762334)

[**Pygame (librería de Python)** 14](#_Toc67762335)

[**Decisión final** 14](#_Toc67762336)

[**Aspectos relevantes del desarrollo del proyecto** 16](#_Toc67762337)

[**Trabajos relacionados** 17](#_Toc67762338)

[***Juegos similares en género y mecánicas*** 17](#_Toc67762339)

[***Evaluación de Plataformer Microgame(2)*** 19](#_Toc67762340)

[**Grid** 19](#_Toc67762341)

[**UI Canvas** 19](#_Toc67762342)

[**Enemies** 19](#_Toc67762343)

[**Tokens** 19](#_Toc67762344)

[**Zones** 19](#_Toc67762345)

[**GameController** 19](#_Toc67762346)

[**Player** 20](#_Toc67762347)

[**Simulation** 20](#_Toc67762348)

[**Pegas** 20](#_Toc67762349)

[**Conclusiones** 21](#_Toc67762350)

[**Conclusiones y líneas de trabajo futuras** 23](#_Toc67762351)

[**Bibliografía** 24](#_Toc67762352)

# **Índice de figuras**

[Ilustración 1: Captura de pantalla del videojuego Celeste. 17](#_Toc67762353)

[Ilustración 2: Captura de pantalla de Super Meat Boy. 18](#_Toc67762354)

[Ilustración 3: Nivel de presentación de Plataformer Microgame. 19](#_Toc67762355)

# **Introducción**

# **Objetivos del proyecto**

El objetivo principal del proyecto será el desarrollo de un videojuego plataformas 2D. De este objetivo principal se derivan otros varios tales como aprender el funcionamiento y estructura del motor gráfico Unity, que será el utilizado durante el desarrollo del videojuego. También se busca analizar, comprender e implementar los elementos que componen un videojuego de plataformas 2D y la arquitectura lógica que permite que funcione.

# **Conceptos teóricos**

## ***Unity***

En este apartado se va a explicar que es Unity y algunos de los elementos que lo componen y de los que hace uso. Este apartado es importante porque durante toda la memoria se va a hacer referencia a estos elementos asumiendo que se ha leído este apartado.

Unity(1) es un motor gráfico orientado al desarrollo de videojuegos que permite desarrollar para ordenador (Microsoft Windows, Mac OS y Linux), consolas, móvil y dispositivos de realidad aumentada.

Unity tiene una serie de elementos, pero los más destacables son:

### **MonoBehaviour(2)**

MonoBehaviour es la clase de la que (en principio) parten todas las clases que utiliza Unity. Esta clase y sus hijas son las que inicializa Unity al crear una escena. La razón por la que esta clase es tan importante y requiere de explicación son los siguientes métodos:

**Awake() y Start():** Tanto Awake como Start son métodos que se ejecutan al crear un objeto que herede de MonoBehaviour, sin embargo funcionan de manera ligeramente distinta.

El método Awake se llama en el instante exacto en el que se carga el script al que esta asociado o en el que se crea la instancia del nuevo objeto de esta clase. Funciona prácticamente como el constructor de una clase. Es un método muy adecuado para inicializar variables ya que se ejecutará justo al crearse la instancia del objeto o cargarse el script.

El método Awake se ejecutará siempre independientemente de que el objeto este activado o no. Con activado se hace referencia al atributo booleano *enabled* de la clase MonoBehaviour. Cuando este atributo este a *false* el objeto no realizará ninguna operación (actuará como si estuviese desactivado) y cuando este a *true* funcionará con normalidad.

Como se ha mencionado el método Start funciona de manera muy similar a Awake pero con dos diferencias clave. El método Start se activa antes de que se llame a cualquier método Update (explicados a continuación), pero no garantiza que se valla a llamar en el mismo momento en el que se crea el objeto o se carga el script (a diferencia del método Awake). La otra diferencia con Awake es que el método Start se llamará solo si el objeto esta activado (*enabled* a *true*) mientras que Awake se llamará siempre.

**Update(), FixedUpdate() y LateUpdate():** Estos tres son métodos que se llaman repetitivamente hasta la desaparición del objeto. Estos métodos son los que utilizará Unity para saber en todo momento que es lo que tiene que hacer y en que momento ha de hacerlo. Pero ¿Por qué hacer tres métodos distintos para albergar instrucciones que se repetirán continuamente? La razón de esto reside en cuándo se ejecutan estos tres métodos.

El método Update se llama cada *frame*. Explicado a grandes rasgos un *frame* representa el momento en el que cambia lo que sale por pantalla. Así que cada vez que cambia lo que se ve en pantalla se llama al método Update. El tiempo que puede pasar entre *frames* no tiene por qué ser siempre el mismo. Es por ello que el periodo entre llamadas al método Update no siempre será el mismo.

El método FixedUpdate se llama repetitivamente, con la excepción de que no lo hace cada *frame*, sino que se repite en el tiempo de manera regular. El periodo entre llamadas al método FixedUpdate será siempre de 0.02 segundos. Este periodo se puede modificar pero la llamada al método Fixed Update siempre será regular. Esto hace al método FixedUpdate un método ideal para realizar cálculos dependientes del momento del tiempo en el que te halles (calcular trayectorias de objetos por ejemplo).

El método LateUpdate es muy similar al método Update. También se ejecuta en cada *frame* pero difiere del método Update en que se ejecutará siempre después de los demás métodos “update”. Esto puede ser útil para la actualización de elementos que requieren que se hayan hecho una serie de cambios antes. Un ejemplo podría ser el seguimiento de una cámara a un objeto. Si se establece el seguimiento de la cámara y luego se actualiza la posición del objeto la cámara va a estar persiguiendo al objeto en una posición en la que no está.

**Mensajes:**

Además de estos métodos hay otros métodos que se identifican por el nombre de “mensajes”. Los “mensajes” funcionan de forma un poco diferente a los métodos normales y corrientes. Son métodos que se pueden activar con normalidad al llamarlos, pero también son métodos que se pueden activar al “recibir un mensaje” utilizando el método SendMessage() de la clase GameObject (esta clase se explicará a continuación). La gracia de estos métodos “mensaje” no está en que se encuentren en la clase MonoBehaviour, sino como interactúan con otras clases y objetos. Es por eso que a continuación se mencionarán los más interesantes, pero la explicación de los métodos se realizará en otro apartado en el que se haga referencia a estos métodos y se comprenda mejor el uso de estos.

Métodos “mensaje”:

* OnCollisionEnter.
* OnCollisionEnter2D.
* OnCollisionExit.
* OnCollisionExit2D.
* OnCollisionStay.
* OnCollisionStay2D.
* OnTriggerEnter.
* OnTriggerEnter2D.
* OnTriggerExit.
* OnTriggerExit2D.
* OnTriggerStay.
* OnTriggerStay2D.
* Start (también es un método “mensaje”).
* Update (también es un método “mensaje”).

### **GameObject(3)**

La clase GameObject es la clase de la que parten todos los objetos que va a utilizar Unity. Todos los elementos que creas en Unity son GameObject. La clase GameObject por defecto añade atributos que ofrecen información básica sobre ese GameObject. Algunos de los más importantes serían: transform para saber que región del espacio ocupa el GameObject, tag para identificar al GameObject y name para identificar tanto al GameObject como a los componentes de este (todos comparten el mismo nombre).

GameObject por si sola es una clase inútil. Pero lo importante de GameObject es la capacidad que posee para añadirse componentes a si mimo (con el método GetComponent entre otros). Los componentes, al ser añadidos a la instancia de GameObject que le corresponda ya pueden ser usados por Unity. Los componentes de GameObject pueden ser objetos de cualquier tipo, sin necesidad de heredar de MonoBehaviour.

GameObject posee varios métodos estáticos, pero se van a explicar dos que se creen dignos de mención. Estos métodos son Destroy e Instantiate. Lo que hacen estos métodos se puede deducir por el nombre de estos. Destroy destruye instancias de GameObject y/o componentes de estos y el método Instantiate las crea. Sn embargo, cabe mencionar que el método Instantiate no crea de verdad los objetos, sino que clona uno existente y devuelve el objeto clonado. Esto es importante porque Unity provee unos objetos especiales que son los Prefabs. Estos Prefabs son un GameObject persistente que guarda la configuración con la que ha sido construido ese GameObject. De esta manera si tienes un Prefab de, por ejemplo, un enemigo del juego puedes crear todas las copias de ese enemigo que quieras llamando al método Instantiate y pasando como argumento ese Prefab. Con los Prefabs se logra tener copias idénticas de objetos sin tener que crearlas manualmente cada vez.

La última característica importante de GameObject es el método SendMessage. Este método permite al GameObject o un componente suyo mandar un mensaje a los demás componentes del GameObject, haciendo que todos los componentes (en realidad solo los que hereden de MonoBehaviour) que tengan un método con nombre igual al pasado como argumento en el método SendMessage ejecutarán ese método. Un ejemplo sería gameobject.SendMessage(“Metodo1”). Al ejecutar ese método se hará que todos los componentes que hereden de MonoBehaviour y tengan un método llamado Metodo1 invoquen ese método.

### **Escena**

Las escenas son los elementos de Unity que contienen otros objetos. Una escena puede ser un nivel o un menú del juego.

### **Físicas en Unity**

Un objeto por defecto no se ve afectado por las físicas. Sin embargo añadiendo el componente Rigidbody (Rigidbody2D para los juegos en dos dimensiones) el GameObject al que este asociado variará su posición como si estuviese afectado por las físicas.

Rigidbody tiene un atributo llamado *velocity*. Este atributo un vector de 3 dimensiones que representa en que dirección se moverá el GameObject afectado por las físicas. Un componente Rigidbody hace que su GameObject se vea afectado por la gravedad o no con el atributo booleano *useGravity*.

Rigidbody tiene un atributo booleano llamado *isKinematic*. Este atributo hace que un objeto no se vea afectado por las colisiones. Un objeto con un Rigidbody con el atributo *isKinematic* igual a *true* (objeto 1) que colisiona con otro con un Rigidbody con el atributo *iskinematic* igual a *false* (objeto 2) provoca que el objeto 1 modifique su movimiento teniendo en cuanta las reacciones físicas generadas por la colisión con el objeto 2. El objeto 2, sin embargo no verá su movimiento afectado por la colisión.

### **Colisión entre objetos**

Unity ofrece por defecto una manera de manejar las colisiones entre objetos. Para que un GameObject ofrezca la posibilidad de colisionar con otro objeto, este debe de tener un componente llamado Collider (para los juegos en dos dimensiones Collider2D). Este componente te permite determinar una región del espacio en la que otro GameObject con un componente Collider se considerará colisionando con el primer objeto. Si dos Colliders comparten algún punto de los espacios que delimitan, se considerará que se ha producido una colisión entre los GameObject a los que pertenecen.

La región del espacio que ocupa un Collider por defecto es estática y no se puede mover, sin embargo moviendo la posición del GameObject moverá la posición del Collider, pues su posición es relativa al GameObject.

La colisión entre objetos se desarrolla mediante la ejecución de los métodos OnCollisionEnter/Stay/Exit de los componentes que heredan de la clase MonoBehaviour que tengan implementados esos métodos.

Si dos objetos tienen un Rigidbody y un Collider, al colisionar ambos objetos variaran su movimiento en consecuencia. Sin embargo, si se desea evitar esto, el componente Collider tiene un atributo booleano llamado *isTrigger*. Si está a *false* el funcionamiento será el explicado anteriormente. Sin embargo, si está a *true* el atributo *isTrigger*, su movimiento no se verá afectado al colisionar pero si que se detectará la colisión. En caso de que el atributo *isTrigger* sea *true*, la colisión no se resolverá ejecutando los métodos anteriormente explicados, sino que se resolverá mediante los métodos OnTriggerEnter/Stay/Exit de los componentes que heredan de la clase MonoBehaviour que tengan implementados esos métodos.

# **Técnicas y herramientas**

## ***Motor de juego***

Para la creación del videojuego se planteó apoyarse en un motor de juego ya creado frente a implementar todo el proyecto desde 0. Se planteó utilizar Unity (hacer uso de un motor gráfico) frente a la librería de Python Pygame (no hacer uso de un motor gráfico).

### **Unity**

Para el desarrollo del videojuego se ha considerado utilizar Unity como motor gráfico, ya que es un motor gráfico gratuito de fácil uso (aunque limitado en algunos aspectos), pero que ofrece los recursos necesarios para el desarrollo. Esta herramienta trae elementos ya implementados que ahorran mucho tiempo de trabajo tales como los Colliders (clases encargadas del manejo de las colisiones entre objetos) y clases encargadas de simular las físicas e interactuar entre estas y los objetos en la escena. Además, Unity ofrece una interfaz que facilite la visualización de los niveles del videojuego.

El argumento final para elegir este motor gráfico y no otros como, por ejemplo, Unreal Engine 4 ha sido completamente subjetivo. Ya se tiene experiencia previa y se ahorrará mucho tiempo del que se invertiría en el proceso de aprendizaje de otro motor gráfico

### **Pygame (librería de Python)**

Pygame es una herramienta que ofrece una serie de clases que ofrecen una solución intermedia entre construir desde cero todo el código relativo al desarrollo de un videojuego y un motor gráfico que ofrece bastantes elementos de un videojuego ya implementados.

Construir desde cero el videojuego podría llevar demasiado tiempo y probablemente no diese tiempo a desarrollar el videojuego entero como un elemento funcional. Sin embargo, hacerlo desde cero ofrece una libertad absoluta en el desarrollo y la funcionalidad.

Utilizar un motor gráfico para el desarrollo del videojuego facilita mucho el desarrollo, sin embargo obliga a ceñirse al modelo que sigue el motor gráfico.

La librería de Pygame ofrece una solución intermedia, ofreciendo bastante libertad y una estructura de clases que limita muy poco ofreciendo las funcionalidades justas y necesarias (creación de la ventana donde se mostrará el juego, visualización de sprites y elementos visuales y poco más.).

### **Decisión final**

Finalmente se ha optado por el uso del motor gráfico Unity en lugar de la librería Pygame porque es un entorno con el que se está más familiarizado (teniéndose un conocimiento mucho más profundo de Unity que de Pygame). Como ya se ha mencionado anteriormente el proceso de aprendizaje puede llevar demasiado tiempo (siendo que para Pygame se posee un conocimiento muy básico). Adicionalmente se teme que, al ser Pygame demasiado abierto (una de sus ventajas), no se tenga tiempo suficiente para crear un videojuego de suficiente calidad.

Debido a su facilidad de uso y los elementos que ya trae por defecto se ha elegido Unity para el desarrollo del videojuego.

# **Aspectos relevantes del desarrollo del proyecto**

# **Trabajos relacionados**

## ***Juegos similares en género y mecánicas***

Este videojuego se inspira en otros dos videojuegos diferentes: Super Meat Boy y Celeste. Esta temática es a nivel de género más que de mecánicas. Ambos juegos son plataformas 2D comprometidos con sus mecánicas y precisos en su jugabilidad (esto es lo que se busca con el proyecto que se va a desarrollar). Es cierto que Celeste está más centrado en la historia que Super Meat Boy, mientras este se centra casi exclusivamente en las mecánicas.

Celeste representa la variedad mecánica que se desea alcanzar, ofreciendo mecánicas distintas como acelerones y portales (igual que el videojuego que se va a desarrollar) e incluso mecánicas que modifican el estado natural del juego (como permitir dar más de un acelerón en el aire cuando no sería normalmente posible). Se aspira a alcanzar la variedad de mecánicas que Celeste provee y la diversión que estas generan.

Imagen de la pantalla de un video juego

Descripción generada automáticamente con confianza baja

Ilustración 1: Captura de pantalla del videojuego Celeste.

El videojuego Super Meat Boy está muy centrado en el movimiento del jugador. La calidad de este juego es tal, que el jugador es en todo momento consciente de dónde está el avatar que controla y qué está haciendo. El juego da mucha importancia a las físicas y cómo el jugador interactúa con ellas. Estas físicas no cambian, pero son un elemento muy bien establecido e intuitivo. En varios niveles el jugador tiene que hacer uso de las físicas y la inercia para superar obstáculos que en condiciones normales no sería capaz de superar.

Imagen que contiene Diagrama

Descripción generada automáticamente

Ilustración 2: Captura de pantalla de Super Meat Boy.

En cuanto a la estructura de los niveles, Celeste y Super Meat Boy difieren ligeramente. En Celeste el nivel está dividido en subniveles que no tienen por qué ser independientes entre sí. El jugador escoge un capítulo y ese capítulo está dividido en una serie de niveles por los que el jugador viaja hasta alcanzar el último nivel y superar el capítulo. En la captura de pantalla anteriormente mostrada se puede observar cómo cada nivel de Celeste es cerrado, con unos límites definidos y una entrada y salida claras. El nivel generalmente se superará resolviendo un puzle que se manifiesta deduciendo un camino que requerirá el uso de distintas mecánicas para recorrerse y llegar al siguiente nivel.

Los niveles de Super Meat Boy, como se puede observar en la captura de pantalla, no están limitados al alcance de la cámara, sino que la meta no se ve inicialmente. En el nivel mostrado en la captura se muestra un camino claro a seguir, pero no tiene por qué ser así.

A diferencia de Celeste, los niveles de Super Meat Boy son independientes entre sí y la salida de un nivel no es la entrada a otro, mas es cierto que se sigue una temática en la que los niveles siguen un patrón visual similar.

Con el videojuego que se va a desarrollar se desea seguir una estructura de niveles similar a Super Meat Boy, con niveles abiertos, independientes entre sí y que no están limitados a la visión de la cámara.

Para las mecánicas se va a seguir el ejemplo de Celeste, incluyendo mecánicas visuales y variadas que generen interacción entre sí.

La temática y mecánicas de modificación de las físicas y el tiempo es la parte original que se va a implementar en el videojuego que se va a desarrollar. Mencionar aun así, que se van a utilizar mecánicas que no son enteramente originales (como el “tiempo bala” que se utiliza en otros géneros y en algunos otros juegos de plataformas) pero que se van a adaptar al género de plataformas 2D.

Existe un juego de plataformas 2D que se llama Braid(1) e implementa mecánicas de viajes en el tiempo. Al no ser estas mecánicas similares a las que se proponen para este proyecto, no ha sido objeto de análisis.

## ***Evaluación de Plataformer Microgame(2)***

Para el desarrollo del videojuego se planteó partir de una plantilla de proyecto que proporcionase las bases del videojuego. Como plantilla de partida se eligió Plataformer Microgame[[1]](#footnote-1) y se realizó un estudio de la plantilla para ver si era válida como punto de partida. Del estudio se obtuvo un análisis de los elementos de Plataformer Microgame.



Ilustración 3: Nivel de presentación de Plataformer Microgame.

### **Grid**

Unity tiene un objeto que es el Tilemap. Este objeto permite de manera sencilla representar escenarios a partir de sprites añadidos al editor de Tilemaps. Grid es un objeto formado por una serie de Tilemaps (Foreground, Background, FarBackground y level) para formar el escenario de la escena del nivel.

### **UI Canvas**

Ofrece una plantilla como punto de partida para la creación de elementos UI (User Interface).

### **Enemies**

Objeto que agrupa todos los enemigos en uno para tenerlos centralizados y fácilmente alcanzables e identificables. En la plantilla de Plataformer Microgame hay un tipo de enemigo implementado por defecto, que es el mostrado en la escena de muestra de la herramienta. Los enemigos pueden estar estáticos en un punto o seguir un patrón de movimiento.

### **Tokens**

Son los típicos coleccionables de los juegos. Estos tokens tienen dos scripts que se encargan de ellos: uno para manejar las animaciones y otro para la colisión y recolección del coleccionable por parte del jugador.

### **Zones**

Objeto encargado de agrupar todas las zonas de cierto interés del nivel. Un ejemplo de estas zonas serían las zonas de victoria y de muerte del nivel (si el jugador toca la zona de victoria ganará y si toca la de muerte morirá).

### **GameController**

Contiene los elementos necesarios para el funcionamiento del juego como elemento conjunto. Principalmente contiene una clase con datos que las clases del nivel utilizaran, una clase encargada de animar los coleccionables (tokens) y una clase encargada de mostrar u ocultar el menú de pausa del juego.

### **Player**

Objeto que representa el avatar que controlará el jugador.

### **Simulation**

Simulation es una clase encargada de manejar los eventos del juego. El objeto GameControler hace uso de esta clase para ir ejecutando los eventos a medida que entran en cola. Esta clase tiene una particularidad de C#. Simulaion es una “partial class”. Esto permite que la clase Simulation se construya en varios ficheros distintos. Para el funcionamiento de la clase Simulation, esta hace uso de otras dos subclases: Simulation.Event y Simulation.InstanceRegister. Se va a explicar a continuación porque son clases que se consideran importantes y claves para entender el funcionamiento de la arquitectura del videojuego.

Simulation: Este fichero contiene la estructura principal del funcionamiento de Simulation. Simulation es una clase estática con una cola, también estática, que guarda eventos (clase Event) y los libera cuando GameController llama al método tick(). Este fichero tiene el método tick() y los métodos necesarios para añadir y remover elementos de la cola.

Simulation.Event: Contiene la clase interna Event que se encarga de ejecutar el comando asociado a ese evento. De esta clase de la que heredan todos los eventos que saltan durante la ejecución del juego (como por ejemplo EnemyDeath, el evento que salta cuando el jugador muere). Los eventos se guardan en su mayoría en la carpeta Assets/Scripts/Gameplay.

Simulation.InstanceRegister: Contiene la clase InstanceRegister. Esta clase simplemente devuelve una instancia nueva de un objeto cualquiera. Esta clase esta creada para que Simulation pueda crear singletons (patrón de diseño) de clases. Es utilizado para que todas las clases trabajen sobre el mismo modelo. Ese modelo es un script denominado PlataformerModel con una clase que exclusivamente tiene una serie de atributos (como el Player, las cámaras o el punto de aparición del jugador) que serán utilizados por varias clases.

### **Pegas**

Hay una serie de pegas importantes que se han encontrado en la plantilla de Plataformer Microgame y que han sido importantes a la hora de elegir si utilizarla o no.

**AnimationController**

AnimationController es la clase que implementas las físicas y la animación de los objetos (en la escena solo se aplica a los enemigos).

Esta clase se encarga de animar y controlar las físicas de los enemigos. Se viola el principio de responsabilidad única, además con dos mecánicas muy distintas como son las físicas y las animaciones. Debería separarse en dos clases distintas, una para la animación y otra para las físicas. Esto es importante porque, actualmente, en caso de querer variar las físicas o las animaciones de un enemigo tienes que reescribir todo el método ComputeVelocity() modificando tanto físicas como animaciones. El script PlayerController adolece de los mismos problemas.

Adicionalmente delega a la animación el movimiento de todo el enemigo, lo cual obliga a las animaciones a encargarse del movimiento, l0 cual no es tarea suya.

**Health**

Health hereda de MonoBehaviour, pero no tiene necesidad de heredar de esta clase, ni heredar sus métodos y responsabilidades. La única función que sobrescribe de MonoBehaviour es Awake(), función que puede ser perfectamente sustituida por un constructor. Adicionalmente, Health no tiene un método para devolver la salud a un estado inicial o por defecto, haciendo que tanto a la hora de reestablecer la salud de un objeto como a la hora de restablecer la salud del Player cuando reaparece después de morir, se aplique el método Increment(), método que no corresponde a esa acción.

**SpawnPoint**

SpawnPoint es un objeto de la escena supuestamente creado para determinar el punto de aparición del jugador, sin embargo esto solo se aplica cuando el jugador muere, de manera que inicia el juego en una posición y reaparece en otra. No sé si es una pega o no, pero es digno de mención.

**JumpState**

La clase PlayerController maneja los estados de salto mediante una enumeración, manejándolos mediante un switch. Esto viola el principio Open/Close y centraliza toda las operaciones correspondientes a los estado en PlayerController agrandando la clase.

En lo relativo a la acción de salto el atributo de deceleración (jumpDeceleration) del salto solo se aplica si no se mantiene el botón de salto pulsado hasta el fin de la acción de salto. Esto hace que el salto corto aplique la deceleración pero el salto largo no, de manera que la misma acción puede desenvolverse de dos formas distintas.

**Rigidbody2D.Cast**

El Player utiliza el método Cast() de su Rigidbody2D para detectar los elementos que tiene a su alrededor y actuar en consecuencia. Esto provoca que todas las superficies con las que choca sean tratadas iguales, ya sean paredes o suelo, lo que desemboca en que el jugador al saltar mientras está al lado de una pared “colisione” con ella y cancele el salto a mitad de la acción. Adicionalmente puede ser un inconveniente utilizar este método a la hora de añadir mecánicas como trepar por las paredes o deslizarse por el suelo.

**PatrolPath.Mover**

Los métodos establecen la forma de obtener la posición que ocupa el objeto en el momento, pero no hay límites explícitos que permitan saber por ejemplo si se ha terminado de ejecutar el movimiento o no. Esto no supone un problema debido a la implementación del código, pero, personalmente, sería preferible establecer unos límites convirtiendo la clase en algo similar a un iterador, que en cada paso calcule la siguiente posición del objeto.

### **Conclusiones**

La clase Simulation es la base del funcionamiento del juego y los eventos la forma de interactuar con esta clase. Los eventos son clases heredadas de la clase abstracta Event. De esta forma se consigue una forma sencilla de crear eventos, los cuales son la clave para interactuar con el mundo. Esto provoca que las demás clases solo tengan que determinar la situación y determinar cuándo lanzar los eventos. Se nota claramente en la separación en carpetas, ya que la carpeta Assets/Scripts/Gameplay está formada enteramente por eventos, mientras que la carpeta Assets/Scripts/Mechanics está formada de clases que determinan cuando lanzar eventos (entre otras responsabilidades de las que se pueden encargar algunas clases).

En líneas generales, salvo la clase Simulation, cuya implementación me parece correcta y muy útil, el resto de los elementos deberían ser restructurados para adecuarse al modelo que se desea implementar. Sin embargo gracias a la utilidad de la clase Simulatión y todos los elementos visuales y de interfaz que ofrece por defecto la plantilla se ha tomado la decisión de desarrollar el videojuego partiendo de la plantilla Plataformer Microgame, eso sí, cambiando mucho la estructura de clases.

Algunos de los cambios que habría que realizar sobre las clase que ofrece la plantilla de Plataformer Microgame serían:

En el proyecto se delega a los propios sujetos (jugador y enemigos) la labor de simular las físicas que les afectan. Personalmente me parece más conveniente crear una clase a que se encargue de la simulación de las físicas y que los objetos jugador y enemigos sean los que le consulten como afectan las físicas.

Health no debe heredar de MonoBehaviour. Adicionalmente Health debe añadir un método para devolver el estado de la salud a un estado inicial o por defecto.

No hay una clase o un script que inicialice el estado del juego, sino que confía en el estado de la escena al ejecutarla, lo cual no me agrada, ya que si quieres añadir cosas al inicio de la ejecución de la escena, como por ejemplo una animación de aparición puede dificultar la labor o segregarlas en distintas clases (haciendo cada clase una serie de operaciones en el método Awake() de la clase y obligando a esas clases a heredar de MonoBehaviour). Añadir una clase que haga esta labor de inicialización no pude empeorar la situación, solo mejorarla.

Hacer una estructura de clases adecuadas a los estados del Player y los comportamientos asociados a estos, aplicando el patrón de diseño Estado.

Cambiar el nombre de algunas clases cuyo nombre resulta confuso. Estas clases son:

• HeathIsZero a PlayerHealthIsZero. Este script solo se aplica al jugador no a todos los elementos cuya salud llega a cero.

• AnimationController a EnemyAnimationController. Este script solo se aplica sobre los enemigos y no sobre cualquier objeto.

• PlayerSpawn a PlayerSpawnAfterDeath. Este script solo se lanza cuando el jugador muere y ha de reaparecer en la escena y no cada vez que el Player aparece en la escena. El script podría conservar su nombre si se aplicase el evento PlayerSpawn también durante la aparición del Player.

# **Conclusiones y líneas de trabajo futuras**

# **Bibliografía**

1. [Unity](#_Unity). [En line] [Ultimo acceso: 23/03/2021] <https://es.wikipedia.org/wiki/Unity_(motor_de_videojuego)>
2. [MonoBehaviour](#_MonoBehaviour(2)). [En línea] [Ultimo acceso: 23/03/2021] <https://docs.unity3d.com/ScriptReference/MonoBehaviour.html>
3. [GameObject](#_GameObject(3)). [En línea] [Ultimo acceso: 23/03/2021] <https://docs.unity3d.com/ScriptReference/GameObject.html>
4. [Braid](#_Juegos_similares_en). [En línea] [Ultimo acceso: 22/03/2021] [https://web.archive.org/web/20110223023113/http://braid-game.com/](https://web.archive.org/web/20110223023113/http:/braid-game.com/)
5. [Plataformer Microgame](#_Evaluación_de_Plataformer). [En línea] [Ultimo acceso: 22/03/2021] <https://assetstore.unity.com/packages/templates/platformer-microgame-151055?_ga=2.85524954.2125376078.1616431073-502923268.1612351479>

1. <https://assetstore.unity.com/packages/templates/platformer-microgame-151055?_ga=2.85524954.2125376078.1616431073-502923268.1612351479> [↑](#footnote-ref-1)