Escuela Politécnica Superior de Jaén

Grado en Ingeniería Informática

Universidad de Jaén

Trabajo Fin de Grado

DESARROLLO DE UN PROTOTIPO DE VIDEOJUEGO DE PLATAFORMAS CON GENERACIÓN ALEATORIA DE NIVELES

Alumno: Garrido Martínez, Sergio

Tutor: Prof. D. Jiménez Pérez, Juan Roberto

Dpto: Informática

Junio, 2018

Contenido

1. Introducción

1.1. Definición del videojuego

El objetivo del proyecto es desarrollar un prototipo de videojuego del género plataformas en tres dimensiones usando el motor Unity5. La principal característica del videojuego es que los niveles que el jugador debe superar se generarán aleatoriamente de manera que cada vez que se inicie una nueva partida, el desafío sea diferente.

Para ello, cada nivel estará compuesto por un determinado número de salas, cada sala con un desafío que el jugador deberá superar. Dependiendo del tiempo que el jugador emplee en completar cada sala, recibirá una determinada cantidad de puntos. A medida que el jugador consiga puntos, completará niveles y la dificultad del juego aumentará en consecuencia.

Se entiende por género de plataformas aquel en el que las principales mecánicas para completar los objetivos están relacionadas con correr, saltar (generalmente entre plataformas) y esquivar peligros a la vez que se recogen objetos que determinan la eficacia del desempeño del jugador.

1.2. Ejemplos de juegos similares y referentes

Para el desarrollo del prototipo, se han tenido en cuenta dos de títulos que han servido de referencia e inspiración:

- Super Mario 64: Lanzado para Nintendo 64 en 1996, este título es un referente dentro del género de plataformas en tres dimensiones por ser uno de los videojuegos que definió los estándares de dicho género. Es un videojuego considerado como revolucionario para su época y que sentó muchas de las bases de la actual industria del videojuego. Se ha tomado como inspiración para el desarrollo del proyecto en lo referente al género de las plataformas, en los desafíos propuestos y en la apariencia gráfica.
- The Binding of Isaac: Juego independiente lanzado para múltiples plataformas en 2011. La característica más innovadora que lo convirtió en un videojuego de éxito es que los niveles y desafíos que componen el mismo son generados aleatoriamente. La idea de que el jugador enfrente diferentes desafíos cada vez que comienza una nueva partida, ha sido uno de los principales objetivos a conseguir en el prototipo desarrollado.

1.3. Estructura del documento

A continuación se describe el contenido de las diferentes secciones que forman la documentación.

En la sección 2, se describe a grandes rasgos y de manera resumida los requisitos que se esperan cumplir del videojuego.

En la sección 3 se describe la metodología de trabajo y planificación que se han seguido además de los costes asociados al proyecto si éste fuera un producto preparado para lanzarse a la venta.

En la sección 4 se describe el proceso de desarrollo del videojuego.

En la sección 5 se describen los medios empleados durante el desarrollo

La sección 6 es un manual de usuario en el que se explican como jugar al videojuego.

La sección 7 recoge las conclusiones del proyecto así como un conjunto de directrices sobre que mejoras o implementaciones se podrían hacer en un futuro.

En el apartado 8 se encuentra la bibliografía utilizada para la realización del proyecto.

2. Visión general

El prototipo es un videojuego del genero de plataformas en tres dimensiones, en el que le jugador deberá progresar a lo largo de diferentes niveles obteniendo puntos y evitando obstáculos.

Los niveles estarán compuestos por salas que propondrán diferentes desafíos al jugador. En estas salas el jugador encontrará peligros y mecanismos que deberá usar o evitar para llegar al final de las mismas.

Estas salas se irán instanciando aleatoriamente a medida que el jugador las complete. Habrá diferentes tipos de salas y dependiendo de ello, el jugador obtendrá puntos de una forma u otra. Generalmente, conseguirá puntos si completa las salas en menos de un tiempo límite; en otros casos si consigue terminar la sala sin ser herido o como recompensa por llegar a lugares difíciles de alcanzar.

El jugador contará con puntos de salud que se irán agotando a medida que reciba daños. Cuando pierda la totalidad de su salud, la partida terminará. Sin embargo, también podrá restablecer su salud cuando encuentre objetos curativos con forma de corazón en ciertas salas especiales.

El juego contará con dos modos:

- Modo normal, en el que el objetivo será completar los niveles. Cada vez que el jugador consiga 10 puntos, completará un nivel y pasará al siguiente. Los desafíos que deberá enfrentar serán mayores cuantos más niveles supere. Al ser un prototipo, el videojuego constará de un total de tres niveles.
- Modo desafío, en el que el objetivo será conseguir la puntuación más alta posible a través de un nivel de duración indeterminada que terminara cuando el jugador pierda la totalidad de su salud.

3. Planificación

3.1. Metodología

Inicialmente antes de comenzar el desarrollo se elaboró un documento que recogía y describía los principales requisitos del proyecto. Dicho documento describe las mecánicas del personaje principal, de la cámara, interfaz, progresión y objetivos del videojuego.

Para el desarrollo del proyecto se ha escogido una metodología de trabajo ágil (E.M 2010) en la que el desarrollo se ha dividido en iteraciones. Cada iteración ha tenido una duración de 4 semanas, con una promedio de 15 horas de trabajo por semana.

Al comienzo de cada iteración, se han escogido los requisitos a implementar durante la misma (empezando por los fundamentales). La metodología de trabajo se ha basado principalmente en diseñar las funcionalidades e implementarlas posteriormente en prototipos que se han probado de manera aislada y posteriormente se han añadido al proyecto.

3.2. Costes

En este apartado se tratarán los costes asociados al desarrollo del proyecto en el hipotético caso de que fuera un producto ideado para ser lanzado a la venta.

En principio, habría que incluir los gastos asociados a las licencias de software utilizadas. El software empleado para el desarrollo del proyecto está detallado en el apartado 5.

Por una parte, hay que tener en cuenta que Unity ofrece dos licencias diferentes, Unity Free y Unity Pro. La primera es gratuita y es la que se usará siempre y cuando los ingresos brutos anuales obtenidos por la empresa desarrolladora no superen los 100.000 USD.

En otro caso, habrá pagar la licencia de Unity Pro cuyo coste es de 125$ mensuales.

Los softwares de edición de audio (Audacity) y modelado 3D (Blender) son distribuidos bajo licencia libre y no hay que pagar nada por su uso comercial.

Además, suponiendo un sueldo base de 8€ por hora y ciñéndose a la planificación descrita en el apartado anterior se han invertido en el desarrollo un total de 300 horas, lo que daría un sueldo de 2400€

Se supone que para el desarrollo del proyecto, no ha sido necesario invertir en hardware puesto que se han usado equipos ya existentes, por lo que tampoco hay un coste asociado a los mismos.

4. Desarrollo del videojuego

A continuación se describe el desarrollo del proyecto en cada una de sus iteraciones. Para cada iteración, se definen los requisitos a implementar su diseño y las partes más destacables de la implementación.

4.1. Primera iteración

En la primera iteración se ha implementado el control del personaje principal y de la cámara, dos características esenciales que definirán el resto del desarrollo del proyecto ya que influirán en todas las mecánicas jugables posteriores.

4.1.1. Requisitos

Personaje principal

El personaje principal será un avatar con forma humanoide, que podrá realizar las siguientes acciones:

- Con W o Arriba, caminará hacia delante.
- Con S o Abajo, caminará hacia atrás.
- Con A o Izquierda, caminará hacia la izquierda.
- Con D o Derecha, caminará hacia la derecha.
- El jugador caminará en diagonal combinando de las anteriores teclas siempre que ambas no sean opuestas (W y S por ejemplo).
- Saltar.
- Recibir daño.

Las acciones de saltar y recibir daño tendrán su correspondiente sonido asociado.

El personaje cuenta con un medidor de salud con un máximo de 5 puntos que indica la cantidad de daño que puede recibir. Cada vez que recibe daño, el jugador volverá al último punto de control (puntos seguros del nivel) con un punto de salud menos. Cuando el indicador de salud llega a 0, la partida termina. Sin embargo, todo lo relativo al sistema de salud y los puntos de control se diseñará e implementará en iteraciones posteriores.

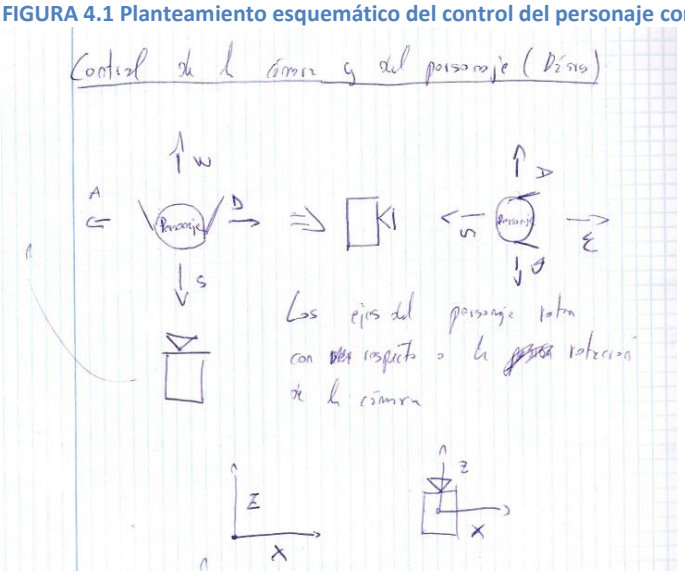
Cámara

Se utilizará la cámara que proporciona Unity5 por defecto para proyectos en tres dimensiones en tercera persona (Unity s.f.). Sus características serán:

- Tercera persona.
- El movimiento del personaje será relativo a la posición de la cámara. Esto quiere decir que cuándo el personaje avance hacia delante, lo hará siempre en la dirección a la que esté mirando la cámara.

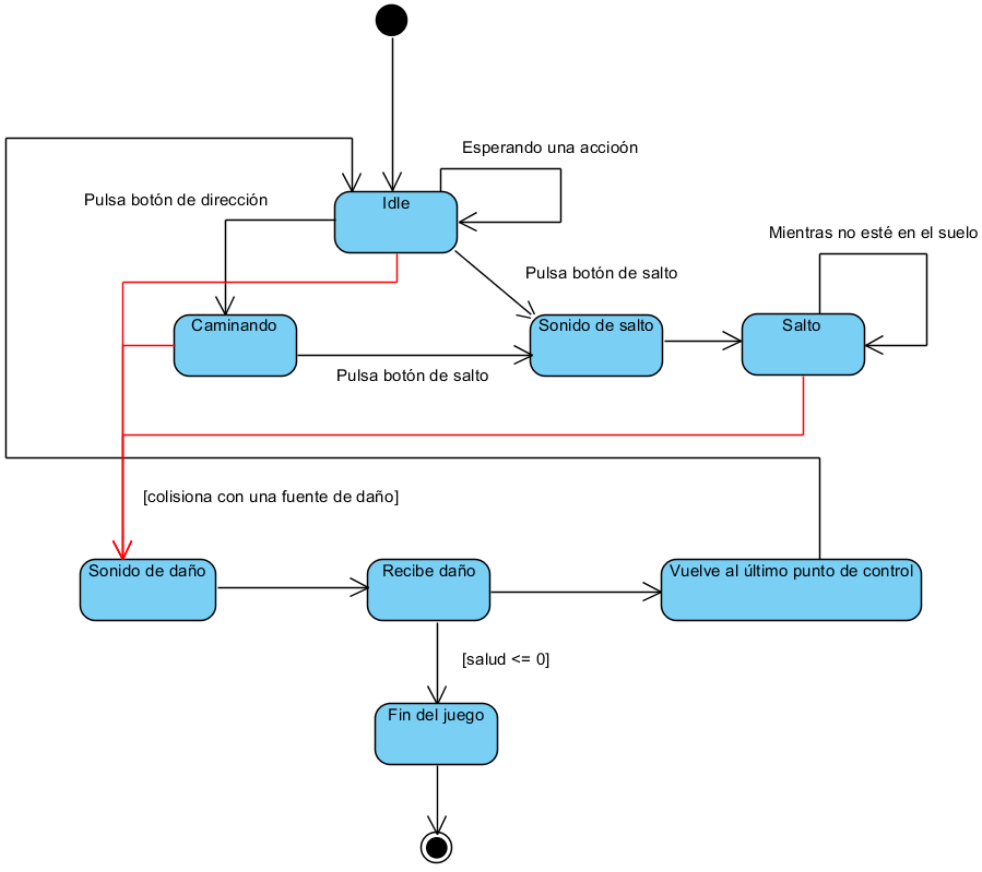
- La cámara podrá rotar libremente entorno a los ejes Z e Y del personaje aunque se definirán ángulos límite en el caso del eje Z.

FIGURA 4.1 Planteamiento esquemático del control del personaje con respecto a la cámara.



La FIGURA 4.1 es un esquema a papel sobre como el posicionamiento de la cámara afecta al movimiento del personaje. Como ya se ha explicado, los controles del personaje son relativos a la posición de la cámara con respecto a éste.

FIGURA 4.2 Diagrama de máquina de estados del personaje



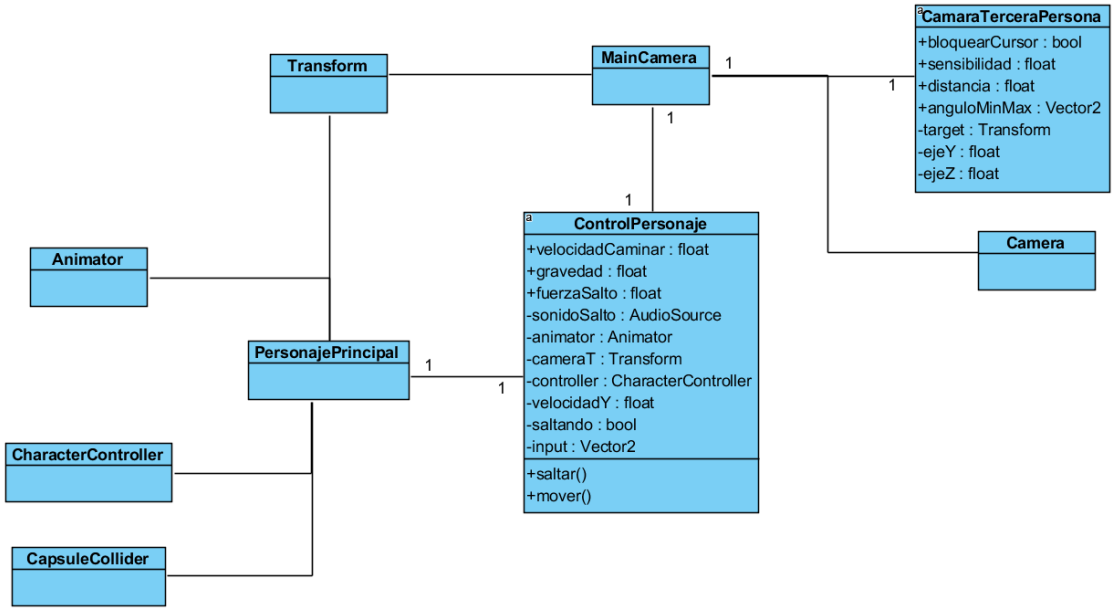
En la FIGURA 4.2 muestra el diagrama de máquina de estados del personaje. En esta iteración son especialmente interesantes los nodos que indican el estado físico del personaje, es decir, si se encuentra parado, andando o saltando.

El estado Idle se refiere al estado en el que el personaje está parado sin realizar ninguna acción. Efectuar alguno de los movimientos de dirección provocará que éste pase a su estado de Caminando. El botón de salto hará que el personaje pase a su estado de Salto. Se considerará que el personaje estará en dicho estado mientras esté suspendido en el aire (no esté tocando el suelo). Si el personaje deja de accionar las teclas de movimiento o deja de estar en el aire, volverá al estado inicial Idle.

Los caminos de color rojo en el diagrama son aquellos que llevan al estado de Daño. Sin embargo, el resto de nodos son referentes al sistema de salud y puntos de control que se implementarán en iteraciones posteriores, por lo que ahora no es necesario tenerlos en cuenta ahora mismo.

4.1.2. Diseño

FIGURA 4.3 Diagrama de clases del control del personaje y la cámara



La FIGURA 4.3 muestra el diagrama con las clases implicadas en el control de la cámara y el personaje principal.

Las clases PersonajePrincipal y MainCamera son los GameObject que aparecerán en la escena mientras que el resto de clases son Components que definen el comportamiento de ambos objetos.

Transform es un Component presente en todos los GameObjects de Unity y define las propiedades relativas a la posición, rotación y escalado del objeto. El personaje principal debe conocer el Transform de la cámara puesto que su movimiento es relativo a ésta.

Animator es un Component que se encarga de las animaciones del personaje principal.

CharacterController es un Component que le indica a Unity que el GameObject es el personaje principal y que sobre éste deberán aplicar físicas y colisiones.

CapsuleCollider establece mediante una cápsula el volumen sobre el que se calcularán las colisiones sobre el PersonajePrincipal.

Camera es el Component que sirve para definir cámaras en Unity.

ControlPersonaje y CámaraTercera persona son Components de tipo script. A continuación se indica su utilidad.

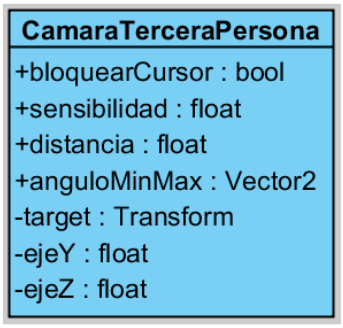
FIGURA 4.4 Clase ControlPersonaje UML

La FIGURA 4.4 muestra el diseño de la clase ControlPersonaje que se encarga del control del mismo.

Descripción de los atributos

- VelocidadCaminar: Velocidad a la que se desplaza el personaje. Un valor alto hará que se mueva a mayor velocidad.
- Gravedad: Determina la fuerza de gravedad que actúa sobre el personaje. Un valor alto hará que caiga antes al suelo si se encuentra en el aire.
- FuerzaSalto: La fuerza (altura) del salto.
- SonidoSalto: Sonido que se reproducirá cuando el personaje realice la acción de saltar.
- Animator (Unity s.f.): Componente usado para controlar las animaciones del personaje.
- CameraT: Referencia al transform de la cámara principal.
- Controller (Unity s.f.): Componente de Unity usado para realizar los movimientos del personaje.
- Saltando: Variable booleana que indica si el jugador ha saltado.
- Input: Registra si el jugador ha pulsado alguna de las teclas direccionales cada frame.

FIGURA 4.5 Clase CamaraTerceraPersona UML



La FIGURA 4.5 muestra el diseño de la clase CamaraTerceraPersona que se encarga del control de la cámara.

Atributos:

- BloquearCursor: Cuando el valor de la variable es true, el cursor se bloquea y desaparece de la pantalla.
- Sensibilidad: Velocidad a la que se mueve la cámara.
- Distancia: Distancia de la cámara al jugador.
- AnguloMinMax: Valores mínimos y máximos para la rotación de la cámara en el eje Z.
- Target: Punto de referencia al que enfoca la cámara. Durante el desarrollo de los prototipos se comprobó que la cámara era más cómoda si en vez de enfocar al personaje directamente, ésta enfocaba a un GameObject vacío Target situado dentro de la jerarquía del personaje principal a la altura de su cabeza.
- EyeZ y EjeY: Almacenan las rotaciones en el eje 'z' y en el eje 'y' de la cámara.

4.1.3. Detalles de la implementación

Personaje Principal

El control del personaje está implementado en el script *Assets/PersonajePrincipal/Scripts/ControlPersonaje.cs*

Cuando el jugador introduce uno de los inputs de movimiento (WASD), el resultado de dichos inputs se almacena en un Vector2 y se normaliza.

En cada Update() (Unity s.f.), se calcula el ángulo que deberá rotar el personaje antes de empezar a caminar (p.e. si camina hacia atrás, deberá rotar 180 grados). A éste ángulo se le suma la rotación de la cámara, con lo cual permite que el movimiento sea relativo a la misma. El resultado de la suma de los dos ángulos se le es asignado al transform del personaje.

FIGURA 4.6 Input del control del personaje

```
input = new Vector2 (Input.GetAxisRaw ("Horizontal"), Input.GetAxisRaw ("Vertical"));
Vector2 inputDir = input.normalized;
if (inputDir != Vector2.zero) {
    transform.eulerAngles = Vector3.up * (Mathf.Atan2 (inputDir.x, inputDir.y) * Mathf.Rad2Deg + cameraT.localEulerAngles.y);
}
```

En la imagen FIGURA 4.6 se muestra el código descrito. GetAxisRaw devuelve 1 si el input indicado ha sido pulsado y 0 en caso contrario. Si el jugador presiona W (avanzar hacia delante), el vector input valdrá (0, 1) y el jugador no tendrá que rotar sobre sí mismo antes de avanzar. Posteriormente se le sumará la rotación de la cámara con CameraT.localEuerAngles.y.

Una vez aplicada la rotación, se calcula el Vector3 velocity que desplaza al personaje en la dirección deseada:

- Calculando la velocidad de movimiento speed = velocidadCaminar * inputDir.magnitude.
  VelocidadCaminar es una variable pública que define la velocidad de movimiento del personaje. inputDir.magnitude será 1 en caso de que el jugador haya pulsado alguna tecla de movimiento y 0 en cualquier otro caso. En otras palabras, si el jugador no pulsa nada, speed será 0.
- Finalmente, speed se multiplica por el vector forward del personaje, lo que hará que avance en la dirección a la que está mirando una vez se ha rotado (su vector forward local).
- Además también habrá que calcular la velocidad en la dirección Y para aplicar gravedad en caso de que el personaje haya saltado. Dicha velocidad se multiplica por el vector up del personaje.

El vector velocity será entonces la suma de speed + velocidadY.

FIGURA 4.7 Cálculo del salto

```
float speed = velocidadCaminar * inputDir.magnitude;

velocidadY += Time.deltaTime * gravedad;

Vector3 velocity = (transform.forward * speed) + (Vector3.up * velocidadY);
controller.Move (velocity * Time.deltaTime);
```

En la FIGURA 4.7 se muestra el código relativo a lo anterior descrito.

Para el salto, están las variables velocidadY, gravedad y fuerzaSalto.

Al presionar espacio, se invoca al método saltar() que modifica e incrementa la velocidadY haciendo que el personaje salte al sumar dicho valor al vector velocity. VelocidadY va decreciendo en función de la gravedad hasta que el personaje llega al suelo, momento en el que vuelve a valer 0.

Para las animaciones, se usa la herramienta Mecanim de Unity (Unity s.f.) que permite crear máquinas de estados gráficas para controlar las animaciones desde los scripts.

FIGURA 4.8 Máquina de estados de las animaciones del personaje principal en Mecanim

Como se muestra en la FIGURA 4.8, el estado IdleCaminar es un Blend Tree (Unity s.f.) que engloba las animaciones de Idle (parado) y caminando. Dependiendo de si el personaje está en movimiento o no, se reproduce una o la otra. Del estado IdleCaminar se puede pasar a Saltar cuando lo indica la variable booleana jumping del Animator. Dicha variable se modifica desde el script *ControlPersonaje.*

Tanto las animaciones como el modelado se han realizado usando el software Blender. Para las animaciones se ha empleado técnicas de animación de cinemática directa e inversa vistas en la asignatura Técnicas de animación y Postprocesamiento 3D (Beane 2012).

Cámara

El control de la cámara está implementado en el script *Assets/PersonajePrincipal/Scripts/CamaraTerceraPersona.cs.* Dicho script está asociado como componente a la cámara principal.

Se ha empleado la cámara que Unity proporciona por defecto para el desarrollo de proyectos en tres dimensiones. Para que la cámara orbite en torno al personaje principal, se ha creado el GameObject LookAt. Dicho GameObject, es un objeto vacío (no tiene ningún componente asociado a excepción del Transform que está presente en todos los objetos de Unity) que se encuentra dentro de la jerarquía del personaje principal, a la altura de la cabeza del mismo. El objetivo es que la cámara mire al LookAt en vez de al centro del personaje principal ya que esto hace que el control sea más cómodo.

El LookAt tiene un tag (Unity s.f.) con el mismo nombre asignado. Este tag servirá al script de la cámara para localizarlo cuando ésta se inicialice al cargar la escena.

FIGURA 4.9 Buscando el LookAt mediante su tag

```
target = GameObject.FindGameObjectWithTag ("LookAt").transform;
```

En la FIGURA 4.9 se muestra la línea de código que se ejecuta en el método Start() (Unity s.f.) del script que sirve para localizar el objeto y asignar sus coordenadas al target.

La cámara orbita en torno al personaje principal usando el ratón. Tanto la distancia de la cámara al personaje como la velocidad a la que ésta se mueve, son valores que el usuario puede determinar.

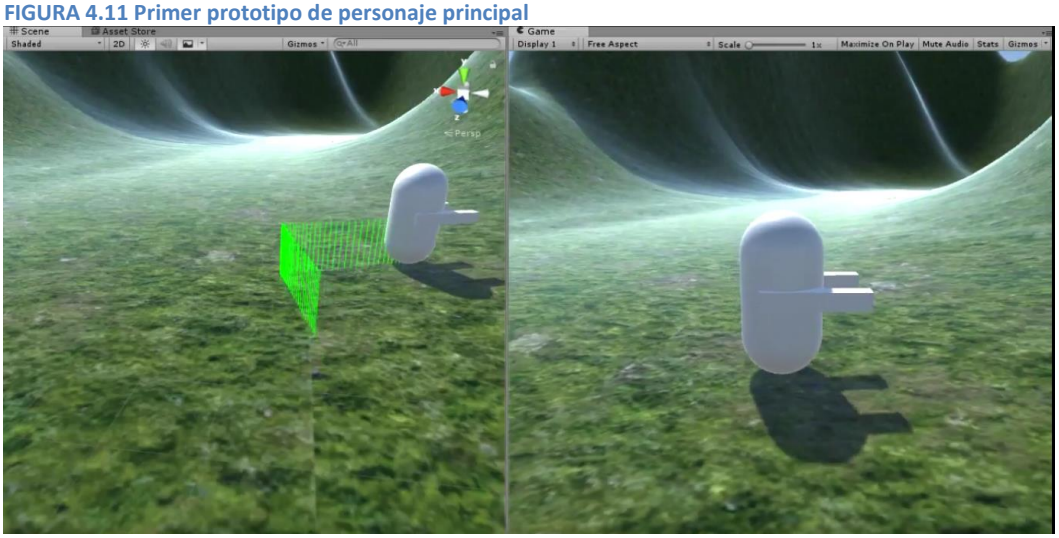
FIGURA 4.10 Control de la cámara

```
ejeY += Input.GetAxis ("Mouse X") * sensibilidad;
ejeZ -= Input.GetAxis ("Mouse Y") * sensibilidad;
ejeZ = Mathf.Clamp (ejeZ, anguloMinMax.x, anguloMinMax.y);

Vector3 rotacion = new Vector3 (ejeZ, ejeY);
transform.eulerAngles = rotacion;

transform.position = target.position - transform.forward * distancia;
```

El código de la FIGURA 4.10 se ejecuta en la función Update() del script.
En los atributos ejeY y ejeZ se almacena el resultado de los input del ratón.
La función Clamp (Unity s.f.) nos asegura que el valor de la rotación en el eje z estará entre el mínimo y el máximo definidos.
Finalmente se usan dichos valores para definir la nueva rotación de la cámara.
La última línea de código sirve para definir la distancia de la cámara al LookAt.

El desarrollo del control del personaje principal y de la cámara pasó por varios prototipos hasta llegar a la versión definitiva.

FIGURA 4.11 Primer prototipo de personaje principal



En la FIGURA 4.11 se muestra uno de los primeros prototipos. El avatar elegido como personaje está realizado con primitivas básicas de Unity ya que en este momento del desarrollo las animaciones y modelado del mismo no eran relevantes.

FIGURA 4.12 Segundo prototipo de personaje principal

La FIGURA 4.12 muestra un segundo prototipo en el que se mejoró el movimiento de la cámara y el personaje ajustando los parámetros de las distancias y velocidad de movimiento entre otras cosas. También se cambió el modelo por uno más similar al definitivo incluyendo las animaciones de caminar e idle (parado).

El modelo y animaciones definitivas se incluirán en la cuarta iteración.

4.2. Segunda iteración

El objetivo de la segunda iteración, ha sido diseñar e implementar un nivel de prueba en el que estén presentes algunas de las principales mecánicas descritas en el documento de diseño y que además sirva para probar el personaje principal.

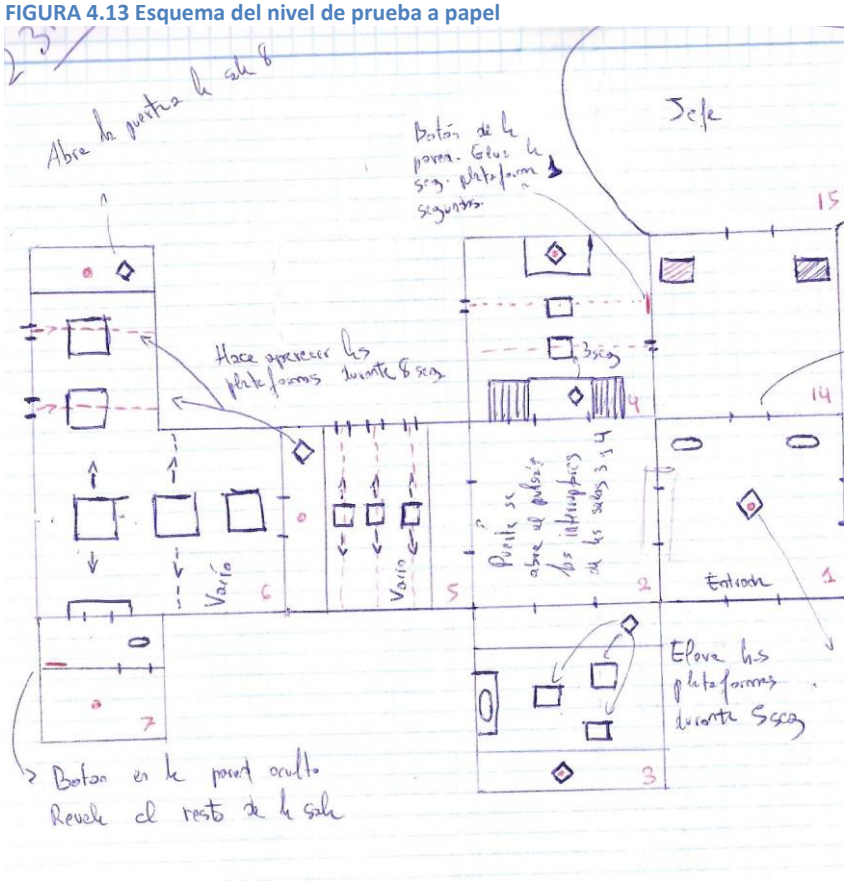
El nivel a construir no se generará de manera aleatoria sino que todo en él estará previamente definido.

4.2.1. Requisitos

La FIGURA 4.13 muestra el esquema a papel del nivel de prueba. En el esquema se muestra la distribución del nivel que se espera construir, con las diferentes salas, su ubicación y los objetivos a completar en cada una de ellas.

El nivel de prueba final consta de 6 salas en las cuales se implementan algunas de las principales mecánicas que se usarán posteriormente en el juego y que se describen a continuación.

FIGURA 4.13 Esquema del nivel de prueba a papel

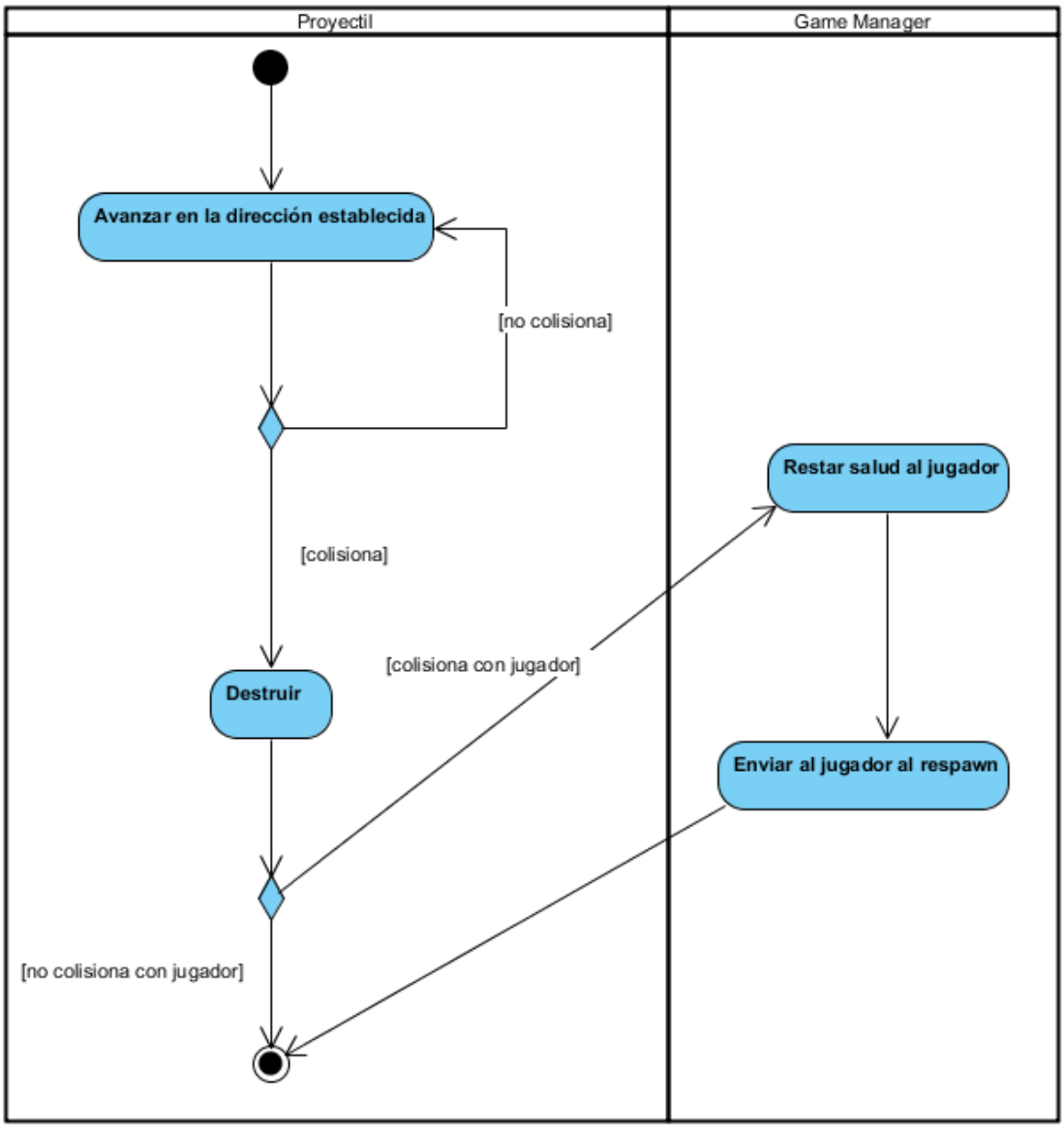


Mecánicas que a implementar en el nivel:

- Plataformas: soportes que ayudarán al jugador a avanzar a través del nivel. Las plataformas pueden ser de varios tipos.
  - Fijas: Permanecen estáticas en un único lugar.
  - Móviles: Se desplazan entre varios puntos del mapa a una determinada velocidad. Si el jugador se sube a una, se desplazará sobre ella al mismo tiempo.

- Disparadores/Cañones: disparan proyectiles cada cierto tiempo. Los proyectiles se mueven con una dirección y a una velocidad. Si impactan contra el jugador, éste perderá un punto de salud.
  Los proyectiles desaparecerán siempre que:
  - Colisionen contra el jugador.
  - Colisionen contra cualquier otro objeto en la escena.
  - Tras un determinado tiempo tras ser disparados.

FIGURA 4.14 Diagrama de actividades del proyectil.



La FIGURA 4.14 ilustra el funcionamiento de los cañones. Para ésta iteración, la parte relativa al GameManager (en iteraciones posteriores GameController) aún no será relevante. Esta parte de la funcionalidad será implementada en la cuarta iteración.

- Botones: Al pulsarlos, hacen aparecer otros mecanismos durante un breve periodo de tiempo. Por ejemplo, el jugador puede pulsar un botón que haga aparecer una plataforma que le permita superar la sala en la que se encuentra.

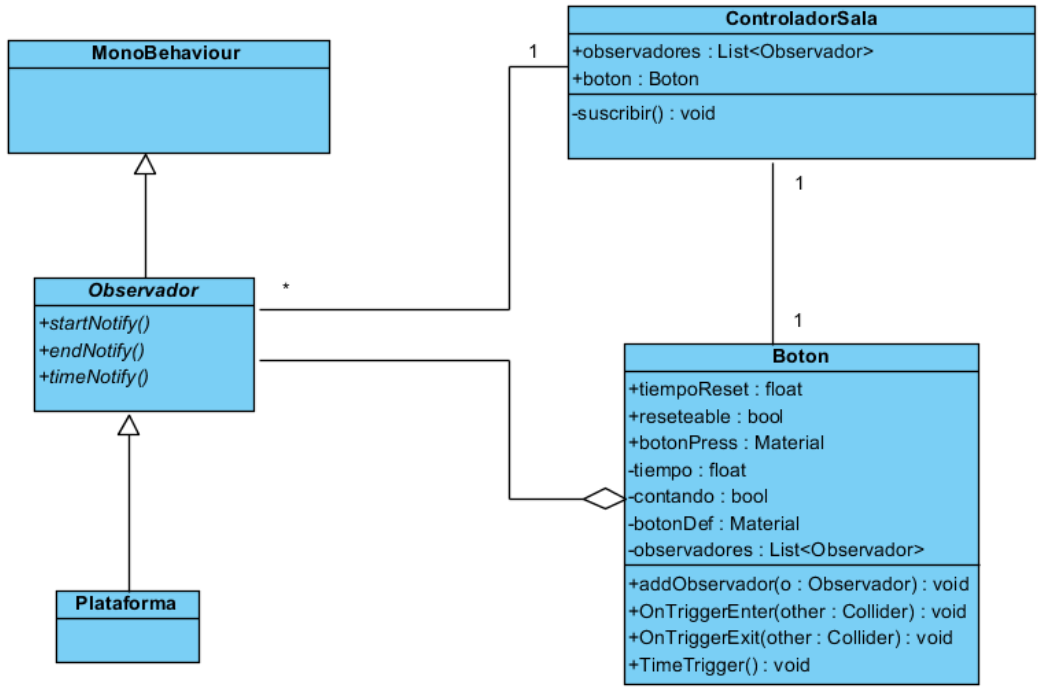
4.2.2. Diseño

Botones

Su función es la de hacer aparecer otros objetos en la escena durante un tiempo limitado cuando del jugador los pulsa. Para su implementación, se ha usado el patrón de diseño

Observer (Freeman 2004), de manera que los objetos 'activables' heredan de la clase abstracta Observador y se suscriben al botón que los notifica cuando sea necesario.

FIGURA 4.15 Diagrama UML Botón



En el diagrama de la FIGURA 4.15 se muestra el ejemplo de un botón que al pulsarlo haría aparecer un conjunto de plataformas. Al transcurrir un tiempo, las plataformas desaparecerían y el jugador tendría que volver a pulsar el botón.

El Controlador de la Sala se encarga de suscribir los observadores al botón.

Los métodos del Observador:

- startNotify(): Para notificar al observador de que el jugador ha pulsado el botón.
- endNotify(): Para notificar al observador de que el jugador ha dejado de pulsar el botón.
- timeNotify(): Para notificar al observador cuando haya transcurrido un determinado tiempo.

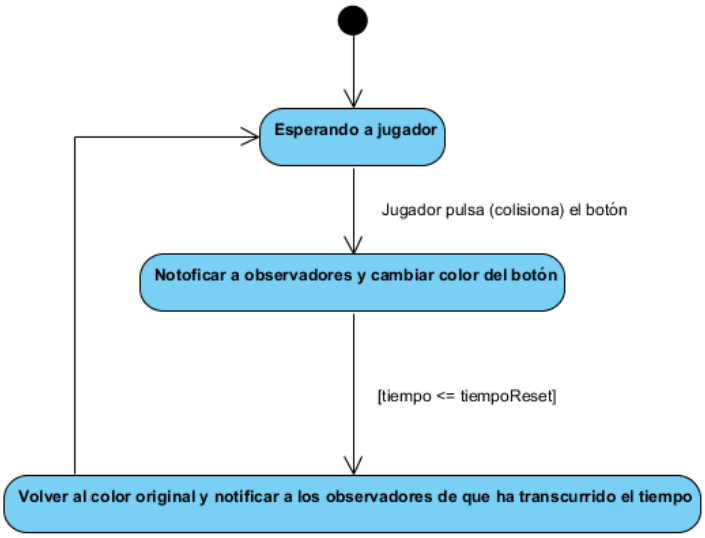
Los atributos del Botón:

- tiempoReset: tiempo que tarda el botón en volver a su estado original.
- Reseteable: indica que el botón es reseteable. Si no lo es, sólo se podrá pulsar una vez.
- botonPress: color del botón cuando está presionado.
- botonDef: color del botón cuando está sin presionar.
- Contando: indica si el botón ha iniciado la cuenta atrás antes de que se resetee.
- Observadores: Lista de observadores suscritos al botón.

Los métodos del botón:

- addObservador: añade un nuevo observador a la lista de suscriptores.
- OnTriggerEnter: Cuando el jugador colisiona con el botón, este se presiona y los observadores son notificados haciendo que aparezcan en la escena.
- OnTriggerExit: Cuando el jugador deja de estar sobre el botón, se notifica a todos los observadores.
- TimeTrigger: Se notifica a los observadores cuando transcurre el tiempoReset haciendo que los mismos desaparezcan de la escena.

FIGURA 4.16 Diagrama Actividades del botón

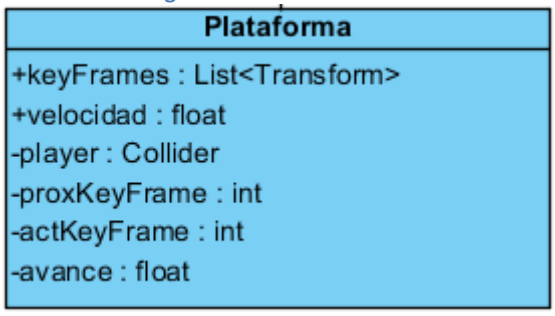


La FIGURA 4.16 muestra el funcionamiento de los botones. Mientas el jugador no interaccione con ellos, estos no harán nada. Cuando el jugador los accione, cambiarán de color y se notificarán a los elementos que estén suscritos al mismo. Finalmente, cuando transcurra cierto tiempo, el botón volverá a su estado inicial y se notificará nuevamente a los observadores.

Objetos móviles

Aclaración: la clase se llama Plataforma porque inicialmente se ideó para ser aplicada a plataformas que se desplazaran por la escena con un recorrido predefinido. Sin embargo, gracias a la arquitectura sistema entidad-componente de Unity, ésta se ha acabado aplicado a otros objetos a lo largo del desarrollo.

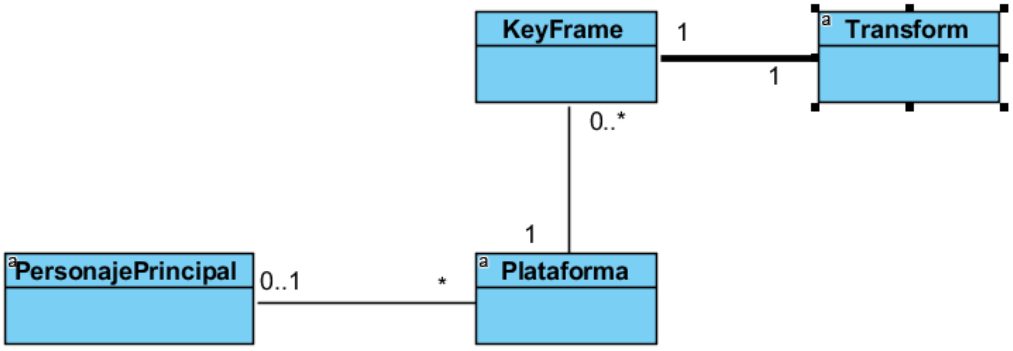
FIGURA 4.17 Diagrama UML Plataforma



Descripción de los atributos de la clase mostrada en la FIGURA 4.17:

- keyFrames: Los keyFrames definen la trayectoria del objeto a lo largo de su recorrido. Son objetos vacíos cuyo único componente es un Transform que indica la posición en la que se encuentran. Dichos Transforms se almacenan en una cola de manera que la plataforma se moverá del keyFrame i al i+1. Una vez llegue a la posición del último, ésta se desplazará de nuevo al primer keyFrame. En el apartado Detalles de la implementación se detalla en más profundidad su funcionamiento.
- Velocidad: velocidad a la que se mueve la plataforma entre los keyFrames.
- Player: inicialmente a null. Referencia al jugador en caso de que éste esté sobre la plataforma.
- proxKeyFrame y actKeyFrame: índices dentro de la cola que indican el punto los dos puntos entre los que se está interpolando.
- Avance: entre 0 y 1, la posición de la plataforma entre los dos keyFrames entre los que se está interpolando.

FIGURA 1.18 Diagrama de clases UML PersonajePrincipal y Plataforma.



La FIGURA 1.18 muestra como la plataforma puede tener de cero a muchos keyFrames. Si no tiene ninguno, entonces la plataforma será inmóvil. En otro caso, ésta se desplazara entre estos.

La plataforma podrá tener asociado al personaje principal si éste está sobre la misma como se indica en el diagrama.

Disparadores/Cañones

Los disparadores arrojan proyectiles en una determinada dirección cada cierto tiempo.

FIGURA 4.19 Diagrama de clases Disparador-Proyectil

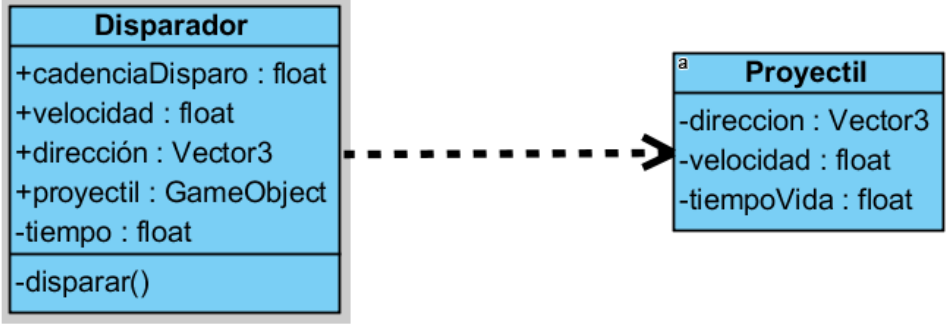


FIGURA 4.19 muestra el diagrama de clases UML de los disparadores y los proyectiles:

- La CadenciaDisparo indica el tiempo que transcurre entre disparo y disparo.
- La dirección es un vector que indica la trayectoria con la que saldrá el disparo a la velocidad indicada.
- Proyectil es una referencia al GameObject que se disparará.

Cada proyectil tiene su propia dirección, velocidad y tiempo de vida (tiempo tras el cual desaparecerá).

4.2.3. Detalles de la implementación

Objetos móviles

El funcionamiento de los ObjetosMoviles se encuentra en el script:
Assets/Mecanismos/Scripts/PlataformaMovil.cs

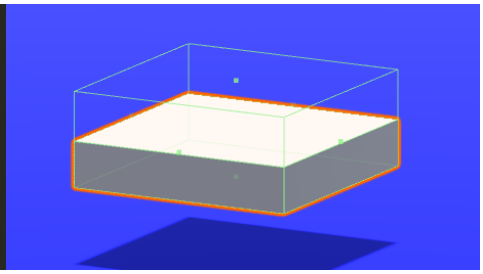
El movimiento de los objetos se realizará por interpolación lineal entre los keyframes que definen los puntos de su trayectoria. Para almacenar los keyFrames, se utiliza una cola de manera que cuando el objeto se encuentre en el último keyFrame, ésta irá desde el mismo hasta el primero de todos (de n a 0, siendo n el último keyFrame de la cola). Si el objeto tiene 1 o menos keyframes, no se desplazará y será fijo.

Además, el objeto contará con una velocidad de movimiento determinada.

Si el objeto es una plataforma y el jugador debe subirse a ella para acceder a otro lugar, es necesario que éste se mueva solidariamente con la plataforma. En otro caso, la plataforma avanzaría dejando atrás al jugador. Para ello, se usa un Collider (Unity s.f.) que le indica a la plataforma si el jugador está sobre la ella como se puede ver en la FIGURA 4.20.

FIGURA 4.20 Collider de la plataforma

```
void OnTriggerEnter(Collider other){
    if (other.tag == "Player") {
        player = other;
    }
}

void OnTriggerExit(Collider other){
    if (other.tag == "Player") {
        player = null;
    }
}
```



Si el jugador entra en el Collider, es asignado al atributo player. De este modo, la plataforma también podrá aplicar el movimiento al mismo.

FIGURA 4.21 Movimiento de la plataforma

```
void LateUpdate () {

    avance += Time.deltaTime * velocidad;
    Vector3 movimiento = Vector3.Lerp (keyFrames[actKeyFrame].position, keyFrames[proxKeyFrame].position, avance);

    if (player != null) {
        Vector3 desplazamiento = movimiento - transform.position;
        player.transform.position += desplazamiento;
    }

    transform.position = movimiento;

    if (transform.position == keyFrames [proxKeyFrame].position) {
        actKeyFrame = proxKeyFrame;
        proxKeyFrame = (proxKeyFrame + 1) % keyFrames.Count;
        avance = 0;
    }

}
```

Si la referencia al player no es nula (el jugador está sobre la plataforma), calculamos el desplazamiento que va a hacer la plataforma en ese Update() y se le suma al jugador tal y como se indica en el código de la FIGRUA 4.21.

Disparadores/Cañones

El funcionamiento de los disparadores/cañones se encuentra en el script *Assets/Trampas/Scripts/Disparador.cs*

FIGURA 4.22 Script del disparador

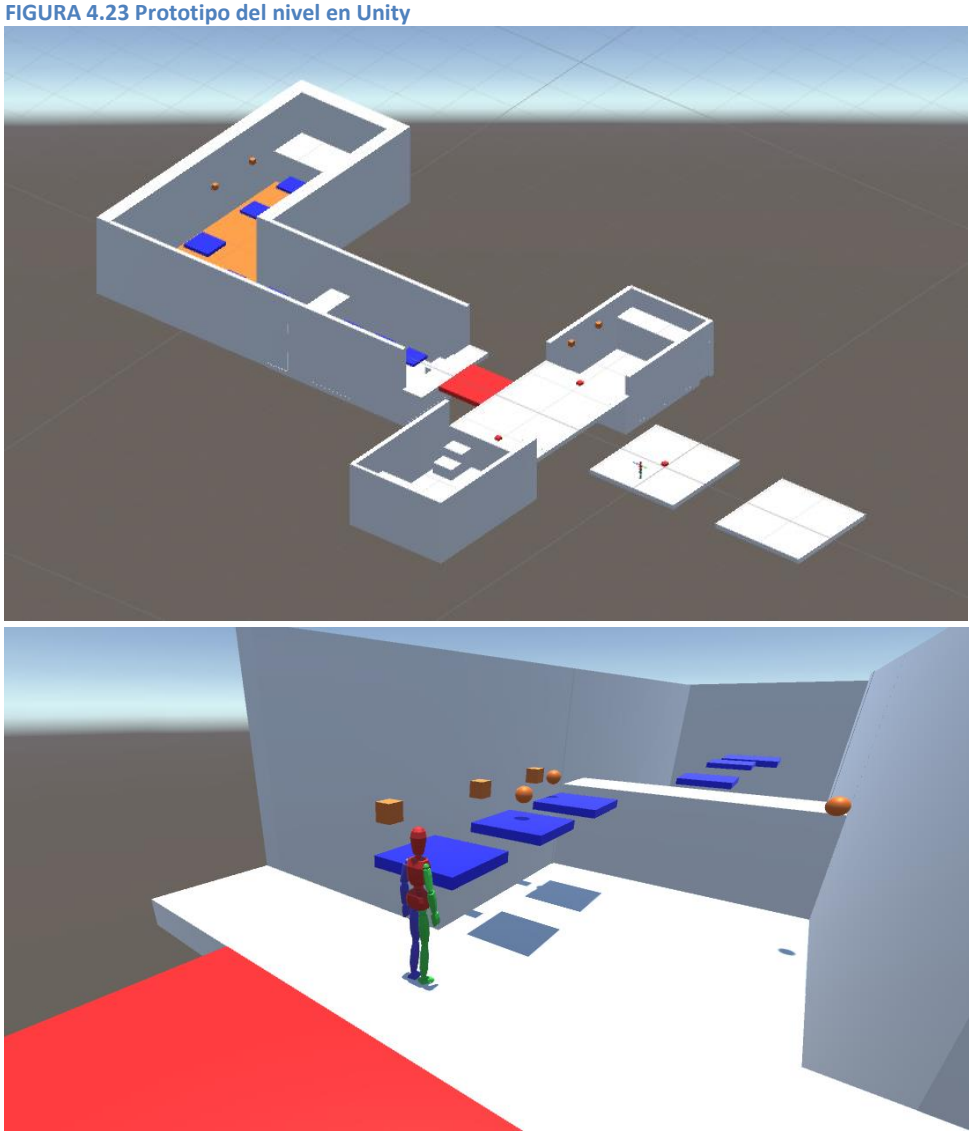
```
void Update () {
    tiempo += Time.deltaTime;
    if (tiempo >= cadenciaDisparo) {
        tiempo = 0.0f;
        disparar();
    }
}

private void disparar() {
    GameObject p = Instantiate(proyectil,
        this.GetComponent<Transform>().position,
        gameObject.transform.rotation);
    p.GetComponent < Proyectil > ().setDireccion (direccion);
    p.GetComponent < Proyectil > ().setVelocidad (velocidad);
}
```

En la función update se comprueba en cada frame si ha transcurrido el tiempo establecido por la cadencia de disparo. Cuando es así, se llama a disparar() y se crea una nueva instancia del proyectil asignándosele dirección y velocidad como se muestra en el código de la FIGURA 4.22.

Instantiate instancia en tiempo de ejecución un nuevo GameObject. En este caso, un GameObject de tipo proyectil.

Finalmente, una vez instanciado, se le asignan la dirección y velocidad con GetComponent <Proyectil> que permite acceder al componente Proyectil del objeto.

FIGURA 4.23 Prototipo del nivel en Unity



La FIGURA 4.23 muestra el nivel de prueba construido en Unity. Para el nivel se han utilizado privitivas básicas (cubos, esferas y cilindros) y texturas planas.

4.3. Tercera iteración

El objetivo de esta iteración ha sido implementar el sistema mediante el cual se puedan generar niveles aleatorios.

Además también se ha implementado el sistema de recompensas por tiempo, premiando con más puntos a jugadores más hábiles además de incluir objetos (monedas y corazones).

4.3.1. Requisitos

Recordamos que el juego consta de dos modalidades:

- Modo normal, en el que el objetivo será completar los niveles. Cada vez que el jugador consiga 10 puntos, completará un nivel y pasará al siguiente. Los desafíos que deberá enfrentar serán mayores cuanto más niveles supere. Al ser un prototipo, el videojuego constará de un total de tres niveles.
- Modo desafío, en el que el objetivo será conseguir la puntuación más alta posible a través de un nivel que sólo terminara cuando el jugador pierda.

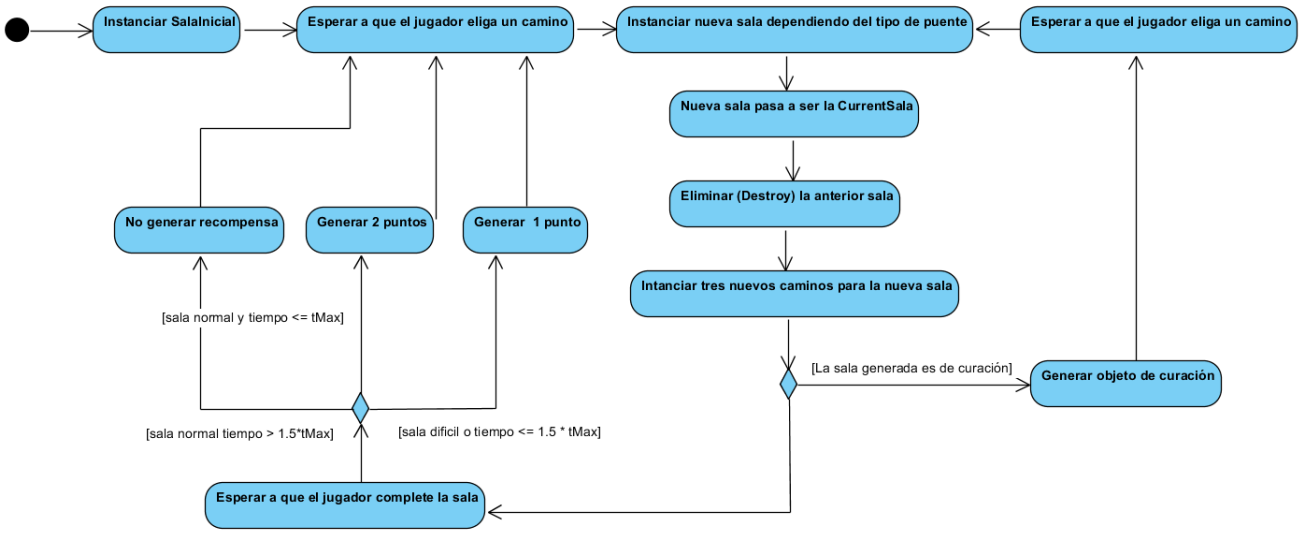
En cualquier caso, al seleccionar uno de los dos modos y comenzar una nueva partida, no existe un nivel predefinido, sino que éste se va generando a medida que el jugador avanza por el mismo. Para ello, existen una serie de salas predefinidas que se instanciarán a medida que el jugador las va completando.

Al final de cada sala, aparecen tres caminos o puentes. El jugador deberá elegir uno de ellos y aleatoriamente se instanciará la siguiente sala que deberá completar. Una vez se genera una nueva sala, la anterior desaparecerá.

Hay tres tipos de caminos:

- Azul, que conduce a una sala de dificultad normal. Estas salas tienen un tiempo máximo para completarlas. Si el jugador lo consigue, obtendrá 2 puntos. Si logra hacerlo en un tiempo menor al tiempo máximo x 1.5, obtendrá 1 punto. En cualquier otro caso, no obtendrá puntuación.
- Rojo, que lleva a una sala de mayor dificultad. Esta sala no tendrá tiempo límite y el jugador siempre conseguirá un punto al completarla. Además, habrá puntos extra repartidos a lo largo de la sala, pero si el jugador recibe daño, estos desaparecerán.
- Verde, que lleva a una sala en la que el jugador podrá reestablecer su salud con un objeto de curación. La probabilidad de que aparezca un camino a una sala verde será mayor cuanto más tiempo haya pasado desde que apareció el último camino verde. Es decir, cuantas más salas complete el jugador sin que aparezca un camino verde, más probable será que en la siguiente sala aparezca uno.

FIGURA 4.24 Diagrama de actividades del sistema de generación del nivel



Los objetos a implementar son:

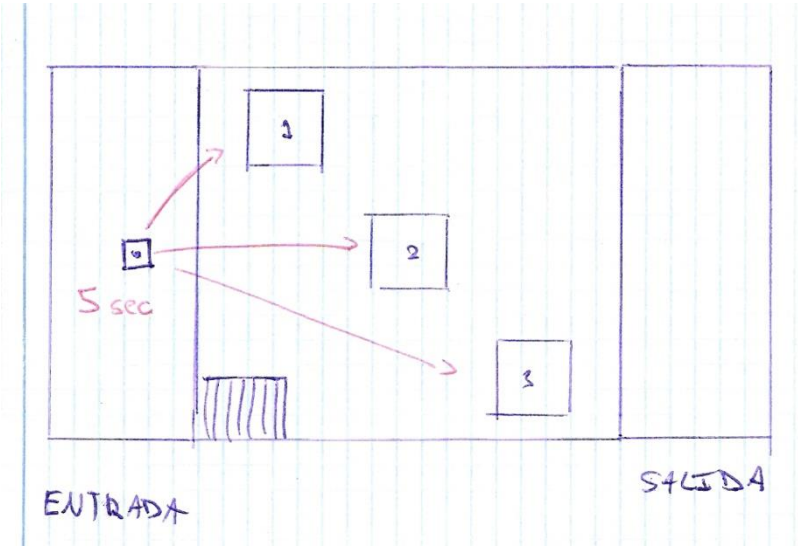
- Monedas/Puntos: Son las recompensas por completar una sala. Se requiere un determinado número de puntos para superar un nivel y avanzar al siguiente. Hay dos tipos de monedas.
  - Monedas doradas: equivalen a 1 punto. Se podrán obtener de tres maneras:
    (1) Como recompensa tras completar una sala normal en un tiempo inferior al 1,5 x tiempo límite.
    (2) Siempre como recompensa tras completar una sala difícil.
    (3) En las salas difíciles habrá monedas doradas extra. Estas desaparecerán si el jugador recibe daño durante el transcurso de dicha sala.
  - Monedas verdes: equivalen a 2 puntos. Se obtienen únicamente como recompensa tras completar una sala normal en un tiempo inferior al tiempo límite.
- Corazones: Restablecen una cantidad de salud del jugador. Únicamente aparecerán en salas de curación. Hay de tres tipos.
  - Medio corazón: restablece 1 punto de salud.
  - Corazón: restablece 2 puntos de salud.
  - Doble corazón: restablece 5 puntos de salud.

Diseño de las primeras salas

Reutilizando el nivel de prueba de la anterior iteración, se han diseñado las 4 primeras salas que servirán como base para implementar el sistema que genere niveles aleatorios.

A continuación se describe el funcionamiento esperado de cada sala.

FIGURA 4.25 Sala 1



El jugador deberá pulsar el BOTÓN (en la FIGURA 4.25 el botón está representado como un cuadrado con un punto en el centro) que se encuentra en la entrada de la sala. Esto hará que aparezcan las plataformas 1,2 y 3 que le permitirán alcanzar la salida. Una vez transcurran 6

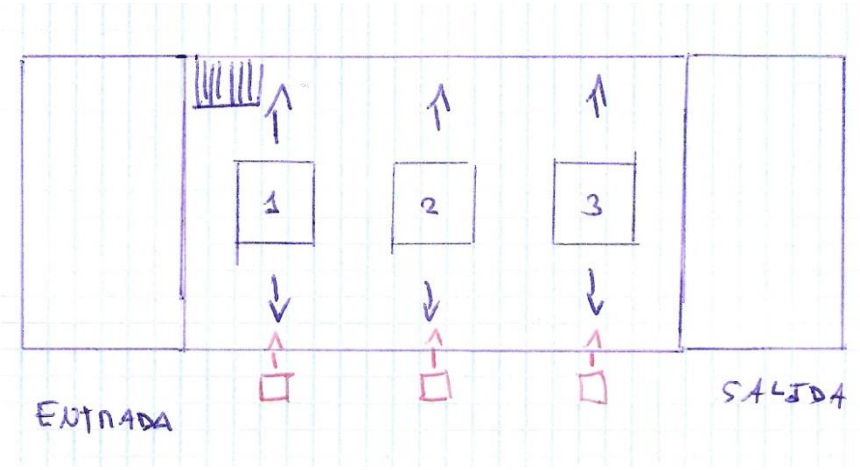
segundos tras pulsar el botón, éste volverá a su estado original y las plataformas desaparecerán.

Si el jugador no lo consigue, caerá a un piso inferior. Para salir del mismo, deberá utilizar las escaleras situadas en la parte inferior izquierda que le llevarán a la entrada.

Los tiempos de recompensa son:

- En menos de 7 segundos, el jugador recibe 2 puntos.
- En menos de 10,5 segundos, el jugador recibe 1 punto.
- En más de 10,5 segundos el jugador no recibe puntos.

FIGURA 4.26 Sala 2

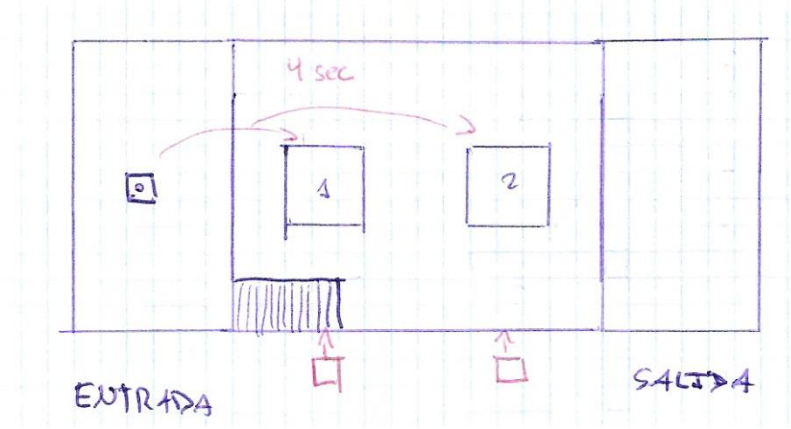


El jugador deberá cruzar entre las plataformas móviles para llegar a la salida. Las plataformas se moverán horizontalmente de izquierda a derecha. Además habrá tres cañones situados en la pared derecha de la sala (dibujados en rojo en la FIGURA 2.26) disparando a intervalos de 3 segundos aunque estos por el momento no serán dañinos.

Los tiempos de la recompensa son:

- En menos de 8,5 segundos el jugador recibe 2 puntos.
- En menos de 12,75 segundos el jugador recibe 1 punto.
- En más de 12,75 segundos el jugador no recibe puntos.

FIGURA 4.27 Sala 3

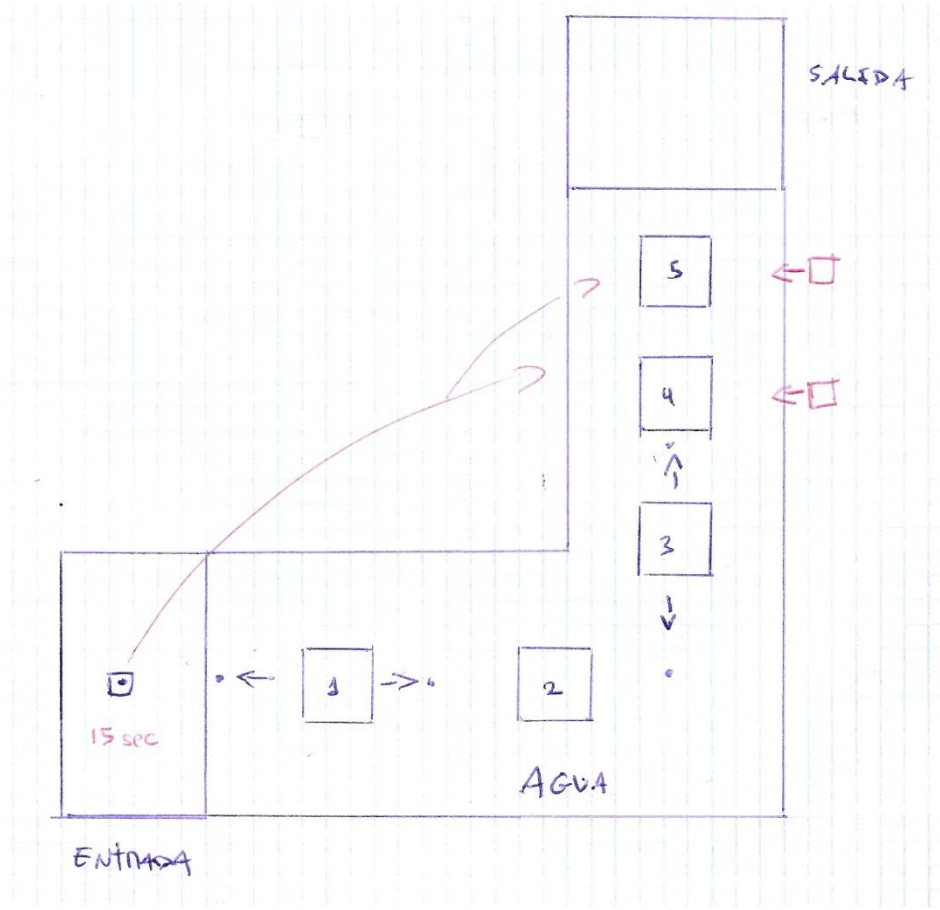


Esta sala combina mecánicas de la sala 1 y 2. En este caso el jugador deberá pulsar el botón que se encuentra al comienzo. Éste hará aparecer a las plataformas 1 y 2 durante 4 segundos. A la derecha, alineados con las plataformas, se encuentran dos cañones que disparan a intervalos de 2 segundos.

Los tiempos de la recompensa son:

- En menos de 5 segundos el jugador recibe 2 puntos.
- En menos de 7,5 segundos el jugador recibe 1 punto.
- En más de 7,5 segundos el jugador no recibe puntos.

FIGURA 4.28 Sala 4



El botón que se encuentra al comienzo de la sala hará aparecer a las plataformas 4 y 5 durante 15, que al igual que en la sala anterior, tienen cañones disparando en la misma dirección en la que se encuentran.

Las plataformas 1 y 3 son móviles y se desplazan en la dirección indicada por la flecha.

Además, en esta sala hay agua bajo las plataformas, por lo que si el jugador cae, perderá salud y volverá al punto de inicio.

Entonces para completar la sala, el jugador deberá pulsar el botón y avanzar por las plataformas 1,2 y 3. Al atravesar las plataformas 4 y 5 deberá hacerlo sin perder tiempo y con cuidado de no ser impactado por los cañones.

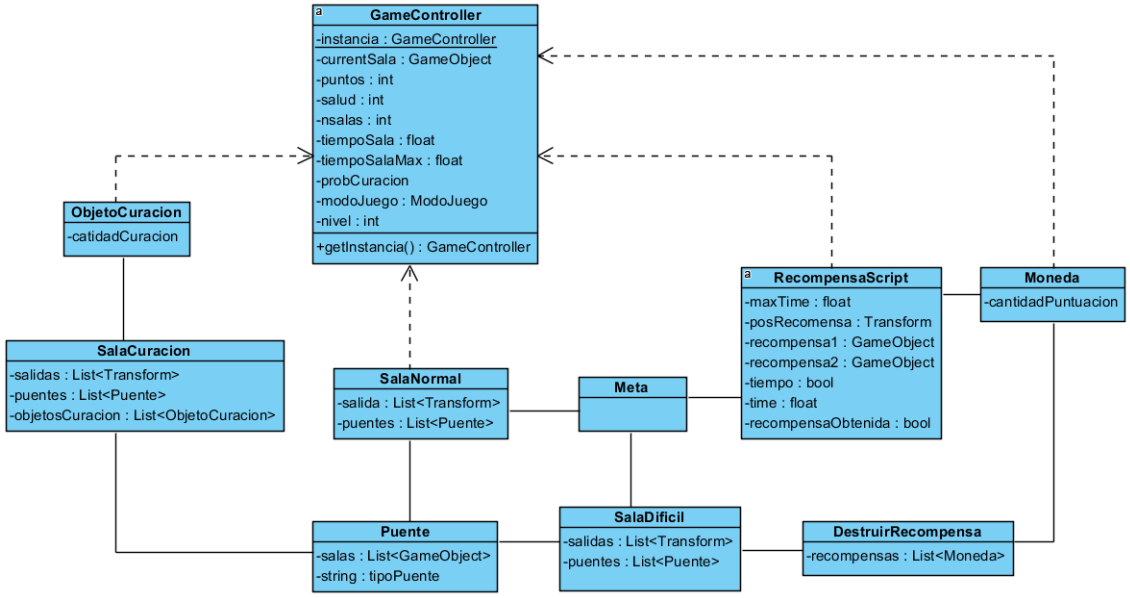
Los tiempos de la recompensa son:

- En menos de 15 segundos el jugador recibe 2 puntos.
- En menos de 22,5 segundos el jugador recibe 1 punto.
- En más de 22,5 segundos el jugador no recibe puntos.

Al ser considerablemente más difícil que las anteriores, a esta sala se le han dado tiempos mayores.

4.3.2. Diseño

FIGURA 4.29 Diagrama de clases UML de generación aleatoria de niveles



La FIGURA 4.29 muestra el diagrama de clases UML diseñado para implementar el sistema de generación de niveles que a continuación se describe.

Cada sala independientemente del tipo que sea tiene:

- Una lista de ubicaciones (Transforms que especifican unas coordenadas) donde estarán los puentes o caminos a la siguiente sala.
- Una lista de puentes o caminos.

Al instanciar una sala, para cada salida de la lista de salidas, se instanciará un puente escogido aleatoriamente entre los disponibles en la lista de puentes.

Cada puente además constará con una lista de salas. Estas serán las salas que podrá generar el puente en caso de que el jugador escoja avanzar por el mismo. La sala que genere se seleccionará aleatoriamente de entre todas las que haya disponibles en la lista.

Además, cada sala constará de un GameObject Meta, el punto a partir del cual se considerará que el jugador ha superado la misma. Dicha meta tendrá el Component RecompensaScript que se describe más adelante y cuyo principal cometido es el de 'evaluar' el desempeño del jugador en la sala.

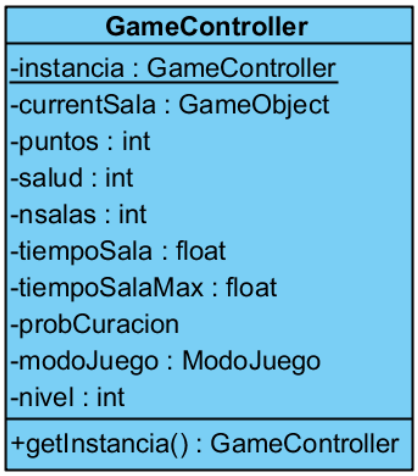
Por otra parte, las salas de curación tendrán también una lista con objetos de curación. Una vez se instancia una sala de curación, se elegirá uno de los posibles objetos de curación de la lista y también se instanciará para que el jugador lo pueda recoger.

Los objetos de curación y las monedas tendrán como atributo la cantidad de vida o puntuación que otorgarán al jugador. Una vez éste los recoja, desaparecerán.

Destruir recompensa será un Componente de tipo script únicamente presente en las salas difíciles y su función será la de destruir todas las recompensas adicionales existentes en dicha sala. Para ello, contará con una lista con todas las recompensas extra, si el jugador es herido, estas serán eliminadas.

Finalmente es necesario el uso de un GameObject que se encargue de registrar todos los parámetros esenciales del juego tales como la puntuación, salud del jugador, número de salas completadas, etc… Dicho GameObject debe además de poder ser accedido por el resto de componentes del juego en cualquier momento. Este será el GameController que se describe a continuación.

FIGURA 4.30 Clase GameController UML



La FIGURA 4.30 muestra el GameController la clase encargada de registrar los principales parámetros que definen el juego. Está implementada mediante el patrón de diseño Singleton (Freeman 2004) de manera que sólo existe una única instancia de la misma que es accedida por el resto de clases que necesitan conocer o modificar parámetros como la salud o la puntuación.

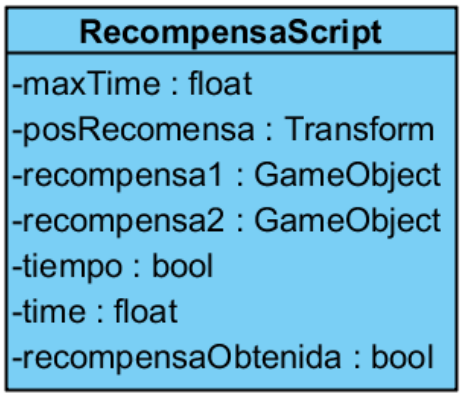
Atributos:

- CurrentSala: Referencia a la sala actual en la que se encuentra el jugador.
- Puntos: Puntuación del jugador.
- Salud: Cantidad de salud que le queda al jugador.
- MaxSalud: Salud máxima del jugador. La salud nunca puede superar a este valor.
- TiempoSala: Tiempo que lleva el jugador en la CurrentSala.
- TimpoSalaMax: Margen de tiempo para conseguir completar la CurrentSala y obtener mayor cantidad de puntos.

- ProbCuración: Probabilidad de que en la próxima sala se genere un camino que lleve a una sala de curación. Esta probabilidad incrementa con cada sala completada.
- modoJuego: Es un enum que le indica al GameController el modo de juego en el que se encuentra. Puede ser Normal o Desafío.
- Nivel: Nivel actual en el que se encuentra el jugador actualmente. Sólo es relevante en el modo Normal.

Para todos los atributos listados existen Getters y Setters que permiten su acceso y escritura.

FIGURA 4.31 Clase RecompensaScript UML



La FIGURA 4.31 muestra la clase encargada de determinar la recompensa que recibirá el jugador al terminar una sala.

Atributos:

- maxTime: Determina el tiempo máximo que tiene el jugador para obtener la puntación al atravesar la meta en una sala normal.
- posRecompensa: Ubicación dónde aparecerá la recompensa una vez alcanzada la Meta.
- Recompensa1 y 2: GameObjects (monedas) que se instanciaran en la posRecompensa.
- Tiempo: Booleano que indica si la sala tiene tiempo máximo o no (si es una sala normal o difícil)
- Time: Tiempo que lleva el jugador dentro de la sala.
- RecompensaObtenida: Indica si el jugador ha atravesado ya la meta o no.

4.3.3. Detalles de la implementación

Generación del nivel

El script *Assets/Salas/GeneradorNivel/Script/SalaScript.cs* define los caminos que aparecerán en una sala.

En el método Start() se selecciona cada una de las Transforms con las coordenadas en las que aparecerá un nuevo camino o puente y escoge al azar uno de los posibles tipos de caminos para instanciarlo.

Si el camino escogido es verde, además habrá una probabilidad de que éste se instancie o no. Dicha probabilidad viene determinada por el atributo ProbCuracion del GameController que aumenta su valor en 0.02 por cada sala completada. Una vez se genera un camino verde, la ProbCuracion es restablecida a 0 otra vez.

FIGURA 4.32 Instanciación de los puentes

```
foreach (Transform salida in salidas) {

    bool seleccionado = true;

    do {
        puente = Random.Range (0, puentes.Count);

        tipoPuente = puentes[puente].GetComponent<PuenteScript>().tipoPuente;

        if (tipoPuente == "Curacion"){
            if (Random.Range(1,100) <= GameController.getInstancia().getProbCuracion()) {
                seleccionado = true;
                GameController.getInstancia().setProbCuracion(0);
            }
            else {
                seleccionado = false;
            }
        }
    } while (tipoPuente == "Curacion" && !seleccionado);

    nuevoPuente = Instantiate (puentes[puente], salida);
    nuevoPuente.transform.position = salida.position;
    nuevoPuente.transform.rotation = salida.rotation;
}
```

El script *Assets/Salas/GeneradorNivel/Script/PuenteScript.cs* implementa el funcionamiento de los caminos.

Cada camino tiene asociada una lista con las posibles salas que se pueden generar en caso de que el jugador lo escoja. Si el jugador está en modo Normal, la lista de salas será un subconjunto del total que dependerá del nivel en el que se encuentre, (a mayor dificultad, se seleccionará un subconjunto de salas más difíciles).

En el modo Desafío las listas de posibles salas de cada camino estarán formadas por el conjunto total de todas las salas. Es decir, al cruzar una salida se podrá instanciar cualquier sala de todas las existentes.

Una vez el jugador atraviesa el camino, se selecciona una de las salas posibles, ésta pasa a ser la CurrentSala en el GameController y se elimina la sala anterior.

FIGURA 4.33 Eliminando la sala antigua

```
GameController.getInstancia ().addNSalas (1);
GameController.getInstancia ().eliminarCurrentSala ();
GameController.getInstancia ().setCurrentSala (nuevaSala);
```

Sistema de recompensas

Dentro de cada sala hay un GameObject Meta. Dicho GameObject tiene un Component de tipo BoxCollider de manera que cuando el jugador lo atraviese, la sala quedará completada. El sistema de recompensas está implementado en el script *Assets/Salas/GeneradorNivel/Script/RecompensaScript.cs* asociado a la Meta.

Si la sala es normal, habrá un tiempo límite en base al cual se generará una recompensa distinta (0, 1 o 2 puntos). En cada Update() se actualizará el tiempo que lleva el jugador en la sala como se puede ver en la FIGURA 4.34.

FIGURA 4.34 El tiempo se actualiza en cada frame

```
// Update is called once per frame
void Update () {
    if (!recompensaObtenida && tiempo) {
        time += Time.deltaTime;
        GameController.getInstancia().setTiempoSala (time);
    }
}
```

Al atravesar el BoxCollider se harán dos comprobaciones:

- El tipo de sala en la que se encuentra el jugador. Si el jugador está en una sala difícil, al completarla siempre se generará una recompensa de un punto.
- En caso de que sea una sala normal, el tipo de recompensa que ha conseguido el jugador en función del tiempo (revisar los requisitos para más detalles).

Una vez hechas dichas comprobaciones, se instanciará la recompensa como se puede ver en la FIGURA 4.35.

FIGURA 4.35 Instanciando la recompensa

```
void OnTriggerEnter(Collider other){
    if (other.tag == "Player" && !recompensaObtenida) {
        if (tiempo) {
            if (time <= maxTime) {
                Instantiate (recompensa1, posRecompensa.position, posRecompensa.rotation);
            } else if (time <= maxTime * 1.5) {
                Instantiate (recompensa2, posRecompensa.position, posRecompensa.rotation);
            }
        } else {
            Instantiate (recompensa2, posRecompensa.position, posRecompensa.rotation);
        }
        recompensaObtenida = true;
    }
}
```

En las salas difíciles, cualquier amenaza capaz de dañar al jugador, tendrá entre sus componentes el script *Asset/Salas/GeneradorNivel/Scripts/DestruirRecompensa.cs.* Dicho script tiene una lista con todas las monedas extra que están repartidas por la sala. Si el jugador recibe daño, el script eliminará las recompensas del juego como se muestra en la FIGURA 4.36.

FIGURA 4.36 Destruir Recompensas

```
void OnTriggerEnter(Collider other){
    if (other.tag == "Player" && !destruidas) {
        foreach (GameObject o in recompensas) {
            Destroy (o);
        }
        destruidas = true;
    }
}
```

Objetos

Todos los objetos que puede recoger el jugador cuentan con un Collider de manera que cuando éste los toca, llaman al GameController y modifican la salud o la puntuación. Una vez realizada la modificación, son destruidos.

Los scripts que implementan la funcionalidad de los objetos son *Assets/Objetos/Scripts/Corazon.cs* y *Assets/Objetos/Scripts/Moneda.cs*.

4.4. Cuarta iteración

Durante esta iteración se ha implementado la interfaz de usuario que mostrará en pantalla los datos relevantes para el mismo. También se ha diseñado en Blender el modelo definitivo del personaje y los objetos así como sus correspondientes animaciones.

Además se han añadido las principales texturas al juego y ha introducido el sistema de puntos de control.

4.4.1. Requisitos

El juego contará con una interfaz que muestre los siguientes datos al jugador:

- Puntos de salud del jugador.
- Monedas recogidas y monedas necesarias para completar el nivel.
- Tiempo que lleva el jugador en la sala actual y tiempo máximo para conseguir recompensa completa. El tiempo se mostrará de color verde cuando el jugador pueda conseguir dos puntos, amarillo cuando pueda conseguir uno y rojo cuando no pueda conseguir ningún punto.
- Número de salas completadas.

Cualquier elemento capaz de dañar al jugador hará que (revisar FIGURA 4.2 en el apartado 4.1.1):

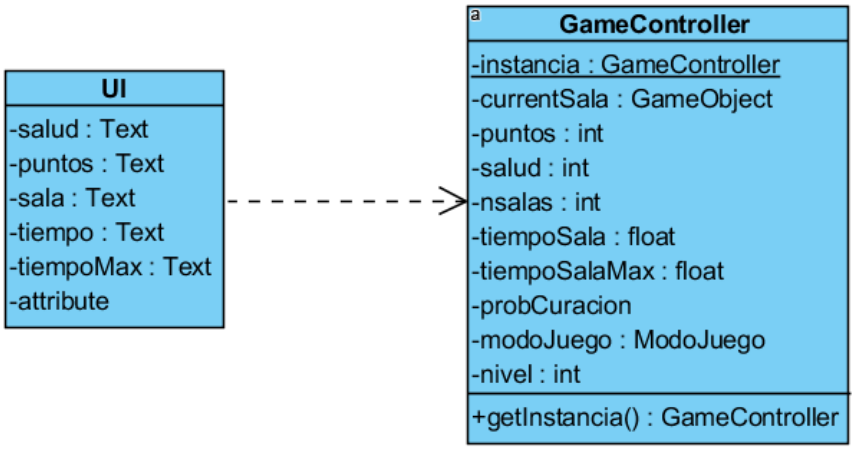
- Éste pierda un punto de salud.
- Sea enviado al punto de control de la sala.

Cada sala contará con un punto de control, el lugar al que será devuelto el jugador una vez reciba daño y que se encontrará al inicio de la sala.

4.4.2. Diseño

La FIGURA 4.37 se muestra el diagrama de clases de la interfaz de usuario. Puesto que el GameController registra todos los datos que se van a mostrar en pantalla, únicamente se requiere una clase UI que accederá a dichos datos y los muestre por pantalla.

FIGURA 4.37 Diagrama de clases de la UI



4.4.3. Implementación

Puntos de control y daño

Para la implementación de los puntos de control, se usa un GameObject vacío (sin componentes) al que se le ha asignado el tag 'Respawn'.

Los GameObjects capaces de herir al jugador, tienen asignado el script *Assets/Trampas/Scripts/Daño.cs* entre sus componentes.

FIGURA 4.38 Script del daño

```
void LateUpdate () {
    if (playerGolpeado) {
        GameObject.FindGameObjectWithTag ("Player").transform.position =
            GameObject.FindGameObjectWithTag ("Respawn").transform.position;
        GameController.getInstancia().addSalud (-1);
        playerGolpeado = false;
    }
}

void OnTriggerEnter (Collider other){
    if (other.tag == "Player") {
        playerGolpeado = true;
    }
}
```

Como se muestra en la FIGURA 4.38 cuando el jugador colisiona con el objeto dañino, el valor de playerGolpeado pasa a true. Entonces en el siguiente frame (LateUpdate()), se busca el GameObject del jugador y del punto de control usando sus respectivas etiquetas. La posición del jugador se iguala a la del punto de control y se accede al GameController para restarle un punto de salud.

Interfaz

La interfaz se ha implementado mediante un canvas (Unity s.f.) que tiene asociado el script *Assets/Salas/UI.cs* al mismo. Dicho script accede al GameController para conocer y mostrar por pantalla los datos que forman parte de la interfaz.

FIGURA 4.39 Script de la UI

```
void Update () {

    salud.text = "Salud: " + GameController.getInstancia().getSalud ().ToString ();
    puntos.text = "Monedas: " + GameController.getInstancia().getPuntos ().ToString () + " / 10";
    sala.text = "Sala: " + GameController.getInstancia().getNSalas().ToString();
    if (tiempo != null) {

        tiempo.text = GameController.getInstancia().getTiempoSala ().ToString ("F2");
        tiempoMax.text = " / " + GameController.getInstancia().getTiempoSalaMax ().ToString ("F2");

        if (GameController.getInstancia().getTiempoSala () <= GameController.getInstancia().getTiempoSalaMax ()) {
            tiempo.color = Color.green;
        } else if (GameController.getInstancia().getTiempoSala () <= GameController.getInstancia().getTiempoSalaMax () * 1.5) {
            tiempo.color = Color.yellow;
        } else {
            tiempo.color = Color.red;
        }
    }
}
```

Como se puede ver en la FIGURA 4.39, todos los datos a mostrar en la interfaz de usuario son accedidos del GameController y convertidos a texto.

En el caso del tiempo, éste se mostrará de diferentes colores dependiendo de cuanto haya pasado el jugador en la sala. Se mostrará de color verde siempre que el tiempo sea menor que el máximo para conseguir dos puntos.

Después pasará a color amarillo si siendo mayor que el máximo, no es más que 1.5 veces el mismo.

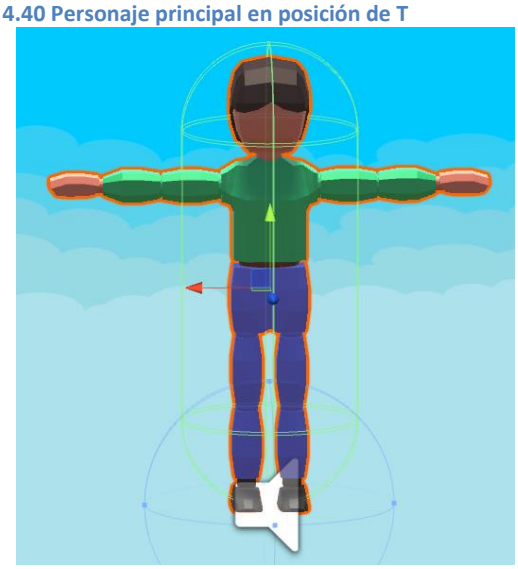
Finalmente, pasará a rojo cuando sea superior a 1.5 veces el máximo.

Texturas, fuentes de texto y skyboxes.

Estos tres elementos se han implementado mediante paquetes de assets gratuitos importados de la Asset Store. Los paquetes son:

- FarlandSkies, para las Skyboxes (Unity s.f.) utilizadas en todas las escenas.
- JazzCreateBubble, tipo de fuente utilizado para los textos de la interfaz y los menús.
- Five Seamless Tileable Ground Textures, paquete de texturas simples usado en la mayor parte del juego.

FIGURA 4.40 Personaje principal en posición de T



En la FIGURA 4.40 se muestra el modelo final del personaje principal realizado en Blender.

4.5. Quinta iteración

La última iteración se ha centrado en la implementación del resto de salas para dar una mayor variedad al videojuego. En este punto, la mayoría de mecánicas ya han sido implementadas por lo que únicamente ha sido necesario reutilizar los Prefabs y scripts ya creados.

Además se ha implementado un menú principal y las pantallas de transición necesarias.

4.5.1. Requisitos

El prototipo contará con un total de 26 salas (22 normales y 4 difíciles).

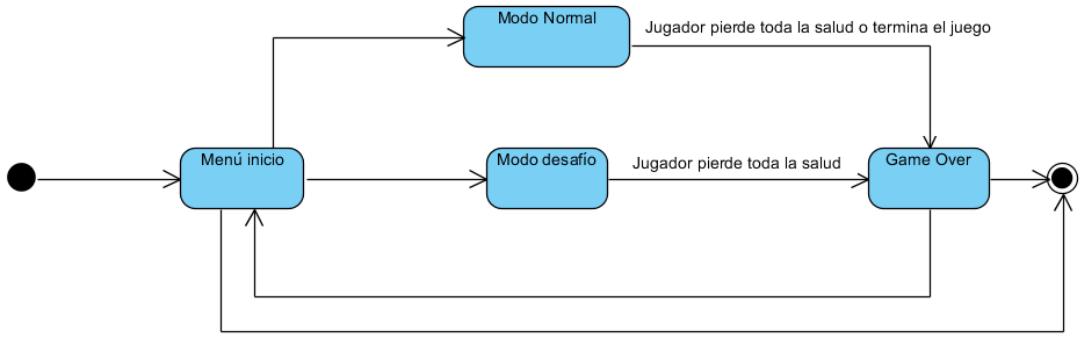
Además se implementarán el mecanismo de las plataformas giratorias. Estas plataformas girarán sobre su propio eje cada cierto tiempo con el objetivo de tirar al jugador de las mismas en caso de que esté sobre ellas. Antes de girar, cambiarán a color rojo de manera intermitente para avisar al jugador.

También habrá una variante de estas plataformas que tendrán pinchos sobre una de sus caras. Al girar, los pinchos quedarán expuestos pudiendo dañar al jugador.

Por otra parte, habrá un menú principal que dará opción a elegir el modo de juego al jugador y a salir del mismo.

También habrá una pantalla de fin del juego (GameOver) que aparecerá cuando el jugador pierda toda la salud o cuando éste complete el tercer nivel. La pantalla de fin del juego dará opción a volver al menú de inicio o a salir del juego. La FIGURA 4.41 muestra un diagrama de máquina de estados con la funcionalidad descrita.

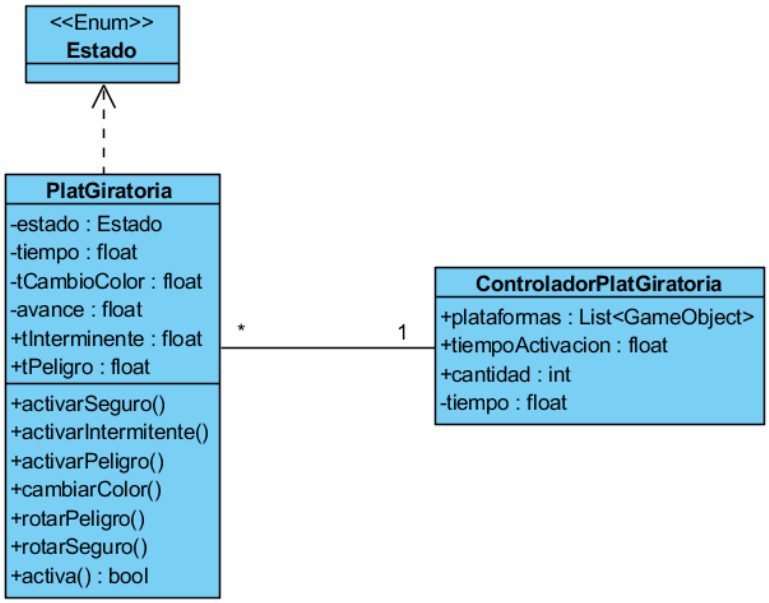
FIGURA 4.41 Diagrama de estados del juego



4.5.2. Diseño

A continuación se describe el diseño de las plataformas giratorias.

FIGURA 4.42 Diagrama de clases de las plataformas giratorias.



Todas las plataformas giratorias de una sala están asociadas a un controlador que las tiene registradas en una lista. Cada cierto tiempo, el Controlador seleccionará una de las plataformas de su lista aleatoriamente y la hará girar.

Las plataformas giratorias se pueden encontrar en tres estados distintos:

- Seguro: La plataforma no ha girado ni lo va a hacer próximamente.
- Intermitente: La plataforma va a girar en breve (cuando pase el tiempo indicado en **tIntermitente**).
- Peligro: La plataforma ha girado. Volverá al estado Seguro cuando pase el tiempo indicado en **tPeligro**.

Lista de funciones de la clase PlatGiratoria:

- activarSeguro(): Hace que una plataforma que está en estado de peligro, vuelva a su estado de seguro.

- activarIntermitente(): Una plataforma que está en estado de seguro, pasa a estado intermitente.
- activarPeligro(): Una plataforma que está en estado intermitente, pasa a estado de peligro.
- cambiarColor(): Hace que la plataforma alterne entre los colores azul y rojo mientras está en estado intermitente.
- rotarPeligro() y rotarSeguro(): Rotan la plataforma cuando pasa a estado de peligro o a estado de seguro.
- activa(): Indica si la plataforma ha sido seleccionada para girar, es decir, si está en estado intermitente o de peligro.

El ControladorPlatGiratoria se encarga de controlar qué plataformas girarán.

Atributos:

- Plataformas: lista de plataformas que el controlador tiene bajo su control.
- tiempoActivacion: tiempo que transcurrirá cada vez que el controlador selecciona un conjunto de plataformas para que se volteen.
- Cantidad: número de plataformas que se voltean cada vez que transcurre el tiempo de activación.

4.5.3. Implementación

El script *Asset/Trampa/Scripts/ControladorPlatGiratoria.cs* implementa el control de las plataformas giratorias.

FIGURA 4.43 script del control de las plataformas giratorias

```
void Update () {
    tiempo += Time.deltaTime;
    if (tiempo >= tiempoActivacion) {
        for (int i = 0; i < cantidad; i++){
            tiempo = 0;
            bool activado = false;

            while (!activado) {

                int rand = Random.Range (0, plataformas.Count);
                if (!plataformas [rand].GetComponent <PlatGiratoria> ().activa ()) {
                    plataformas [rand].GetComponent <PlatGiratoria> ().activarIntermitente ();
                    activado = true;
                }
            }
        }
    }
}
```

En la FIGURA 4.43 se muestra la parte del código relativo a las plataformas giratorias. Una vez trascurre el tiempo de activación, se seleccionan i plataformas de manera aleatoria. Si estas no han sido activadas anteriormente, se las activa para que cambien de estado seguro a estado intermitente.

5. Medios empleados

5.1. Unity5

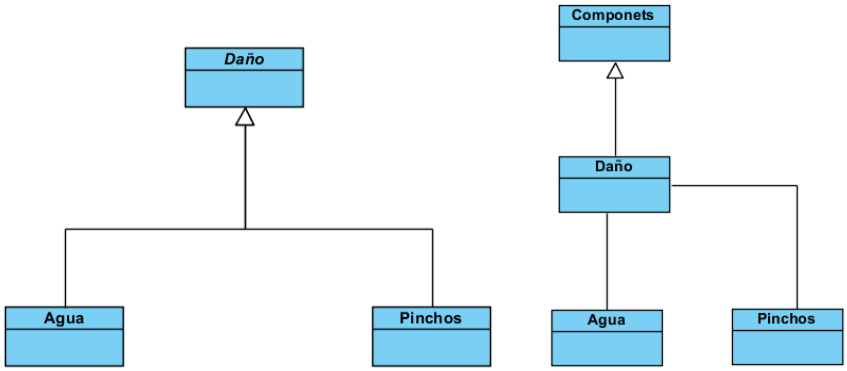
Para el desarrollo del prototipo, se ha empleado el motor Unity5. Además de ser gratuito, cuenta con una gran cantidad de documentación disponible así como tutoriales para iniciarse en su uso. Además de esto, otras características que han sido determinantes a la hora de escoger Unity han sido:

Arquitectura Sistema Entidad-Componente

En Unity todos los objetos presentes en una escena son GameObjects (Unity s.f.). Los GameObjects tienen asociados Components (Unity s.f.) que les dan diversas características como puedan ser físicas, texturas, scripts, colisiones, etc.

Con esta arquitectura, se elimina la necesidad de utilizar la herencia descartando con ello todos los problemas derivados de la misma. Además permite una gran reutilización de código dotando a todo el proyecto de gran flexibilidad.

FIGURA 5.1 Ejemplo de herencia vs Sistema Entidad-Componente



La FIGURA 5.1 muestra la diferencia entre implementar un sistema de daño con herencia y componentes.

Mediante la arquitectura basada en componentes, las fuentes de daño pueden ser tratadas como GameObjects independientes que llevan asociado el componente de daño. De esta forma, podemos por ejemplo hacer que el agua, además de hacer daño, se mueva en una dirección añadiéndole un componente que permita dicho comportamiento.

Así se pueden entender los GameObjects como contenedores de Components que definen la manera en que estos se comportan. Unity provee al usuario de un conjunto de Components que le permiten definir físicas, colisiones, sistemas de partículas o fuentes de luz entre otras muchas cosas o crear propios scripts en C# o en Javasript.

Mecanim

Es una herramienta que Unity proporciona para facilitar el control de las animaciones asociadas a los GameObjects. Su uso se basa en la creación de máquinas de estados para

conjuntos de animaciones. Dichas máquinas de estados son visuales por lo que se pueden crear y enlazar nodos fácilmente.

Hay que aclarar que Mecanim no es una herramienta para crear animaciones, su función es la de estructurar conjuntos de animaciones de una manera cómoda e intuitiva.

Asset Store

La Asset Store es una tienda digital en la que los usuarios publican contenido (gratuito o no) que puede ser descargado e importado a los proyectos fácilmente.

Prefabs

Unity permite la creación de Prefabs (Unity s.f.), GameObjects que han sido almacenados junto con todos sus componentes y características para su posterior reutilización lo cual conlleva un gran ahorro de tiempo y esfuerzo. Además las múltiples instancias de un mismo Prefab pueden ser editadas simultáneamente editando el original.

Otro uso sumamente interesante de los Prefabs es que estos pueden ser instanciados en la escena en tiempo de ejecución.

5.2. Otros medios

Adicionalmente, se han utilizado los siguientes medios durante el desarrollo del proyecto:

Blender

Software libre especializado en la creación de modelado, iluminación, renderizado y desarrollo de gráficos en tres dimensiones.

Durante el desarrollo del prototipo, se ha usado Blender para el modelado del personaje principal y otros objetos como las monedas y corazones además de las correspondientes animaciones asociadas a los mismos.

Audacity

Software libre cuya principal función es la grabación y edición de audio.

Para el desarrollo del prototipo, se ha usado Audacity para la edición de lo audio usado en el mismo.

MonoDevelop

Entorno de programación que integra Unity5 para la creación de scripts en C# o Javascript.

Todos los scripts del prototipo se han programado en C# usando este entorno de desarrollo.

6. Manual de usuario

6.1. Modos de juego y menú principal

Time Runner es un videojuego de plataformas en 3D en el que nuestro objetivo será avanzar por diferentes niveles generados aleatoriamente superando pruebas y consiguiendo la mayor puntuación posible.

Al iniciar el juego, apareceremos en la pantalla de inicio con un menú que nos dará a escoger entre las siguientes opciones.

FIGURA 6.1 Menú de inicio

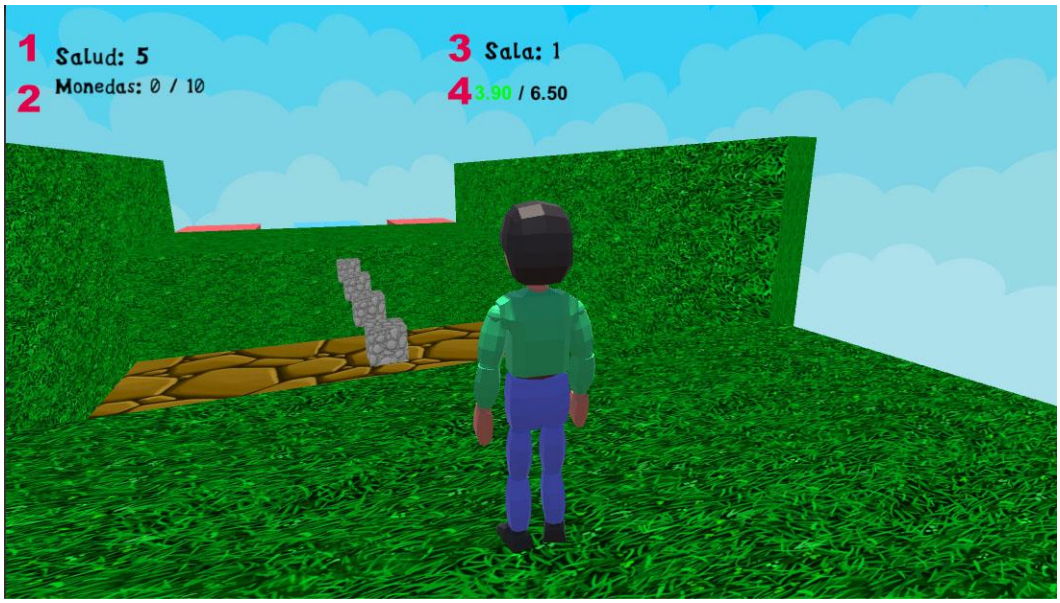
La primera opción nos permitirá jugar al modo normal, en el que deberemos superar niveles obteniendo una determinada puntación para avanzar entre los mismos. El modo normal consta de 3 niveles en total.

El modo desafío consta de un único nivel sin fin. El objetivo será el jugador sobreviva obteniendo la mayor puntación posible.

El botón salir cerrará el juego.

6.2. Interfaz

FIGURA 6.2 Interfaz



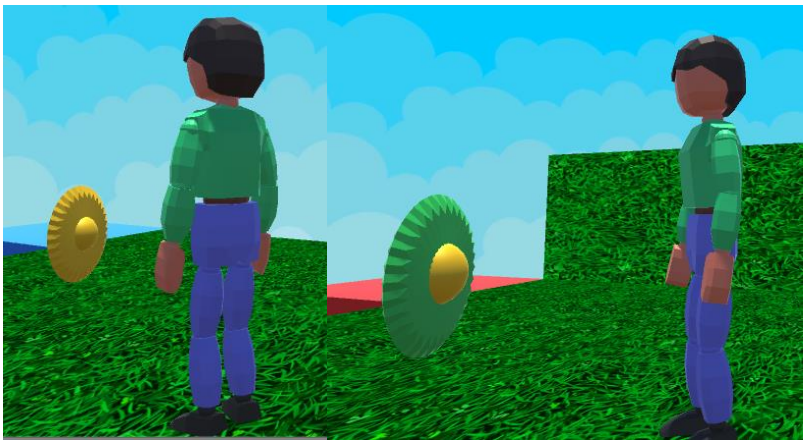
- 1: Salud del jugador. Como máximo tendrá 5 puntos. Cada vez que reciba daño, perderá un punto.
- 2: Monedas recogidas. A la izquierda el número actual de monedas, a la derecha, el número necesario para avanzar al siguiente nivel.
- 3: Sala en la que se encuentra el jugador. Cada vez que supera una sala, se incrementa en uno.
- 4: Tiempo para completar la sala. A la izquierda el tiempo que lleva en esa sala el jugador. A la derecha, el tiempo máximo para obtener una puntuación perfecta en esa sala. Mientras el número esté en verde, el jugador obtendrá 2 puntos, cuando pase a color amarillo, obtendrá 1 punto. Cuando sea de color rojo, no obtendrá puntos.

6.3. Controles

- Para caminar, se pueden usar las teclas WASD o las flechas direccionales.
- Para saltar, pulsa espacio. Cuidado, mientras estás en el aire pierdes parte de tú movilidad.
- Se puede mover la cámara en torno al personaje usando el ratón.

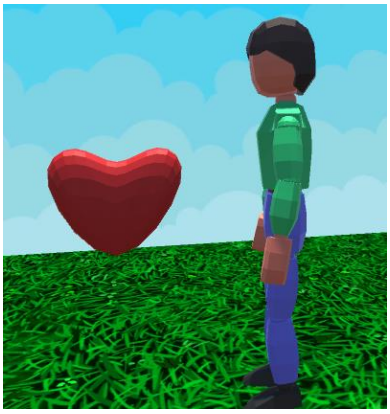
6.4. Objetos

FIGURA 6.3 Monedas



Las monedas amarillas darán 1 punto al jugador.
Las monedas verdes darán 2 puntos al jugador.

FIGURA 6.4 Corazones



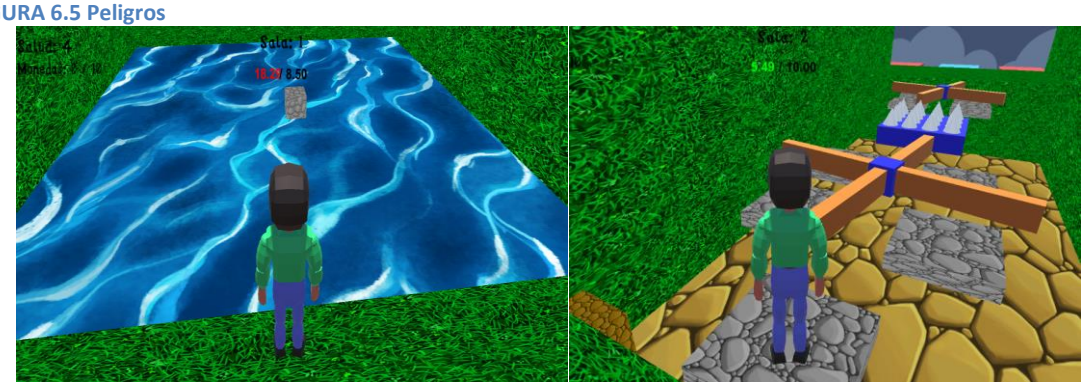
Los corazones restablecen la salud del jugador.

Medio corazón restablece un punto de salud.
Un corazón entero restablece dos puntos de salud.
Un corazón doble restablece toda la salud.

6.5. Peligros y salas

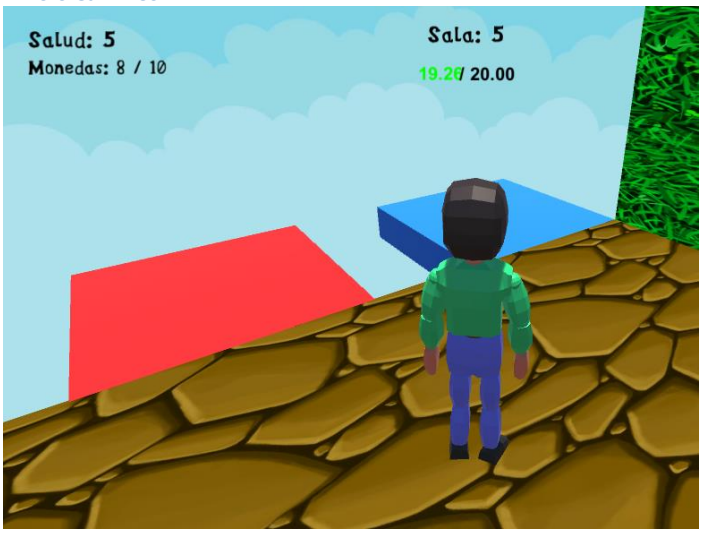
A lo largo de los niveles, como jugador enfrentarás multitud de peligros y riesgos. Ten cuidado y trata de evitarlos en la medida de lo posible. Ser alcanzado por un peligro hará que pierdas un punto de salud y te devolverá al comienzo de la sala haciéndote perder valioso tiempo y futuras recompensas.

FIGURA 6.5 Peligros



El agua, las hélices o las superficies puntiagudas serán algunos de los peligros deberás evitar.

FIGURA 6.6 Caminos



Cada vez que completes una sala, aparecerán varios caminos para continuar. Los caminos azules conducen a salas de dificultad normal en las que el tiempo determinará la recompensa que recibirás.

Los caminos rojos conducen a salas de mayor dificultad en las que las recompensas también serán mayores.

Los caminos verdes son menos usuales y conducen a salas en las que podrás recuperar salud.

Recuerda, una vez elegido un camino, no habrá vuelta atrás.

7. Conclusiones

El desarrollo de un videojuego, al igual que con cualquier otro software, es una tarea que requiere de una gran dedicación y esfuerzo. Es necesario escoger y seguir una buena planificación y ceñirse a metas realistas, pues resulta fácil perderse entre montones de funcionalidades y características y acabar desarrollando muchas cosas, pero nada concreto y funcional.

Ese es en gran medida uno de los mayores desafíos para desarrollador, pasar de lo que está escrito en el papel, al producto funcionando. Dividir el trabajo y no tratar de abarcarlo todo al mismo tiempo, enfocarse en cosas concretas y tener siempre claro el horizonte al que nos dirigimos es fundamental para acabar con éxito un trabajo de esta magnitud. Por eso, para el desarrollo de este proyecto, se escogieron dos ideas simples pero que han estado presentes durante todo el proceso y que han determinado por completo el rumbo del mismo: crear un videojuego del género plataformas en tres dimensiones; y que cada vez que se juegue, la experiencia sea distinta.

Ahora que el prototipo está funcionando y se han completado las metas que se establecieron en su comienzo, queda por delante expandirlo y mejorarlo. Una de las más evidentes mejoras es la relativa al apartado visual y sonoro ya que durante el desarrollo, al ser elementos pertenecientes al apartado artístico del videojuego y ser éste un prototipo, no se hizo especial hincapié en los mismos.

Pero también espero en un futuro poder añadir nuevos cambios y características que enriquezcan y hagan más interesante y desafiante jugar a mi videojuego. Nuevas mecánicas, más niveles, una mayor cantidad de desafíos serán algunos de esos cambios.

Bibliografía

Beane, Andy. *3D Animation Essentials.* Indianapolis, 2012.

E.M, Jiménez. *Ingeniería de Software Ágil.* Madrid: Bubok, 2010.

Freeman, Eric. *Head First design patterns.* Sebastopol: O'Reilly, 2004.

Unity. *https://docs.unity3d.com.* s.f. https://docs.unity3d.com/Manual/class-Animator.html (último acceso: Enero de 2018).

—. *https://docs.unity3d.com.* s.f. https://docs.unity3d.com/Manual/class-CharacterController.html (último acceso: Enero de 2018).

—. *https://docs.unity3d.com.* s.f. https://docs.unity3d.com/462/Documentation/Manual/MecanimAnimationSystem.html (último acceso: Junio de 2018).

—. *https://docs.unity3d.com.* s.f. https://docs.unity3d.com/Manual/class-Camera.html?_ga=2.117490697.1869778707.1518392958-192241298.1516269017 (último acceso: Junio de 2018).

—. *https://docs.unity3d.com.* s.f. https://docs.unity3d.com/ScriptReference/MonoBehaviour.Update.html (último acceso: Junio de 2018).

—. *https://docs.unity3d.com.* s.f. https://docs.unity3d.com/Manual/class-BlendTree.html (último acceso: Junio de 2018).

—. *https://docs.unity3d.com.* s.f. https://docs.unity3d.com/560/Documentation/Manual/UICanvas.html (último acceso: Junio de 2018).

—. *https://docs.unity3d.com.* s.f. https://docs.unity3d.com/Manual/Tags.html (último acceso: Junio de 2018).

—. *https://docs.unity3d.com.* s.f. https://docs.unity3d.com/ScriptReference/Mathf.Clamp.html (último acceso: Junio de 2018).

—. *https://docs.unity3d.com.* s.f. https://docs.unity3d.com/ScriptReference/BoxCollider.html (último acceso: Junio de 2018).

—. *https://docs.unity3d.com.* s.f. https://docs.unity3d.com/Manual/class-Skybox.html (último acceso: Junio de 2018).

—. *https://docs.unity3d.com.* s.f. https://docs.unity3d.com/560/Documentation/Manual/Prefabs.html (último acceso: Junio de 2018).

—. *https://docs.unity3d.com.* s.f. https://docs.unity3d.com/Manual/class-GameObject.html (último acceso: Junio de 2018).

—. *https://docs.unity3d.com.* s.f. https://docs.unity3d.com/Manual/Components.html (último acceso: Junio de 2018).

—. *https://docs.unity3d.com.* s.f. https://docs.unity3d.com/ScriptReference/MonoBehaviour.Start.html (último acceso: Junio de 2018).