UNIVERSIDAD DE CHILE
FACULTAD DE CIENCIAS FÍSICAS Y MATEMÁTICAS
DEPARTAMENTO DE CIENCIAS DE LA COMPUTACIÓN

AJUSTE DINÁMICO DE DIFICULTAD EN VIDEOJUEGOS MEDIANTE BIOFEEDBACK

MEMORIA PARA OPTAR AL TÍTULO DE
INGENIERO CIVIL EN COMPUTACIÓN

ELÍAS MOISÉS ZELADA BAEZA

PROFESOR GUÍA:
FRANCISCO GUTIÉRREZ FIGUEROA

MIEMBROS DE LA COMISIÓN:
MARÍA CECILIA BASTARRICA PIÑEYRO
EDUARDO GODOY VEGA

SANTIAGO DE CHILE
2019

Resumen

Los videojuegos se diseñan para presentar un desafío al usuario, ya sea medir su habilidad para resolver un problema, su coordinación para presionar teclas o su conocimiento sobre algún tema en particular. Para los usuarios, el atractivo de un videojuego viene principalmente dado por el sentimiento de satisfacción al completar esos desafíos. No obstante, si el jugador no es capaz de progresar debido a la presencia de una dificultad muy alta, eventualmente se frustrará y dejará de jugar. En cambio, si encuentra los desafíos muy fáciles y fugaces, se aburrirá y abandonará el juego. Debido a esto, un correcto balance en la dificultad de un juego es esencial para lograr su éxito. Por un lado, la dificultad no es sino una percepción, la cual es diferente para cada jugador. Por otro lado, la gran mayoría de videojuegos actualmente existentes ofrece una cantidad muy limitada de niveles de dificultad.

En este trabajo de título se utilizaron señales biológicas, en particular el pulso, para brindar una experiencia personalizada a cada jugador. La dificultad percibida por el usuario es variable a lo largo de todo el juego, modificándose a partir de los cambios que el usuario sufra. Para lograr esto, primero se escogió un sensor para medir el pulso, el cual debía ser cómodo, preciso, con buena conectividad y de bajo costo. Luego, se implementó una forma de transformar las medidas de pulso obtenidas a un grado de dificultad de manera dinámica. Después, se diseñó e implementó un juego de plataformas cuya dificultad, representada por la cantidad y disposición de obstáculos, puede ser variable sin que se pierda la esencia de cada nivel. Finalmente, se realizó un experimento con usuarios a baja escala, para poner a prueba el juego desarrollado y observar el efecto provocado por el ajuste dinámico de la dificultad.

Los resultados del estudio lograron evidenciar cómo los jugadores en el grupo con ajuste dinámico de dificultad mejoran su experiencia de usuario. En efecto, en general completaron los niveles en un menor número de intentos, requiriendo también una menor cantidad de tiempo. De igual manera, sus opiniones expresadas respecto al juego permiten concluir que tuvieron una experiencia más agradable y satisfactoria, a pesar de haber percibido un menor nivel de dificultad. Sin embargo, esta percepción es sólo aparente, ya que en promedio la dificultad experimentada fue la misma para ambos grupos, permitiendo así notar que la modificación de dificultad actuó en los momentos precisos en los que fue requerida.

Este trabajo de título da primeras luces sobre la factibilidad de diseñar un mecanismo tal que, utilizando señales biológicas obtenidas por tecnologías de fácil acceso, logre efectuar una regulación dinámica de dificultad en un videojuego. El siguiente paso en este trabajo consistirá en realizar una prueba de usuarios formal, para así validar empíricamente la adopción del mecanismo de regulación diseñado, en términos de desempeño y experiencia de usuario.

Agradecimientos

Gracias a mi profesor guía por su ayuda a lo largo de la realización de esta memoria, y a mi familia.

Tabla de Contenido

Índice de Tablas

Índice de Figuras

Capítulo 1

Introducción

Por más interesante o llamativo que pueda lucir un videojuego, si su alta dificultad impide al jugador realizar progresos significativos, este se sentirá impotente y frustrado. Por otro lado, si la dificultad es muy baja, el jugador se aburrirá al no encontrar el nivel de desafío que estaba buscando. En ambos casos, tanto la frustración como el aburrimiento pueden ocasionar que el jugador abandone el juego por completo, lo cual es un acontecimiento para nada deseable dentro de la industria.

Como la dificultad dentro de un videojuego depende de la percepción propia de cada usuario, no es posible establecer un nivel de desafío estándar que le brinde a todos una experiencia de usuario satisfactoria. Es por esto que muchos juegos permiten seleccionar de entre una lista acotada de opciones la dificultad a la que se desea jugar. Generalmente, estas categorías son tres: fácil, medio y difícil. Si bien el crear diferentes modos de dificultad permite una mayor flexibilidad para el jugador a la hora de enfrentarse a un juego, tanto el significado de cada modo como el adecuado ajuste a lo largo del tiempo quedan a total criterio del desarrollador, pudiendo no ser lo que los jugadores esperan.

1.1. Problema abordado

Históricamente, se han implementado métodos dinámicos de ajuste de dificultad en videojuegos, principalmente basados en la habilidad del jugador, su desempeño y cuánto tiempo ha pasado en un mismo nivel [15, 35]. Este análisis se realiza sobre información que se debe ir recolectando a lo largo del tiempo, con la cual se puede ir construyendo un perfil de usuario en base a predicciones. Sin embargo, no considera explícitamente lo que está sintiendo el jugador mientras juega, como por ejemplo, si un nivel o zona le está resultado demasiado estresante o si ha perdido el interés de continuar por falta de desafío.

Una manera de obtener más información sobre lo que siente un jugador de manera inmediata consiste en medir y procesar señales biológicas, como por ejemplo el pulso [14]. Este proceso, conocido en la literatura como *biofeedback*, permite que una aplicación de software interactivo, como lo es un videojuego, adapte su comportamiento en función de algún ti-

po de señal obtenida desde el usuario o jugador [38]. Esto implica calcular un estado base del jugador, sin ser expuesto a interferencias externas, y a partir de este valor monitorear constantemente su estado actual para detectar cualquier cambio abrupto. En particular, la aplicación de este tipo de técnicas no ha sido ampliamente explorada en el diseño de experiencias de usuario en videojuegos comerciales, existiendo pocos ejemplos como *Bring to Light* [4] y *O.zen* [13].

1.2. Solución propuesta

En este trabajo de título se diseñó e implementó un videojuego de plataformas dividido en niveles, cuya dificultad es modificada de acuerdo a las variaciones del pulso del jugador. Al respecto, la dificultad del juego se entiende como la cantidad de intentos que le toma al usuario en completar un determinado nivel. Para mantener el interés del jugador, cada nivel implica un mayor desafío en comparación al anterior. Sin embargo, la particularidad de la solución propuesta es que este desafío es regulado directamente por el pulso, medido en latidos por minuto. En efecto, estudios médicos han demostrado una alta correlación entre el nivel de excitación o interés de una persona en realizar una tarea y la manifestación física en su pulso cardiaco [1]. Si se detecta una variación significativa en el pulso del jugador respecto al valor promedio que ha mantenido durante la sesión, se realizan ajustes al nivel de tal forma que la dificultad se adecúe con las variaciones del pulso. De esta manera, si un jugador está teniendo complicaciones dentro de un nivel, es decir, le está tomando muchos intentos completarlo y alterándolo al punto de subir su pulso, se modifica el nivel para hacerlo más fácil. En contraste, si el pulso no experimenta ninguna variación considerable a lo largo del tiempo o incluso si empieza a bajar, se aumenta el grado de desafío para así no perder el interés del jugador.

En la figura 1.1 se presenta el tercer nivel del videojuego implementado, el cual es parte del tutorial. Cada nivel se compone de un punto de partida y una meta representada por unas escaleras. Mediante el uso de un control, el usuario debe conducir al gato desde el punto de partida hasta la meta para así avanzar al siguiente nivel. En este proceso, debe evitar los diversos obstáculos que se le presentan, ya que en caso contrario deberá intentar nuevamente el nivel desde el punto de partida. Tanto la cantidad de obstáculos como su disposición depende del nivel de dificultad en el que se encuentre el jugador, el cuál se determina mediante las variaciones de pulso.

El desarrollo de un videojuego que utilice el pulso como señal adicional de entrada no es una tarea directa. Dado el poco progreso de este campo dentro de la industria no existe una forma estándar de realizar un enlace directo entre las señales biológicas y los videojuegos. En particular, el obtener estas medidas implica tener que utilizar sensores especializados que las capturen.

Por un lado, si el foco es integrar señales biológicas en el campo de los videojuegos, no es realista ni aceptable tener que instrumentar de manera excesiva al jugador. El hecho de estar equipado con muchos sensores y cables durante una sesión de juego conduce de manera inmediata a una degradación en su experiencia de usuario. Por otro lado, dado que

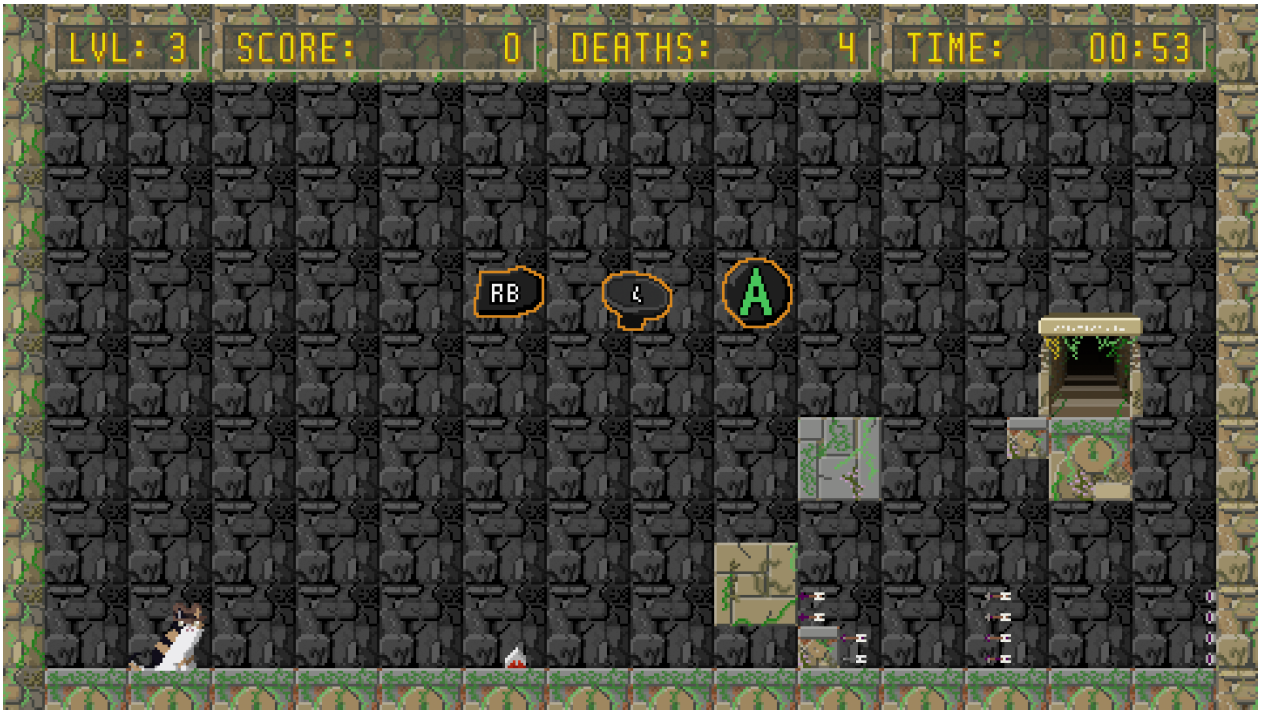


Figura 1.1: Tercer nivel del juego

los videojuegos pertenecen al mercado de masas, el alto costo adicional en que el jugador debería incurrir al tener que comprar estos sensores es un punto importante a considerar.

Dado lo anterior, el videojuego desarrollado en este trabajo de título toma el apronte de utilizar tecnología ya disponible en el mercado de masas—como monitores de actividad—integrado a una implementación acorde, como una manera de facilitar la adopción y uso del juego por parte de los usuarios. Así se evita desarrollar hardware personalizado que no asegure un resultado oportuno ni un costo bajo. Luego, el trabajo realizado constituye una prueba de concepto de que es factible y viable desarrollar un videojuego (potencialmente) comercial que utilice el pulso para controlar su dificultad, donde este ajuste sea usable y considerado como útil por el usuario.

Como el juego implementado en el contexto de este trabajo de título requiere de un sensor de pulso, fue necesario evaluar detenidamente qué tipo de sensor se debe utilizar, teniendo en consideración que no debe ser considerado como una molestia por el jugador y que pueda estar disponible en el mercado de masas. Por otro lado, fue necesario diseñar un algoritmo tal que, dadas las medidas de pulso obtenidas desde el usuario durante la sesión de juego, se pueda inferir cuándo se produce una alteración significativa que implique tener que modificar la dificultad del juego. Esta no es una tarea trivial, dado que las medidas—y, por ende, variaciones—del pulso no son las mismas para todas las personas.

Finalmente, la creación de un videojuego que permita darle un uso apropiado al pulso requirió de una buena planificación y un desarrollo minucioso, ya que la estructura de la aplicación debe permitir modificar la dificultad del juego sin que se pierda la esencia del mismo. Esto implica que no se puede utilizar de manera directa un videojuego ya existente, dado que los videojuegos de plataforma presentes en el mercado no fueron diseñados con un

ajuste de dificultad inmediato en mente, y al ser productos comerciales no exponen su código de manera libre para poder modificarlos. Crear un juego desde cero permitió una mayor flexibilidad a la hora de realizar pruebas y ajustes según fue necesario, así como también tener presente la calidad del código para así asegurar su mantenibilidad y facilidad de uso en algún trabajo futuro.

1.3. Objetivos

A continuación se presenta el objetivo general y los objetivos específicos a cumplir en este este trabajo de título.

Objetivo General

Construir un mecanismo que regule de manera automática la dificultad de un videojuego, potencialmente comercial, desarrollado utilizando tecnología de fácil acceso en el mercado de masas, basándose en las variaciones en el pulso del jugador, sin degradar la experiencia de usuario.

Objetivos Específicos

Para lograr el objetivo general expuesto en este trabajo de título, se definieron los siguientes objetivos específicos:

1. Seleccionar un sensor o método para medir el pulso asequible dentro del mercado de masas, es decir, su precio no debe ser mayor a la mitad del promedio de los precios de las consolas de videojuegos de las últimas dos generaciones, lo que corresponde aproximadamente a $199 dólares [5]. Este punto es importante ya que el sensor es utilizado para un videojuego comercial desarrollado y el precio es un factor importante a la hora del éxito o fracaso comercial de un producto [13]. Por otro lado, el sensor debe ser lo suficientemente preciso como para ser capaz de medir de manera correcta las variaciones de pulso en a lo menos 5 bpm respecto al estado de reposo. Este objetivo se logró realizando una comparación del precio de la solución escogida con los límites anteriormente establecidos, y para cuantificar la precisión se comparó los bpm obtenidos en determinado estado del jugador con los bpm reportados por un esfigmomanómetro digital.

2. Diseñar un algoritmo que detecte de manera correcta las variaciones significativas de pulso durante el videojuego, es decir, determine qué cambios en el pulso corresponden a una variación que deba modificar la dificultad y cuáles no. Se logró lo anterior mostrando el pulso del jugador y las variaciones evidenciadas como elementos de depuración de errores, indicando cuándo se produce un cambio a partir de estas.

3. Dar una respuesta a las variaciones de pulso del jugador, la cual debe hacerse presente a lo más en el siguiente nivel luego de detectar un cambio significativo en el pulso del jugador. Se logró esta meta mostrando el nivel de dificultad actual como un elemento de depuración de errores, verificando que esta cambia luego de haberse producido una variación en el pulso del jugador.

4. Modificar de manera efectiva la dificultad del juego, es decir, que tanto el nuevo número de obstáculos que aparecen en escena así como su distribución permitan que el número de reintentos promedio para completar un nivel sea directamente proporcional a la magnitud de la variación en el pulso del jugador. Esto se logró jugando un número determinado de veces las distintas variaciones de un nivel y comprobando que la proporción promedio entre partidas ganadas y los reintentos estuviera acorde al nivel de dificultad.

5. Alcanzar una mejor experiencia de usuarios al interactuar con el juego modificado con el ajuste dinámico de dificultad, respecto a la misma versión del juego pero sin regulación. Se logró este requerimiento mediante una medición del número de intentos y tiempo que el jugador tardaba en completar cada nivel, así como también la satisfacción reportada luego de jugar a la versión con o sin el ajuste de dificultad, obteniéndose mejores resultados en el grupo que usó el ajuste.

1.4. Metodología

El videojuego desarrollado pertenece al género de plataformas y fue implementado en C++ utilizando el motor de videojuegos *Unreal Engine 4*. Se eligió este género de videojuegos debido a su versatilidad en cuanto al uso de diferentes elementos dentro de un nivel. También porque, dependiendo de cómo esté construido, un nivel puede tomar muchos intentos antes de lograr terminarlo, lo que se conecta de manera directa con la medición de dificultad según el número de intentos utilizada en el juego implementado. La elección de Unreal Engine 4 se justifica en tanto que este es un motor estándar en la industria para el desarrollo de videojuegos [12], el cual utiliza C++ como lenguaje de programación.

A continuación se listan los pasos seguidos durante el desarrollo del videojuego, para así cumplir con los objetivos propuestos en el presente trabajo de título:

1. Se realizó un análisis comparativo entre sensores y métodos existentes para medir el pulso del jugador. Este análisis consideró la viabilidad de distintas alternativas existentes en el mercado, basándose en tres factores principales: precio, precisión y comodidad. El precio debe ser tal que no genere un mayor problema a los jugadores el adquirir la tecnología seleccionada. La precisión debe permitir que el juego pueda estar al tanto de los cambios del pulso en comparación a un estado de reposo. Finalmente, la comodidad implica que no sea una molestia utilizar el sensor durante el juego, lo que podría romper la inmersión o hacer que el jugador evite volver a jugar para no tener que utilizar el sensor nuevamente. Por otro lado, el análisis también consideró la complejidad del proceso de obtener el valor del pulso y hacerlo llegar mediante C++ a Unreal Engine, para

así ser ocupado como variable dentro del juego. Por ejemplo, se privilegiaron aquellas alternativas que exponen una API ya implementada, frente a opciones que requieren implementar la interfaz de comunicación desde cero. Como aproximación al problema, se estudiaron desde un punto de vista técnico sensores de pulso que incorporan los monitores de actividad, smartwatch, bandas de pecho y sensores de pulso ópticos.

2. Se integró el sensor escogido con Unreal Engine a través de C++, comenzando por mostrar en pantalla el pulso medido por el sensor como un elemento de depuración, ya que el juego final no le muestra su pulso al jugador. Lograr lo anterior implicó la posibilidad de poder utilizar el valor del pulso como variable dentro del juego.

3. Se desarrolló un módulo que toma el pulso del jugador y realiza los cálculos necesarios para determinar cuándo se produce una variación significativa que deba ser considerada para la modificación de la dificultad. En etapas iniciales del desarrollo, dicho resultado se mostró en pantalla como un elemento de depuración de las variaciones detectadas.

4. Se diseñó e implementó un videojuego cuya dificultad es modificada por el pulso. El diseño del juego siguió un proceso centrado en el usuario, con foco en maximizar la jugabilidad del mismo [9]. Así pues, éste se divide en 12 niveles, en los cuales el jugador debe llegar desde un punto de partida a una meta sin morir en el proceso. De perder, se debe empezar nuevamente desde el punto de partida. Los niveles ocupan sólo una pantalla, de tal manera que no pasa mucho tiempo entre que el jugador comienza el nivel y lo completa o muere. Que los niveles no sean muy largos es fundamental para que, de esta manera, se pueda entregar una pronta respuesta a los cambios en el pulso del jugador. Se decide no modificar de forma dinámica el nivel actual que se está jugando para no introducir confusión innecesaria en el jugador, por lo que la nivelación de dificultad se realiza luego de que el jugador pierde y debe intentar nuevamente el nivel, o en el siguiente nivel si es que gana el actual.

5. Se incorporaron nuevos elementos con los que interactuar y nuevas forma de utilizar las habilidades del personaje a medida que el jugador avanza en los niveles para así mantener su nivel de interés. Esto incluye nuevos tipos de obstáculos y variaciones en la estrategia necesaria para completar el nivel.

6. Se agregó un argumento de consola que permite desactivar el efecto que tiene el pulso, dejando fija la dificultad en un estado medio. En contraste, la modificación realizada a partir del pulso permite abarcar el rango de dificultad desde fácil, pasando por medio y finalmente difícil dependiendo de las variaciones de pulso que experimenta el jugador.

7. Se crearon 5 etapas iniciales extra a modo de tutorial que introducen el juego a los usuarios, mientras se realiza en paralelo un proceso de calibración para obtener una medida de pulso base a utilizar para detectar posteriores variaciones. Si bien este pulso base se puede calcular nuevamente a lo largo del juego, tener una fase de calibración permite tener una medida de control sobre la cual realizar los cálculos para los niveles posteriores.

8. Se validó el efecto del ajuste dinámico de dificultad dentro del videojuego a través de

una comparación de la experiencia de usuario percibida por un grupo de jugadores enfrentados al juego con y sin el ajuste. Se utilizaron como métricas el número de intentos utilizado por los jugadores para completar el juego, el tiempo usado en completar los niveles y el grado de satisfacción reportado por los usuarios mediante la aplicación de una encuesta, analizando las curvas de pulso para ambos casos. Como instrumento de medición se utilizó el *Game Experience Questionnaire* [16], un cuestionario ampliamente usado en el área de Interacción Humano-Computador en Juegos, para medir experiencias de usuario en este dominio de aplicación [17].

Al realizar lo anterior se logró completar con los objetivos propuestos, desde: (1) definir una forma de medir el pulso, (2) integrarla a Unreal Engine, (3) detectar las variaciones significativas, (4) utilizarlas de manera eficiente y efectiva, y finalmente (5) mejorar la experiencia de usuario del jugador. En particular, el primer paso permitió cumplir el objetivo específico #1, el segundo y tercero logran el objetivo específico #2, el juego descrito en el cuarto paso cubre las expectativas de los objetivos específicos #3 y #4, apoyándose también este último en el quinto paso, en tanto los pasos restantes apuntan al objetivo específico #5.

1.5. Estructura del documento

El resto de este documento se estructura como sigue. En el capítulo 2 se presenta y discute la literatura relacionada que sustenta el trabajo ejecutado en esta memoria de título. El capítulo 3 está dedicado a presentar el diseño de los elementos del juego desarrollado. A continuación, en el capítulo 4 se describen las decisiones de diseño e ingeniería tomadas respecto a: (1) la selección del dispositivo de monitoreo de pulso, (2) el desarrollo de un protocolo de comunicación entre el dispositivo y el juego, y (3) el algoritmo de ajuste dinámico de dificultad. Por su parte, el capítulo 5 presenta la implementación de los distintos elementos que componen el juego. En el capítulo 6 se presentan los resultados de una prueba de concepto para verificar el cumplimiento de los objetivos propuestos al inicio de este trabajo. Finalmente, en el capítulo 7 se presentan las conclusiones y se ofrecen perspectivas de trabajo futuro.

Capítulo 2

Trabajo Relacionado

Dado que el biofeedback aún no es una técnica comúnmente usada en la industria de los videojuegos, hay muy pocos ejemplos de aplicaciones que utilicen un sensor de pulso para alterar de alguna manera la jugabilidad. *Bring to Light* [4] es un juego de terror que fue lanzado en julio de 2018, cuyo objetivo principal es asustar al jugador. Así, se introduce el pulso como una forma de medir cuán asustado está el jugador, de tal manera que si está muy relajado se elevará el grado de miedo que produce el juego para mantener al jugador en un estado de tensión constante. Sin embargo, a pesar de que el ajuste de jugabilidad según el pulso se promueve como uno de los puntos fuertes del juego, esta funcionalidad es completamente opcional, estando limitada sólo al uso de sensores compatibles con la tecnología Ant+, la cual corresponde a un protocolo de comunicación inalámbrica diseñada para dispositivos de monitoreo.

En 2009, Nintendo anunció el periférico *Wii Vitality Sensor* para la consola Nintendo Wii [25]. Este periférico consiste en un oxímetro que permite medir el pulso, a partir del cual se extrae información como qué tan nervioso o enfocado está el jugador en determinado momento. A pesar de las características anunciadas, junto con una visión de que los juegos podrían utilizarse como una forma de relajación, no se mostró ninguna aplicación concreta que fuera compatible con este sensor. Posterior a su presentación, no se volvió a mencionar a este dispositivo sino hasta el 5 de julio de 2013 donde se confirmó su cancelación [25].

Por su parte, Ubisoft sacó al mercado, el año 2015 en Francia, un oxímetro de pulso llamado *O.zen* [13]. Este sensor fue vendido a un costo de € 99.99 y sólo es compatible con una aplicación de iOS del mismo nombre, la que consiste en 14 minijuegos enfocados a controlar el nivel de estrés de un jugador a partir de ejercicios de respiración. Sin embargo, en la actualidad, las valoraciones de esta aplicación en la AppStore sólo llegan a 2,7 de 5 estrellas [34]. Esto se debe principalmente al alto costo del sensor, teniendo en cuenta que sólo puede ser utilizado con el juego anteriormente mencionado.

Respecto a métodos actuales para medir el pulso, hoy en día existe una amplia variedad de dispositivos como monitores de actividad y smartwatches [19], por ejemplo, la pulsera *Mi Band* [37] destaca por su bajo costo y accesibilidad, así como las pulseras de la marca *Fitbit* exponen una API web pública [8], facilitando su comunicación con otras aplicaciones

de software externas. Por otro lado, también es posible medir el pulso con sólo una cámara y una fuente de luz, lo cual implica que cualquier smartphone podría utilizarse como un sensor de pulso [22].

En un contexto de laboratorio, Nenonen et al. [23] utilizan el pulso para controlar un juego, demostrando así la factibilidad técnica de poder realizar esta tarea. Asimismo, Drachen et al. [6], mediante un estudio en el que utilizan tres videojuegos de disparos en primera persona, comprueban que existe una relación entre señales biológicas como el pulso y la experiencia de usuarios en el juego. Por otro lado, Liu et al. [18], utilizan el nivel de ansiedad como factor determinante en el manejo de la dificultad de un videojuego. La ansiedad es estimada mediante el análisis de una gran cantidad de señales biológicas como actividad y ruido cardiaco, bioimpedancia, actividad electromiográfica, entre otras, para lo cual se requirió tener equipo especializado. Debido a esto, no es posible transferir de manera directa a la industria de videojuegos los resultados de su estudio, ya que por un lado el jugador debería utilizar una gran variedad de sensores distintos perturbando su experiencia de juego, y por otro lado el costo que implicaría poseer todos estos sensores sería muy elevado, considerando que los sensores deberían además ser bastante precisos.

En resumen, el bajo uso de *biofeedback* en la industria de los videojuegos se ve justificado por una variedad de periféricos que no han alcanzado una adopción efectiva por parte de usuarios finales, ya sea por su dificultad de uso o elevado costo. Asimismo, el estado del arte indica que sí es posible lograr esta interacción, pero en un escenario de laboratorio y usando equipamiento de alta complejidad técnica. Inspirándose en todo este conocimiento previo, el presente trabajo de título pretende desarrollar una prueba de concepto que permita integrar tecnología de amplia penetración (como smartphones o monitores de actividad de bajo costo), para regular de manera automática la dificultad ofrecida por un videojuego a desarrollar, y así mejorar su experiencia de usuarios.

Capítulo 3

Diseño del Juego

El juego desarrollado en este trabajo de título pertenece al género de plataformas. Esto es, se tiene un personaje con la habilidad de moverse y saltar, el cual se desplaza a través de un nivel compuesto de plataformas. El objetivo principal del juego es que el usuario debe completar algún objetivo, evitando los obstáculos que se encuentre.

Para facilitar tanto la creación de elementos visuales como la implementación del juego, se determina que este será en 2 dimensiones (2D). Como estilo gráfico se adopta el estilo *pixel art*, es decir, tanto el personaje como el entorno en que este se desenvuelve y los elementos con los que interactúa son representados por imágenes o *sprites* consistentes de pocos píxeles, los cuales son claramente distinguibles unos de otros. La definición de los componentes principales del juego adhiere a las recomendaciones propuestas por Jesse Schell [29], un diseñador de videojuegos ampliamente reconocido por la comunidad académica y profesional.

3.1. Componentes principales

El juego se encuentra ambientado en un entorno de ruinas con temática de búsqueda de tesoros y posee tres elementos fundamentales: los niveles, el personaje y los obstáculos. Los componentes gráficos del juego fueron diseñados por el autor de este trabajo de título.

3.1.1. Diseño de niveles

Se desarrollaron un total de 12 niveles más 5 niveles iniciales a modo de tutorial. Estos últimos, además de servir para introducir las mecánicas principales al jugador, también sirven para acumular datos de su pulso, los cuales serán utilizados en niveles posteriores para ajustar de manera oportuna la dificultad. Un nivel ocupa sólo una pantalla, de manera que el jugador pueda ver al mismo tiempo todos los elementos presentes, siendo capaz de planificar una estrategia para completar el nivel mientras evita los obstáculos. El tamaño de un nivel es de 14 bloques horizontales por 7 verticales, donde cada bloque se constituye por un

espacio de 32 por 32 píxeles. La idea es que los elementos estructurales del nivel, así como los obstáculos y el mismo personaje no sean más grandes que el tamaño de un bloque. Como la resolución objetivo del juego es 1920x1080 se utiliza un zoom de 400 % para que así se pueda ver correctamente el nivel. En la figura 3.1 es posible observar los cuatro bloques que son utilizados para dar estructura a cada nivel, así como también brindan los bordes que delimitan el espacio de juego.

Figura 3.1: Bloques estructurales del juego

Cada nivel posee un punto de partida, donde es posicionado el personaje cuando comienza el nivel o cuando muere y debe intentarlo de nuevo. También posee escaleras que permiten pasar al siguiente nivel, las cuales se muestran en la figura 3.2. Para progresar dentro del juego se debe llegar desde el punto de partida hasta las escaleras sin morir, debiendo presionar un botón determinado para avanzar al siguiente nivel mientras se encuentra sobre el lugar en que se ubican las escaleras. La posición, tanto del punto de partida como de las escaleras, se mantiene constante a lo largo del nivel, no siendo afectada por el ajuste de dificultad para así mantener una experiencia consistente para el jugador. De esta manera se busca que el usuario tenga claro los extremos de la ruta que debe seguir para completar el nivel.

Figura 3.2: Escaleras de cambio de nivel

Si bien se utiliza un bloque como unidad para el diseño de un nivel, no se limita el posicionamiento de objetos ni el movimiento del personaje a una grilla compuesta de bloques. Sin embargo, sí se restringe la posición de objetos de tal manera que se mantenga la consistencia del diseño escogido. Es decir, se busca que los píxeles de cada elemento estén alineados entre sí, ya que en caso contrario se producirían inconsistencias en el proceso de escalamiento y

despliegue (*rendering*) del juego. Sólo el personaje puede no estar alineado a los píxeles del nivel, ya que así se permite que el jugador experimente un movimiento continuo, brindándole una mayor precisión y una mejor retroalimentación visual de los controles de entrada (i.e., *inputs*) que está efectuando. Esto es en contraste a tener un movimiento discreto, en cuyo caso, si bien los píxeles del personaje estarían alineados a los demás elementos del nivel, si el jugador se moviera muy levemente el personaje no mostraría ningún movimiento aparente.

3.1.2. Diseño de la interacción

En cuanto a la interacción del jugador, esta se diseñó utilizando como base un mando con palancas analógicas, como los de las consolas Xbox o PlayStation. Se implementó de igual manera la posibilidad de utilizar teclado, sin embargo este sólo es utilizado para testeo, permitiendo de esta manera que todos los usuarios sean capaces de vivir la misma experiencia. Así, el tipo de interacción con el juego no constituye un factor variable dentro de las pruebas realizadas.

Para la disposición de los inputs en el mando se adopta un diseño convencional, estando el movimiento del personaje a cargo de la palanca izquierda, su salto en los botones de la derecha y su capacidad de correr en los botones superiores o traseros.

3.1.3. Diseño del personaje

Como personaje principal del juego, se creó un gato que viste un sombrero al estilo de cazador de tesoros como se observa en la figura 3.3. Posee tres modos de movimiento que le permiten progresar dentro de un nivel y llegar a las escaleras para pasar al siguiente: caminar, saltar y caminar por el muro de fondo.

Figura 3.3: Personaje principal

El modo de caminar, el cual junto al salto componen las bases de un juego de plataformas, posee una velocidad que es dependiente de los inputs que utilice el jugador. Existen dos fases distinguibles: caminar y correr. Si bien caminar requiere de la utilización de un sólo input, correr requiere dos, para de esta manera ser claramente diferenciados y ofrecerle un mayor control al jugador. También, el poseer velocidad variable abre la posibilidad de diseñar niveles

cuya estructura y posicionamiento de obstáculos requieran de determinada precisión en la velocidad del personaje para así ser completados.

Las velocidades de caminar y correr se designan de manera que sean distinguibles una de otra. Se acompañan con un cambio en la velocidad de animación del personaje, así como también con la adición de un efecto visual extra. De esta forma el jugador puede notar claramente en qué estado se encuentra. Los valores específicos se establecen de acuerdo a la capacidad que se quiere dar al personaje cuando se enfrenta con los distintos obstáculos del juego. También se tiene en consideración el tiempo mínimo que le tomaría recorrer todo un nivel de inicio a fin si no existieran obstáculos, cuidando que este no sea muy largo, provocando que el jugador se aburra. Asimismo, este tiempo tampoco puede ser muy corto, haciendo que el jugador no pueda experimentar cada nivel en su completitud.

El modo de salto posee sólo un estado, el cual es ejecutado mediante un input por parte del usuario, donde el efecto depende del tiempo que el jugador mantiene presionado el input. Este modo de movimiento consiste en la aplicación de una fuerza inicial vertical sobre el personaje. Este valor es luego corregido por un factor de gravedad, a partir de la cual se calcula la velocidad y siguiente posición dentro del nivel. Esta iteración ocurre hasta detectar alguna colisión con una superficie sobre la que se puede caminar, procediendo a cambiar de modo. La corrección sobre el efecto de salto corresponde a sólo una fracción de la última fuerza aplicada, hasta que esta se vuelve tan pequeña que deja de ser significativa, lográndose así un salto preciso, continuo y suave.

La velocidad horizontal inicial del salto corresponde a la misma que el personaje poseía en el anterior modo de movimiento. Una vez en el aire, el jugador posee completo control sobre la velocidad horizontal de la misma manera que en el modo anterior. Si bien esto no es realista, corresponde a un estándar dentro de los juegos de plataforma.

Finalmente, está el modo de caminar por el muro de fondo, en donde el personaje mantiene la velocidad horizontal que posee en el aire durante un período de un segundo. Este modo sólo puede ser utilizado una vez mientras se está en modo salto, pudiendo volver a usarse luego de pasar por el modo caminar. Durante el segundo de duración, si bien no es posible aumentar la velocidad de movimiento, sí lo es disminuirla. Esta última acción se ejecuta dependiendo de si el jugador tenía presionado el input para correr y lo suelta, o bien si modifica la manera en que utiliza el input para caminar, ya sea cambiando de dirección o reduciendo la distancia en un control análogo. Mientras el personaje camina por el muro no se ve afectado por la gravedad, por lo que es una manera confiable de pasar por zonas estrechas o también para sortear obstáculos muy largos en distancia. Se agrega este modo para tener una mecánica propia del juego y así hacerlo más novedoso para el usuario. De igual modo, no se agregan más mecánicas distintas para que le sea más fácil al jugador asimilar las habilidades del personaje y así tener un control más preciso y eficiente sobre éste.

3.1.4. Diseño de obstáculos

En cuanto a los obstáculos, se decide que sólo existan tres, para que así el jugador sea capaz de comprenderlos fácilmente a lo largo de los niveles. De esta manera se evita introducir

dificultad extra (de manera innecesaria) al usuario, la cual pueda afectar los resultados de las pruebas realizadas posteriormente.

El primer obstáculo corresponde a una púa de metal, pudiéndose observar cuatro de estas en la figura 3.4. Si el personaje colisiona con una púa se considera como una muerte y debe empezar nuevamente el nivel. Este tipo de obstáculos ocupa sólo un cuarto del área de un bloque, ya que así se pueden ubicar de mejor manera dentro de un nivel. Luego, resulta posible ir variando su cantidad, dependiendo de qué tan preciso se requiere que sean los movimientos del jugador a fin de evadirlas. Al ser un obstáculo simple, se puede manipular fácilmente su cantidad y posicionamiento, de tal manera de crear un único nivel que posea distintas configuraciones de púas que le impliquen diversos grados de desafío al jugador.

Figura 3.4: Obstáculo: púas

Un ejemplo de utilización de este obstáculo corresponde a poner una determinada cantidad de púas una al lado de la otra sobre el piso. La cantidad de púas que se utilice será proporcional a la precisión que debe tener el jugador para evadirlas, y también influirá en la estrategia que éste utilice, ya sea simplemente saltar, saltar mientras corre o caminar por el muro. Así pues, el agregar o quitar una sola púa, aumentaría o disminuiría la dificultad del obstáculo respectivamente, permitiendo crear transiciones fluidas entre una dificultad y otra, las cuales pueden pasar desapercibidas para el usuario.

El segundo obstáculo está constituido por dos elementos: (1) un punto de partida que posee una dirección y (2) flechas. Luego de un tiempo inicial, desde el punto de partida se crean periódicamente flechas que se mueven en línea recta en la dirección indicada y a una velocidad constante previamente especificada. En la figura 3.5 se observan cuatro puntos de partida junto a cuatro flechas que ya han sido creadas, y por lo tanto, moviéndose hacia la derecha. En este caso no existe colisión entre el personaje y los puntos de partida, por lo que este puede tocarlos o caminar sobre ellos sin algún tipo de penalización.

Sin embargo sí se define un punto de colisión con las flechas que son lanzadas. De la misma forma que con las púas, si el personaje colisiona con las flechas se considera como una muerte y debe volver a empezar el nivel. Cada flecha posee aproximadamente un cuarto del área de un bloque, por lo que se pueden posicionar de manera precisa para limitar o influenciar los movimientos del jugador como se desee, por lo que también constituyen un buen obstáculo que puede ser modificado de tal manera de aparentar un cambio de dificultad continuo.

Figura 3.5: Obstáculo: flechas

Como tercer y último obstáculo se encuentran las piedras, mostrándose una en la figura 3.6. Estas poseen un área de un cuarto de un bloque, y si bien son clasificadas como obstáculos, por sí solas no son capaces de hacer que el jugador muera. En cambio, es su posicionamiento estratégico el que limita los movimientos del personaje de tal manera que sea más probable que este colisione con una púa o flecha. Por otro lado, a pesar de ser un obstáculo, también son utilizadas para facilitar el desplazamiento del personaje dentro del nivel, reduciendo por ejemplo la distancia necesaria para un salto. A diferencia de los bloques estructurales del nivel, la presencia de las piedras depende directamente de la dificultad en la que se encuentre jugando el usuario, apareciendo o desapareciendo de acuerdo a ésta.

Figura 3.6: Obstáculo: piedras

El dinamismo y flexibilidad que ofrece la existencia de las piedras permite crear por completo un nivel que no utilice bloques estructurales más allá de los que cumplen el papel de límites. Esto permite lograr una experiencia totalmente guiada por la dificultad y los cambios que ésta experimente mientras el jugador continúe intentando el nivel. Esta idea es utilizada en tres de los niveles desarrollados.

3.1.5. Diseño de narrativa

Pasando a la historia del juego, ésta nos presenta al gato calicó de la figura 3.7 viendo un programa de televisión, en el cual se muestra a un cazador de tesoros en una expedición

dentro de unas ruinas. En esta expedición el cazador se encuentra con una caja de cartón, la cual contiene un tesoro valioso. El gato se encuentra muy interesado en el programa y se entusiasma al ver al cazador con el tesoro. Sin embargo, a pesar de las apariencias, el gato no está interesado en el tesoro, sino en la caja en que este se encontraba. Es así como el gato se escapa de su casa a través de una ventana y comienza a caminar en busca de las ruinas. Su viaje lo lleva a diversos lugares, como un bosque y un desierto, hasta que finalmente llega a unas ruinas en cuya entrada se encuentra un cartel con la imagen del tesoro que estaba dentro de la caja. Antes de embarcarse en esta aventura, el gato nota la presencia de un sombrero de aventurero detrás de una roca cercana a la entrada. Se lo pone y entra a las ruinas. Sin embargo luego de entrar, el cartel se cae y revela un símbolo de peligro, indicando así lo inseguras que son las ruinas. En el Apéndice A se presenta, de manera gráfica, el detalle de la historia del juego.

Figura 3.7: Personaje principal dentro de la historia

La presencia de una historia para el juego es importante debido a varios aspectos [29]. El primero consiste en la completitud del juego, ya que es necesario un contexto para que los acontecimientos que ocurren dentro del juego tengan sentido. De igual manera la historia permite la existencia de un punto de término para el juego, presentándose este como la llegada del gato a la caja y logrando así que el jugador se sienta recompensado por todo el esfuerzo realizado al guiar el personaje a través de los diversos niveles. También la historia crea un vínculo entre el personaje y el jugador, infundiendo en este la motivación necesaria para afrontar los niveles con la esperanza de que el gato logre su objetivo. Finalmente, al igual que con los niveles de tutorial, mientras el jugador está viendo la historia, se está tomando su pulso, contribuyendo estas medidas al cálculo del pulso en estado de reposo que es utilizado como base para el ajuste de dificultad en los niveles posteriores.

3.2. Mecánica del juego

El juego posee un total de 17 niveles, donde 5 son utilizados como tutorial para introducir las mecánicas disponibles al jugador. Los 12 niveles restantes están afectos a un ajuste dinámico de dificultad en función de las variaciones en el pulso del jugador, implementado en esta memoria.

La dificultad del juego se mide como un número entre 0 y 1, donde 1 representa la mayor dificultad disponible. Dependiendo de la dificultad actual, los obstáculos pueden o no estar presentes dentro de un nivel. Cada obstáculo posee un número que representa su dificultad, y su aparición o desaparición se rige mediante dos modos distintos. El primero es el más intuitivo y relaciona de manera directa la dificultad de un obstáculo con su presencia en el nivel. Por ejemplo, si un obstáculo posee dificultad 0.7, este sólo aparecerá si la dificultad en la que se encuentra el jugador es mayor a igual a este valor. En contraste, el segundo modo relaciona la dificultad de manera inversa, es decir, si un obstáculo tiene dificultad 0.3, este será visible mientras la dificultad del jugador sea menor o igual a este número. Este último modo es principalmente utilizado por las piedras, permitiendo que estas aparezcan sólo en niveles inferiores de dificultad, siendo posicionadas de manera tal que permiten ayudar al usuario a progresar dentro del nivel en vez de entorpecer sus movimientos.

El diseño general de cada nivel se centra en enfatizar ya sea el uso de una mecánica en particular, o bien, algún tema recurrente como la presencia del mismo tipo de obstáculo o de la ejecución de la misma combinación de inputs en reiteradas ocasiones para completar el nivel. Durante la etapa de diseño no existió un orden en particular para la creación de los niveles, sino que fue un proceso de definir una mecánica o tema que pudiera ser interesante para el jugador, y luego construir un nivel alrededor de esta. Cada nivel es creado primero en su dificultad estándar, la cual corresponde al valor 0.5. Este valor se mapea de manera tal que el nivel utiliza la cantidad de obstáculos necesaria para crear un nivel donde cualquier jugador promedio—con menor o mayor esfuerzo—sea capaz de completarlo satisfactoriamente, sin utilizar mucho ni muy poco tiempo en el proceso.

Para hacer factible la presencia de una dificultad dinámica, luego de crear un nivel se procede a crear variaciones del mismo, eliminando o agregando obstáculos. En general se crearon 10 o más por cada nivel, procurando que exista al menos una variación por cada incremento de 0.1 en dificultad. También se tuvo en cuenta que entre dos variaciones los cambios no deben ser muy significativos, es decir, si el jugador no está prestando demasiada atención a la estructura del nivel, no le debería ser fácil notar que éste ha sido modificado. Sin embargo, este cambio sí debe impactar de manera directa, ya sea al tiempo que le toma el jugador completar el nivel y/o a la cantidad de intentos requerida para el mismo propósito.

Se opta por este tipo de aproximación a la hora del diseño de los niveles, ya que así se logra crear un juego que aparente ser continuo y suave para el jugador. En cambio, si el jugador de la nada evidencia la presencia de cambios abruptos dentro del mismo nivel, por un lado rompería su inmersión dentro del juego, y por otro lo haría más consciente de la relación directa entre su pulso y la dificultad que está experimentando. Esto puede llevarlo a intentar alterarlo de manera artificial para así conseguir una ventaja injusta a lo largo de los distintos niveles.

Para dar cuenta de la metodología de diseño utilizada para los niveles del juego, a continuación se presentan cuatro ejemplos. En cada caso se explica el concepto sobre el cual se estructura el nivel, qué es lo que se espera que el jugador realice para completarlo y el porqué de su existencia en el contexto del juego. También se ejemplificará cómo la dificultad varía dentro de un nivel en particular, pudiendo observarse cómo a través de la cantidad y certero posicionamiento de obstáculos es posible crear distintas experiencias de juego mientras se

mantiene la esencia base del nivel. En otras palabras, siguen estando presentes los desafíos principales, sin embargo completarlos es más fácil o difícil dependiendo del caso. En el Apéndice B se presentan, gráficamente, cada uno de los niveles del juego en tres dificultades: 0, 0.5 y 1, correspondiendo así a la menor dificultad dentro del juego, la dificultad media en que se espera que cualquier jugador sea capaz de completar, y la máxima dificultad diseñada.

3.2.1. Ejemplo #1: nivel 9

El esqueleto de este nivel se puede observar en la figura 3.8. La idea detrás del nivel es simple: explorar la existencia de las distintas velocidades de movimiento horizontal, por lo que se presentan dos principales obstáculos.

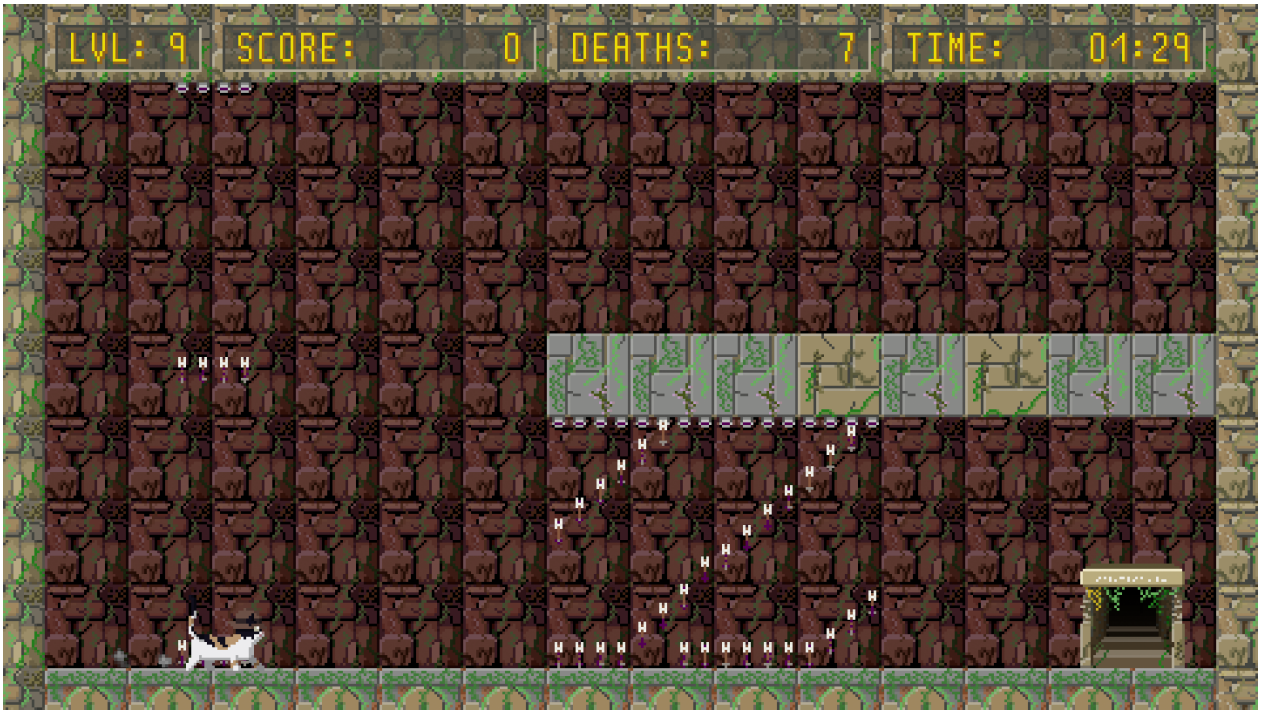


Figura 3.8: Nivel 9 – dificultad 0.5

El primero se compone de cuatro flechas que son lanzadas a gran velocidad desde el techo, cuya localización y frecuencia de generación está calculada para que el jugador no pueda atravesarlas si no corre; de pasar caminando, alcanzarán al personaje y tendrá que intentar nuevamente el nivel. El segundo obstáculo son flechas que también aparecen desde el techo. Sin embargo, la altura de su creación es mucho menor, al igual que su velocidad y frecuencia de creación, estando estos parámetros configurados de tal manera que el jugador pueda cruzar fácilmente si pasa caminando, pero no así si corre o salta. Se espera que el este sea capaz de notar el requerimiento de usar diferentes velocidad de movimiento a partir de la velocidad y disposición de las flechas. Así, se busca que el jugador que tome mayor conciencia de la existencia de esta mecánica, la cual le puede resultar útil para los siguientes niveles.

3.2.2. Ejemplo #2: nivel 17

Como segundo ejemplo se puede observar el nivel final del juego en la figura 3.9. Este se centra en la mecánica del salto y requiere que a lo largo del juego el usuario haya comprendido en su totalidad que dependiendo de cuánto deja presionado el input para saltar, la altura que alcance el personaje será mayor o menor.

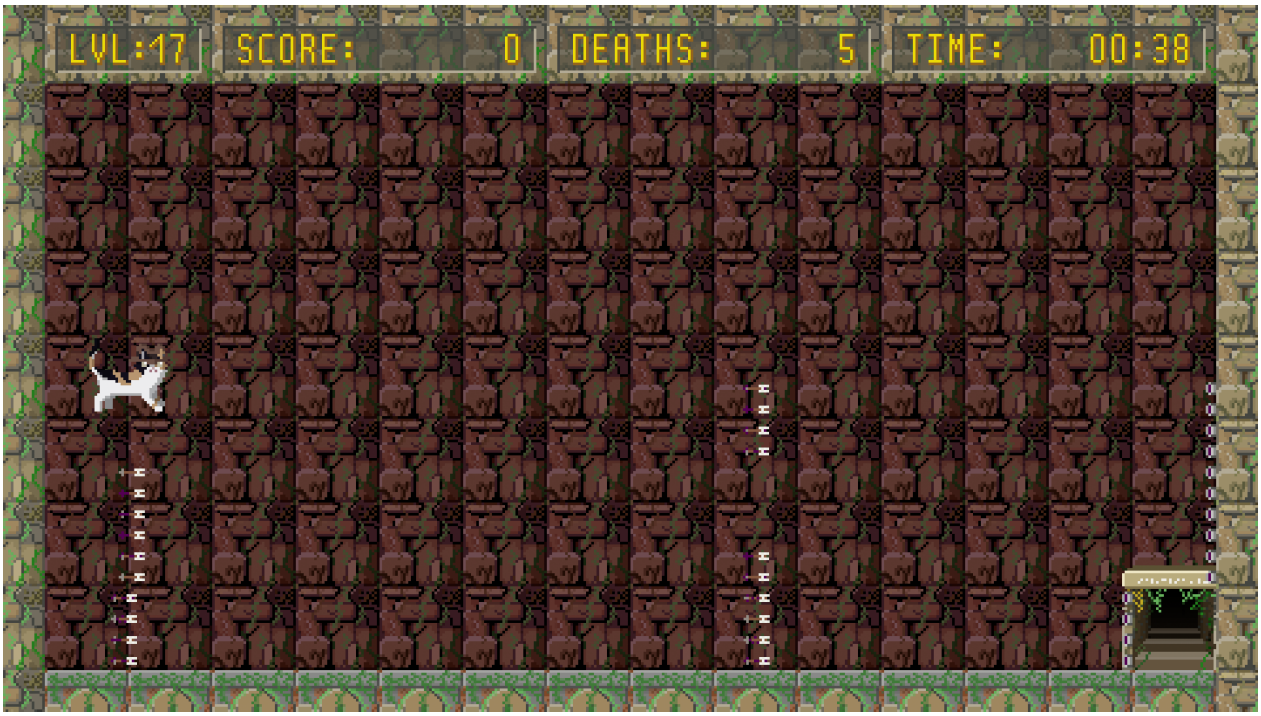


Figura 3.9: Nivel 17 – dificultad 0.7

Este nivel presenta dos ciclos de flechas que forman barreras móviles, a través de las cuales el jugador debe pasar mientras avanza hacia las escaleras. La altura del primer ciclo de flechas fuerza a que el jugador ejecute un salto alto, es decir, deje apretado el input de salto la mayor cantidad de tiempo posible. El segundo ciclo de flechas, por su lado, posee dos grupos de flechas. Éstas se encuentran posicionadas de manera que el jugador deba ejecutar un salto bajo, es decir, soltar el input de salto apenas es presionado; de lo contrario, chocará con el grupo de flechas superior, teniendo que iniciar nuevamente el nivel. Tanto la frecuencia como el orden de aparición de ambos ciclos es constante a lo largo del nivel, por lo que la cantidad de veces que el jugador deba realizar un salto alto seguido de uno bajo sólo dependerá de cuánto demore en llegar hacia las escaleras. De esta forma puede mantener un ritmo constante mientras avanza.

3.2.3. Ejemplo #3: nivel 11

El desafío principal de este nivel corresponde a pasar entre los pilares del centro, tal como se observa en la figura 3.10. Aquí el jugador debe sacar provecho de su habilidad para

caminar por el muro de fondo, teniendo que calcular el momento preciso para accionar el input correspondiente.

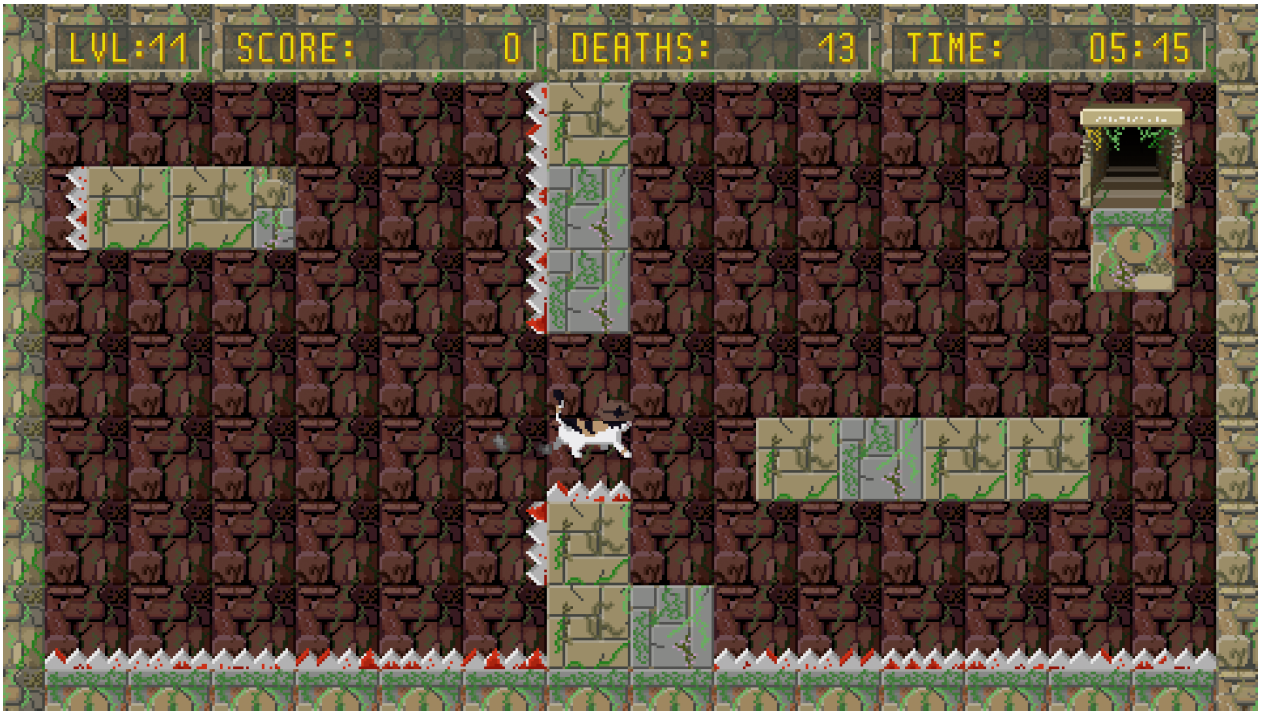


Figura 3.10: Nivel 11 – dificultad 0.4

Si bien este nivel muestra un desafío menor que el esperado para un jugador promedio, ya que corresponde a su variación de dificultad 0.4, se elige como ejemplo ya que permite evidenciar la presencia de piedras, las cuales se encuentran en la parte derecha de la primera plataforma. Estas piedras usan el segundo modo de generación de obstáculos, es decir, sólo aparecen si la dificultad del nivel es igual o inferior a 0.4, siendo de esta forma no un obstáculo en sí, sino una ayuda para los jugadores que están teniendo dificultades al completar el nivel. De esta manera, proporcionan un largo mayor a la primera plataforma, ofreciendo así más tiempo disponible para estar en el aire. Luego, se vuelve más sencillo el atravesar por los pilares centrales al no terminarse tan antes el tiempo de caminar por el muro y, por lo tanto, el personaje no cae directo a las púas.

3.2.4. Ejemplo #4: nivel 14

En la figura 3.11 se presenta un ejemplo de uno de los niveles que no posee ningún bloque estructural, estando construido en su totalidad por obstáculos. De esta manera, se hace posible la creación de variantes que poseen mayor continuidad para el usuario a lo largo de las distintas dificultades.

A diferencia del ejemplo anterior, aquí se puede ver un uso de las piedras como obstáculos en vez de como ayuda para el jugador. Su posición limita la altura máxima a la que el personaje puede saltar, provocando así que termine más fácilmente sobre las púas y teniendo por lo tanto que intentar nuevamente el nivel. Dada la estructura del nivel, se puede variar de

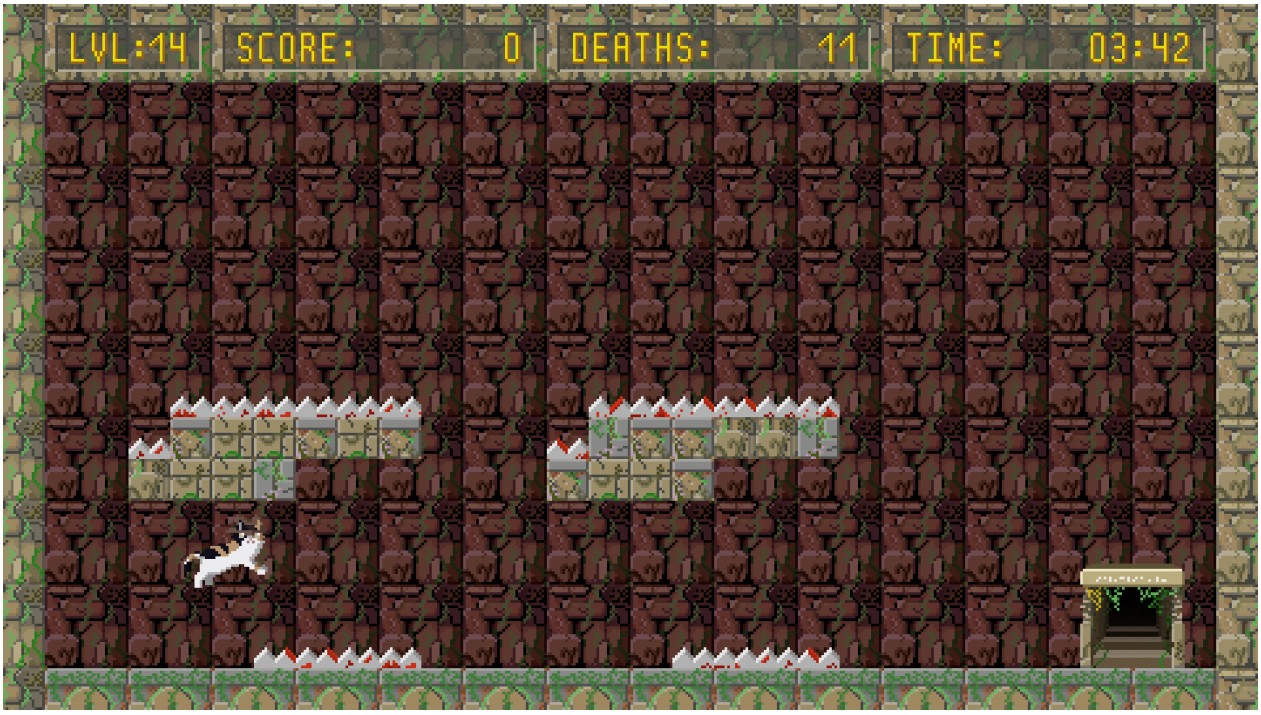


Figura 3.11: Nivel 14 – dificultad 0.5

manera directa su dificultad, tanto extendiendo la cantidad de púas ubicadas en el piso como incorporando nuevas pierdas que limiten aún más la capacidad de salto del personaje. Así, se fuerza al jugador a tener una mayor precisión en su movimiento y a tener eventualmente que caminar por el muro para lograr llegar de un lado a otro. Ya que el primer salto puede ser complicado, dependiendo de la habilidad el jugador y de la dificultad en que se esté jugando actualmente, el segundo salto es exactamente el mismo y no presenta dificultad agregada, permitiendo también mantener la consistencia en el desafío impuesto al usuario para superar el nivel.

En cada uno de los cuatro ejemplos mostrados anteriormente se puede observar cómo su creación está planificada para enmarcar el uso de alguna mecánica particular, o bien, enfrentar al jugador a alguno de los obstáculos de forma creativa. De esta manera se logra, por un lado, tener poca variedad de elementos presentes a lo largo de los niveles, pero al mismo tiempo un gran número de posibilidades de creación de niveles que capten la atención del usuario.

3.2.5. Ejemplo #5: nivel 8

En la figura 3.12 se muestra en detalle tres de las variantes del octavo nivel del juego. Se presentan tanto la dificultad estándar, que corresponde a la que cualquier jugador debiera ser capaz de completar, como la dificultad mínima y máxima, correspondiendo de arriba hacia abajo a los valores 0.0, 0.5 y 1.0. Se elige este nivel ya que muestra de manera clara las diferencias presenten en las distintas variaciones creadas para un nivel, mostrando cómo ajustes en la cantidad y posición de obstáculos logran influenciar el tiempo y cantidad de

intentos requeridos para completarlo.

Este nivel está construido de tal manera que dependiendo de la dificultad, la estrategia necesaria para completarlo cambie en cada caso. Sin embargo, en cada una de sus variaciones se mantiene constante el mismo desafío, el cual corresponde a saltar sobre los primeros dos grupos de púas, pasar de la parte superior del nivel a la inferior a través de las púas ubicadas a la derecha de la plataforma central, y finalmente saltar sobre el grupo de púas inferior. Dado que es uno de los primeros niveles luego del tutorial, se enfatiza la consistencia en el manejo del personaje por parte del jugador, presentándole tres veces el mismo obstáculo en distintas posiciones. El primer grupo de púas se encuentra al lado del punto de partida del nivel para que el jugador pueda experimentar de manera rápida la temática del nivel y se familiarice con ésta. De igual forma, el segundo grupo de púas está al lado del primero para que una vez que el jugador haya logrado encontrar la técnica necesaria para pasarlo, pueda volver a ponerla a prueba. Ya completados los primeros dos desafíos, se presenta algo distinto para que la experiencia del jugador no sea repetitiva. Este nuevo obstáculo corresponde a un salto que debe realizar entre dos muros con púas. Finalmente se presenta nuevamente un grupo de púas para que así el jugador pueda hacer uso de la confianza que obtuvo al superar los dos primeros obstáculos, llegando de esta forma a las escaleras.

Por el lado del ajuste en la dificultad, se puede ver que los principales cambios corresponden a la cantidad de púas y al espacio disponible para pasar de la parte superior a la inferior. El tamaño de los grupos de púas va creciendo de manera constante según la dificultad del nivel, y está planificado para limitar las opciones del personaje para sortear este obstáculo. En la primera variante presentada, ya que hay sólo hay una púa por grupo, el jugador únicamente necesita saltar sobre esta para evadirla, no requiere ni correr ni caminar por la muralla, pero no por esto está limitado a utilizar estas opciones si le son más cómodas durante este nivel. En cambio, para la segunda y tercera variante, si el jugador sólo salta mientras camina, la distancia que recorre no será suficiente para pasar sin caer en las púas, debiendo al menos saltar mientras corre para la segunda variante, y caminar por el muro mientras corre en la tercera. En cuanto al salto hacia la parte inferior, se utilizan nuevamente las piedras como obstáculo, acotando el espacio que el personaje dispone y por lo tanto requiriendo una combinación de inputs más precisa por parte del usuario. A diferencia de los grupos de púas, este cambio no ocurre de manera gradual, sino que aparece a partir de la dificultad 0.7.

Es así como a lo largo de los niveles, el ajuste de dificultad se traduce en la aparición y desaparición de púas, flechas y piedras de manera oportuna, manteniendo un diseño simple, conciso y fácil de entender por parte del jugador. Así pues, esto permite al usuario enfocarse en la experiencia de jugar, más que en la modificación de dificultad que está ocurriendo mientras juega.

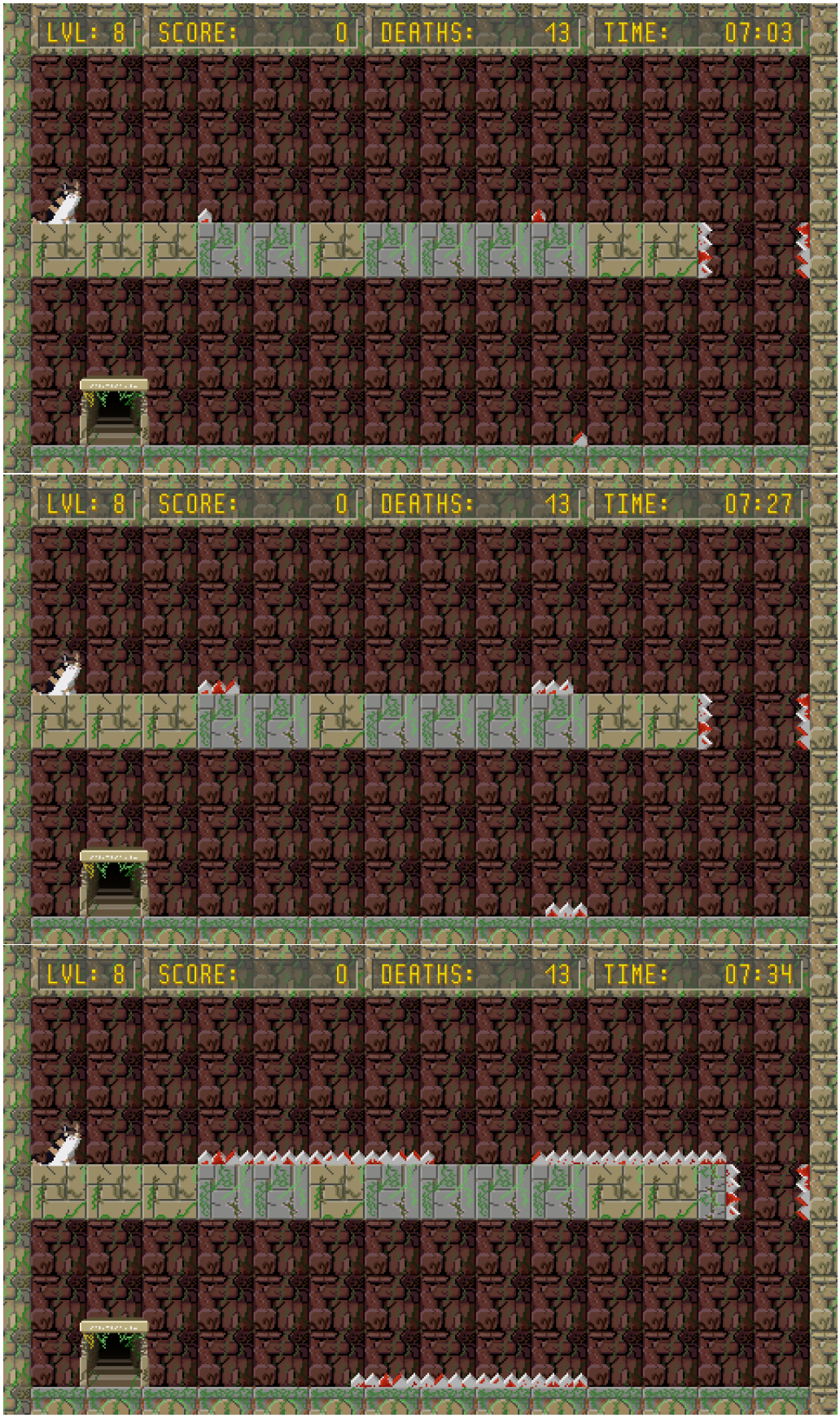


Figura 3.12: Nivel 8 – dificultad 0.0, 0.5 y 1.0 de arriba hacia abajo

Finalmente, para confirmar el diseño adecuando de los niveles, se realizó una validación piloto de los prototipos de diseño con dos usuarios usando la técnica de *thinking aloud*. En primera instancia, se les pidió jugar con el ajuste de dificultad activado hasta que completaran el juego, procediendo luego a mostrar las distintas variaciones existentes para cada nivel a modo de obtener sus opiniones y cómo sintieron la sinergia general de los niveles en sus distintas variaciones. Como resultado de esta prueba se obtuvieron buenos resultados, ya que ambos evaluadores mencionaron la consistencia entre los distintos niveles y la factibilidad de completarlos en su dificultad media al invertir algo de esfuerzo.

Capítulo 4

Regulación del Juego con Biofeedback

Dado que la principal característica del juego desarrollado corresponde al ajuste de dificultad dinámica debido al pulso, el primer paso para conseguir esto consiste en determinar qué dispositivo será utilizado para obtener esta medida. Así pues, se procede a realizar un estudio de las distintas soluciones actualmente presentes en el mercado.

4.1. Selección del dispositivo de monitoreo de pulso

Se hace un sondeo utilizando el motor de búsqueda Google con las palabras claves *best heart rate monitors* [7, 32]. Se elige este criterio de búsqueda con el objetivo de encontrar los tipos de sensores que posean mayor probabilidad tanto de pertenecer al jugador como de ser adquiridos en el futuro. De esto se pudo identificar cinco principales tipos de dispositivos que permiten medir el pulso de manera consistente: (1) monitores de actividad o pulseras, (2) smartwatches, (3) bandas de pecho, (4) sensores de pulso óptico o clip de dedo, y (5) audífonos con sensor de pulso integrado.

4.1.1. Definición del tipo de dispositivo a utilizar

Para cada uno de estos dispositivos se analizaron los pros y contras de su utilización enfocados en su aplicación a esta memoria y considerando cuatro aspectos esenciales: precio, precisión, comodidad y conectividad, es decir, la facilidad de utilizar el pulso obtenido por el sensor en otra plataforma como por ejemplo un computador. Cabe destacar que estos criterios fueron elegidos debido a que el objetivo final es utilizar el sensor como parte de un juego potencialmente comercial, es decir, el resultado obtenido debe ser capaz de ser llevado al mercado y no ser sólo un ejercicio experimental.

La información requerida para realizar este análisis fue obtenida mediante la búsqueda de cada uno de los distintos tipos de dispositivos en Google, enfocándose en sus características físicas, especificaciones técnicas, publicaciones comparativas y comentarios de dueños de los

distintos equipos. El análisis realizado se puede resumir en los siguientes puntos clave por dispositivo:

- **Monitores de actividad:** Presentan un amplio rango de precios dependiendo del aspecto, hardware y funcionalidades que posean [31], lo cual permite buscar de manera más precisa aquel sensor que se adapte mejor a las características buscadas. A pesar de las diferencias de precio entre los distintos modelos presentes en el mercado, todos poseen un sensor de pulso cuya precisión es lo suficientemente aceptable [28], es decir, permiten detectar las variaciones de pulso importantes de manera oportuna. También ganan puntos por el lado de la comodidad dependiendo del tamaño físico que posean, ya que consisten en sólo una pulsera, la cual no provocaría una mayor molestia al usuario. En cuanto a la conectividad, los datos generados por el monitor suelen ser enviados por Bluetooth, siendo así posible diseñar algún mecanismo de comunicación con el computador, de no existir previamente, mediante el análisis de paquetes. Sin embargo, también es posible encontrar dispositivos con APIs web, como es el caso de los productos de la marca Fitbit.

- **Smartwatches:** Si bien suelen ser de costo elevado, sobrepasando generalmente los $150 dólares [27], su punto fuerte reside en que las probabilidades de que el usuario ya posea uno son mayores que para los demás dispositivos [33]. Por el lado de la precisión en la medición del pulso, esta suele ser considerablemente buena [21], a pesar de que sus funcionalidades no están enfocadas alrededor de esta característica. De igual manera que los monitores de actividad, su forma de pulsera permite que usarlos no sea una molestia para el usuario mientras realiza otras actividades, como jugar en este caso. Su mecanismo de comunicación suele estar enfocado a aplicaciones tanto de Android como de iOS, desde las cuales sería posible enviar los datos a otras plataformas buscando alguna aplicación auxiliar o desarrollándola.

- **Bandas de pecho:** El precio de las bandas más populares se encuentra en el rango de los $70 a $90 dólares [7], el cual corresponde a un punto medio comparado con los demás dispositivos analizados. Poseen la mejor precisión a la hora de medir el pulso ya que generalmente esta es su única funcionalidad. Considerando la comodidad de usar una de estas bandas, si bien no son necesariamente molestas de usar, corresponden a elementos no cotidianos, por lo que es altamente posible que los potenciales jugadores no estén acostumbrados a su uso. Asimismo, el proceso para equiparlas no es tan directo como para el caso de los monitores de actividad o smartwatches, en los que sólo basta ponerse una pulsera. Como las bandas son inalámbricas, poseen mecanismos de comunicación que interactúan ya sea con una aplicación de celular o directamente el computador, usando Bluetooth y/o ANT+ como protocolos.

- **Sensores de pulso óptico:** Corresponden a los dispositivos más baratos de todos, difícilmente superando los $30 dólares [20]. A pesar de su rango de precio, su precisión es bastante buena [36], ya que de manera similar a las bandas de pecho, una de sus principales funciones consiste en medir el pulso junto a la saturación de oxígeno en la sangre, por lo que constituyen una buena alternativa tanto en precio como en precisión. Sin embargo, al considerar la comodidad pierden muchos puntos, sobre todo si el usua-

rio debe estar utilizando sus manos, como sería el caso si éste está jugando un juego. Si bien existen sensores que se utilizan en otras partes del cuerpo como los pies o las orejas, lo típico es utilizarlos en alguno de los dedos de las manos. También pierden muchos puntos, si es que no todos, en el hecho de que ofrecen opciones de conectividad muy pobres, pues su objetivo es ser utilizados como dispositivos independientes, teniendo a lo más algunos conexión mediante USB.

- **Audífonos:** Pertenecen a un sector bastante nuevo aunque no por ello menos popular, lo que hace que los audífonos equipados con un sensor de pulso integrado posean precios elevados, mas no superiores a los de un smartwatch [26]. Esto se debe a que miden el pulso desde el interior de la oreja, por lo que los valores obtenidos son bastante precisos [26]. Sin embargo lo anterior implica que los audífonos deben ser *in-ear*, lo que puede ser una molestia para algunos usuarios que encuentran este tipo de audífonos invasivos. Son inalámbricos por lo que se comunican por medio de Bluetooth, medio por el cual envían tanto sonido como los datos obtenidos por el sensor de pulso.

Debido a lo anterior, cuyo resumen se aprecia en la tabla 4.1, se decide utilizar los monitores de actividad. Esto obedece a que proveen una precisión aceptable, no provocan incomodidad en el usuario y existe una gran variedad tanto de precio como de conectividad.

Tabla 4.1: Resumen análisis de dispositivos para medir el pulso

Tipo	Precio	Precisión	Comodidad	Conectividad
Monitor de actividad	Rango amplio	Aceptable	La mejor	Bluetooth/PC
Smartwatch	>$150	Muy buena	Muy buena	Smartphone
Banda de pecho	$70 - $90	La mejor	Regular	Ant+/PC
Sensor de pulso óptico	<$30	Muy buena	Mala	USB/Nula
Audífono	>$150	Muy buena	Buena	Bluetooth

4.1.2. Definición del modelo específico de pulsera a utilizar

Ya con el dispositivo seleccionado se procedió a buscar la marca y modelo a utilizar. Este proceso implicó realizar una comparación de muchos productos. Sin embargo, no se llegó a un consenso al no encontrar un sensor que fuera simultáneamente económico y que tuviera herramientas de conectividad que permitieran un fácil acceso a los datos desde el computador. Debido a esto, se escogió a un representante de cada categoría. Por un lado está el *Xiaomi Mi Band 3* destacando por su bajo costo, el que no supera los $30 dólares. Por otro lado, están las pulseras de la marca *Fitbit*, las que si bien tienen un precio mucho mayor (más del doble que el Mi Band los modelos más económicos), poseen una API web la cual permite tener una comunicación directa con la pulsera.

Investigando más sobre el Mi Band se pudo encontrar la existencia de un trabajo realizado por Nikishaev [24] en el cual se explica cómo analizando los paquetes Bluetooth que se envían entre la pulsera y la aplicación de celular *Mi Fit* es posible realizar ingeniería inversa y así comprender el funcionamiento interno de la pulsera. De esta manera, se puede recrear este

proceso desde un programa de computador. Asimismo, Shymanskyy [30] provee una biblioteca en Javascript para el dispositivo Mi Band. A pesar de que el trabajo de ambos utiliza como objetivo el Mi Band 2, sería posible usar este para desarrollar una biblioteca en C++ que permita comunicar los valores de pulso entregado por el sensor del Mi Band 3 con Unreal Engine mediante el uso de la API Bluetooth de Windows RT disponible para Windows 10.

Considerando lo anterior, se decidió finalmente elegir a la pulsera Xiaomi Mi Band 3 como el sensor de pulso a utilizar para esta memoria, ya que si bien carece de una interfaz de aplicación (API) como tal, sí es posible desarrollar una utilizando como base el trabajo existente. De igual manera, el esfuerzo en tiempo de desarrollar esta API se justifica al considerar el costo excesivo en que se hubiera incurrido de elegir una pulsera Fitbit, sobre todo si se considera que este costo lo tendría que hacer el usuario para poder utilizar el juego. Esto introduce innecesariamente un potencial riesgo de adopción, lo cual podría llevar al fracaso del producto a desarrollar como se evidencia con casos reales como el de *O.Zen* explicado con anterioridad.

4.2. Conexión con el juego

Una vez en posesión del Mi Band se comenzó con el desarrollo de la API que finalmente permita la transmisión del pulso obtenido por el sensor hacia el computador. Para su implementación se utilizó el lenguaje C++ debido a este ser el lenguaje de programación que utiliza Unreal Engine, así como también uno de los lenguajes en los que Windows provee acceso a sus APIs, junto a C#, JavaScript y Visual Basic.

La comunicación con el Mi Band se realizó a través de Bluetooth Low Energy (BLE), el cual usa el perfil GATT para la transferencia de información entre la pulsera y otros dispositivos. Dado que este perfil constituye la base de toda la comunicación con el Mi Band, y por lo tanto la estructura fundamental de la API a implementar, vale la pena explicarlo.

4.2.1. Comunicación con el dispositivo Mi Band

Generic Attribute Profile (GATT) [2, 3] es un perfil que define la manera en que un dispositivo BLE se comunica, existiendo dos actores en esta comunicación: el cliente y el servidor. Su relación se establece mediante el envío y recepción de datos. Un cliente se encarga de enviar solicitudes al servidor y recibir respuestas acorde a ellas. En primera instancia el cliente no conoce nada del servidor, así que antes de poder leer o escribir datos debe realizar un reconocimiento previo de sus atributos. De aquí se concluye que la función del servidor consiste en estar constantemente escuchando por posibles solicitudes, responderles si es que llegan y además es posible configurarlo para que emita notificaciones de una variable. Esto es, cada vez que esta variable ve alterado su valor, o luego de que ha pasado determinado tiempo, es el servidor el que envía un mensaje al cliente.

Para el manejo de la información, GATT utiliza el protocolo de información genérica o Attribute Protocol (ATT), el cual divide los datos de manera jerárquica en distintas secciones,

como se observa en la figura 4.1. A un conjunto de grupos de datos relacionados entre sí se le asigna el nombre de servicio, donde estos grupos son llamados a su vez características. Así, un servidor posee uno o más servicios, los cuales poseen una o más características con las que puede interactuar el cliente.

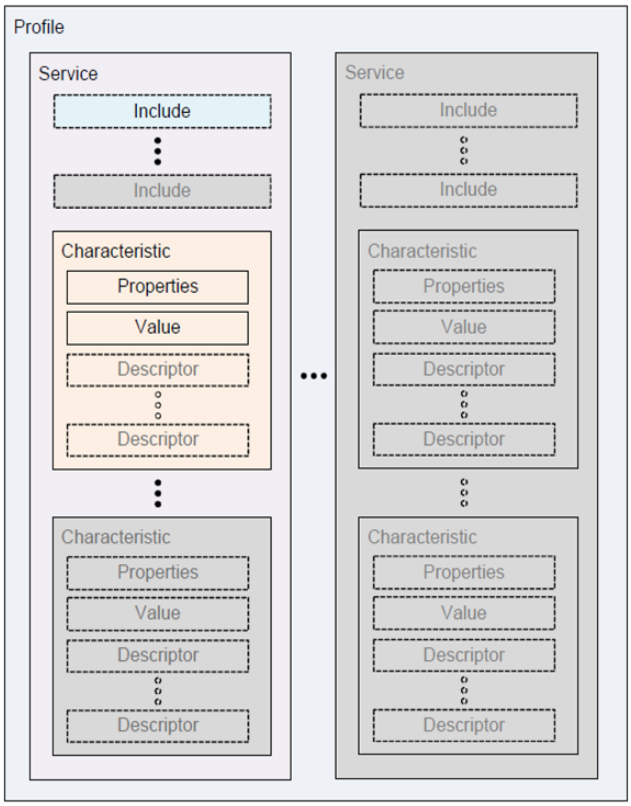


Figura 4.1: Jerarquía presente en el perfil GATT para dispositivos BLE [2]

Una característica posee por defecto propiedades y un valor, siendo este último el que lee o escribe el cliente. Sin embargo, también se pueden tener cero o más descriptores. Un descriptor no posee información propia, sino que su papel es mantener información de la información, lo que se conoce comúnmente como *metadata*.

El proceso de identificación, tanto de servicios como de características, se realiza mediante la utilización de un identificador único universal (UUID), el cual corresponde a un número de 128 bits (16 bytes). Gracias a éste, se hace posible establecer la comunicación entre el cliente y el servidor. Sin embargo 16 bytes implica un costo significativo de tamaño para la capa de enlace encargada de llevar a cabo la comunicación. Es debido a esto que se definen dos formatos cortos de UUID de 16 y 32 bits. Para ello se establece la máscara `XXXXXXXX-0000-1000-8000-00805F9B34FB`, la cual permite ser combinada con la UUID corta para así obtener su versión de 128 bits. Estos identificadores de menor longitud son utilizados para servicios o características previamente definidos por la organización de estándares *Bluetooth Special Interest Group* (Bluetooth SIG), para que de esta manera los desarrolladores posean una interfaz común de identificación de atributos conocidos.

4.2.2. Autenticación con el dispositivo Mi Band

Si bien un dispositivo BLE posee características que cualquier cliente puede leer y/o escribir, como por ejemplo la hora o su estado actual, es común que se establezca algún sistema de autorización que permita tener acceso a todas las características del dispositivo. De esta manera se impide que terceros puedan obtener datos o modificarlo sin permiso. El Mi Band no constituye un excepción a lo anterior, por lo que es necesario completar un proceso de autenticación antes de poder realizar acciones como iniciar el sensor de pulso o consultar los valores que este produce. Por otro lado, ya que no todas las funcionalidades de un dispositivo BLE están estandarizadas, y para utilizar UUID de 16 o 32 bits, se suele crear una máscara propia del dispositivo con la que se puedan generar UUID personalizadas. Para el Mi Band esta máscara corresponde a `0000XXX-0000-3512-2118-0009af100700`, cuyos primeros 16 bits son cero ya que no se ocupan direcciones de 32 bits.

A continuación se detallan los pasos específicos para acceder al Mi Band, incluyendo el proceso de autenticación [1]:

- Escanear todos los dispositivos cercanos.
- Comprobar si el dispositivo encontrado posee entre sus servicios uno con la UUID `0xfee0` (Si se tiene más de una pulsera se puede filtrar por la dirección MAC).
- Una vez identificado el dispositivo, dejar de buscar más.
- Escanear y guardar referencias a los distintos servicios, características y descriptores que se desee utilizar. Para este caso los servicios importantes son el de autenticación (`0xfee1`), y el del pulso (`0x180d`). El servicio de autenticación posee una sola característica (`0009`) y un descriptor (`0x2902`). El servicio de pulso posee dos características, un punto de control (`0x2a39`) utilizado para enviar comandos y un encargado de obtener el pulso (`0x2a37`), el cual a su vez posee un descriptor (`0x2902`). Cabe destacar que el descriptor `0x2902` corresponde al encargado de habilitar las notificaciones de la actualización del valor de una característica. Este procedimiento se logra al escribirle los bytes { `0x01`, `0x00` } y designando un método que se quede escuchando estas actualizaciones.
- Habilitar las notificaciones de la característica de autenticación.
- Generar una `key` de 16 bytes. Es ideal que sea generada nuevamente cada vez que se produzca una conexión nueva, sin embargo se puede utilizar una `key` fija para fines prácticos.
- Escribir { `0x01`, `0x00`, `key` } en la característica de autenticación. Las respuestas del servidor corresponden a actualizar el valor de la característica, debido a esto se activan

[1] Cuando se utiliza una UUID estándar se utiliza el formato "`0xXXXX`", mientras que las UUID personalizadas se denotan como "`XXXX`". Todas las UUID son de 16 bits

las notificaciones previamente.

- Si la respuesta del servidor es { `0x10`, `0x01`, `0x01` }, la `key` se recibió correctamente y se debe solicitar un key de autenticación *random* escribiendo { `0x02`, `0x00`, `0x02` }.

- Si los primeros bytes de la respuesta del servidor son { `0x10`, `0x02`, `0x01` }, los siguientes 16 bytes corresponde a la key de autenticación solicitada. Se debe encriptar esta key con la `key` generada en un inicio utilizando el esquema de cifrado por bloques AES (*Advanced Encryption Standard*), el modo de operación ECB (*Electronic codebook*), 128 bits y sin padding. Se envía el resultado escribiendo { `0x03`, `0x00`, `key` encriptada } en la característica.

- Si la respuesta del servidor es { `0x10`, `0x03`, `0x01` }, la autenticación fue un éxito.

4.2.3. Activación de la lectura del pulso y habilitación de sockets

Ya con la autenticación realizada, fue posible interactuar con todos los servicios y características que posee el Mi Band, en particular el lector de pulso, por lo que se procedió a activar las notificaciones de la característica que contiene el pulso. Existen dos modos de medición: inmediata y continua. En la medición inmediata se calcula el pulso por 12 segundos y se entrega un único valor correspondiente al resultado. Para activar este modo se debe escribir { `0x15`, `0x02`, `0x01` } en la característica de punto de control. En contraste, la medición continua entrega todos los valores de pulso registrados mientras ésta permanezca activa.

De manera similar, la activación se realiza escribiendo { `0x15`, `0x01`, `0x01` }. Para evitar problemas, se debe escribir previamente { `0x15`, `0x01`, `0x00` } y { `0x15`, `0x02`, `0x00` } en ambas activaciones, para así desactivar cualquier medición que se encontrara en curso. Dado que la pulsera no tiene forma de saber por cuánto tiempo debe realizar la medición continua, se establece una regla que permita regular este tiempo. La medición seguirá efectuándose siempre y cuando cada doce segundos se escriba { `0x16` } a la característica de control. Las medidas de pulso son obtenidas del método asignado al recibir las notificaciones de la característica correspondiente, siendo éste el punto de partida para utilizar el pulso como sea necesario.

Con todo el proceso anterior, fue posible obtener el pulso del usuario. Sin embargo aún no es directa su comunicación con Unreal Engine. Si bien la medición se obtiene mediante C++, también se utiliza código específico de Windows, tanto para el manejo de tareas asíncronas (C++/CX) como para la comunicación mediante Bluetooth (C++/WinRT). Es por esto que se opta por no integrar de manera directa el código a Unreal Engine, sino simplemente enviar el pulso mediante la utilización de sockets. Con este objetivo se utiliza la biblioteca de sockets de Windows y la implementación de sockets presente en Unreal Engine. Es necesario mencionar que la implementación presente en Unreal Engine está en desarrollo y puede ser modificada en cualquier momento por una nueva actualización del software. Sin embargo, cumple con su propósito para el trabajo actual.

La comunicación se realiza mediante el puerto 1242. Dentro de Unreal Engine se crea un servidor encargado de aceptar conexiones entrantes, para que de esta forma el código encargado de obtener el pulso se conecte como cliente, enviándoselo apenas es recibido por el método a cargo de las notificaciones de esta característica. Así, se logra usar el pulso dentro de Unreal Engine como se puede ver en la figura 4.2, siendo presentado como un elemento de depuración.

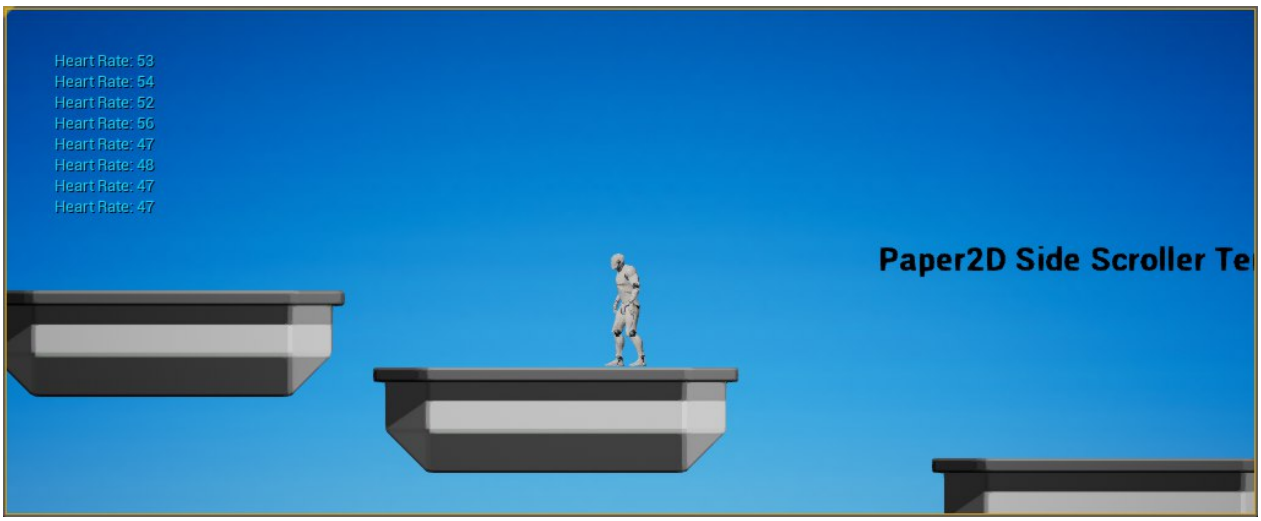


Figura 4.2: Pulso dentro de Unreal Engine

Implementar la comunicación con la pulsera como un módulo externo al código del juego permite una mayor portabilidad, ya que si se desea portar el juego a otra plataforma sólo sería necesario crear un nuevo módulo encargado de la comunicación Bluetooth. Para este fin, se pueden utilizar las herramientas que estén disponibles para este propósito en la plataforma objetivo, como es el caso del módulo original donde se usa código propio de Windows. Es este mismo punto el que evidencia el beneficio de haber realizado el juego en un motor como Unreal Engine, ya que éste permite, con muy poco esfuerzo de por medio, compilar el juego para una gran variedad de plataformas.

4.3. Proceso de comunicación

Si bien con lo anterior ya se cuenta con un sistema de comunicación entre la aplicación encargada de conectarse al Mi Band y Unreal Engine, esta comunicación no es bidireccional. Desde dentro del juego, el usuario no es capaz de comunicarse con la aplicación, lo que imposibilitaría acciones básicas como seleccionar qué dispositivo se desea ocupar e indicar cuándo se desea iniciar o detener la medición del pulso. Asimismo, no sería posible incorporar elementos de comunicación adicionales, como por ejemplo mandar un mensaje a la pulsera para que lo muestre en su pantalla, o bien ser capaz de hacerla vibrar a voluntad, lo cual se utiliza cada vez que el personaje muere como feedback extra hacia el jugador.

Debido a esto, además del servidor que está corriendo en el lado del juego, se creó un servidor que se ejecuta desde la aplicación que se comunica con el Mi Band. Para inicializar

el servidor, se utiliza el puerto 1243 y se implementó un método encargado de estar constantemente escuchando por comandos si es que se conecta algún cliente. Estos comandos se componen de un número que representa su id y una lista de argumentos dependiendo de que comando se quiera ejecutar. También se hace una distinción entre dos estados a la hora del procesamiento de la información recibida, siendo la barrera que los divide si la pulsera ha pasado o no por el proceso de autenticación. Así pues, si se le encarga la tarea de vibrar pero el Mi Band no ha sido correctamente autenticado, entonces aún cuando reciba el paquete Bluetooth que lo comanda a vibrar, no se presentaría ningún resultado.

A continuación se presentan los comandos disponibles junto a los argumentos que requiere cada uno. El id que espera recibir el servidor corresponde a un `byte`, `uint8` o `unsigned char` Si se desea portar el módulo que se comunica con el Mi Band a otra plataforma, es necesario que este nuevo módulo sea capaz de responder a estos mismos comandos y ejecute las mismas acciones.

- **Id:** `0`, **argumentos:** `{ bool }`. Este primer comando se encarga de iniciar o detener el cliente en el lado de la aplicación, que es el encargado de enviar información hacia el servidor del juego mediante el puerto 1242. El argumento que recibe determina si se debe iniciar el servidor al recibir `true`, o detenerlo si se recibe `false`. Como no se efectúa ningún tipo de comunicación con la pulsera, no es un requerimiento que esté autenticada. La ejecución de este comando es necesaria como un primer paso para iniciar la conexión con la pulsera, ya que es el cliente quien está constantemente mandando la lista de dispositivos disponibles encontrados por la aplicación. Esta lista es recibida por el juego una vez completada la conexión con su servidor.

- **Id:** `1`, **argumentos:** `{ uint32 uint8[] }`. Con este comando se inicia la comunicación de la pulsera, la cual lleva al proceso de autenticación. Luego, es necesario saber a qué dispositivo se desea conectar. Cada dispositivo posee una dirección MAC única, por lo cual se utiliza este número para diferenciarlos, siendo también el valor requerido para iniciar la comunicación dentro de la aplicación. El segundo argumento corresponde a la dirección MAC en el formato `XX:XX:XX:XX:XX:XX`, siendo la aplicación la encargada de transformarla a un número entero para su posterior utilización. Como la MAC es recibida como un arreglo de bytes, el servidor requiere de antemano conocer el tamaño del arreglo para así leer correctamente la información que le será enviada. Es por esta razón que el primer argumento corresponde precisamente al tamaño del arreglo que será enviado. En este caso, dado el formato fijo utilizado, el tamaño siempre será 17. Sin embargo, se solicitan dos argumentos, ya que esta corresponde a un forma de comunicación estándar cuando se envían elementos de largo variable como es el caso de un arreglo.

- **Id:** `2`, **argumentos:** `{ uint32 uint8[] }`. De aquí en adelante los comandos requieren acceder a funcionalidades de la pulsera, por lo que sólo tendrán efecto si es ya se encuentra conectado a un dispositivo y ya se ha ejecutado correctamente el proceso de autenticación. Este comando se encarga de mostrar un mensaje en la pantalla del Mi Band. Es por esta razón que como argumentos requiere el tamaño del mensaje y luego el mensaje en sí. Se crea este comando a modo de prueba para confirmar de manera sencilla si se obtuvo acceso a las funcionalidades de la pulsera luego del proceso de autenticación.

- **Id:** `3`, **argumentos:** `{ bool }`. Este comando se encarga de iniciar o detener el proceso encargado de la medición continua del pulso, siendo su argumento quién determina si se inicia, al ser `true`, o detiene, si es `false`. Una vez iniciada la medición del pulso, el cliente de la aplicación es el encargado de mandar constantemente cada uno de los pulsos registrados por la pulsera hacia el servidor del juego, para su uso en la modificación de dificultad de los niveles.

- **Id:** `4`, **argumentos:** `{ uint16 }`. Para hacer que el Mi Band vibre, sólo es necesario indicar la cantidad de tiempo que se quiere que esta esté vibrando desde que recibió la instrucción de hacerlo. Este comando sirve para indicar a la pulsera que inicie el proceso de vibración por una determinada cantidad de tiempo en milisegundos, siendo ésta especificada por el argumento que recibe. A diferencia de los demás comandos, se utiliza `uint16` y no `uint32`, ya que el Mi Band sólo pude estar vibrando de manera continua aproximadamente 65 segundos dado que internamente usa un entero de 16 bits para almacenar la cantidad de milisegundos restantes que le quedan por vibrar. Debido a lo anterior, si se desea obtener una vibración continua, es necesario enviarle un nuevo comando antes de que finalice al tiempo del anterior.

- **Id:** `5`, **argumentos:** `{}`. Este corresponde al comando más simple y al igual que el tercer comando se utiliza a modo de prueba. No recibe argumentos, y una vez ejecutado la pulsera realizará dos vibraciones cortas consecutivas. Si bien el mismo resultado se puede lograr con el comando anterior utilizado dos veces, se utiliza un servicio distinto del Mi Band en el proceso. Éste es el servicio de alerta inmediata, el cual puede ser utilizado para simular distintos tipos de alerta. Dado que escapan del foco de esta memoria, no serán explicadas con mayor detalle.

El tener funcionando una comunicación bidireccional entre la aplicación encargada del Mi Band y el juego desarrollado permite proceder tanto con el desarrollo del procedimiento de comunicación utilizado por el juego durante cada ejecución, como también con la implementación del algoritmo necesario para transformar los pulsos recibidos. De esta manera, se puede contar con información útil que permita modificar la dificultad de los niveles en los que se encuentra el jugador.

4.4. Medición del pulso y comunicación con el juego

El procedimiento requerido para llevar la medición del pulso desde la pulsera al juego posee varios pasos. Primero, se inicia la aplicación que se comunica con el Mi Band. Cuando esta aplicación se ejecuta, inmediatamente realiza dos acciones. La primera corresponde a iniciar el servidor que será el encargado de estar escuchando los comandos que le envíe tanto el usuario como las rutinas del juego. La segunda se encarga de iniciar la comunicación Bluetooth y analizar todos los dispositivos cercanos, comprobando si éstos corresponden a una Mi Band. El proceso de descubrimiento de nuevos dispositivos no es inmediato y van apareciendo primero aquellos que se encuentren más cercanos. Si se identifica una Mi Band

y el servidor del juego se encuentra inicializado, así como también el cliente de la aplicación, se envía su dirección MAC. Cabe destacar que no es posible conectar un cliente mediante sockets si es que no existe previamente un servidor activo. Es por esta razón que se separa la inicialización del cliente como un comando por parte de la aplicación, para que de esta manera sólo sea ejecutado cuando ya se haya completado de manera correcta la configuración del servidor.

Ya con la aplicación iniciada se procede a ejecutar el juego. Al mismo tiempo en que éste comienza, se realiza el proceso de configuración para el servidor que recibirá información desde la aplicación. Cuando este servidor queda funcionando, se envía el primer comando a la aplicación. Así, se indica que debe crear y conectar el cliente para que de esta manera el juego pueda recibir los dispositivos disponibles de los que podrá elegir el jugador para obtener su pulso. Para ello se le presenta con la pantalla de la figura 4.3, donde se observa un grupo de botones en cuyo texto indican la dirección MAC de los dispositivos cercanos.

Figura 4.3: Pantalla de selección de dispositivos

En esta pantalla irán apareciendo los dispositivos que son detectados por la aplicación. Sin embargo, no hay una forma exacta de saber si alguno de los dispositivos mostrados sigue a una distancia a la cual sea posible su conexión. Es por esto que cada vez que se recibe un dispositivo, se le otorga una vida de 30 segundos, de tal manera que si no se vuelve a escuchar se este dispositivo en ese lapso, se procede a retirar de la lista. Por otro lado, dada la naturaleza del juego, no se da la opción de iniciar una sesión sin conectarse previamente a una pulsera. Sin embargo, se incorporó un botón *Cancel*, el cual se activa si no se encuentra ninguna Mi Band cercana. También se agregó el argumento de línea de comandos `-nodevice`, el cual puede ser utilizado para saltarse la pantalla de selección de dispositivos si así fuera necesario, lo cual claramente provocaría que el juego no cuente con algún tipo de ajuste de dificultad.

Es importante notar que la conexión del Mi Band con la aplicación requiere que ésta no esté previamente vinculada con otro dispositivo, como por ejemplo el celular. De caso contrario, no será posible establecer una comunicación nueva al ya estar ocupado el canal.

Cuando el jugador selecciona un dispositivo, se envía la dirección MAC como comando a la aplicación y esta se encarga de realizar tanto la conexión como la autenticación del Mi Band elegido. Una vez este proceso termina de manera satisfactoria, la aplicación detiene su búsqueda de dispositivos cercanos, por lo cual el cliente detiene su envío de direcciones MAC y manda el mensaje `Connected` al servidor del juego. También para comprobar que la autenticación con la pulsera fue un éxito, la aplicación envía un mensaje a la pulsera para que sea mostrado en su pantalla.

Ya con el Mi Band conectado, se envía un comando para que comience la medición del pulso. Es en este momento que se presenta al usuario la historia del juego. Entre la solicitud realizada y el primer valor de pulso obtenido existe un lapso de 12 segundos. Es por esta razón que se muestra la historia luego de realizar la conexión con la pulsera y no antes, para así tener una ventana mayor en la que se pueden obtener mediciones del pulso base del jugador, sobre el cual se evaluaran las variaciones detectadas durante el juego. La historia del juego dura 68 segundos, y como el período de obtención del pulso por parte de la pulsera es de aproximadamente 4 segundos, se obtienen alrededor de 15 mediciones gracias a la presencia de la historia.

Una vez terminada la historia comienza el tutorial del juego. Las mediciones de pulso obtenidas durante estos 5 niveles también son consideradas para el cálculo del pulso base del jugador, siendo su cantidad dependiente del tiempo que permanezca el usuario dentro de estos niveles. Por el lado de la comunicación con el servidor de la aplicación, desde este punto se establece que cada vez que el jugador muera se enviará un comando para que la pulsera vibre por 100 milisegundos, para que de esta forma el usuario esté consiente de la cantidad de muertes que lleva e intente ser más preciso con los inputs que presiona para así mantener el valor del contador de muertes bajo y, por lo tanto, reducir la frecuencia de vibración del Mi Band.

Una vez que el jugador completa los niveles de tutorial, y por lo tanto se ha familiarizado con las mecánicas del juego, se comienza con la fase de ajuste de dificultad. Es importante mencionar que el diseño de los niveles del tutorial está pensado exclusivamente para el aprendizaje del jugador y no plantean una dificultad adicional. De esta forma, se busca no afectar la medición del pulso base y que así los valores obtenidos sean consistentes con los obtenidos mientras el jugador estaba viendo la historia.

4.5. Algoritmo de ajuste dinámico

El algoritmo encargado de realizar el ajuste de dificultad de manera dinámica no puede depender de valores preestablecidos, ya que el pulso base difiere de persona a persona. También puede ocurrir la situación en que una persona comience con un pulso base, sin embargo a lo largo del tiempo su pulso se estabilice en otro valor, lo que provocaría que su dificultad

siempre fuera muy alta o muy baja dependiendo del caso. Es por esto que, si bien se establecen puntos de partida a partir de las mediciones obtenidas durante la historia y el tutorial, estos valores deber irse modificando a lo largo del juego.

Para el ajuste de dificultad, el primer paso corresponde a obtener la mediana de las medidas recolectadas hasta antes de que empiecen los niveles que no son tutorial. Se utiliza la mediana y no el promedio para así ignorar posibles cambios repentinos en el pulso que pudieron haber ocurrido durante la fase de calibración. Como la dificultad dentro del juego es representada como un número continuo entre 0 y 1, es necesario establecer límites entre los cuales es posible que se encuentre el pulso del usuario durante el juego, para de esta manera poder obtener la dificultad actual correspondiente de acuerdo a la posición del pulso con respecto a los límites.

4.5.1. Definición de límites dinámicos

Ya que no se pueden tener límites fijos debido a lo explicado anteriormente, se establece un sistema dinámico que utiliza dos variables. La primera, llamada `scope`, corresponde a un porcentaje sobre el valor de la mediana, el cual determinará el pulso mínimo y máximo a utilizar. Es posible observar un ejemplo visual en la figura 4.4 para su mejor comprensión. Por sí sólo, el `scope` no ofrece una manera de modificar su valor a lo largo del tiempo, es por esto que se utiliza una segunda variable, llamada `deadzone`. Esta variable corresponde a un porcentaje que se aplica sobre el `scope` para así obtener la zona roja mostrada en la figura.

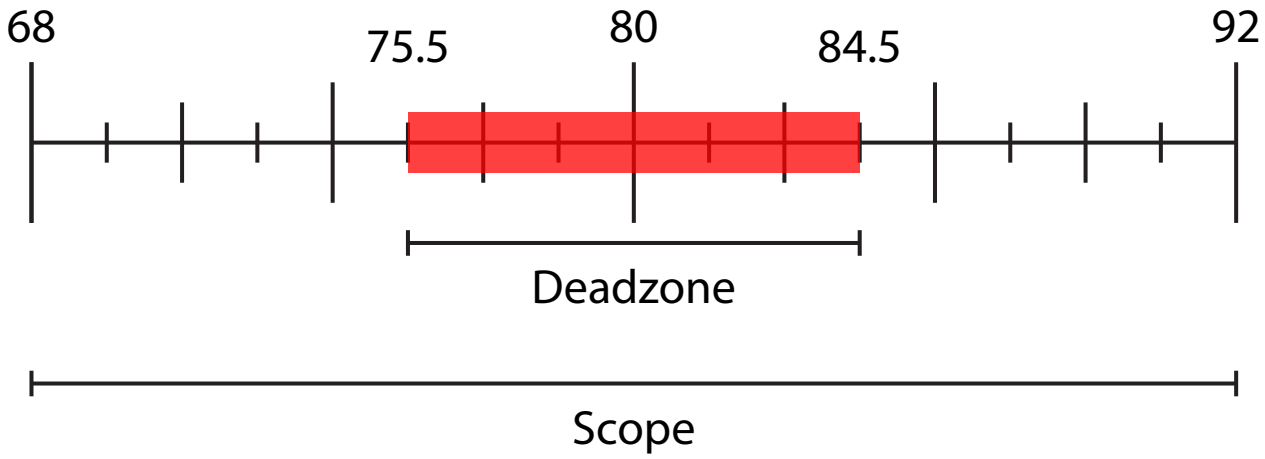


Figura 4.4: Ilustración del algoritmo utilizado con `scope` 0.15 y `deadzone` 0.375

Si el pulso actual del jugador se ubica dentro del `deadzone` esto implica que es posible que los límites actuales del `scope` sean muy grandes, por lo que es necesario reducirlo. En contraste, si el pulso se ubica fuera, entonces puede que el `scope` sea muy pequeño, por lo que se procede a incrementarlo. El factor de cambio de `scope` depende de qué tan alejado esté el pulso actual del borde del `deadzone`. Usando de ejemplo la figura 4.4, si el pulso actual del jugador es 80, el factor de cambio será -100 %, si es 75.5 correspondería a 0 % y si es menor o igual a 71 sería 100 %, donde para valores intermedios del pulso se obtendría un porcentaje proporcional entre -100 % y 100 %.

Si bien con el factor anterior es posible modificar el `scope` de manera directa, el cambio provocado de este proceso sería demasiado grande, y hay que tener en consideración que este ajuste no corresponde más que a una heurística que permite tener mejores límites de pulso para el cálculo de la dificultad. Debido a esto se establece la regla que el `scope` no debería modificarse en más de un 5 % por minuto. Como el período de muestreo de la pulsera es de aproximadamente 4 segundos, el factor de modificación obtenido es multiplicado por 0.05 y dividido por 15.

En la figura 4.5 es posible ver un fragmento del código del algoritmo, en el cual se evidencia el procedimiento realizado cada vez que se recibe una nueva medida del pulso, correspondiendo a la variable `HeartRate`.

```
1 float Change = HeartRate - HeartRateMedian;
2 HeartRateMedian += Change / 30;
3 float DeadZone = HeartRateMedian * ScopePercentage * DeadZoneFactor;
4 float PercentageOutside = FMath::Clamp((FMath::Abs(Change) - DeadZone) / ↩
    DeadZone, -1.f, 1.f);
5 ScopePercentage += 0.05f * PercentageOutside / 15;
6 HeartRateMin = (1.f - ScopePercentage) * HeartRateMedian;
7 HeartRateMax = (1.f + ScopePercentage) * HeartRateMedian;
8 float TargetDifficulty = FMath::Clamp((HeartRateMax - HeartRate) / (↩
    HeartRateMax - HeartRateMin), 0.f, 1.f);
```

Figura 4.5: Extracto de algoritmo de ajuste dinámico

4.5.2. Aplicación del algoritmo de ajuste dinámico

Al modificar el `scope` de manera dinámica, se tiene en consideración que si bien el pulso base de cada usuario es distinto al de los demás, también lo es la magnitud de la variación que este puede experimentar. Por ejemplo, dos jugadores pueden tener el mismo pulso base; sin embargo, para uno de ellos su pulso máximo alcanzable puede ser de sólo 10 bpm sobre su pulso base, en cambio para el otro el máximo se puede alcanzar en más del doble de esta variación.

Durante las pruebas realizadas en esta memoria se utiliza un valor inicial de 15 % para el `scope` y 37.5 % para el `deadzone`. Sin embargo se realiza un primer ajuste al `scope` luego del cálculo de la mediana, utilizando como datos los mismos pulsos obtenidos durante la fase de calibración, y teniendo como objetivo incrementar el valor del `scope` si así fuera necesario. Considerando que los pulsos ya fueron ordenados para la obtención de la mediana, se calcula tanto el promedio de la primera mitad como de la segunda mitad del arreglo de pulsos. Si el `scope` fue escogido de manera correcta, estos promedios deberían encontrarse dentro del `deadzone`, por lo que en caso contrario se calcula la distancia absoluta que los separa de esta.

Utilizando nuevamente de ejemplo la figura 4.4, si el primer promedio es 73 y el segundo es 83, sus distancias absolutas serían 2.5 y 0. Luego se calcula el promedio de estos valores

y se divide el resultado por la mediana, para así obtener el porcentaje que será agregado al `scope`. Dada la naturaleza del algoritmo desarrollado, el valor del `scope` se iría modificando de manera dinámica a lo largo del juego. Sin embargo se decide realizar este primer ajuste ya que el valor inicial escogido no deja de ser una estimación, por lo que de esta forma se logra llegar más rápido al valor adecuado para el jugador actual.

El valor del `deadzone` es el único que no cambia durante la ejecución del juego y este afecta de manera directa qué tan rápido es modificado el `scope` cada vez que se recibe una medición nueva del pulso, sin embargo sólo influye en el tiempo en que le toma al `scope` en converger a un valor estable que represente de manera certera a los límites del jugador. Es importante tener en cuenta que el `deadzone` no es más que un porcentaje del `scope`, por lo que si bien su valor no es modificado, sí lo es la distancia de la zona delimitada por este. Se escoge el valor de 37.5 % luego de realizar pequeñas pruebas para ver el comportamiento obtenido con distintos valores, lográndose que la modificación no sea ni tan lenta al punto de no ser perceptible, ni tan rápida al punto de nunca alcanzar un valor estable para el `scope`.

Finalmente, se debe tener en cuenta la posibilidad de que el pulso de un jugador varíe de manera permanente hacia un nuevo valor estable por razones externas al juego, en cuyo caso se debería modificar el pulso base utilizado para el cálculo de la dificultad. Este escenario presenta un mayor grado de dificultad, ya que se debe diferenciar de manera correcta cuando se está en presencia de esta situación o cuando la variación del pulso es temporal para luego regresar a su valor base, que es lo que se espera que ocurra durante el juego para que sea efectiva la modificación dinámica de dificultad. Para superar esta posibilidad se establece una nueva regla, la cual indica que si el pulso se estabiliza en un nuevo valor, el pulso base utilizado para el cálculo de dificultad debe convertirse en este valor si se mantiene en él por más de dos minutos. Para lograr cumplir esta regla, y teniendo en cuenta que dos minutos equivalen aproximadamente a 30 mediciones del pulso, se calcula la diferencia entre el pulso actual y el base, se divide por 30 y se suma al pulso base para que de esta manera se alcance eventualmente el nuevo pulso estable de ocurrir la situación anteriormente descrita. Por otro lado, esta modificación permite que el pulso base no sea un valor estable, otorgándole así más dinamismo al algoritmo.

Capítulo 5

Implementación del Juego

Para la implementación del juego se utilizó el motor Unreal Engine 4, debido a la variedad de herramientas que ofrece y a que utiliza C++ como lenguaje de programación. De esta manera, se permite que el código se ejecute con menos pérdida de rendimiento con respecto a otros lenguajes de más alto nivel. De igual manera, en un inicio se prioriza el uso de C++ debido a la posibilidad de incorporar directamente la aplicación encargada de comunicarse con el Mi Band al código del juego. Sin embargo, luego se descarta esta alternativa en favor la portabilidad del juego, separando la aplicación a un módulo independiente programado en C++ que se comunica con el juego mediante el uso de sockets. El código implementado para el juego y el módulo puede encontrarse en los repositorios de GitHub del autor, *Ragelite* [1] y *HRM* [2] respectivamente.

5.1. Entorno de desarrollo

Si bien Unreal Engine es un motor enfocado principalmente al desarrollo de videojuegos en 3D, posee un complemento llamado Paper2D que brinda la posibilidad de trabajar con elementos 2D de manera más simple. Sin embargo como no existe un motor 2D dedicado dentro de Unreal, los objetos en dos dimensiones que utiliza Paper2D existen dentro de un mundo en tres dimensiones, siendo su correcto posicionamiento a lo largo de los ejes y la utilización de una cámara con proyección ortográfica lo que simula estar dentro de un mundo en dos dimensiones para el usuario.

La base de los elementos que ofrece Paper2D corresponde a los *sprites*. Éstos permiten transformar una imagen a un objeto en dos dimensiones que puede ser situado dentro de un nivel. De igual manera, ofrece configuraciones predeterminadas para eliminar la compresión y suavizado automático que aplica Unreal a las imágenes, para así lograr un estilo pixel art, siendo este el utilizado por el juego desarrollado.

[1] `https://github.com/elixs/Ragelite`
[2] `https://github.com/elixs/HRM`

Como un *sprite* no es más que una única imagen que se mantiene constante durante su existencia, también existen los *flipbooks*. Éstos permiten crear animaciones a partir de una secuencia de imágenes, siendo asignado a cada una de estas un tiempo relativo durante el cual permanecerán a la vista del jugador antes de continuar con la siguiente, y una frecuencia que controla la velocidad en que se realiza el cambio de imágenes.

Paper2D también posee de forma nativa soporte para trabajar con *tiles*, siendo estos imágenes diseñadas para construir niveles. Ya que un solo *tile* crearía un nivel monótono, se suelen crear diversos diseños los que se agrupan dentro de una única imagen, como por ejemplo los 4 *tiles* mostrados anteriormente en la figura 3.1. Debido a esto, dentro de Paper2D se implementan *tilesets*, los cuales permiten separar una única imagen en la cantidad indicada de *tiles*, así como también otorgar la posibilidad de crear colisiones independientes para cada uno de estos.

Si bien con los *tilesets* ya es posible crear *tiles*, es necesario un espacio para poder posicionarlos dentro del mundo. Es por esto que también se tienen los *tilemaps*. Éstos corresponden a una serie de planos 2D de dimensiones configurables, los que permiten el posicionamiento de *tiles* en su interior. Poseen más de un plano para así poder utilizar el concepto de capas al momento de diseñar un nivel, permitiendo poner un *tile* adelante de otro, siendo así especialmente útiles para separar el diseño del fondo de los objetos con los que puede interactuar el usuario.

A pesar de la existencia de las herramientas anteriormente mencionadas, Paper2D nunca ha sido integrado de manera definitiva al código base del motor, siendo considerado actualmente como un plugin más. Por otro lado, este complemento no ha visto el desarrollo de nuevas funcionalidades desde el lanzamiento de la versión 4.8 de Unreal Engine [10], evento acontecido el 10 de junio de 2015. Desde esta fecha hasta el día de hoy, el equipo encargado del desarrollo del motor no ha invertido más tiempo en Paper2D. Es por esta razón que si se requieren funcionalidades más específicas para el manejo de elementos 2D, éstas deben ser implementadas desde cero. Asimismo, si se desea otorgar más características a los elementos ya existentes, se hace necesario modificar el código del motor.

Debido a lo anterior, sólo el manejo de imágenes, la construcción de los elementos estructurales del nivel y las animaciones del personaje, son desarrolladas utilizando las herramientas que ofrece por defecto Paper2D. Así pues, se vuelve necesario implementar clases y componentes independientes que se encargaran de los demás elementos del juego.

Para construir tanto la estructura de cada nivel como los límites que lo contienen, se utiliza un *tilemap*. Este componente obtiene *tiles* de un *tileset* creado a partir de la imagen de la la figura 3.1. Si bien de esta forma se pueden construir fácilmente una gran variedad de niveles, no es posible utilizar este método para posicionar los obstáculos, ya que un *tile* sólo guarda información sobre si posee o no colisión. Por consiguiente, esto limitaría fuertemente las mecánicas disponibles para el desarrollo de los obstáculos, los cuales deberían por sí solos ser capaces de interactuar con los demás elementos del mundo y comportarse según sea requerido.

En cuanto a la representación visual del personaje principal, ésta se configura utilizando como base un *flipbook*. No se usa un *sprite* ya que su imagen no permanecerá fija durante

todo el juego, sino que se crean distintas animaciones para cada uno de los estados por los que puede pasar dependiendo de los inputs que presione el jugador. El primer estado corresponde a cuando no se presiona ningún input, en cuyo caso el personaje se sienta, mueve la cola y ocasionalmente pestañea o mueve las orejas de manea aleatoria. Esto último se logra al modificar las imágenes de la secuencia en tiempo real. El segundo estado acontece cuando el personaje está caminando; aquí se utiliza un ciclo de caminata, el cual varia su velocidad de acuerdo a la velocidad con que se mueve el personaje, dando así la ilusión de que este está caminando o corriendo mientras se utiliza la misma animación. El tercer estado corresponde al personaje saltando, para el cual no se utiliza un ciclo, sino que se crea una secuencia de imágenes en los distintos estados de salto y caída, la cual es recorrida de acuerdo a la posición del salto en que se encuentre el personaje y su velocidad vertical. Finalmente, el cuarto estado corresponde a cuando el personaje está caminando por el muro del fondo, utilizándose la misma lógica que para cuando camina pero con una animación distinta.

El juego se creó utilizando como base la plantilla *2D Side Scroller* presente dentro de Unreal Engine. Esta corresponde a un ejemplo desarrollado para mostrar las funcionalidades de Paper2D, proveyendo un fondo, una serie de plataformas y un personaje el cual puede caminar y saltar. Si bien pareciera que los elementos presentados pueden ser utilizados de manera directa para implementar el juego de esta memoria, estos presentan variados aspectos que difieren del objetivo buscado. Un ejemplo de esto corresponde a la composición del personaje y sus sistemas de movimiento, teniendo que ser implementados de manera independiente, aunque utilizando de base los ya existentes. Antes de explicar los aspectos técnicos de la implementación del juego, es necesario comprender primero los pilares sobre los que se estructuran los elementos creados dentro del motor.

5.2. Estructura del motor de desarrollo

Al igual que la mayoría de motores, Unreal Engine se basa en una sistema de niveles, donde cada uno corresponde al mundo explorable por el jugador. Dentro de este mundo se pueden crear objetos de distintas naturalezas y con distintos comportamientos. A su vez, estos objetos pueden tener distintos componentes, los cuales se especializan en alguna funcionalidad en particular. Como Unreal Engine utiliza C++, la mayoría de los elementos dentro del motor utilizan a su beneficio la presencia de herencia en este lenguaje. Es así como existe un objeto base sobre el cual todos los demás elementos que constituyen el juego deben heredar, llamándose esta clase `UObject`. Existe una gran variedad de clases que heredan de `UObject`, como por ejemplo animaciones, mallas poligonales, luces y materiales. Sin embargo, hay una clase que posee mayor importancia, ya que constituye la base de todos los elementos que pueden ser creados y posicionados dentro de un nivel, llamada `AActor`. A su vez, los objetos instanciados de `AActor` pueden tener una serie de componentes que heredan de `UActorComponent`, otorgándoles nuevas funcionalidades como una representación física, colisiones, simulaciones de partículas entre muchas más.

La estructura utilizada para la implementación del juego posee diversos elementos que cumplen distintos propósitos, heredando de `UObject` o `AActor` dependiendo del caso. Se explicarán estos elementos desde los más generales a los más específicos de acuerdo a la

estructura interna de Unreal Engine.

Unreal Engine posee un flujo de ejecución que es recorrido por cada juego cuando es iniciado. En la figura 5.1 puede observarse una versión parcial de este flujo, donde sólo se muestra el camino seguido por un juego ejecutado de manera independiente. Es posible encontrar la versión original completa en la documentación oficial [11] desde donde se extrajo y tradujo el diagrama presentado. En la versión completa se muestra también el camino seguido por el juego si es que este es ejecutado de manera directa desde el editor que ofrece Unreal.

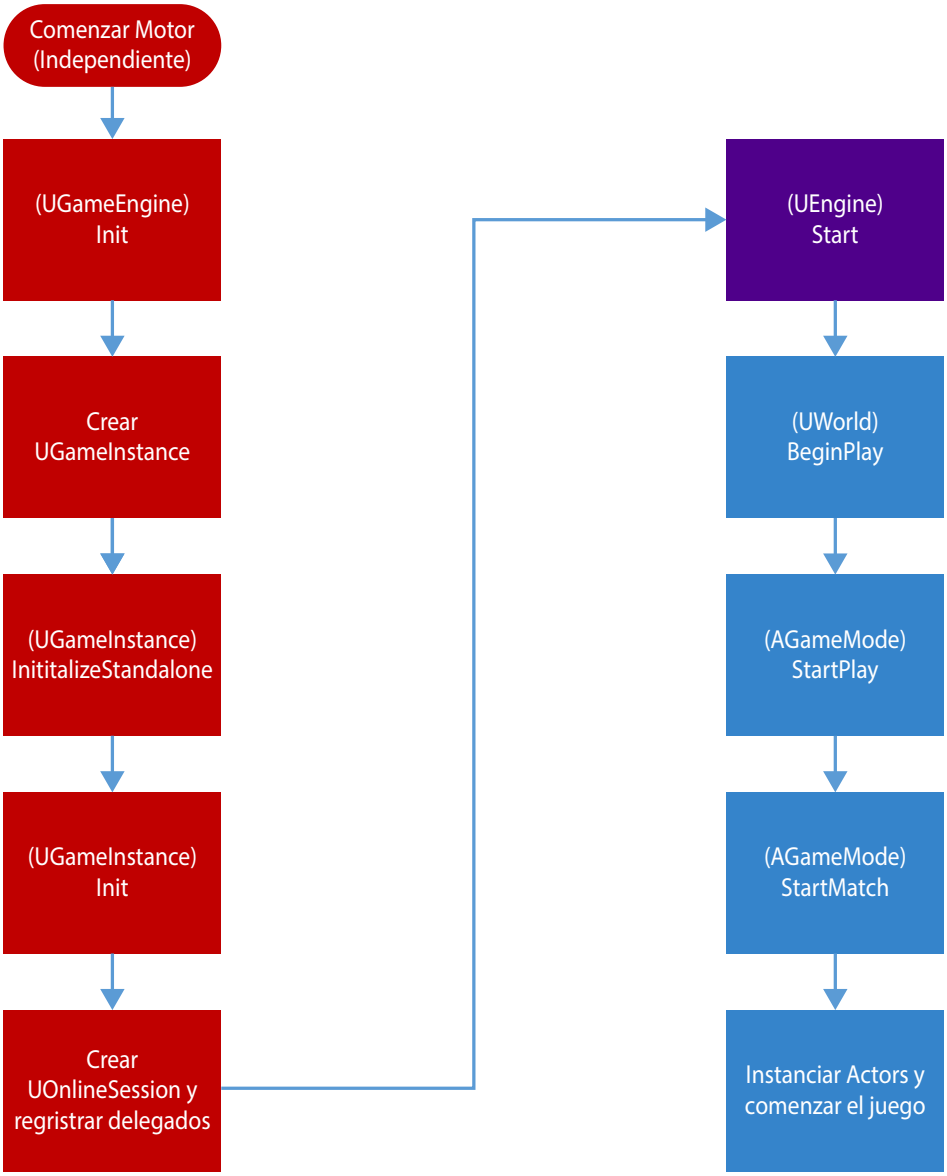


Figura 5.1: Flujo del juego resumido [11]

Dentro de los elementos del flujo se encuentra la clase `UGameInstance`. Esta clase posee la particularidad de siempre existir y al mismo tiempo ser única, asemejándose en términos de implementación al *singleton pattern*. Dada su naturaleza, esta clase puede ser accedida por todos los elementos del juego durante tiempo de ejecución, siendo así utilizada tanto

para guardar información única a lo largo de cada partida, como para ofrecer funcionalidades genéricas que pueden ser accedidas por cualquier clase. Es así como se crea la clase `URlGameInstance`, la cual hereda de `UGameInstace`, y se encarga de mantener las variables importantes del juego, como el nivel actual, el tiempo y la cantidad de muertes. También maneja parámetros de internos, tales como el uso del Mi Band por parte del jugador, o si éste se encuentra sujeto al cambio de dificultad dinámica.

Para facilitar el acceso a funcionalidades comunes dentro del juego se crean tres administradores o *managers*, los cuales se encargan de distintos aspectos del juego. Éstos corresponden a las clases `ULevelManager`, `UWidgetManager` y `AAudioManager`. Se guarda una referencia a cada uno de ellos como una variable de clase dentro de `URlGameInstance`, permitiendo de esta forma que puedan ser accedidos por cualquier elemento del juego. En la figura 5.2 es posible ver un diagrama de clases para `URlGameInstance`.

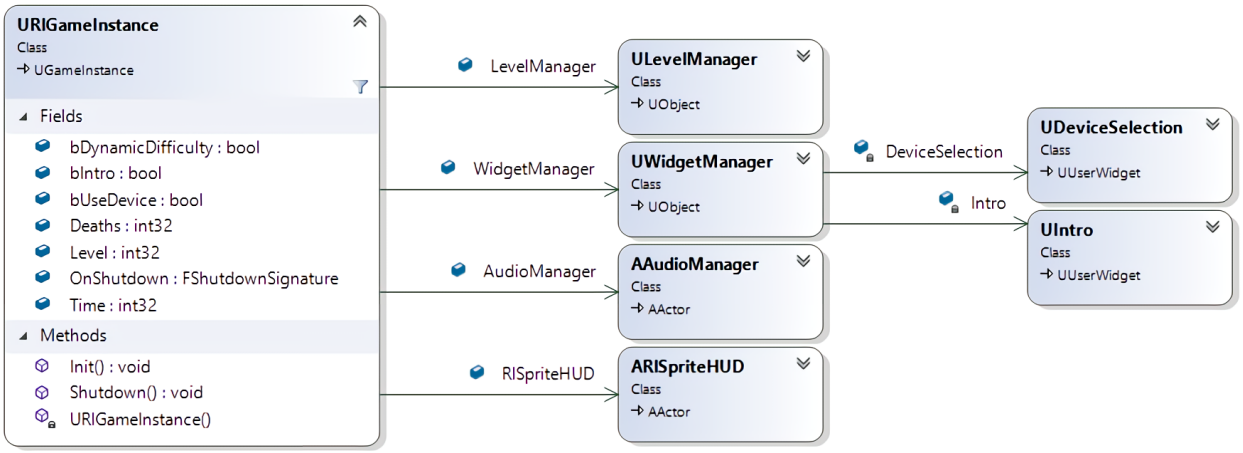


Figura 5.2: Diagrama de clases de `URlGameInstance`

`UWidgetManager` y `AAudioManager` poseen una implementación similar, ya que ambos se encargan de mantener referencias a recursos del juego de manera que puedan ser utilizados fácilmente desde código. `UWidgetManager` contiene elementos de la interfaz y métodos que permiten alternar entre las distintas fases del juego, como mostrar su historia o presentar la pantalla final cuando un usuario ha logrado llegar al nivel final. `AAudioManager` almacena las distintas músicas de fondo utilizadas, tanto para la historia, el juego y la pantalla final, teniendo también métodos que permiten seleccionar que música utilizar y cuando. A diferencia de los otros *manager*, `AAudioManager` hereda de `AActor` y no de `UObject`, ya que para ser capaz de emitir sonidos debe estar presente de manera física dentro de mundo. Es por esto que si bien `ULevelManager` y `UWidgetManager` se instancian dentro del método `Init` de `UGameInstace`, que se muestra en la figura 5.1, `AAudioManager` debe ser creado luego de que se inicial el mundo, siendo escogido el método `BeginPlay` de `AGameMode`.

5.3. Creación de niveles

`ULevelManager` es una de las clases principales de todo el juego, ya que es la encargada de administrar todo lo referente al manejo de niveles. Dentro de sus variables se encuentran referencias a todos los *sprites* que son utilizados por el juego de manera individual, es decir, que no pertenecen a algún *tilemap* o al personaje, como por ejemplo las escaleras y cada uno de los obstáculos. También contiene un arreglo con todos los niveles del juego, y es el encargado de instanciar la clase que maneja la creación y posicionamiento de los obstáculos, pudiéndose observar en la figura 5.3 su diagrama de clases.

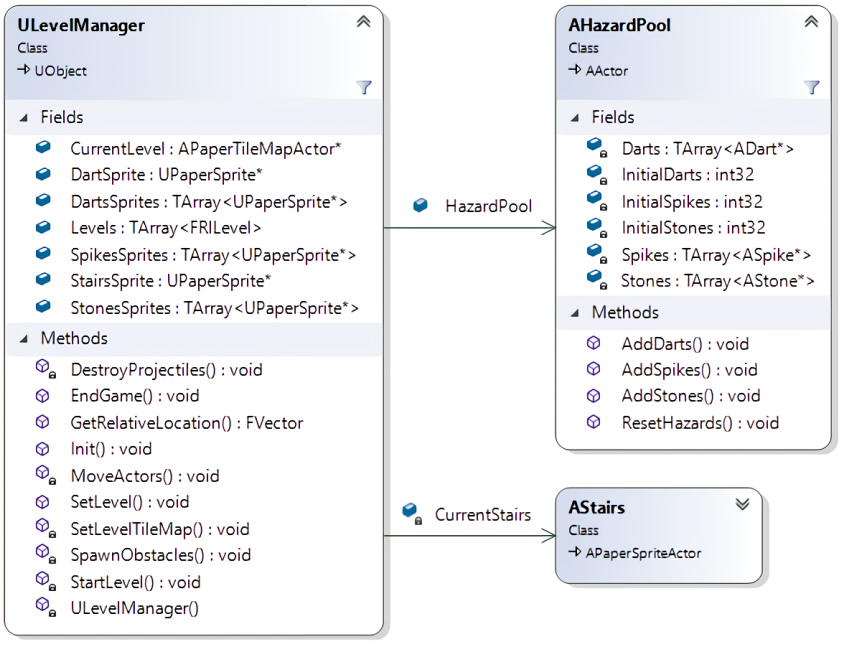


Figura 5.3: Diagrama de clases de `ULevelManager`

Cada nivel del juego es representado por una estructura llamada `FRlLevel`. Se crea una estructura y no una clase, ya que Unreal Engine ofrece la posibilidad hacer que las variables creadas en C++ sean visibles y modificables dentro de su editor a través de *Blueprints*, los cuales son la alternativa que posee Unreal al uso de código. Si bien con los *Blueprints* se pueden programar funcionalidades a través de uso de un lenguaje en base a nodos, sólo son usados como puente entre los recursos del editor y el código en C++, ya que en caso contrario se debería hacer referencia a los recursos a través de su ubicación y nombre exacto dentro de las carpetas del juego, provocando errores si son movidos o su nombre es modificado. Debido a lo anterior, para agregar un nuevo nivel sólo es necesario incorporar un nuevo `FRlLevel` al arreglo de niveles de `ULevelManager` y configurar sus variables sin la necesidad de modificar el código. La figura 5.4 muestra un ejemplo de la estructura de un nivel.

Si bien, como se mencionó con anterioridad, el *tilemap* que conforma un nivel se compone por 14 bloques horizontales y 7 verticales—donde cada bloque se conforma de 32x32 píxeles—para el posicionamiento dinámico de objetos dentro del nivel se utiliza un sistema de coordenadas que utiliza bloques de 16x16, teniéndose así 28 bloques horizontales y 14 verticales. De hecho, la estructura interna de cada *tilemap* se configura con bloques de 16x16

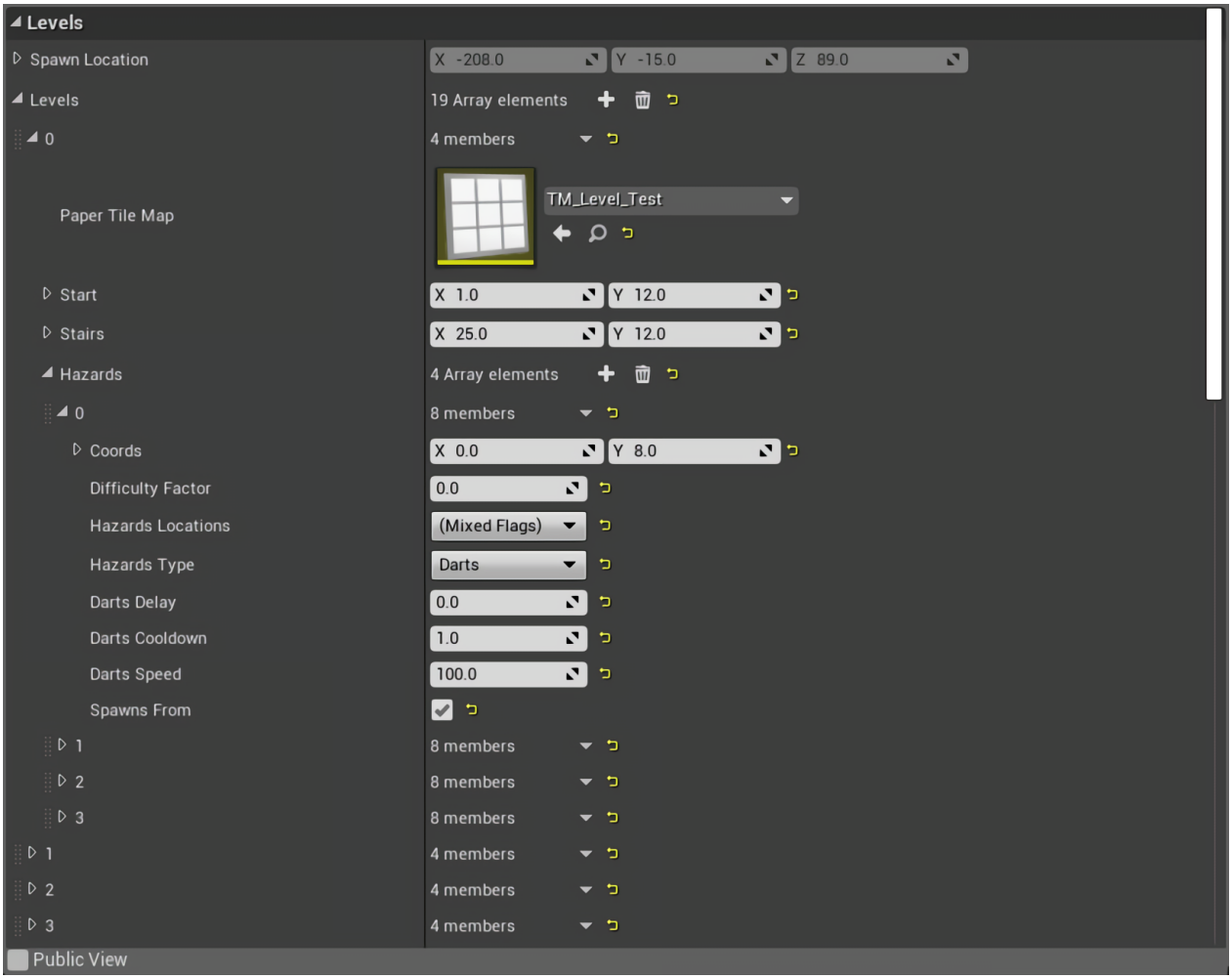


Figura 5.4: Ventana de creación de niveles

para así mantener consistencia entre las coordenadas. Es así como cada `FRlLevel` se compone de una referencia al *tilemap* que le brinda los elementos estructurales al nivel, una vector que indica las coordenadas en las que se posiciona al personaje cuando comienza el nivel o cuando muere y debe intentarlo nuevamente, un vector con las coordenadas de las escaleras y un arreglo de `FHazardsData`. Este último corresponde a una estructura compuesta por la data de cada obstáculo que está presente en el nivel.

5.4. Creación de obstáculos

En la figura 5.5 se muestra un diagrama con las clases relacionadas a la implementación de los obstáculos. Para representar a un obstáculo se implementa la clase `AHazard`. Esta clase posee como componente a un *sprite* debido a su herencia de `APaperSpriteActor`, y puede ser creada dentro del mundo gracias a su herencia de `AActor`. Se implementa también un método que permite posicionar y orientar al obstáculo dentro del mundo según sea requerido. Para esto, dado un punto de partida, se definen 16 posiciones con su respectiva orientación, las

cuales se organizan dentro de un bloque de 32x32 como se puede observar en la figura 5.6. La implementación de algún sistema de posicionamiento para los obstáculos era imperativa, ya que como se explicó con anterioridad, dadas las limitaciones en aspectos de personalización que poseen los *tilemaps*, no fue posible manejarlos como un *tile*, teniendo así que crearse una clase que los representara. Combinando este sistema con el sistema de coordenadas descrito con anterioridad, y considerando que la data de los obstáculos está expuesta al editor al ser parte de un `FRlLevel`, fue posible crear nuevos obstáculos dentro de un nivel simplemente indicando sus coordenadas y que combinación de posiciones se desea que tengan. De esta manera, se permitió la rápida construcción de niveles, a pesar de no poder utilizar de manera directa el entorno gráfico de los *tilemaps*.

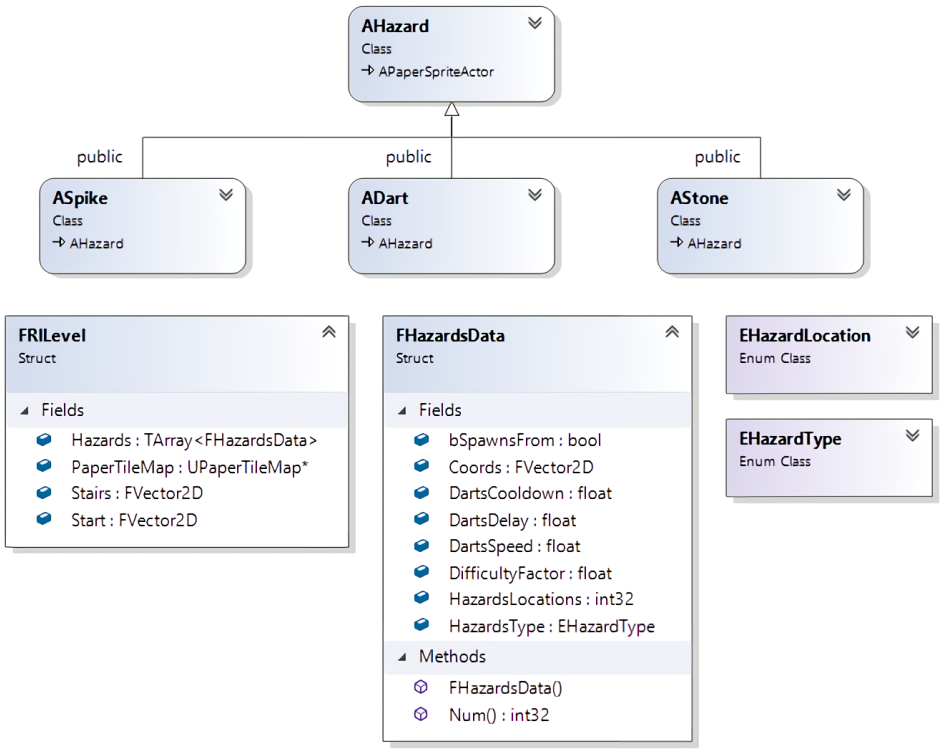


Figura 5.5: Diagrama de clases de los obstáculos

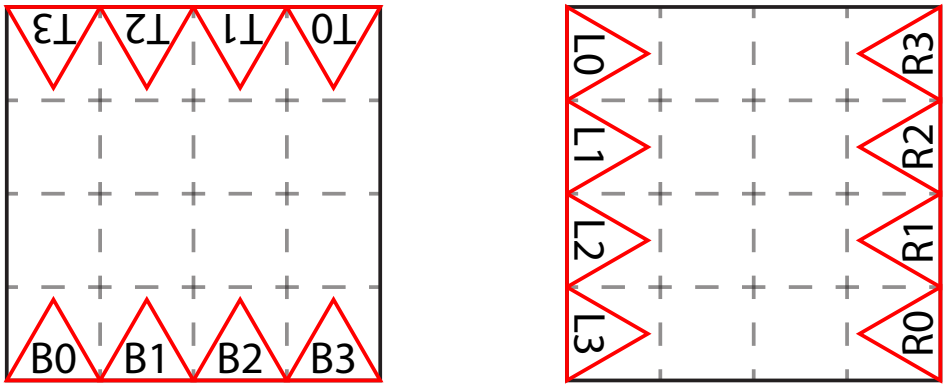


Figura 5.6: Sistema de posicionamiento de obstáculos

Como un `AHazard` sólo posee un *sprite*, se crean tres clases que heredan de éste: `ASpike`,

`ADart` y `AStone`. Por un lado, `ASpike` representa a las púas de metal, por lo que sólo es necesario activar sus colisiones e implementar un método que se ejecute cuando el jugador entre en contacto con éstas, lo que provoca su muerte. Por otro lado, `ADart` simboliza el punto de partida de las flechas, el cual a diferencia de `ASpike` no posee colisión. Entre sus variables se encuentra un retraso inicial que se espera antes de empezar a crear flechas, el tiempo que se espera entre la creación sucesiva de dos flechas y la velocidad a la que éstas se moverán. También permite activar y desactivar la creación de flechas según sea necesario. Finalmente, `AStone` corresponde a las piedras, las que si bien no pueden producir la muerte del jugador, sí colisionan con este, siendo así necesario activar sus colisiones. Por otro lado, ya que las piedras ocupan el doble de área que los demás obstáculos y no poseen una orientación, es necesario reescribir el sistema de posicionamiento. Para mantener la consistencia se siguen utilizando las 16 posiciones anteriormente definidas; sin embargo, éstas se ordenan en 4 grupos correspondientes a las esquinas del bloque de 32x32, por lo que por ejemplo B0, B1, L2 y L3 producirían el mismo resultado.

Si bien una flecha no es más que un objeto que se mueve de manera constante de acuerdo a una velocidad y dirección inicialmente definidas, es interesante explicar su implementación ya que posteriormente se utilizan las mismas bases para permitir el movimiento del jugador. Para las flechas se implementa la clase `AProjectile` heredando de `APaperSpriteActor`, por lo que posee un *sprite* al cual se le activan las colisiones. Para que un `AActor` sea capaz de moverse, éste debe poseer un componente que le brinde esta habilidad. Para esto existe una clase llamada `UMovementComponent`, la cual ofrece métodos que permiten su movimiento mientras se tienen en consideración los posibles objetos con colisión que este pude encontrar a lo largo de su camino. Dado que esta clase es definida como abstracta en los macros que utiliza de manera interna Unreal Engine, no es posible crear un componente que la utilice como base. Es por esta razón que se crea la clase `UDefaultMovementComponent`, la que sólo hereda de `UMovementComponent` y no implementa ninguna funcionalidad extra. Luego, en cada iteración del juego, `AProjectile` calcula la posición a la que una flecha se debería mover de acuerdo a su posición y velocidad actual, y utiliza a su componente de movimiento para intentar moverse a su siguiente destino. De encontrarse un elemento con colisión, como los muros o incluso otros obstáculos, detiene su movimiento, desactiva su colisión y activa un temporizador para ser destruida luego de un segundo. Al igual que `ASpike`, la colisión de una flecha con el jugador produce su muerte; sin embargo, también produce la destrucción inmediata de la flecha.

A pesar de que instanciar un `AActor` dentro del mundo no consume muchos recursos, si se instancian muchos al mismo tiempo puede ocasionar que el juego se detenga un instante. Esto inevitablemente provocaría molestia por parte del jugador. Si bien los obstáculos no poseen muchos componentes y por lo tanto no es muy probable que se produzca una interrupción al iniciar un nuevo nivel, se implementa de todas formas un sistema de *object pooling* para eliminar esta posibilidad. Este sistema consiste en instanciar previamente una determinada cantidad de obstáculos que podrían ser utilizados dentro de los niveles, de tal manera que al iniciar un nuevo nivel ya se cuente con obstáculos listos para ser utilizados, siendo necesario solamente ser posicionados en su ubicación correspondiente. Debido a esto se crea la clase `AHazardPool` la cual hereda de `AActor` ya que debe ser capaz de instanciar los obstáculos dentro del mundo. Esta clase posee tres arreglos diferentes, uno para cada tipo de obstáculo, así como también tres valores que representan la cantidad inicial que se creará de cada uno

de ellos. Ya que determinado nivel puede requerir de más obstáculos que los actualmente disponibles, se crean métodos que permiten instanciar nuevos obstáculos y añadirlos a sus arreglos correspondientes.

Los obstáculos instanciados por `AHazardPool` son posicionados inicialmente fuera de los límites del juego, a una distancia de 10.000 unidades en cada eje. Cuando se comienza un nuevo nivel es necesario ubicar de manera correcta cada obstáculo en la posición establecida dentro del nivel. Como se mencionó anteriormente, la información de los obstáculos dentro de un `FRlLevel` corresponde a un arreglo de `FHazardsData`. Esta estructura posee las coordenadas en las que se debe encontrar un obstáculo, su dificultad, su posicionamiento y su tipo, así como también variables específicas para las flechas como su retraso inicial, el tiempo a esperar entre su creación consecutiva y su velocidad, finalizando con un booleano que indica que lógica debe ser utilizada para su aparición dentro del nivel, es decir, si debe aparecer desde que la dificultad del juego sea mayor o igual que la del obstáculo, o si debe ser parte del nivel mientras la dificultad del juego sea menor o igual a la del obstáculo, desapareciendo para valores mayores. Es así como en `AHazardPool` se implementa un método que se encarga de recorrer todos los `FHazardsData` y por cada uno comprueba si debería estar presente según su dificultad, en cuyo caso toma el obstáculo del tipo correspondiente de los que tiene disponibles y los ubica en la posición indicada. De igual manera, si el obstáculo corresponde a una flecha, se encarga de activarlo y de configurarlo de manera adecuado. Finalmente, si no llegara a tener más obstáculos disponibles de determinado tipo, se encarga de crear más.

5.5. Manejo de niveles

Ya habiendo comprendido cómo se administran los obstáculos, es posible volver a la clase `ULevelManager`. Dado que esta posee toda la información necesaria, es la encargada de construir el juego y mantener el correcto flujo de acuerdo a las acciones del jugador. Para ello, se implementan dos métodos principales: `SetLevel` y `StartLevel`, siendo mencionados sus nombres debido al importante rol que poseen ya que sin ellos no existirían niveles que jugar.

`SetLevel` recibe como argumento un modo que le indica que acción se desea ejecutar, empezar el juego, reiniciar un nivel o avanzar al siguiente. Si se desea comenzar el juego se restablece el contador de niveles. En cambio, si se quiere avanzar al siguiente, este contador es incrementado, no existiendo cambios si el cambio corresponde a un reinicio. Luego dependiendo del nivel en el que se encuentra el jugador se le activan o desactivan distintas habilidades del personaje, proceso que es utilizado para las etapas de tutorial del juego, en las cuales se van introduciendo una a una las mecánicas del juego al jugador. También se verifica si ya se ha completado el tutorial y, por tanto, resulta necesario detener el proceso de calibración del pulso para así empezar con el ajuste dinámico de dificultad. De igual manera, comprueba si el usuario ha llegado al nivel final, en cuyo caso procede a mostrar la pantalla de victoria del juego. Finalmente activa un proceso en el que se lleva de manera continua la pantalla a negro para revelar posteriormente el nuevo nivel, ya sea el mismo si es que el personaje acaba de morir, o el siguiente si logró llegar hasta las escaleras. Cuando la pantalla está completamente en negro es cuando se llama a `StartLevel`.

Al igual que `SetLevel`, `StartLevel` sólo recibe el tipo de acción que se desea ejecutar. Este método se encarga de crear los componentes que forman un nivel, así como también de reorganizarlos dependiendo del caso. Si se quiere comenzar el juego, primero se destruyen todos los elementos existentes, eliminando cada una de sus referencias para así comenzar desde cero. Luego se instancian y configuran las escaleras, se crea el *tilemap* que estará encargado de contener el nivel actual, así como también el *tilemap* que contienen los inputs mostrados al jugador durante el tutorial, junto a la clase encargada de animarlo. Finalmente se crea el manejador de obstáculos `AHazardPool`. Si no se está reiniciando el nivel, se procede a establecer el *tilemap* a utilizar se acuerdo al nivel actual y se verifica si no se tienen suficientes obstáculos disponibles para el nivel, en cuyo caso se agregan más del tipo que corresponda. Independiente de qué modo se haya recibido como argumento, se destruyen todas las flechas presentes en el nivel anterior, se mueve el personaje, las escaleras y los obstáculos a las posiciones requeridas, se actualiza la animación del tutorial y si es que el personaje acaba de morir se vuelve a hacer visible. Así pues, se procede a revelar el nuevo nivel al hacer desaparecer de manera continua la pantalla de color negro que estuvo ocultado todo el proceso anterior.

Dado que `ULevelManager`, al estar referenciado dentro de `URlGameInstance` es accesible para todas las clases, el método `SetLevel` es llamado en sus distintos modos por diferentes componentes del juego según corresponda. Uno de esos corresponde al personaje cuando muere, comandando que se reinicie el nivel, o cuando intenta entrar a través de las escaleras, permitiendo así progresar al siguiente nivel. Lograr esto último requiere que las escaleras sean implementadas como un elemento independiente, por lo que se crea la clase `AStairs` heredando de `APaperSpriteActor`. La configuración de las colisiones utilizadas para este objeto es diferente a todos las demás, ya que no debe bloquear los movimientos del personaje. Es así como a las escaleras se les agrega un componente que si bien no posee una colisión bloqueante, es capaz de determinar si un ente ha entrado o salido del volumen que cubre. Luego, si este ente corresponde al personaje, se le otorga la facultad de avanzar al siguiente nivel presionando el input adecuado, capacidad que le es removida si sale de la zona que cubre el componente.

Al iniciar el juego, se invoca a `SetLevel` con el modo correspondiente, dependiendo de cuál es la configuración actual que se haya solicitado de acuerdo a los argumentos de línea de comandos utilizados. Éstos pueden ser: `-nodevice` para indicar que no será utilizado un dispositivo de medición del pulso a lo largo del juego, y `-nointro` si no se desea ver la historia del juego. Considerando la figura 5.1 anteriormente mostrada, el primer chequeo se realiza cuando se inicializa la clase `AGameMode`. Más concretamente se crea la clase `ARlGameMode` que la extiende. El objetivo de esta clase es definir las reglas que poseerá el juego, y como es instanciada inmediatamente luego de la creación del mundo, es buen punto de partida para decidir qué ruta tomará el juego. El chequeo realizado corresponde a analizar si será utilizado un Mi Band, llevando al usuario a la pantalla de selección de dispositivos. En caso contrario, se realiza una nueva comprobación acerca si se desea ver la historia del juego. Si lo anterior es cierto, se procede a mostrar la pantalla que contiene la historia del juego, sino se llama de manera directa a `SetLevel` para así iniciar el juego.

5.6. Interfaces de usuario

Pasando a los elementos de la interfaz del juego se pueden mencionar dos tipos. El primero lo conforman las distintas pantallas mostradas a lo largo del juego, como la selección de dispositivos, la historia del juego y la imagen de victoria. Estas pantallas se implementan utilizando *UMG*, el cual es una herramienta visual de creación de interfaces provista por el editor de Unreal Engine. De esta manera, se tiene a disposición una gran variedad de elementos como texto, botones, y contenedores para el correcto posicionamiento de objetos. También posee una línea de tiempo, la cual permite la animación de cada una de las propiedades de cada uno de estos elementos.

Las interfaces creadas por esta herramienta se llaman *Widget Blueprint* e internamente utilizan a la clase `UUserWidget` como base, siendo posible crear nuevas clases que la extiendan para así conectar la funcionalidad implementada en el *Blueprint* con código en C++. Es así como se crean las clases `UDeviceSelection` y `UIntro`. `UDeviceSelection` se encarga de implementar las funcionalidades de la pantalla mostrada anteriormente en la figura 4.3, recibiendo la lista de los dispositivos disponibles desde el servidor. Cuando el usuario selecciona un botón, se envía la dirección MAC asociada a la aplicación a cargo de la comunicación Bluetooth, manejando casos bordes como cuando la conexión no se lleva a cabo, o dando la opción de cancelar el uso del Mi Band si no hay ningún dispositivo cercano. `UIntro` por su lado se relaciona con el *Widget Blueprint* que contiene tanto la secuencia de imágenes que componen la historia del juego, como la imagen mostrada cuando el usuario completa todos los niveles, junto a sus respectivas animaciones. Es así como se encarga de comenzar las animaciones según se cumplan las condiciones necesarias y de continuar con las siguientes fases del juego, como por ejemplo llamar a `SetLevel` cuando termina la historia.

El segundo tipo de interfaz corresponde a la utilizada para mostrar información al usuario durante el juego o *HUD*. Ésta no se implementa utilizando *Widget Blueprint*, ya que no ofrecen la posibilidad de utilizar una fuente de texto personalizada basa en imágenes. Es aquí donde se observa una de las falencias de Paper2D debido al abandono en que se encuentra por parte de los desarrolladores, ya que la mayoría de los motores que ofrecen el desarrollo de juegos en dos dimensiones poseen de manera nativa la capacidad de crear fuentes de texto basadas en una imagen que contenga cada uno de los caracteres que se desee utilizar. En efecto, esto es esencial para mantener la estética del juego, sobre todo si se está trabajando con *pixel art*. Este tipo de fuente se conoce como *bitmap font* o *sprite font*, dado que cada letra es en sí un *sprite* que posee un diseño y colores según sea necesario, en comparación a una fuente regular con la cual sólo se podría generar texto plano y de color negro.

Es así como se debe implementar una herramienta que haga posible el uso de *sprite fonts*, por lo cual se crea la clase `ASpriteTextActor` la cual hereda de `AActor`. La variable principal de esta clase está formada por un diccionario el cual se encarga de traducir entre caracteres y *sprites*. Esta variable es expuesta al editor de Unreal Engine para que así pueda ser completado de manera directa con los caracteres que sean necesarios según el caso. De igual manera, se posee como variable el texto que debe ser mostrado, así como también distintos modos de alineación horizontal, vertical y espaciado. En particular, esta clase no hace más que tomar el texto objetivo e instanciar un *sprite* por cada letra que lo compone en

la posición correspondiente de acuerdo a los parámetros que le son entregados. Una solución más permanente hubiera sido implementar esta funcionalidad dentro del motor en sí; sin embargo, esto dificultaría tanto el futuro desarrollo del proyecto al tener que adaptar Unreal Engine cada vez que sea lanzada un actualización, así como también su distribución a terceros que deseen ver como funciona el código y probarlo, ya que cada uno debería realizar la modificación por cuenta propia.

Para controlar y actualizar la información del *HUD* del juego se crea la clase `ARlSpriteHUD`, la que guarda una referencia a los `ASpriteTextActor` que son utilizados para mostrar el nivel actual, la cantidad de muertes y el tiempo transcurrido. Esta clase se encarga de inicializar la interfaz del usuario cuando comienza el juego y ofrece métodos para incrementar el nivel y la cantidad de muertes por otros elementos del juego, además de manejar de manera autónoma el contador de tiempo del juego con su respectivo formato en horas, minutos y segundos.

Las interfaces creadas mediante *Widget Blueprint* son agregadas de manera directa frente al mundo que ve el jugador por parte de Unreal Engine. Sin embargo, para que éste pueda visualizar de manera correcta este mundo es necesaria la presencia de una cámara. Como el juego desarrollado se compone de niveles de una pantalla, a diferencia de otros juegos de plataforma, la cámara no sigue al personaje sino que se mantiene fija en el centro con tal de mostrar la completitud del nivel y las interfaces creadas manualmente. La estructura general de Unreal Engine indica que cada mundo debe considerando como un nivel independiente. Es por esta razón que existe un elemento llamado *Level Blueprint*, el cual es utilizado para implementar funcionalidades que tengan que ver con el nivel actual. Como en este caso se utiliza un sólo mundo durante todo el juego, se posee un único *Level Blueprint*, siendo así utilizado para crear y configurar la cámara que será en todos los niveles. Para esto se crea la clase `ARlLevelScriptActor` y se configura como padre de *Level Blueprint*. Dentro de esta clase se crea una cámara, se establece que utilice proyección ortográfica en vez de perspectivas y se le comanda utilizar un ancho específico de tal manera que sea posible ver de manera precisa toda la extensión del nivel y se mantengan las proporciones requeridas, considerando el zoom aplicado sobre el juego.

5.7. Desarrollo del personaje

Finalmente se tienen las dos últimas clases, que junto a `ULevelManager` permiten la existencia y funcionamiento base del juego. Estas corresponden al personaje y el componente encargado de habilitar su movimiento de acuerdo a los inputs del jugador. Como se mencionó con anterioridad, la plantilla *2D Side Scroller* utilizada ya contiene implementado un personaje, el cual puede colisionar con obstáculos, saltar y caminar. Sin embargo, éste no es utilizado en el juego sino que se implementa uno nuevo que, si bien posee similitudes, difiere en aspectos clave. Para comprender por qué no se utilizó el personaje que viene por defecto es necesario primero conocer cómo éste está estructurado. En la figura 5.7 se presenta un diagrama de clases para el personaje y su movimiento.

Como los elementos que conforman la plantilla adoptan como prefijo el nombre del proyecto, el personaje está implementado en la clase `ARageliteCharacter`, la cual hereda de

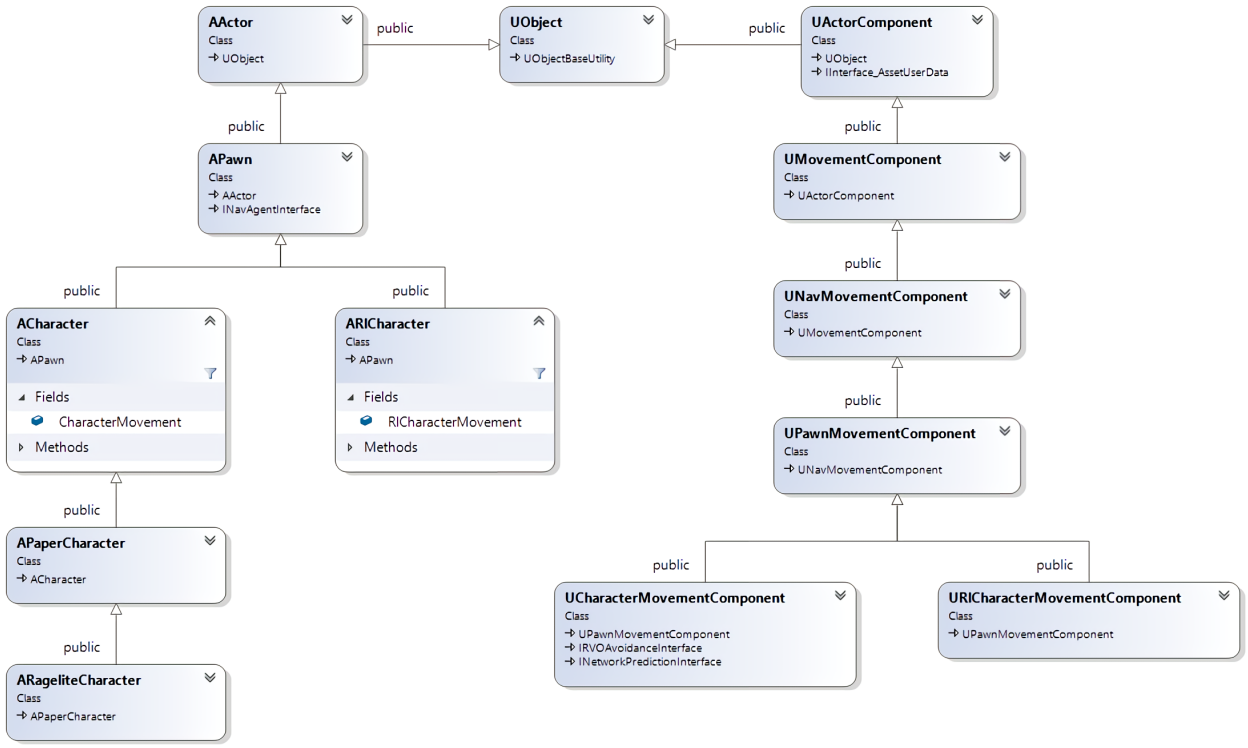


Figura 5.7: Diagrama de clases del personaje y su movimiento

`APaperCharacter`. Dado que esta clase está pensada para ser utilizada como base de un juego de plataformas típico, dos de sus componentes corresponden a una cámara y un soporte que la mantiene constantemente siguiendo al personaje. De igual forma posee una referencia hacia dos *flipbooks*, uno que contiene la animación de cuando corre y otro con la animación que se reproduce al no existir la presencia de inputs por parte del usuario. También posee un método que determina qué *flipbook* utilizar en base a la velocidad el personaje y se preocupa de igualmente rotar la animación actual de acuerdo a si el movimiento indicado por el jugador es hacia la derecha o la izquierda. Por otro lado se encarga de conectar los inputs del usuario con los métodos que permiten su movimiento y salto.

Ya que `ARageliteCharacter` hereda de `APaperCharacter`, recibe un componente que le permite utilizar cualquier *flipbook* que sea necesario, brindándole así una representación gráfica dentro del mundo. Si bien con esto pareciera que `APaperCharacter` es una clase diseñada específicamente para representar un personaje en dos dimensiones, lo único que hace esta clase es heredar toda su funcionalidad de la implementación que posee Unreal Engine para un personaje en tres dimensiones `ACharacter`, y ocultar el componente que posee el modelo 3D que normalmente sería el que le permitiría ser visible dentro del mundo.

`ACharacter` en una clase que está diseñada para proveer colisión y lógica de movimiento a un personaje cuyo diseño se encuentra orientado de forma vertical, como personas o cualquier bípedo, implementando los estados de caminar, saltar, volar y nadar gracias que posee un componente de movimiento llamado `UCharacterMovementComponent`. Aquí es donde se presenta el problema principal, dado que esta clase está pensada para ser usada para personajes bípedos, toda su implementación y lógica de movimiento está construida a partir del uso de

una cápsula vertical como componente de colisiones, la cual si bien puede ser escalada según sea necesario, no puede ser rotada, ya que esto invalidaría toda su implementación. Dado que el personaje principal del juego no es bípedo sino que corresponde a un gato calicó, no es posible adaptar una cápsula vertical para que represente de manera confiable sus colisiones con el entorno, siendo este un requerimiento imperativo debido a la naturaleza del juego. Así pues, el éxito del jugador corresponde en su totalidad la relación que se forma entre los inputs que éste presiona y el comportamiento esperado por el personaje. También, dado el diseño del personaje y que el estilo del juego es *pixel art*, el uso de una cápsula resultaría en un comportamiento indeseado debido a que su parte inferior es esférica. Esto provoca, por ejemplo, que si el jugador se acerca al borde de un bloque, el personaje descendiera en altura mientras sigue aún sobre este, en vez de caer de manera inmediata que sería lo esperado. Es por esto que el elemento de colisión más acorde correspondía a un cubo. Sin embargo de la misma manera en que es imposible rotar la cápsula, también lo es cambiar que elemento es utilizado para gestionar las colisiones.

Debido a lo anterior no es factible utilizar `ACharacter` como base del personaje, lo cual al seguir la cadena de herencias provoca que la clase `ARageliteCharacter`, que viene como ejemplo dentro de la plantilla utilizada, no cumpla con los requerimientos necesarios para dar vida al gato que controlará el jugador. Es así como se hace necesario implementar una nueva clase que represente al personaje. Al analizar `ACharacter` es posible ver que ésta hereda de la clase `APawn`, la cual corresponde a base de todos los `AActor` que pueden ser controlados por el jugador, correspondiendo así a la representación física de jugadores y criaturas dentro de un nivel. Como esto es lo mínimo necesario, para la implementación del personaje se crea la clase `ARlCharacter` heredando de `APawn`.

Utilizar una clase nueva implica crear todas las funcionalidades necesarias desde cero, haciéndose esto más evidente al considerar el movimiento del personaje. Como se mencionó con anterioridad, `ACharacter` utiliza a `UCharacterMovementComponent` para hacer factible su movimiento. Sin embargo, este componente está específicamente diseñado para su uso dentro de un `ACharacter`, a tal punto que dentro del código de ambos realizan chequeos mutuos para asegurarse de que se están utilizando estas clases y ninguna otra. Es así como no es posible utilizar este componente para proveerle lógica de movimiento a `ARlCharacter`, siendo también necesario implementar desde cero una clase que constituya un nuevo complemento de movimiento, la cual es llamada `URlCharacterMovementComponent`. Si bien estas nuevas clases se encuentran fuertemente entrelazadas, llamando cada una de manera consecutivas a los métodos implementados dentro de la otra, se continuará con la explicación de `ARlCharacter` para luego continuar con `URlCharacterMovementComponent`.

`ARlCharacter` es la clase que posee la mayor cantidad de funcionalidades dentro del juego. Se encarga desde la coordinación entre los inputs que presiona el usuario y su componente de movimiento, hasta la creación y mantenimiento del cliente y el servidor que se comunica con la aplicación a cargo del Mi Band. Para su implementación se realiza un análisis del código de `ACharacter` en su totalidad, por lo cual se adoptan algunas funcionalidades y se crean otras basadas en las ya existentes. Dado que corresponde a la representación física del personaje, posee un *flipbook* como variable, así como también una referencia a cada una de las animaciones utilizadas creadas para otorgarle vida al personaje.

Para manejar de manera correcta las distintas transiciones entre las animaciones del personaje, se modela la máquina de estados de la figura 5.8, donde cada nodo está ligado a una animación independiente.

Los estados son los siguientes:

- `Idle`: El jugador no está presionando ningún input. Para salir de este estado se debe presionar el input para caminar, saltar, o bien caer desde alguna plataforma.

- `ToIdle`: El jugador estaba presionando previamente el input para caminar, sin embargo dejó de hacerlo. Al terminar la animación de este estado se pasa inmediatamente a `Idle`, sin embargo es posible llegar a otros estados de la misma que manera que `Idle` mediante el uso del input para caminar, saltar o caer desde una plataforma.

- `ToIdleR`: El jugador no estaba previamente presionando ningún input, sin embargo ahora presionó el input de caminar. Cuando acaba la animación de este estado se continúa con `Running`, aunque se puede volver a `ToIdle` si se deja de presionar inputs.

- `Running`: El jugador está activamente presionando el input para caminar. Salir de este estado implica dejar de caminar, saltar, o caer desde una plataforma, no siendo afectado por los cambios de dirección del personaje, a pesar de que en estos la velocidad pasa por cero.

- `ToJumping`: El jugador acaba de presionar el input para saltar. Cuando se termina la animación se da paso a `Jumping`.

- `Jumping`: El jugador presionó previamente el input para saltar. Se mantiene el estado mientras el personaje continúe ascendiendo producto de la fuerza inicial del salto, pasando en caso contrario a `ToFalling`.

- `ToFalling`: El personaje llegó al punto más alto del salto y se prepara para comenzar a caer. El término de la animación de este estado da inicio a `Falling`.

- `Falling`: El personaje está cayendo producto de la fuerza de gravedad. Al chocar con objeto en el que pueda caminar se realiza un transición a `Landing`.

- `Landing`: El personaje estaba previamente cayendo. Sin embargo llegó a una superficie sólida en la cual puede aterrizar, por que al terminar la animación de este estado se vuelve a `Running`.

Para implementar esta máquina de estados se crean dos estructuras que representan los nodos y sus transiciones. Un nodo corresponde a un `FRlAnimation`, el cual posee como variables: (1) una función que le indica que animación utilizar, (2) un indicador que determina si la animación corresponde a un ciclo, (3) otro indicador que dice si la animación debe ser reproducida en reversa, (4) un número que establece el frame inicial y otro el final de la animación, (5) un indicador sobre si se desea empezar la animación desde el frame actual,

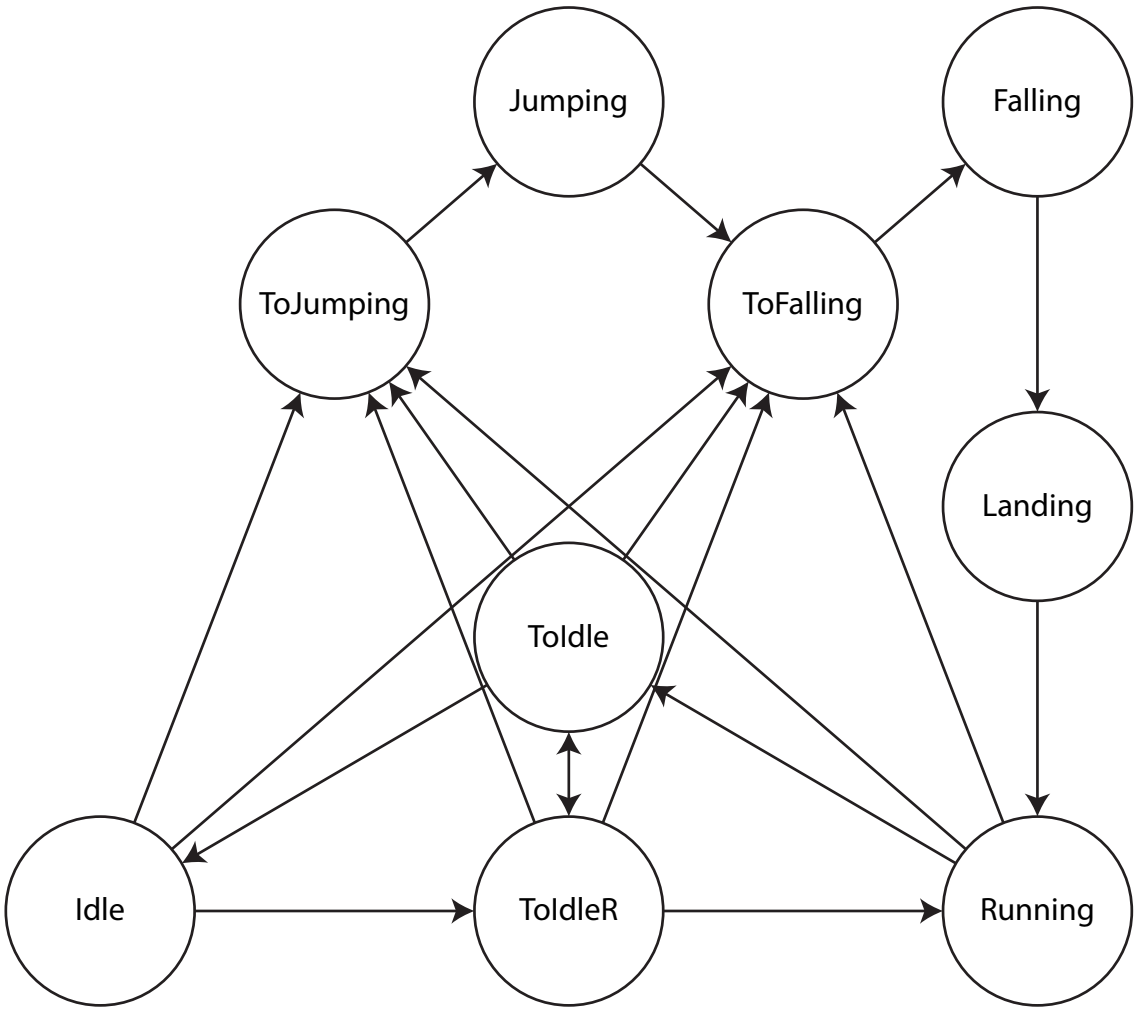


Figura 5.8: Máquina de estados de la animación del personaje

lo que es útil por ejemplo cuando se cancelan movimientos y se utiliza la misma animación pero en reversa, y (6) finalmente un arreglo de transiciones. A esta estructura se le entrega una función y no una animación en concreto debido a que el diseño del gato no es simétrico, teniendo distintos colores a lo largo de su cara y cuerpo, por lo que por cada animación existe su versión alterna donde el gato mira hacia la dirección opuesta. Poseer una función permite que ésta sea ejecutada cuando se requiera conocer que animación utilizar, siendo su resultado determinado por el contexto actual del personaje.

Una transición se representa por la estructura `FRlTransition`, la cual contiene un predicado y el siguiente estado. Un predicado corresponde a una función que al ejecutarse retorna un booleano. Es así como en cada iteración del juego se recorren las transiciones del `FRlAnimation` asociado al estado actual del personaje y si alguno de sus predicados se cumple se procede a cambiar el estado actual.

Los estados son manejados como un *enum*, por lo que para poder obtener el `FRlAnimation` correspondiente se crea un diccionario que asocia un estado con su estructura análoga. Cada vez que ocurre un cambio de estado se cambia la animación a la indicada por la función presente en `FRlAnimation` y se ejecuta según las especificación presentes en esta misma es-

tructura. Como por defecto la implementación del componente encargado de manejar los *flipbook* sólo posee controles simples como reproducir la animación desde su inicio, o ejecutarla en reversa desde el final, se implementan una gran variedad de métodos auxiliares que manipulan los frames de tal manera que permitan lograr el grado de control requerido para la visualización fluida de las animaciones y sus transiciones.

Aparte de las animaciones, se crean dos efectos de partículas: uno para representar la muerte del jugador y otro para dar más énfasis a los cambios de velocidad cuando el jugador camina, permitiéndole diferenciar con mayor facilidad cuando el personaje está corriendo. Como la muerte del personaje es un evento que sólo ocurre una vez dentro de cada nivel, para mostrar su efecto asociado sólo es necesario crear una instancia de él dentro del mundo en la posición en la que se encontraba el personaje y destruirlo cuando haya terminado de reproducirse. Sin embargo, como el efecto utilizado cuando el personaje corre depende en su totalidad de los inputs presionados por el usuario, y por lo tanto no posee una duración definida, no es factible utilizar el mismo apronte que para el efecto anterior, debiendo crearse un componente dentro de `ARlCharacter` que se encargue de mostrar y ocultar este efecto según corresponda de acuerdo al estado actual del personaje y los inputs del usuario.

Dado que el objetivo principal de crear una nueva clase corresponde a poder utilizar un elemento de colisión diferente a la cápsula que posee `ACharacter`, se utiliza un cubo que cubra de manera correcta las dimensiones del *sprite* del personaje en sus distintos estados, dejando algunos píxeles de tolerancia para que así el jugador pueda acercarse un poco a los obstáculos sin provocar su muerte de manera inmediata. Se utiliza un cubo ya que Unreal no posee un motor dedicado para el manejo de elementos en dos dimensiones, lo que implica que todas las colisiones se realicen en un espacio tridimensional, independientemente de si se está utilizando elementos de Paper2D. De hecho cuando se le otorgan colisiones a un *sprite*, o a un *tile* dentro de un *tileset*, estas corresponden a cubos o prismas dependiendo de la forma que se les asigne. Esto implica que se debe prestar especial atención respecto al posicionamiento de objetos dentro del eje de profundidad. Si bien sus posiciones horizontales y verticales pueden ser correctas, es posible que las colisiones no se estén ejecutando debido a su lejanía respecto a este eje. Esto se puede evidenciar principalmente al utilizar los *tilemap*, pues como se mencionó con anterioridad, sus sistema de capas no en más que ubicar a los objetos dentro de distintos planos ubicados a lo largo del eje de profundidad, siendo así necesario configurar la distancia entre las capas de tal manera que todos los elementos que deben colisionar sean capaces de hacerlo.

Las consecuencias de las colisiones en las que incurre el personaje son manejadas por su componente de movimiento y por los distintos obstáculos. Luego, si el personaje colisiona por ejemplo con una púa, ésta es quien llama al método de `ARlCharacter` encargado de procesar este evento, comprobando previamente que el personaje no se encuentre muerto o que esté invencible, estado que adquiere cuando entra por las escalas y pierde al inicio del nuevo nivel. Cuando el personaje muere, su *sprite* es ocultado, se crea su respectiva simulación de partículas dentro del mundo, se envía un comando para que el Mi Band vibre, se desactivan los inputs por parte del usuario, se incrementa el contador de muertes y se llama a `SetLevel` para así reiniciar el nivel actual.

Es también dentro de `ARlCharacter` donde se establece la conexión entre los inputs que

presiona el usuario y los métodos correspondientes que permiten que el personaje reaccione adecuadamente a éstos, implementándose de tal manera que puedan ser utilizados con un mando o *gamepad*. Los inputs disponibles corresponden a moverse horizontalmente, saltar, entrar a las escaleras, y correr. Se establecen también variables que permiten ignorar diferentes inputs según sea necesario, las cuales son utilizadas durante el tutorial para que así el jugador pueda comprender las mecánicas el juego una a una.

Cuando se recibe un input de movimiento horizontal, éste es derivado directamente al componente de movimiento del personaje, luego de realizarle un pequeño ajuste si es que se está presionando de manera simultanea el input para correr. Sin embargo este no es el caso al detectar la presencia del input de salto, debido a que su efecto depende de qué tanto tiempo se mantenga presionado, alterándose la altura a la que llega el personaje. También si ya se ha ejecutado un salto y el usuario presiona nuevamente este input, se debe comandar al personaje a caminar por el muro del fondo en vez de realizar un nuevo salto, impidiendo que este input posea algún efecto más hasta que no se aterrice en alguna superficie sobre la que se pueda caminar. Como tanto saltar como caminar por el muro poseen distintos resultado dependiendo de cuánto se presione el input, para cada uno de estos estados se crean dos variables: una que indica cuánto tiempo ha estado presionando el input y otra que establece el tiempo máximo a considerar. De esta forma, no hay ningún efecto adicional si el jugador continúa presionando el input pasado este límite. Son estas variables las que utiliza el componente de movimiento para controlar al personaje según corresponda.

Finalmente, `ARlCharacter` se encarga de la conexión entre el juego y la aplicación que se comunica con el Mi Band. Es así como al iniciar el juego se crea un servidor y se asigna un método encargado de ejecutarse cuando algún cliente desee conectarse para enviar información, configurando adecuadamente el socket y buffer a utilizar. Para recibir los mensajes, en cada iteración del juego se revisa si el buffer encargado de almacenar la información entrante contiene algún dato. Dependiendo del estado general del juego, este buffer puede contener las direcciones MAC de los dispositivos, el mensaje `Connected`, o el pulso del jugador. La transición entre los estados se realiza mediante el uso de variables que se modifican de acuerdo a los eventos ocurridos. Cuando se recibe una dirección MAC, ésta es enviada a la pantalla de selección de dispositivos para así ser agregada como una opción seleccionable por el usuario. Si el mensaje recibido es `Connected`, se remueve la pantalla de selección de dispositivos, se comienza la medición del pulso, se indica al módulo encargado de procesar el pulso que debe empezar a almacenar información para el posterior proceso de calibración, y se lleva al jugador al siguiente estado, el cual puede corresponder a la historia del juego o a la ejecución de `SetLevel` para empezar los niveles si es que se ha indicado no mostrar la historia. Si se recibe el pulso del jugador este es redirigido al módulo encargado de su procesamiento, contribuyendo como información si aún no se comienza con el proceso de ajuste de dificultad, o afectando de manera directa la experiencia del jugador si este proceso ya ha sido iniciado.

Por otro lado, para el envío de comandos desde el juego hacia la aplicación que maneja los dispositivos se implementan una serie de métodos que se comunican con el servidor de esta aplicación mediante la creación de un cliente que envía el mensaje correspondiente a través de sockets. Entre estos métodos se encuentra el encargado de iniciar el cliente dentro de la aplicación, para que así el juego reciba tanto la lista de los dispositivos cercanos disponibles como las mediciones del pulso. De igual manera se implementa un método que dado una

dirección MAC escogida por el usuario la envíe de vuelta para que así sea posible establecer la conexión con la pulsera y acceder sus funcionalidades, como vibrar, escribir un mensaje en pantalla e iniciar la medición del pulso, siendo creado un método distinto para cada uno. Cabe destacar que cada vez que se envía un comando al servidor de la aplicación se está creando un nuevo cliente, ya que los comandos enviados dependen totalmente de las acciones del jugador. Así, no existe una certeza de en qué momento y si es que serán ejecutados. Luego, tener a un único cliente conectado de manera constante no correspondería a una solución razonable, sobre todo considerando que la ejecución de estos comandos puede ser solicitada de manera simultánea, y por lo tanto se debería implementar un sistema de cesión del cliente cuando estuviera disponible si es que sólo se utilizara uno. Este no es el caso del cliente que se ejecuta dentro de la aplicación, debido a que este está continuamente enviando información, ya sea de las direcciones MAC de las pulseras en un inicio, o de los pulsos del jugador luego de realizar la conexión. Por lo tanto, tanto se encarga de una sola tarea a la vez, siendo así necesario un sólo cliente que esté continuamente conectado. Una vez finalizado el juego, se liberan los recursos utilizados por ambas partes, procurando cerrar adecuadamente los sockets para que así puedan ser ocupados en una posterior ejecución del juego.

5.8. Implementación del movimiento

Continuando con el último de los elementos base del juego, se encuentra el componente utilizado por `ARlCharacter` para comandar su movimiento. Como ya se explicó con anterioridad, como no se está usando la clase `ACharacter` no es posible utilizar el componente de movimiento `UCharacterMovementComponent` ya que este está implementado de manera específica para su uso con esta clase. Es así como se procede a realizar un análisis completo de su código para así tener una idea general de cómo está manejando el movimiento y poder recrear algo similar que cumpla con los requerimientos necesarios para el juego desarrollado. Este componente hereda de la clase `UPawnMovementComponent`, la cual está diseñada para gestionar el movimiento de un `APawn`, otorgando de igual manera la posibilidad de acumular los inputs recibidos en forma de un vector que puede ser utilizado para comandar el movimiento del objeto al que este componente pertenezca. A su vez, esta clase hereda de `UMovementComponent` la cual está definida como una clase abstracta de acuerdo a los macros internos de Unreal Engine y provee funcionalidades para mover un `UPrimitiveComponent` teniendo en cuenta colisiones. Estos últimos corresponden a componentes que contienen o general algún tipo de geometría, siendo utilizados generalmente para otorgar una representación visual a un objeto dentro del mundo o como información que es utilizada para las colisiones. Un ejemplo de estos componentes son los distintos cuerpos geométricos utilizados para proveer colisión a un `AActor` como la cápsula utilizada por `ACharacter` o el cubo creado para `ARlCharacter`. Es así como se crea la clase `URlCharacterMovementComponent` heredando de `UPawnMovementComponent` y utilizando como elemento base al cubo a cargo de las colisiones de `ARlCharacter`. Es importante mencionar que los componentes de un `AActor` siguen un sistema de jerarquía, existiendo un componente base o `RootComponent`, mientras que los demás deben ser hijos de este o los demás componentes. Esta estructura en forma de árbol es utilizada para manejar el posicionamiento relativo entre los componentes así como su respectiva ubicación dentro del mundo. Si un componente es movido o rotado, todos sus

hijos son afectados por esta transformación; sin embargo el padre no lo es, lográndose así realizar movimientos absolutos que estén basados de acuerdo a las coordenadas del mundo, o movimientos relativos que consideran como punto de origen a su componente padre. Ya que en `ARlCharacter` se desea mover todos los componentes que éste posee al mismo tiempo, se designa al cubo de colisión como el componente base.

Una de las primeras reglas que se establece en `URlCharacterMovementComponent` corresponde a restringir el movimiento del personaje de tal forma que sólo su posición horizontal y vertical puedan ser modificadas, no así su posición en el eje de profundidad. Si bien podría parecer que esta regla no es completamente necesaria ya que dada la forma en que se implementa el movimiento el personaje no debería ser capaz de moverse a lo largo del eje de profundidad, hay que considerar que el manejo de colisiones lo realiza Unreal Engine de manera interna, y estas colisiones están diseñadas para estar presentes dentro de un mundo tridimensional. Si se junta lo anterior al hecho de que por límites computacionales no es posible calcular la existencia de colisiones perfectas—ya que la precisión de un número real no es infinita en ningún lenguaje de programación debido a que esto requeriría memoria infinita—cuando el personaje colisiona con un algún elemento puede darse el caso de que su posición en el eje de profundidad sea modificada levemente. Si bien, dado que para mostrar el juego al usuario se está ocupando una cámara que utiliza proyección ortográfica, y por lo tanto este no notaría las variaciones de posición en el eje de profundidad, las colisiones sí se verían afectadas por este hecho. Esto último se debe a que éstas se calculan dentro de un mundo en tres dimensiones y los elementos poseen un volumen de colisión limitado. Por ejemplo, si el personaje se aleja una cantidad considerable en alguna de las direcciones del eje de profundidad, podría caer al vacío debido a que sobrepasaría a la zona de colisión de los *tiles* que construyen el piso.

Para mover el personaje, en cada iteración del juego se llaman a los métodos correspondientes de acuerdo al estado en que se encuentre el personaje debido a los inputs del usuario. Todo comienza obteniendo el vector de movimiento definido en `UPawnMovementComponent`, el que representa los inputs acumulados del usuario. Para lograr un mayor grado de realismo, los cálculos realizados para la ejecución del movimiento se basan en un sistema que considera aceleraciones y roce, en vez de sólo utilizar velocidades. Es así como el vector de movimiento obtenido representa la aceleración a la que el usuario desea que el personaje se mueva, realizando el cálculo de la velocidad de acuerdo a las fuerzas presentes en cada momento. En particular, la aceleración recibida en este vector permite comandar sólo el movimiento horizontal del personaje, ya que su movimiento vertical depende exclusivamente del input de salto y la gravedad.

Si bien las animaciones del personaje dependen de un total de nueve estados, su movimiento se define utilizando sólo tres, los cuales se basan en las distintas fuerzas y mecánicas que gobiernan cada uno de éstos. Es así como el primer estado llamado `Walking` corresponde a cuando el personaje se encuentra sobre una superficie sólida en la cual puede caminar, el segundo nombrado `Falling` se da cuando el personaje no posee una superficie bajo sus pies, sino que está moviéndose en el aire, y el tercero `WallWalking` hace referencia a cuando el personaje camina por el muro del fondo, siendo posible observar estos estados en la figura 5.9

Antes de procesar la aceleración obtenida para el movimiento horizontal, se comprueba

Figura 5.9: Estados de movimiento del personaje

si se ha presionado el input de salto, ya que de ser así se produciría un cambio de estado el cual es mejor realizarlo antes de que se calcule la velocidad final del personaje. El input de salto está sujeto a dos eventos, los cuales hacen referencia a cuando el jugador lo presiona y cuando lo suelta. Durante el primer evento se reinician las variables encargadas de contar el tiempo que el jugador ha presionado el input, tanto para el salto como para la caminata por el muro, y se activa una variable que indica si el jugador quiere saltar o si quiere caminar por el muro. Durante el segundo evento se reinician nuevamente los tiempos y se desactiva cualquier variable que hubiera sido activada durante el primer evento. Se utiliza este sistema de variables que indican las acciones que desea ejecutar el jugador en vez de directamente realizarlas ya que `URlCharacterMovementComponent` es el encargado de efectuar los movimientos en el orden que se estime conveniente, para así no interferir con el cálculo de las físicas que se realiza en cada iteración y también mantener la consistencia de estados.

Si el jugador desea saltar y es capaz de hacerlo, se le otorga velocidad vertical al personaje y se cambia al estado `Falling`. La magnitud de la velocidad utilizada depende de la cantidad de tiempo que se ha mantenido presionado el botón de salto, decayendo de manera exponencial hasta llegar a cero al alcanzarse el tiempo máximo. Por otro lado, si el jugador presionó el input de salto para caminar por el muro de fondo en vez de saltar, la única acción requerida corresponde a establecer el estado actual como `WallWalking`.

Luego de comprobar el input de salto, se continúa con el procesamiento del vector de aceleración obtenido, eliminando cualquier valor que no esté en su componente horizontal. Dado que el valor de los componentes del vector se encuentran entre 0 y 1, es necesario escalarlo dentro de los límites establecidos, teniendo en cuenta que la aceleración máxima del personaje es mayor si se está actualmente presionando el input para correr. Ya con la aceleración horizontal lista para ser usada, se comprueba en qué estado se está actualmente y en base a este se elige que camino continuar. Por cada estado se crea un método encargado de realizar los cálculos correspondientes de acuerdo a qué tipo de física está sujeto. De esta forma, los procedimientos realizados y el resultado obtenido en cada iteración depende total y exclusivamente del estado actual en que se encuentre el personaje, no siendo así necesario preocuparse o mantener registro de los estados anteriores por los que pudo haber pasado.

Antes de explicar cada uno de los tres métodos que se encargan de realizar el cálculo de las físicas que aplican al personaje, es necesario mencionar el funcionamiento del método encargado de modificar la velocidad a partir de la aceleración actual y una fuerza de roce y desaceleración indicadas. Este método es usado en cada uno de los modos de movimiento luego de que se ha calculado la aceleración correspondiente o para cálculos previos, nombrándose `CalcVelocity` para futuras referencias. Si bien dada la implementación de este método, cada modo de movimiento pude poseer su propia fuerza de roce y desaceleración, sólo los modos `Walking` y `WallWalking` poseen valores distintos de cero, ya que para facilitar el control del personaje por parte del jugador se decide que el aire no ejercerá fuerzas que modifiquen la velocidad. Si bien la fuerza de roce y la desaceleración pudieran parecer simulares, éstas actúan en momentos totalmente distintos. La fuerza de roce está presente mientras el personaje este afecto a algún tipo de aceleración, lo que se cumple mientras el jugador esté presionando el input de movimiento horizontal. Debido a esto, el roce sólo afecta la capacidad del personaje de cambiar de dirección. Por otro lado, la desaceleración actúa en la ausencia de aceleración, siendo la encargada de disminuir la velocidad del personaje cuando el jugador dejó de presionar el input correspondiente.

Como se está utilizando un sistema de movimiento basado en aceleración, el primer paso para realizar el cálculo de la velocidad consiste en determinar la máxima velocidad a la que se puede mover el personaje, ya que de no existir un límite el jugador sería capaz de moverse cada vez más rápido al mantener presionado el input de movimiento. Esta velocidad máxima depende de dos factores. El primero hace referencia a si el jugador tiene actualmente presionado o no el input para correr, en cuyo caso el límite de velocidad es más alto que cuando no existe presencia de este input. El segundo tiene que ver con el tipo de input que utiliza el usuario, ya que al estar jugando con mando, los controles análogos entregan un rango de valores entre 0 y 1 de acuerdo a la distancia en que se encuentre el análogo de su punto de origen. Debido a esto, se limita la velocidad máxima de acuerdo al valor actual del análogo.

Con la velocidad máxima obtenida, se procede a determinar si existe la presencia de aceleración sobre el personaje por parte del usuario. Si no existe aceleración se debe eventualmente detener al personaje, por lo que entra en juego la desaceleración. Para el cálculo de su efecto se utilizan dos valores: un componente de roce, que puede o no ser la misma fuerza de roce que afecta los cambios de dirección, y el componente de desaceleración. Con estos componentes se crean dos vectores: el primero consiste en multiplicar el roce por un vector de igual magnitud que la velocidad pero con dirección opuesta, mientras que el segundo se forma al multiplicar la desaceleración por un vector unitario que representa la dirección opuesta de la velocidad actual. Es así como para obtener la nueva velocidad, se suman los dos vectores anteriores y se multiplican por el tiempo transcurrido en la iteración actual, sumando este resultado a la velocidad. De esta manera, su magnitud va decreciendo a lo largo del tiempo mientras no exista la presencia al algún input. Dado que la velocidad está siendo reducida mediante la resta de vectores, es posible que en vez de llegar a magnitud cero se invierta la dirección de la velocidad, en cuyo caso se detiene al personaje inmediatamente. Ahora bien, si existe aceleración, entonces se aplica la fuerza de roce. En otras palabras, se resta a la velocidad actual una fracción de un vector que posee la misma magnitud de la velocidad pero su dirección corresponde a la dirección de la aceleración, de tal manera que si tanto la velocidad como la aceleración poseen la misma dirección, la velocidad no sea modificada. Por

lo tanto, la fuerza de roce sólo afecta qué tan rápido el personaje puede ejecutar un cambio de dirección.

Luego de que se limitó la velocidad actual y se aplicaron los efectos tanto de la fuerza de roce como de la desaceleración según el caso, el paso final para el cálculo de la velocidad consiste en aplicar los efectos de la aceleración actual. Esto no es más que agregar a la velocidad el resultado de multiplicar la aceleración por el tiempo transcurrido en la iteración actual.

Ya que el tiempo que toma en ejecutarse una iteración es variable, dependiendo de qué tan ocupado esté el sistema que corre el juego en determinado momento y qué hardware posee, para el cálculo de las físicas de los distintos modos de movimiento se divide el tiempo disponible en partes pequeñas y se itera hasta que se haya utilizado por completo. Así pues, se crean subiteraciones dentro de la iteración principal. Realizar este procedimiento permite que el cálculo sea más preciso y, por lo tanto, el personaje interactúe de manera más fluida con su entorno.

5.9. Modos de movimiento

Pasando a la implementación de los distintos modos de movimiento se comienza con `Walking`. Al estar en este modo se asume que el personaje está sobre una superficie sobre la cual puede caminar, por lo que la acción principal realizada consiste en mover al personaje horizontalmente a velocidades variables las cuales dependen íntegramente de los inputs que esté presionando el jugador. Ya que no existe la presencia de movimiento vertical, y los bloques sobre los que puede caminar el personaje no poseen algún tipo de reacción por el peso que este podría producir al estar sobre ellos, no se hace presente la fuerza de gravedad en este modo. En cada subiteración se comprueba inicialmente que la velocidad del personaje sea estrictamente horizontal, eliminando cualquier valor que pueda existir en los componentes de los otros ejes, realizando el mismo proceso para la aceleración. Luego se obtiene la nueva velocidad correspondiente llamando a `CalcVelocity` y se utiliza *ray casting* para determinar si bajo del personaje se encuentran púas, en cuyo caso implicaría su muerte. *Ray casting*, o *line trace* según la terminología de Unreal Engine, no es más que lanzar una linea desde un punto del espacio hasta otro, comprobando si ha colisionado con algún objeto de manera que se pueda reaccionar según el caso. En la figura 5.10 es posible observar una representación gráfica del procedimiento anterior, donde la línea se muestra de color verde si está detectando colisiones y rojo si no lo está. Como se puede ver, se deja una tolerancia para que el personaje pueda posicionarse levemente sobre una púa sin morir. También el largo de la línea mostrada es sólo ilustrativo ya que en la práctica se utilizan sólo cuatro píxeles para optimizar los cálculos.

Si no existe la presencia de púas, entonces se procede a mover al personaje. Para esto se obtiene el vector de desplazamiento a partir de la velocidad previamente calculada y el tiempo de la subiteración actual. Luego se utiliza un método de `UMovementComponent` para intentar mover al personaje a la posición deseada. Este método tiene en consideración la presencia de colisiones, retornando una estructura que indica si existió alguna, en qué momento y con

Figura 5.10: Ejemplo de *line trace*

qué elemento se colisionó. De esta forma, si el personaje choca contra un muro, su velocidad se anula para que no pueda seguir avanzando.

Dado que luego de realizar un movimiento horizontal el personaje puede encontrarse en el aire, es necesario actualizar la referencia que posee respecto a su piso actual. Para esto se utiliza un *shape trace*, el cual sigue la misma idea que un *line trace* sólo que utiliza un volumen en vez de una línea. Así, se utiliza el cubo de colisión del personaje, desplazándolo hacia abajo una determinada distancia para encontrar el nuevo piso sobre el que este esta posicionado. Si no existe algún piso, entonces es necesario cambiar al modo de movimiento `Falling` para empezar un nuevo cálculo de las física utilizando el tiempo restante de la iteración actual.

El modo `Falling` corresponde a cuando el personaje está en el aire sin una base sobre sus pies, lo cual puede ocurrir ya que el jugador utilizó el input de salto, el personaje se cayó por el borde de una plataforma, o bien se terminó el tiempo de su caminata por el muro de fondo. En este modo las aceleraciones que comandan la personaje corresponden al movimiento horizontal proporcionado por el jugador, y a la fuerza de gravedad. Para cumplir lo anterior, cuando se realiza un salto no contribuye a la aceleración del personaje, sino simplemente modifica su velocidad vertical. De esta manera, se incrementaría el valor de esta última de manera exponencialmente decreciente mientras se mantenga presionado el input de salto.

Si bien para este modo se realizan muchos chequeos y ajustes tanto para separar el peso del cálculo de las físicas como para estar preparado ante cualquier acontecimiento respecto a las colisiones, sólo se explicará la idea base detrás de éstas. Al inicio de cada subiteración se elimina cualquier valor que pueda tener el componente vertical de la aceleración y se procede a calcular al velocidad con `CalcVelocity`. Se calcula también una velocidad que sólo depende de la velocidad anterior, anulando por completo la aceleración actual, la cual es utilizada posteriormente para ajustes de la colisiones. Luego se obtiene la fuerza máxima de gravedad utilizada y con ella se crea un vector donde su único componente no nulo corresponde al del eje vertical. Este vector es multiplicado por el tiempo de la subiteración y el resultado se

suma a la primera velocidad calculada. Para lograr una caída suave, se realiza un promedio entre la velocidad inicial que tenía el personaje y la velocidad actual calculada, procediendo a multiplicarse por el tiempo correspondiente para así obtener el vector de desplazamiento. Para esto, se utiliza el mismo método que en `Walking` para mover al personaje mientras se está al tanto de las colisiones. Si se produce una colisión y corresponde a una plataforma en la que el jugador puede caminar, se cambia al modo `Walking` y se comienza con la ejecución de un nuevo cálculo de físicas con el tiempo no utilizado por las subiteraciones. Si la colisión corresponde a un muro, en vez de simplemente anular la velocidad horizontal como ocurre en `Walking`, se calcula un vector de movimiento en la dirección opuesta de tal manera que desvíe el impacto proporcionado por el personaje y se realizan una serie de pasos finales para asegurarse que este movimiento sea exitoso y no provoque el personaje quede atrapado en otro muro.

Finalmente se tiene al modo `WallWalking`, el cual es muy similar a `Walking` ya que está exento de la presencia de gravedad. Sin embargo, el tiempo máximo en que el personaje puede permanecer en este estado corresponde a un segundo. Por otro lado, al igual que `Falling`, utiliza las variables presentes en `ARlCharacter` para determinar el efecto conseguido de acuerdo a cuánto tiempo se ha estado presionado el input de salto. Es así como al inicio de cada subiteración se incrementa la cantidad de tiempo que se ha estado presionado el input y se comprueba que esta cantidad no sea mayor que el tiempo máximo permitido o bien que se haya dejado de presionar el input. En este último caso, se dicta un cambio de modo a `Falling` y se inicia un nuevo cálculo de las físicas con el tiempo restante de la iteración actual, lo que también ocurre si es que la aceleración del personaje o su velocidad son nulas. Así pues, se busca impedir que el jugador pueda cambiar de dirección mientras camina por el muro de fondo, o bien quedarse detenido en el aire al no poseer velocidad horizontal. Si se continua en este estado, se calcula la velocidad correspondiente con `CalcVelocity` y se mueve al personaje utilizando los mismos procedimientos implementados para `Walking`.

Capítulo 6

Prueba de Concepto

Con el objetivo de poner a prueba el juego realizado y comprobar si se cumple la premisa de mejorar la experiencia de juego, se reúne un grupo de usuarios a los que se les pide que interactúen con la aplicación. Para ello, se diseñó un experimento de tipo prueba A/B, en el cual se asignaron aleatoriamente a los participantes del estudio a dos condiciones: (1) juego **con** ajuste dinámico y (2) juego **sin** ajuste dinámico. Para mantener controladas las condiciones experimentales, todos los participantes usaron una pulsera de monitoreo de pulso y siguieron el mismo protocolo.

6.1. Metodología

Dado que el objetivo de esta validación es preliminar, en el sentido de verificar el correcto funcionamiento del juego, el presente experimento toma la forma de una *prueba de concepto*. Así, el objetivo principal de esta evaluación es explorar si el protocolo experimental llevado a cabo puede ser extendido para aplicarlo a una muestra más grande de usuarios.

6.1.1. Participantes

Para la realización de esta prueba se realizó un muestreo por conveniencia, reclutando un total de 23 participantes. Para el proceso de selección se presentó la idea de probar un nuevo juego a un grupo de aproximadamente 70 personas, de las cuales aquellas con interés y suficiente tiempo pasaron a conformar el grupo de participantes para este experimento. Respecto a su distribución, todo el grupo estuvo compuesto por estudiantes de la Facultad de Ciencias Físicas y Matemáticas de la Universidad de Chile, presentándose 3 mujeres y 20 hombres de edades entre 18 y 28 años.

6.1.2. Materiales

Cada participante fue equipado con una pulsera de monitoreo de pulso, todas del mismo modelo y especificaciones técnicas (Xiaomi Mi Band 3). Como mecanismo de control de entrada durante la sesión de juego, los participantes utilizaron un mando de Xbox One. El juego fue ejecutado en un laptop MSI GE63 Raider RGB 8RE y no experimentó ningún tipo de retraso durante cada una de las pruebas.

6.1.3. Procedimiento

Los pasos seguidos para la realización de la prueba consistieron en primero explicar a cada participante el contexto de la creación del juego, es decir, que el juego había sido desarrollado para una memoria y que se utilizaba la pulsera para recolectar información, no especificando si estos datos eran utilizados o no durante el juego. Luego se iniciaba la aplicación encargada de la comunicación con la pulsera y se procedía a ejecutar el juego. Dentro de éste, el participante debía seleccionar de manera correcta la dirección MAC del Mi Band que estaba ocupando, la cual le era previamente proporcionada. Una vez finalizado con el proceso de conexión con la pulsera, comenzaba a mostrarse la historia del juego al mismo tiempo en que se iniciaba la medición de su pulso. Ya finalizada la historia se le presentaba el primer nivel del juego, convirtiéndose en su objetivo llegar hasta al último nivel mientras sorteaba cada uno de los obstáculos presentes en el juego. Finalmente se le agradeció su participación y se le pedió que contestara una encuesta implementada en Google Forms.

La encuesta solicitada a los usuarios estuvo compuesta de cuatro secciones. La primera consistió en información general, particularmente su sexo, edad y la cantidad de horas que juega videojuegos a la semana. Las siguientes dos secciones correspondieron a una traducción del módulo base y post juego de *The Game Experience Questionnaire* [16], un cuestionario válido y confiable para medir experiencias de juego, utilizado ampliamente en investigación académica e industria en Computación Centrada en las Personas. En la última sección se presentaron preguntas de respuesta libre, las que permiten saber qué le gustó del juego, qué aspectos mejoraría y si volvería a jugar y por qué. Se le consultó también si se percataron del cambio de dificultad que ocurría cuando intentaba nuevamente un nivel, independientemente si este cambio estaba o no presente dependiendo de a que grupo pertenecía el jugador. Finalmente se agregó un espacio para comentarios y sugerencias generales.

6.1.4. Recolección y análisis de datos

Durante el juego se fue almacenando información de los distintos participantes para ser utilizada en el posterior análisis. Esta información corresponde a todos los resultados logrados al finalizar cada nivel, en los cuales se encuentra el número de intentos utilizados para completarlo y el tiempo transcurrido hasta ese momento.

Si bien se realizaron pruebas estadísticas para evaluar significancia y medir magnitudes de efecto, es importante notar que dado el tamaño pequeño de la muestra experimental en

cada grupo, los resultados obtenidos deben ser entendidos únicamente de manera descriptiva. En particular, no se busca generalizar para la población de estudio. Todos los análisis fueron realizados usando un nivel de significancia $\alpha = 0.05$, sobre una instalación base del software R versión 3.5.

Se recolectaron y analizaron las siguientes variables para cada grupo:

- **Número de intentos en completar cada nivel.** Para medir una posible diferencia entre la dificultad efectiva del juego en ambas condiciones experimentales, se aplicó una prueba T de muestras independientes entre las medias de los números de intentos en cubrir cada nivel.

- **Tiempo en completar cada nivel.** Al igual que en el caso anterior, se aplicó una prueba T de muestras independientes para estudiar una posible diferencia entre los tiempos promedios empleados en cumplir cada nivel, para ambos grupos.

- **Grado de satisfacción de los participantes.** Se computaron los resultados obtenidos de tabular las respuestas a los cuestionarios GEQ de cada participante, siguiendo dos subescalas: experiencia durante el juego y experiencia después del juego. Se estudió la significancia estadística de las diferencias de medianas obtenidas para cada dimensión de las escalas, aplicando pruebas U de Mann-Whitney. Asimismo, se estudió la dominancia relativa entre grupos, calculando la delta de Cliff como medida de magnitud de efecto.

- **Usabilidad y utilidad percibida del juego.** Se analizaron cualitativamente las respuestas proporcionadas por los participantes en la encuesta de salida. Asimismo, se buscó contrastar las opiniones entre los participantes asignados a cada condición experimental.

6.2. Resultados

Para el estudio se reclutaron $N = 23$ participantes. Sin embargo, debido a problemas de registro de datos (en etapas iniciales de la experimentación) y dos participantes que no completaron el cuestionario correctamente, se decidió no considerar dichos puntos en el análisis. Así, la muestra efectiva estuvo compuesta por $N = 17$ sujetos, 10 de los cuales fueron asignados al grupo *con ajuste* y 7 al grupo *sin ajuste*.

A continuación se presentan los resultados principales obtenidos durante la prueba de concepto.

6.2.1. Métricas de rendimiento

Para ambos grupos, primero se obtuvieron los datos que fueron necesarios para la realización de la prueba T de muestras independientes, correspondiendo a la media y derivación estándar del número de intentos y el tiempo. Estos valores están resumidos en la tabla 6.1.

Tabla 6.1: Número de intentos y tiempo empleado promedio en completar el juego

	Número de Intentos		Tiempo [s]	
Grupos	Prom.	Desv. Est.	Prom.	Desv. Est.
CON (N = 10)	141.60	84.54	734.00	427.18
SIN (N = 7)	182.14	71.23	852.57	337.94

Si se comparan a simple vista los promedios del número de intentos y del tiempo, se puede notar que el grupo que utilizó el ajuste dinámico de dificultad logró completar el juego en un menor número de intentos y en menos tiempo. Dado que los datos recolectados para la muestra de estudio siguen una distribución aproximadamente normal, verificado a través de histogramas y pruebas de Shapiro-Wilk, se aplican pruebas T de muestras independientes para comparar las medias de ambas variables. Los resultados se presentan en la tabla 6.2.

Tabla 6.2: Pruebas T independientes para métricas de rendimiento

Métrica	df	t	p	d
Número de intentos	14.359	–1.069	0.303	–0.510
Tiempo [s]	14.683	–0.638	0.533	–0.301

En la tabla anterior, *df* se refiere a los grados de libertad con los que se computó el análisis y t es el estadístico de prueba. El valor p representa la probabilidad de que los datos obtenidos sean posibles bajo la hipótesis nula, la cual para este caso establece que no hay diferencias entre las medias obtenidas para ambos grupos. Luego, si el valor de p es igual o menor que el nivel de significancia (i.e., $\alpha = 0{,}05$), entonces se rechaza la hipótesis nula, siendo así posible establecer que sí existe una diferencia entre los dos grupos, y que este valor no fue producto del azar. Dado que los resultados obtenidos para el número de intentos y el tiempo para el valor p son 0.303 y 0.533 respectivamente, y ambos son mayores que el nivel de significancia, no es posible rechazar la hipótesis nula. Por tanto, no es posible afirmar que existe una diferencia entre la población de ambos grupos dada la muestra utilizada.

Si bien, dado lo anterior no es posible establecer una diferencia a nivel de población, se puede estudiar la presencia de una diferencia en de la muestra analizada. Esto se consigue a través del cálculo de la magnitud de efecto. En este caso, se utiliza la d de Cohen, representado en la última columna de la tabla 6.2. Como es usual, si el valor absoluto de d se encuentra alrededor de 0.2, se habla de un efecto pequeño; si está cercano a 0.5, corresponde a un efecto mediano; y si el valor es igual o mayor a 0.8, se tiene un efecto grande. Los valores obtenidos son -0.510 y -0.301 para el número de intentos y el tiempo respectivamente, permitiendo así establecer la presencia de un efecto mediano para el número de intentos y un efecto pequeño en la cantidad de tiempo, donde el signo sólo indica que los valores del primer grupo son menores que los del segundo.

De esta manera, los resultados obtenidos indican que, si bien las diferencias analizadas entre el número de intentos y tiempo empleado en completar el juego no son estadísticamente significativas (y, por ende, no es posible generalizarlas a la población de estudio), sí se observan diferencias apreciables para ambos grupos a nivel muestral.

6.2.2. Curvas de evolución

Para analizar los resultados obtenidos en ambos grupos en función de cada nivel, se calcula la cantidad de intentos y el tiempo promedio por nivel. Asimismo, se calcula el error estándar, teniendo en cuenta el tamaño de la muestra. Estos datos se representan en la figura 6.1 para el número de intentos, y en la figura 6.2 para el tiempo.

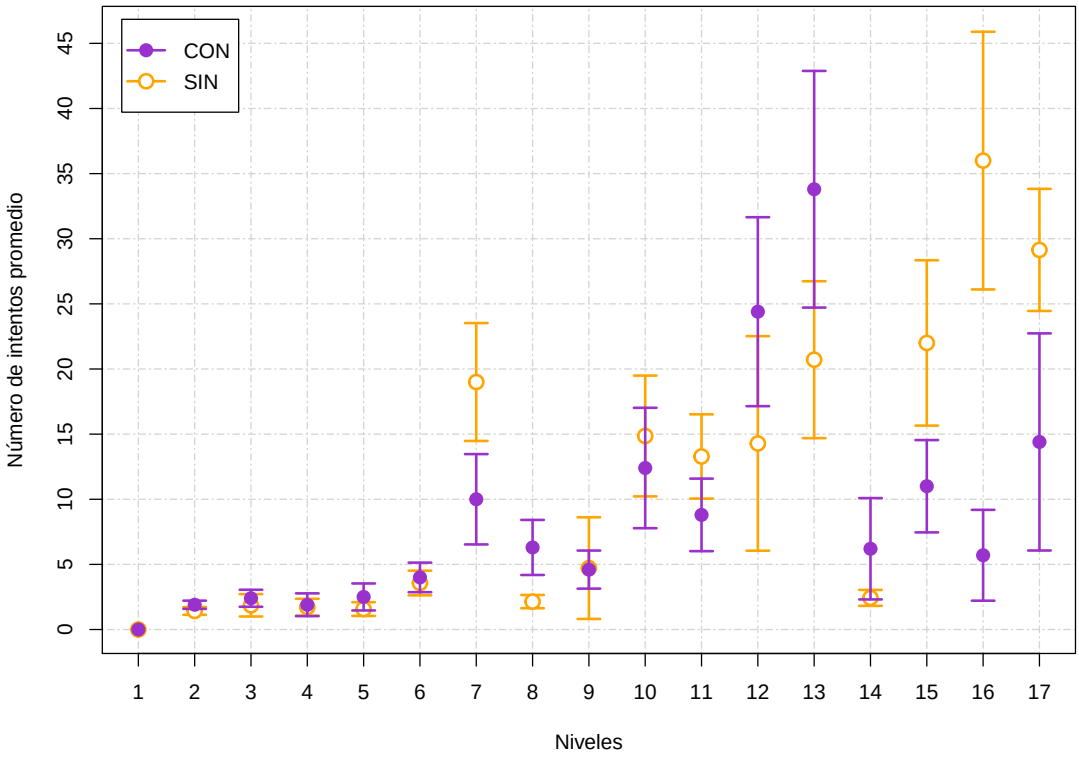


Figura 6.1: Número de intentos promedio por nivel con su error estándar

Respecto a la cantidad de intentos, es posible ver que durante los cinco primeros niveles—que conforman el tutorial—los resultados obtenidos para ambos grupos son muy similares. Esto no es de sorprender, dado que hasta este punto ambos grupos han experimentado lo mismo. En efecto, el ajuste dinámico de dificultad se hace presente luego de finalizar el tutorial. Sin embargo, el sexto nivel también presenta resultados similares para ambos grupos, aunque con diferencias ligeramente más marcadas a las observadas durante el tutorial. Como este nivel es el primero en el que se ejecuta el ajuste, ambos grupos comienzan con la misma dificultad de 0.5, la cual se modifica a lo largo del nivel, siendo su efecto claramente visible en

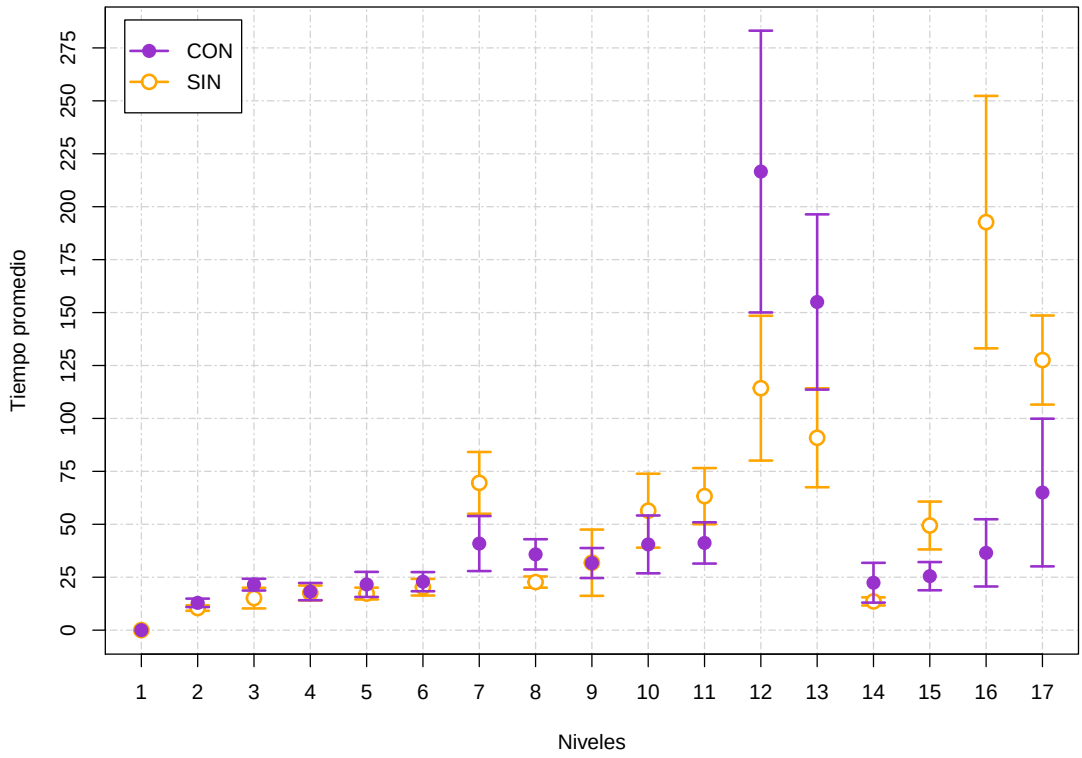


Figura 6.2: Tiempo promedio por nivel con su error estándar

el séptimo nivel. Aquí se evidencia la primera diferencia notable entre los resultados de ambos grupos, donde aquellos sujetos al ajuste logran superar el nivel con una menor cantidad de intentos respecto al otro grupo, pudiéndose notar cómo el ajuste dinámico de dificultad logra cumplir su objetivo.

A pesar de que el séptimo nivel es favorable para el grupo con el ajuste, en el octavo nivel se presenta la situación opuesta. El grupo sin ajuste logra completar el nivel con una cantidad de intentos sustancialmente menor, superando al grupo con ajuste. Esto se podría deber a dos motivos. El primero corresponde a que los jugadores del grupo con ajuste, al haber encontrado más fácil el séptimo nivel, podrían haberse tomado más a la ligera el octavo y por lo tanto la cantidad de intentos que requirieron para completarlo fue mayor a su posible óptimo. El segundo se sustentaría en el hecho de que el grupo sin ajuste, al haber realizado una mayor cantidad de intentos para completar el nivel anterior, tuvieron en consecuencia más oportunidades de comprender cómo funcionaba el juego y, por tanto, mejorar su control sobre las mecánicas del mismo. Esto permitiría que para el octavo nivel su desempeño fuera superior. En una etapa posterior a este trabajo, se estudiarán más detenidamente estas hipótesis.

Entre el noveno y decimocuarto nivel se observa nuevamente la tendencia anterior, aunque a lo largo de una mayor cantidad de niveles. El grupo con ajuste requiere cada vez menos intentos hasta que eventualmente son superados por el grupo sin ajuste. Sin embargo, al con-

siderar del nivel decimoquinto al decimoséptimo, se presenta la situación opuesta, registrando mejor rendimiento el grupo con ajuste por un amplio margen. En esta situación es donde sale a la luz con mayor fuerza el efecto del ajuste de dificultad, ya que al ser los últimos niveles del juego, su diseño se realiza de tal manera que posean un mayor grado de desafío que los anteriores. Así, el grupo con ajuste posee una mayor ventaja respecto al otro grupo, el cual debió completar estos niveles más complejos en su dificultad normal.

Al analizar el gráfico que contiene el tiempo por nivel, se puede observar que la tendencia de los resultados es muy similar a la del gráfico de intentos por nivel. Luego, a partir de éste es posible llegar a las mismas conclusiones. Es aquí donde se hace notoria una posible correlación entre el número de intentos y el tiempo respecto a su efecto en la dificultad de cada nivel. Para verificar esto se calcula el coeficiente de correlación de Pearson entre el número de intentos y el tiempo para completar un nivel para ambos grupos. Así pues, se obtuvo un valor de $r = 0.885$, $p < 0.05$ para el grupo con ajuste y $r = 0.918$, $p < 0.05$ para el grupo sin ajuste. Esto demuestra una alta correlación entre ambas variables, por lo que es posible concluir que se puede ocupar indistintamente cualquiera de las dos curvas para medir la dificultad del juego.

6.2.3. Cuestionario GEQ

Se utilizaron dos módulos del cuestionario GEQ. El primero corresponde a su módulo base, el cual se compone de 33 enunciados en los que el usuario puede responder con un número de 0 a 4 de acuerdo a qué tanto aplica cada enunciado a lo que él experimentó durante el juego. El segundo módulo posee la misma estructura que el primero. Sin embargo consiste en 17 enunciados y da cuenta de lo que sintió el usuario luego de haber completado la totalidad del juego. En ambos módulos, cada enunciado se asocia con alguna dimensión en particular, por lo cual el resultado final corresponde al promedio de las respuestas obtenidas para cada dimensión. En el Apéndice C se presenta el instrumento utilizado (traducido al español, desde [16]).

Para el primer módulo, las dimensiones estudiadas son: *habilidad* (qué tan cómodo se sintió el jugador con los desafíos presentes dentro del juego), *inmersión* (grado de interés mostrado por los distintos elementos del juego), *flujo de juego* (qué tan absorto se encontró el usuario durante el juego), *fastidio* (desagrado producido en su experiencia), *desafío* (dificultad y reto experimentado), *afecto positivo* (nivel de satisfacción general y agrado) y *afecto negativo* (nivel de disgusto general y desinterés). Para el segundo módulo, las dimensiones analizadas corresponden a: *experiencia positiva* (qué tanto disfrutaron el juego), *experiencia negativa* (desagrado general del juego), *cansancio* (nivel de esfuerzo requerido) y *vuelta a la realidad* (grado de abstracción producido por el juego).

Para este caso, dado que los puntajes medidos no siguen una distribución normal, no es posible utilizar una prueba T de muestras independientes para comparar los resultados obtenidos por ambos grupos. Es así como se aplica una prueba U de Mann-Whitney sobre diferencia de medianas. Debido a la misma razón no es posible utilizar la d de Cohen para medir el tamaño del efecto existente, por lo que se calcula la delta de Cliff, la cual indica qué tan frecuentemente los valores de una primera distribución son mayores que los de una

segunda distribución (o, en otras palabras, qué tan dominante es una variable por sobre otra). Los resultados de los análisis para ambos módulos se presentan en las tablas 6.3 (para las dimensiones de la escala *durante* el juego) y 6.4 (para las dimensiones *después* de terminar el juego).

Tabla 6.3: Pruebas U para dimensiones de análisis durante el juego

Dimensión	Med. CON (N = 10)	Med. SIN (N = 7)	U	p	δ
Habilidad	2.60	2.40	44.5	0.374	0.271
Inmersión	2.00	2.50	32.5	0.844	–0.071
Flujo de juego	2.90	2.80	39.0	0.730	0.114
Fastidio	0.67	0.67	27.0	0.457	–0.229
Desafío	1.90	2.20	18.5	0.115	–0.471
Afecto Positivo	3.60	3.00	48.0	0.215	0.371
Afecto Negativo	0.63	0.75	29.5	0.622	–0.157

Tabla 6.4: Pruebas U para dimensiones de análisis después del juego

Dimensión	Med. CON (N = 10)	Med. SIN (N = 7)	U	p	δ
Experiencia positiva	2.42	2.00	40.5	0.623	0.157
Experiencia negativa	0.08	0.50	26.0	0.388	–0.257
Cansancio	0.50	0.00	41.5	0.534	0.186
Vuelta a realidad	0.50	0.67	30.0	0.651	–0.143

En los resultados del módulo que mide la experiencia durante el juego, es posible notar que ninguno de los valores p es menor que el nivel de significancia. Esto quiere decir que no se puede rechazar la hipótesis nula y, por tanto, estos resultados no son generalizables a la población. Lo mismo sucede para el módulo después del juego, donde todos los valores p son mayores que el nivel de significancia.

Ahora bien, los valores obtenidos para la delta de Cliff (δ) permiten realizar un análisis más a fondo de la experiencia para la muestra estudiada. Dado los resultados del primer módulo, es posible notar que los participantes que se vieron afectados por el ajuste dinámico de dificultad encontraron el juego menos desafiante. Sin embargo, al mismo tiempo demostraron mayor afectividad, es decir, se sintieron mejor jugando el juego. Esto permite evidenciar cómo el ajuste dinámico de dificultad provocó una mejor experiencia de juego para los participantes de este grupo. En cuanto a los resultados del segundo módulo, si bien se aprecian efectos menores, es posible notar que la experiencia negativa de los usuario que usaron el ajuste fue menor, lo cual confirma los resultados del módulo anterior.

6.2.4. Usabilidad y utilidad percibida

Dentro de los comentarios recibidos en la sección de preguntas abiertas, los jugadores que experimentaron el ajuste dinámico de dificultad tendieron a expresar que les gustó tanto

la dificultad del juego como el diseño y adaptación de los diferentes niveles. En cambio, los participantes cuya experiencia mantuvo una dificultad constante, expresaron su agrado por las mecánicas presentes. Esto permite extrapolar que el ajuste dinámico de dificultad fue perceptible por parte de los jugadores, siendo al mismo tiempo un factor a considerar en cuanto a la satisfacción conseguida durante el juego, mejorando así su experiencia de usuario. Por otro lado, los usuarios no afectos a los cambios de dificultad tuvieron que familiarizarse más con las mecánicas del juego para así ser capaces de completarlo de manera más amena, comprendiendo sus ventajas y desventajas para distintas situaciones.

Asimismo, uno de los factores destacados por los participantes corresponde a la usabilidad del juego, argumentando que es controlado por mecanismos intuitivos, fáciles de aprender y recordar. En particular, en ambos grupos se recibieron comentarios aprobando tanto el diseño general del juego como del personaje y de los niveles. Luego, estos factores lograron exitosamente no generar discrepancias que pudieran afectar su experiencia de usuario, y así favorecer la aceptación del juego desarrollado. De esta manera, es posible aislar el efecto del ajuste dinámico de dificultad para lograr un análisis más concluyente. Esto último implica que el juego fue fácil de adoptar por ambos grupos de usuarios, destacando así su usabilidad y utilidad percibida.

En cuanto a los aspectos a mejorar, la mayoría de los participantes indicó la sensibilidad y respuesta de los controles como motivo de preocupación, sobre todo el retraso existente entre el comando y ejecución del salto por parte del personaje. Si bien esto podría considerarse como un defecto no previsto, la presencia del retraso en el salto fue introducida de manera intencional. Por un lado se buscó proveer fluidez a la animación del personaje (estados `ToIdle` y `ToIdleR`). Por otro, se intentó reducir el sesgo que se podría producir en los resultados por la presencia de algún participante que fuera experto en juegos de plataforma, debiendo acostumbrarse al igual que los demás al funcionamiento de los controles. Ahora bien, dado que este efecto fue percibido por todos los participantes de este estudio en particular, se puede argumentar que la experiencia de juego fue afectada para todos los usuarios de la misma manera, cumpliendo el objetivo buscado al no producir diferencias percibidas que afecten los resultados.

Finalmente, la mayoría de los participantes expuso sus intenciones de querer volver a jugar. Sin embargo, sus razones fueron diferentes de acuerdo al grupo al que fueron asignados. Aquéllos que percibieron el ajuste de dificultad indicaron que les gustaría ver y explorar las distintas variaciones que tiene el juego, mientras que aquéllos que no, sólo señalaron que este les trajo entretención. Esto permite notar cómo la presencia de la modificación de dificultad tuvo un impacto positivo para el juego, logrando incluso convertirse en su atractivo principal.

6.3. Discusión

A partir de los resultados es posible observar que sí es factible diseñar mecanismos de ajuste dinámico de dificultad que utilicen biofeedback, tales que sean aceptados por los usuarios y permitan al mismo tiempo mejorar su experiencia dentro del juego.

Si bien los resultados obtenidos para los distintos grupos no son estadísticamente significativos, y que el tamaño de la muestra fue de sólo 17 participantes, esto no impidió que se lograran observar magnitudes de efecto apreciables.

Dada la correlación encontrada entre el número de intentos y la cantidad de tiempo, y su posible uso como medida de la dificultad de un nivel, sería posible controlar estos factores en conjunto al pulso para así modificar el nivel de dificultad que presente el juego. Por ejemplo, se podría manipular la dificultad de un videojuego en caso de que un jugador haya utilizado muchos intentos o se haya quedado detenido en el mismo nivel por mucho tiempo.

En cuanto a los resultados del cuestionario, es interesante notar que si bien la dificultad percibida por los jugadores que utilizaron el ajuste fue menor, el promedio de la dificultad efectiva a la que estuvieron expuestos fue de 0.48 ± 0.02, lo cual es increíblemente cercano a la dificultad que experimentó de manera constante el grupo sin el ajuste, la cual—por definición—fue de 0.5. Esto permite evidenciar cómo el ajuste de dificultad logró entrar en efecto cuando los jugadores realmente lo necesitaban, lo que llevó a que percibieran el juego más simple, a pesar de que algunos niveles los completaron con una dificultad superior a la media. De igual manera, considerando las respuestas abiertas obtenidas por la encuesta, éstas permiten confirmar los efectos esperados con los cuales se realizó el diseño de cada nivel. En particular, los participantes del estudio destacaron cómo la experiencia de juego fue satisfactoria, señalando varios factores que afectarían sus expectativas a la hora de enfrentarse a un juego como el que les fue presentado. Debido a esto, los resultados ofrecen información relevante a la hora de diseñar mejores experiencias de usuario.

Volviendo a los objetivos específicos propuestos inicialmente, es posible analizar su cumplimiento dados los resultados obtenidos. Respecto al primero, se logra escoger un sensor asequible según los parámetros establecidos, ya que sólo cuesta alrededor de $30 dólares. Con este sensor fue posible obtener mediciones precisas, las que permitieron realizar un ajuste de dificultad que derivó a los resultados anteriormente presentados. Pasando al segundo objetivo, para realizar el ajuste de dificultad se implementó un algoritmo que detecta de manera correcta las variaciones significativas del pulso, el cual modifica sus parámetros de manera dinámica para permitir que la traducción entre pulso y dificultad pueda anticipar distintos eventos. De igual modo, la respuesta entre una variación significativa del pulso y su efecto dentro del juego se hace presente al instante en que se intenta nuevamente un nivel o se inicia uno nuevo, lográndose así una respuesta rápida al estado actual del usuario, lo que cumple el tercer objetivo. Continuando con el cuarto objetivo, se asocian valores de dificultad a cada obstáculo y se implementa un mecanismo encargado de mostrarlos u ocultarlos de acuerdo a la dificultad actual del nivel, lo cual permitió modificar la dificultad percibida por los jugadores como se puede notar con los resultados de los últimos tres niveles del juego. Finalmente, de acuerdo tanto al número de intentos y tiempo, como a los respuestas obtenidas con la encuesta, es posible confirmar que se logró una mejor experiencia de usuario al utilizar el ajuste dinámico de dificultad desarrollado, ya que si bien el juego les pareció menos desafiante, su nivel de agrado y satisfacción fue mayor.

Resumiendo lo anterior, se presenta cada objetivo, su estado y cómo se verifica:

- **Selección de sensor adecuado:** Cumplido, el sensor es barato, eficaz y se obtienen buenos resultados.

- **Detección de variaciones significativas del pulso:** Cumplido, se diseña un algoritmo que ajusta sus parámetros dinámicamente para obtener una medida acertada de la dificultad actual.

- **Respuesta rápida:** Cumplido, el efecto se hace presente inmediatamente luego de un reintento o una victoria.

- **Modificación efectiva:** Cumplido, el nivel se adapta de manera correspondiente a la dificultad actual, permitiendo que a menores dificultades exista un menor número de intentos y tiempo requerido.

- **Mejora de la experiencia de usuario:** Cumplido, el grupo sujeto al ajuste presenta mejores resultados y muestra una respuesta más positiva hacia el juego.

6.4. Limitaciones de la evaluación

Si bien los resultados obtenidos en esta memoria son valiosos, no por ello están exentos de amenazas que pueden poner en duda su validez interna (es decir, si son confiables) y externa (es decir, si son generalizables).

En primer lugar, debido a que el experimento fue realizado con un número relativamente pequeño de usuarios ($N = 17$), los resultados obtenidos no pueden ser generalizables a la población de estudio. No obstante lo anterior, dado el éxito (relativo) de los resultados obtenidos, se buscará reclutar a una muestra más grande—y representativa—de participantes, para estudiar la validez de las hipótesis (i.e., mejora comparativa de la experiencia de juego) en etapas posteriores de este trabajo.

Por otro lado, ya que se utilizó un muestreo por conveniencia, no fue posible extrapolar la existencia de diferencias respecto a distintos parámetros, como el género, la edad o la cantidad de horas semanales de juego. Debido a lo anterior, las respuesta obtenidas para estas variables sólo se utilizan de manera referencial, y de igual manera que para el caso anterior, una muestra más grande y mejor seleccionada proporcionaría la posibilidad de hacer análisis comparativos de los parámetros mencionados.

Finalmente, como el cuestionario utilizado para la encuesta (GEQ [16]) se encuentra en inglés, fue necesario realizar una traducción al español de este, para que así los participantes no tuvieran problemas al responder. Debido a que técnicamente no se utiliza el cuestionario original, y a que las traducciones podrían no corresponder con la intención original de alguna de las preguntas, se introduce una posible imprecisión en los resultados, la cual podría ser evitada con la existencia de una traducción oficial.

Capítulo 7

Conclusión y Trabajo Futuro

A lo largo del desarrollo del trabajo realizado, es posible notar cómo lograr una mejora en la experiencia de usuario a través de un ajuste dinámico de dificultad basado biofeedback no corresponde a un proceso que pueda ser desarrollado de manera inmediata. En efecto, lograr lo anterior requirió de una gran cantidad de fases, las cuales al trabajar en unísono permitieron obtener resultados favorables dentro de los parámetros establecidos.

De esta manera fue necesario como punto de partida realizar un estudio para seleccionar el dispositivo adecuado para obtener el pulso del usuario. Luego se debió implementar una base que permitiera la conexión de este dispositivo con una aplicación a través de bluetooth. Para tener un medio sobre el cual realizar el ajuste de dificultad se tuvo que diseñar e implementar un juego de plataforma cuya estructura permitiera variaciones de dificultad conservado su esencia. Ya con el juego se debió desarrollar un sistema de comunicación entre éste y la aplicación previa para permitir el intercambio de comandos e información. Finalizando, se tuvo que diseñar un algoritmo que recibiera esta información y la procesara de manera dinámica, de tal manera de adaptar sus resultados a cada tipo de usuario, teniendo la capacidad de sobreponerse a situaciones inesperadas.

Para evaluar el trabajo realizado, se realizó una prueba de concepto con una muestra—por conveniencia—de usuarios reales. Los resultados obtenidos permitieron que el esfuerzo requerido fuera recompensado. Dada la muestra que participó en el experimento, fue posible establecer cómo se mejoró la experiencia de usuario de aquellos participantes que fueron afectos al ajuste dinámico de dificultad durante la sesión de juego. En efecto, éstos cometieron menos errores y requirieron menos tiempo para completar los niveles. Si bien estos resultados no fueron estadísticamente significativos, ni tienen el poder estadístico suficiente para generalizarse (debido al tamaño de la muestra), si se observó una magnitud de efecto considerable para el grupo de estudio.

Los resultados del ajuste dinámico de dificultad también pudieron ser evidenciados en cada nivel por separado.Así pues, se observaron tendencias interesantes como el hecho de que después de completar un nivel en una dificultad menor debido al ajuste, se hacía más factible que el siguiente nivel se percibiera como más complicado. También, a medida que aumentaba la dificultad base de los niveles al acercarse al final del juego, se hizo más presente el efecto

del ajuste, permitiendo observar una clara ventaja para este grupo.

Respecto a las percepciones mostradas por los participantes, se pudo notar cómo el grupo con ajuste se mostró más cómodo con el juego, expresando que si bien lo encontró más fácil, su experiencia fue más enriquecedora. Este hecho se vuelve particularmente interesante al considerar que la dificultad promedio para ambos grupos fue prácticamente la misma. De esta manera, se pudo evidenciar que la dificultad de los niveles fue ajustada adecuadamente de acuerdo a los requerimientos del usuario, haciéndolos más fáciles cuando tenían dificultades e incrementando la dificultad cuando buscaban un mayor desafío que el presentado por defecto.

A pesar de lo anterior, ambos grupos demostraron su agrado general hacia el juego, tanto en su diseño de niveles, en los elementos gráficos utilizados y en la progresión implementada para este. También expresaron deseos de mejora en los mismos aspectos específicos del juego, cuyo diseño fue realizado de manera premeditada para el desarrollo del experimento. Así, si bien son aspectos mejorables para una versión definitiva del juego, fue pertinente utilizarlos en su estado actual para lograr el éxito de las pruebas realizadas.

Tanto el proceso de desarrollo como el de evaluación permiten demostrar que es factible utilizar biofeedback para ejecutar un ajuste dinámico de dificultad, no necesitándose equipo especializado y lográndose resultados favorables por parte de los usuarios.

Debido la utilización del Xiaomi Mi Band como sensor de pulso, cualquier usuario que posea este monitor de actividad puede ser capaz de utilizar el ajuste de dificultad. En caso contrario, es posible adquirirlo a un precio mercado inferior a $30 dólares. Esto permite que el juego desarrollado pueda ser lanzado comercialmente, sin que el jugador deba realizar una gran inversión para disfrutar de manera completa las funcionalidades implementadas.

Por otro lado, dado que se creó una aplicación independiente para la comunicación con la pulsera, no es necesario desarrollar un juego nuevo si se quiere dar soporte a más plataformas. De esta manera, se requeriría únicamente escribir una nueva aplicación que siga las mismas guías de comunicación descritas, lo que abre la posibilidad de dar soporte a más variedad de monitores de actividad.

Análogamente, el algoritmo dinámico de ajuste implementado demostró ser eficiente durante las pruebas realizadas, cumpliendo con las expectativas con las que fue creado. Esto demuestra su factibilidad de uso y abre el abanico para desarrollar aplicaciones de ajuste dinámico en otros contextos que requieran mantener un rango de trabajo el cual se vaya modificando de manera dinámica a la velocidad y con la fluidez que especifiquen sus parámetros iniciales, pudiendo recuperarse de diversas situaciones inesperadas y logrando eventualmente converger a un punto estable. Esto se podría traducir en la extensión y exposición de una API de servicios que pueda ser utilizada para adaptar la capa de dificultad de algún nuevo juego que se esté desarrollando a partir de los sistemas implementados en este memoria. Otras variables a estudiar podrían ser el foco de atención o los rasgos de personalidad del jugador

A partir de los resultados obtenidos es posible desprender distintos elementos que pueden ser utilizados para la realización de trabajos futuros. En primer lugar, si bien se obtuvo una respuesta favorable por parte de la muestra escogida, para poder generalizar los resultados

hacia la población de estudio es necesario replicar el experimento pero un mayor número de participantes, los cuales sean escogidos de manera aleatoria y con distribuciones que permitan estudiar el efecto conseguido tanto por género, edad y experiencia previa con videojuegos.

También fue posible observar que tanto el número de intentos como el tiempo utilizado para terminar un nivel pueden ser utilizados de manera indistinta como medidores de la dificultad. Sin embargo, éstos por sí mismos podrían pertenecer al algoritmo que controla la dificultad, logrando de esta manera poder regular situaciones en las que al jugador le está tomando una gran cantidad de reintentos completar un nivel, o bien si ha estado mucho tiempo sin lograr realizar un avance real. Así sería posible mantener una curva de dificultad más nivelada a lo largo de los niveles, previniendo la existencia de algún conjunto particular de niveles que requieran mucho más esfuerzo que los demás, y permitiendo modelar más fácilmente la progresión de desafío que se busque conseguir.

De igual manera, se podrían introducir nuevos elementos al juego. Entre éstos se encontrarían el generar una mayor interacción con la historia del juego, la cual podría ir mostrando progresión en el arco argumental a medida que el jugador avanza dentro de los niveles. También resultaría interesante crear nuevas mecánicas que permitan al jugador interactuar con su entorno, las cuales podrían ser habilitadas de manera gradual a lo largo del juego, o luego de completar algún objetivo en particular. Esto último permitiría el diseño de muchos más tipos de niveles que exploten las nuevas habilidades del personaje. Por otro lado, se podrían introducir nuevos tipos de obstáculos y/o enemigos, los cuales se encarguen de poner a prueba al usuario de algún modo en específico que sea de interés analizar. Y, a pesar de que no afecta de manera directa la jugabilidad, no por eso no sería menos importante desarrollar música específica para el juego, así como agregar efectos de sonido para los distintos elementos presentes en los niveles, modos de movimiento e interacciones con el personaje.

Finalmente, si bien se utiliza el pulso como base para realizar el ajuste de dificultad, la lógica y fundamentos expuestos en esta memoria pueden ser utilizados para construir experimentos que utilicen fuentes distintas de biofeedback que estén ligadas a la dificultad. También se puede utilizar este trabajo para investigar si nuevas fuentes de biofeedback puedan tener algún efecto utilizable para controlar la dificultad de un juego, como por ejemplo el seguimiento de la mirada del usuario dentro de la pantalla.

Bibliografía

[1] Azarbarzin, A., Ostrowski, M., Hanly, P., and Younes, M. Relationship between Arousal Intensity and Heart Rate Response to Arousal. *Sleep 37*, 4 (04 2014), 645–653.

[2] Bluetooth SIG, I. Gatt overview. `https://www.bluetooth.com/specifications/gatt/generic-attributes-overview`, 2019. [Online; accessed 05-January-2019].

[3] Davidson, R., Akiba, Cufí, C., and Townsend, K. Chapter 4. gatt (services and characteristics). `https://www.oreilly.com/library/view/getting-started-with/9781491900550/ch04.html`, 2019. [Online; accessed 05-January-2019].

[4] Dormehl, L. New vr horror game gets scarier if your heart rate isn't fast enough. `https://www.digitaltrends.com/cool-tech/bring-to-light-heart-rate-vr/`, 2018. [Online; accessed 28-October-2018].

[5] Dornbush, J. Update: Comparing the price of every game console, with inflation. `https://www.ign.com/articles/2016/10/04/comparing-the-price-of-every-game-console-with-inflation`, 2016. [Online; accessed 05-January-2019].

[6] Drachen, A., Nacke, L. E., Yannakakis, G., and Pedersen, A. L. Correlation between heart rate, electrodermal activity and player experience in first-person shooter games. In *Proceedings of the 5th ACM SIGGRAPH Symposium on Video Games* (New York, NY, USA, 2010), Sandbox '10, ACM, pp. 49–54.

[7] Duffy, J. The best heart rate monitors for 2019. `https://www.pcmag.com/roundup/352257/the-best-heart-rate-monitors`, 2018. [Online; accessed 05-January-2019].

[8] Fitbit. Web api. `https://dev.fitbit.com/build/reference/web-api/`, 2016. [Online; accessed 05-January-2019].

[9] Fullerton, T. *Game Design Workshop: A Playcentric Approach to Creating Innovative Games*. CRC Press, 2019.

[10] Games, E. Unreal engine 4.8 release notes. `https://docs.unrealengine.com/en-US/Support/Builds/ReleaseNotes/2015/4_8/index.html`, 2015. [Online; accessed 04-July-2019].

[11] Games, E. Game flow overview. `https://docs.unrealengine.com/en-US/Gameplay/`

Framework/GameFlow/index.html, 2016. [Online; accessed 04-July-2019].

[12] Games, E. Unreal engine games. https://wiki.unrealengine.com/Unreal_Engine_Games, 2018. [Online; accessed 05-January-2019].

[13] Gilleade, K. Ubisoft's o.zen is now available in france. http://justkiel.com/2015/10/14/ubisofts-o-zen-released/, 2015. [Online; accessed 28-October-2018].

[14] Huang, Y.-C., and Luk, C.-H. Heartbeat jenga: A biofeedback board game to improve coordination and emotional control. In *Design, User Experience, and Usability: Interactive Experience Design* (Cham, 2015), A. Marcus, Ed., Springer International Publishing, pp. 263–270.

[15] Hunicke, R. The case for dynamic difficulty adjustment in games. In *Proceedings of the 2005 ACM SIGCHI International Conference on Advances in Computer Entertainment Technology* (New York, NY, USA, 2005), ACE '05, ACM, pp. 429–433.

[16] IJsselsteijn, W. A., de Kort, Y. A. W., and Poels, K. *The Game Experience Questionnaire*. Technische Universiteit Eindhoven, 2013.

[17] Law, E. L.-C., Brühlmann, F., and Mekler, E. D. Systematic review and validation of the game experience questionnaire (geq) - implications for citation and reporting practice. In *Proceedings of the 2018 Annual Symposium on Computer-Human Interaction in Play* (New York, NY, USA, 2018), CHI PLAY '18, ACM, pp. 257–270.

[18] Liu, C., Agrawal, P., Sarkar, N., and Chen, S. Dynamic difficulty adjustment in computer games through real-time anxiety-based affective feedback. *International Journal of Human–Computer Interaction 25*, 6 (2009), 506–529.

[19] Malleret, T. The wellness wearables market is booming—but so are concerns about data privacy. https://globalwellnessinstitute.org/global-wellness-institute-blog/2018/07/03/the-wellness-wearables-market-is-booming-but-so-are-concerns-about-data-privacy/, 2018. [Online; accessed 28-October-2018].

[20] Maslakovic, M. Guide to buying a pulse oximeter. https://gadgetsandwearables.com/2018/10/01/best-pulse-oximeter/, 2018. [Online; accessed 05-January-2019].

[21] Mcgarry, C. Who has the most accurate heart rate monitor? https://www.tomsguide.com/us/heart-rate-monitor,review-2885.html, 2019. [Online; accessed 05-January-2019].

[22] Mellado, I. Measuring heart rate with a smartphone camera. http://www.ignaciomellado.es/blog/Measuring-heart-rate-with-a-smartphone-camera, 2013. [Online; accessed 28-October-2018].

[23] Nenonen, V., Lindblad, A., Häkkinen, V., Laitinen, T., Jouhtio, M., and Hämäläinen, P. Using heart rate to control an interactive game. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (New York, NY, USA, 2007), CHI '07, ACM, pp. 853–856.

[24] Nikishaev, A. How i hacked my xiaomi miband 2 fitness tracker - a step-by-step linux guide. https://medium.com/machine-learning-world/how-i-hacked-xiaomi-miband-2-to-control-it-from-linux-a5bd2f36d3ad, 2018. [Online; accessed 02-January-2019].

[25] Nintendo Wiki contributors. Wii vitality sensor. http://nintendo.wikia.com/wiki/Wii_Vitality_Sensor, 2018. [Online; accessed 28-October-2018].

[26] Palladino, V. Hear the pulse: Heart rate monitoring fitness earbuds tested. https://arstechnica.com/gadgets/2016/12/hear-the-pulse-heart-rate-monitoring-fitness-earbuds-tested/, 2016. [Online; accessed 05-January-2019].

[27] Peckham, J. Best smartwatch 2019: the top choices you can buy. https://www.techradar.com/news/wearables/best-smart-watches-what-s-the-best-wearable-tech-for-you-1154074, 2018. [Online; accessed 05-January-2019].

[28] Rettner, R. How accurate are fitness tracker heart rate monitors? https://www.livescience.com/56459-fitness-tracker-heart-rate-monitors-accuracy.html, 2016. [Online; accessed 05-January-2019].

[29] Schell, J. *The Art of Game Design*. CRC Press, 2015.

[30] Shymanskyy, V. Miband 2 js library for node.js and html5 (webbluetooth). https://github.com/vshymanskyy/miband-js, 2018. [Online; accessed 02-January-2019].

[31] Stables, J. Best fitness tracker guide 2019: Fitbit, garmin, xiaomi and more. https://www.wareable.com/fitness-trackers/the-best-fitness-tracker, 2018. [Online; accessed 05-January-2019].

[32] Stables, J. Best heart rate monitors: Top watches, chest straps and fitness trackers. https://www.wareable.com/fitness-trackers/best-heart-rate-monitor-and-watches, 2018. [Online; accessed 05-January-2019].

[33] Statista. Wearables. https://www.statista.com/outlook/319/109/wearables/united-states, 2019. [Online; accessed 05-January-2019].

[34] Store, A. O.zen. https://itunes.apple.com/fr/app/o-zen/id481680612, 2018. [Online; accessed 05-January-2019].

[35] Tan, C. H., Tan, K. C., and Tay, A. Dynamic game difficulty scaling using adaptive behavior-based ai. *IEEE Transactions on Computational Intelligence and AI in Games 3*, 4 (Dec 2011), 289–301.

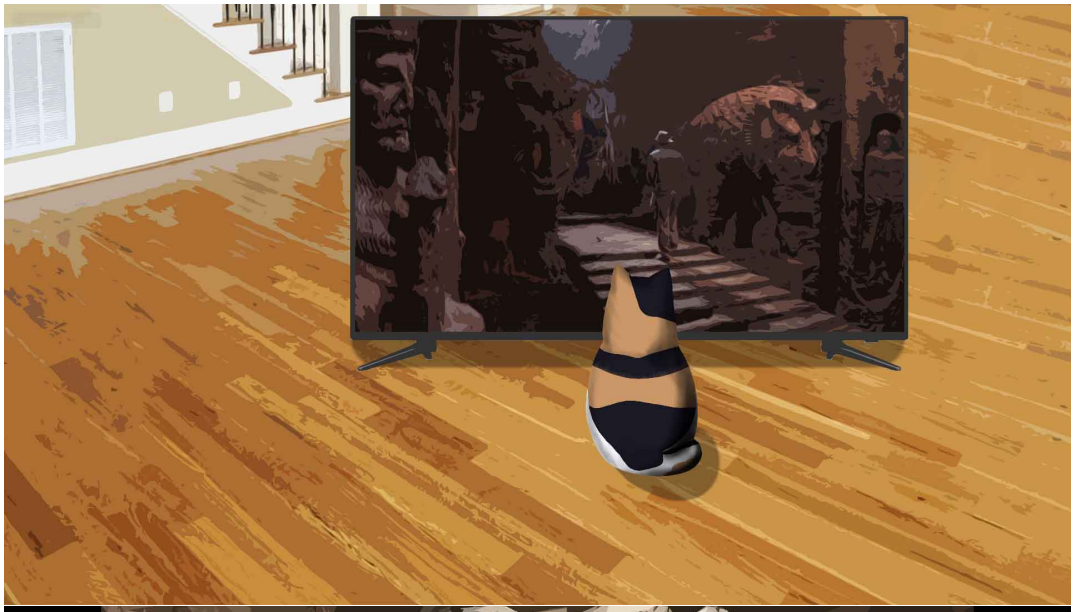
[36] Theory, N. 8 best pulse oximeters (reviews & buyers guide). https://www.nursetheory.com/best-pulse-oximeters/, 2015. [Online; accessed 05-January-2019].

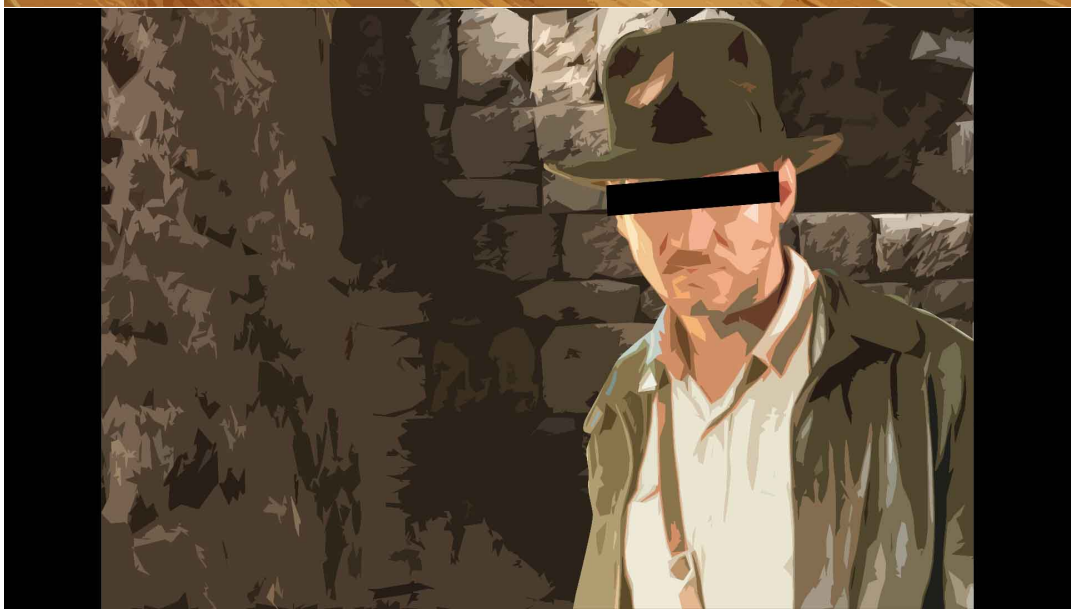
[37] Xiaomi. Mi band 3. https://www.mi.com/cl/mi-band-3/, 2018. [Online; accessed 05-January-2019].

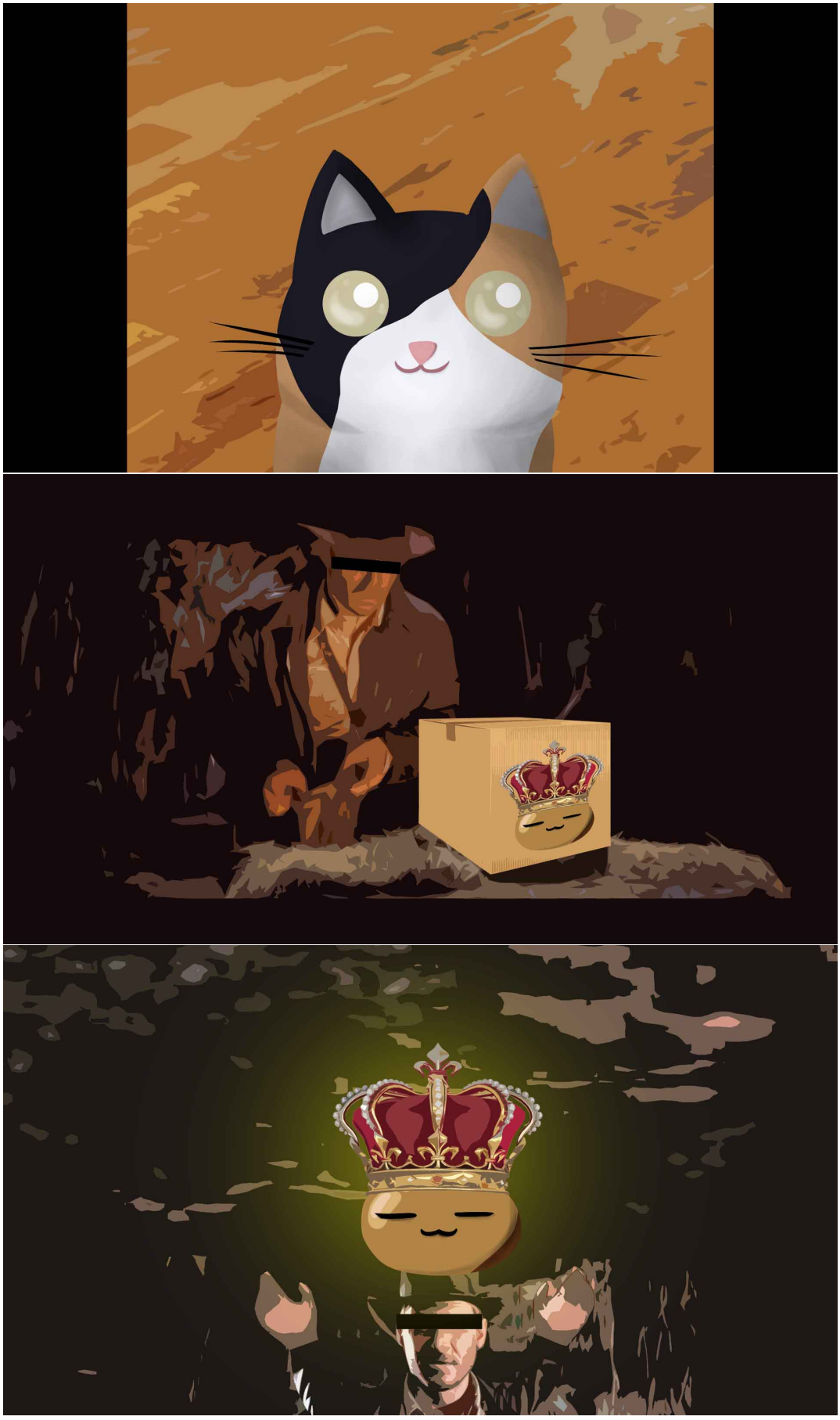
[38] Yu, B. Adaptive biofeedback for mind-body practices. In *Extended Abstracts of the ACM SIGCHI Conference on Human Factors in Computing Systems* (New York, NY, USA, 2016), CHI '16, ACM, pp. 260–264.

Apéndices

Apéndice A: Historia del juego

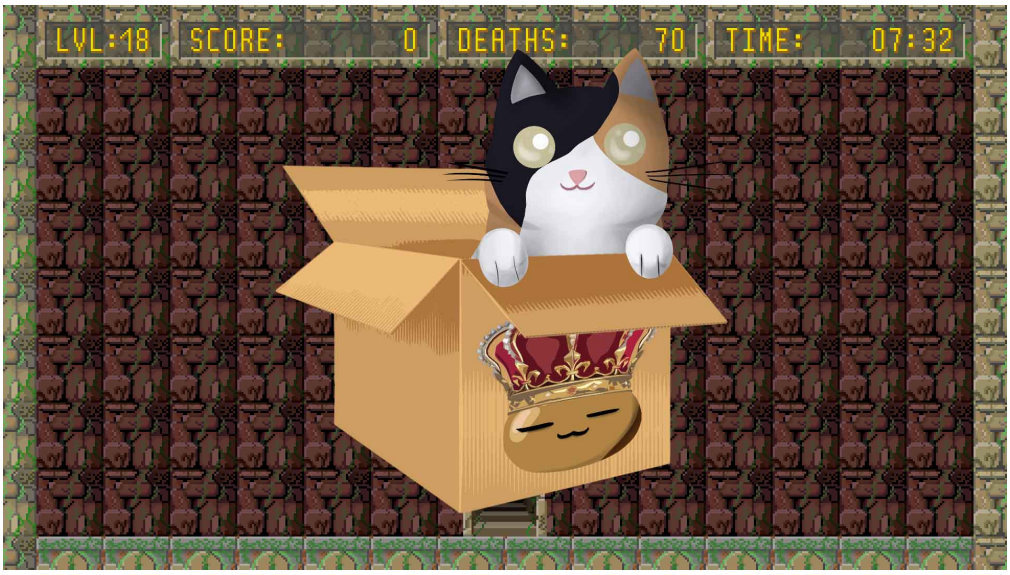




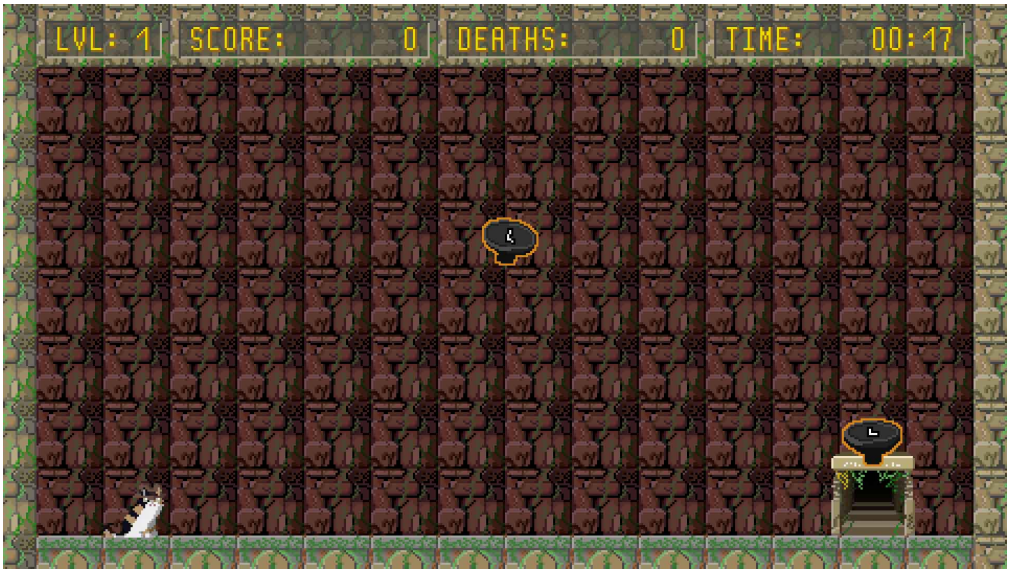


Apéndice B: Niveles del juego

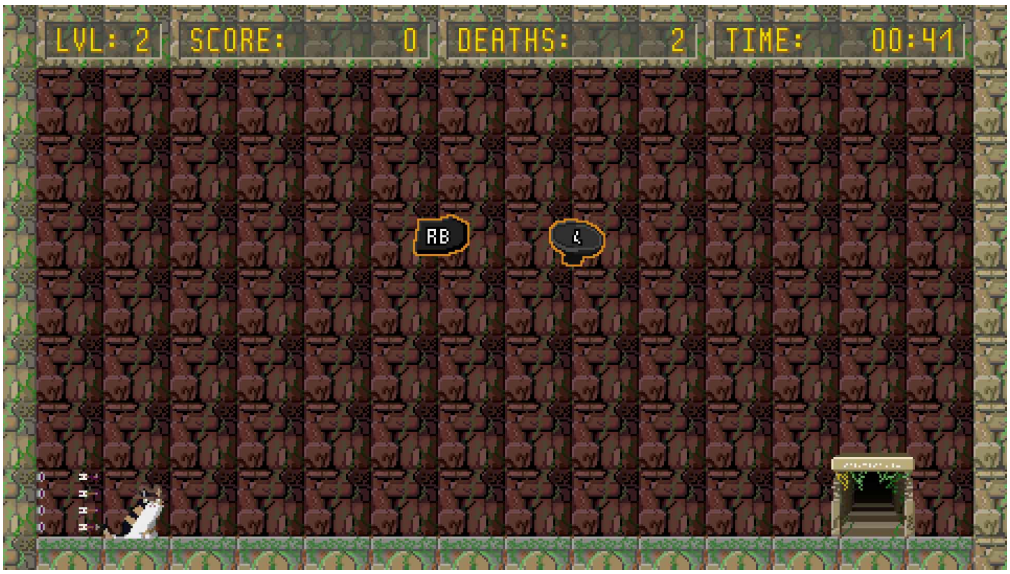
Pantalla de victoria



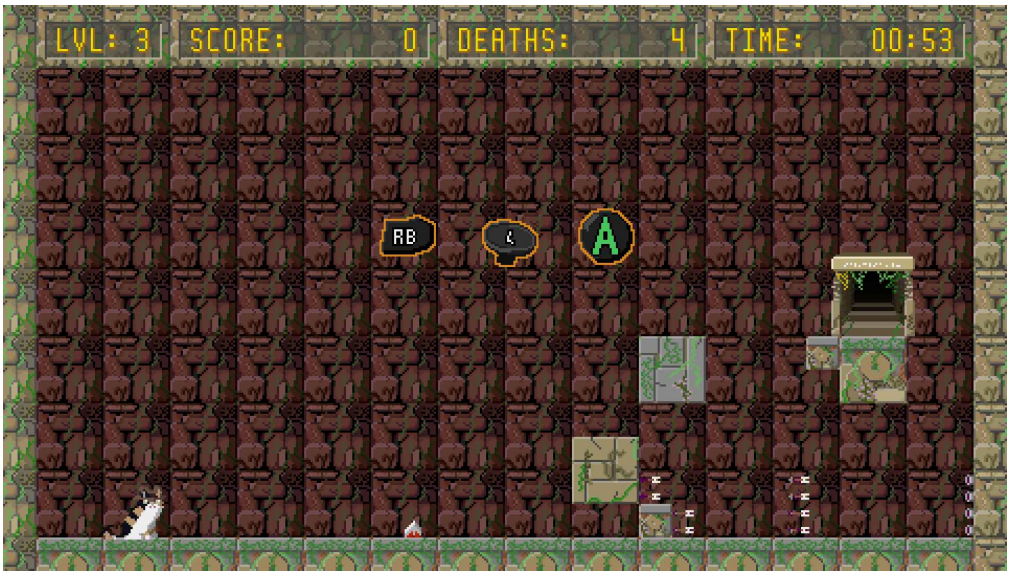
Nivel 1



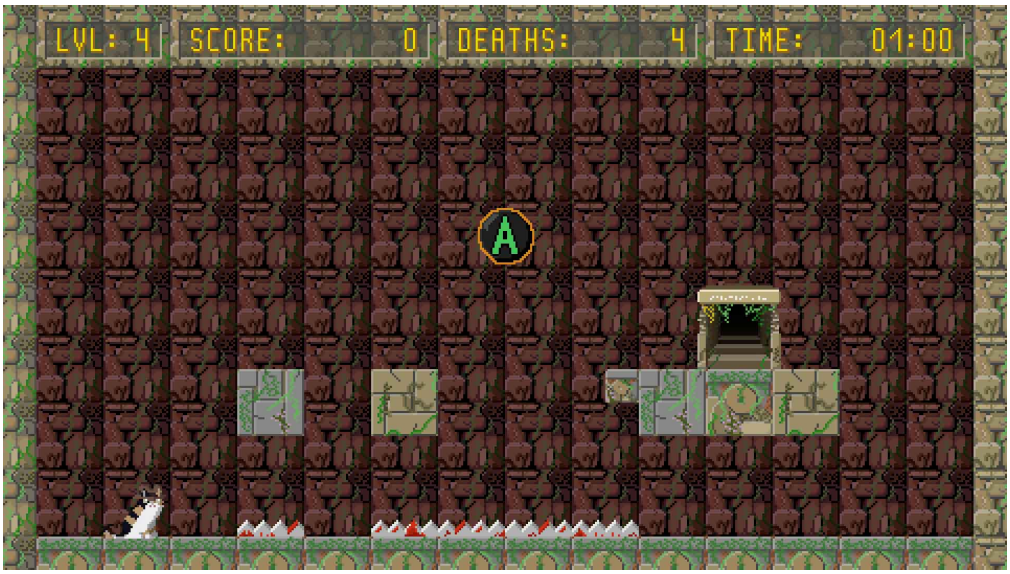
Nivel 2



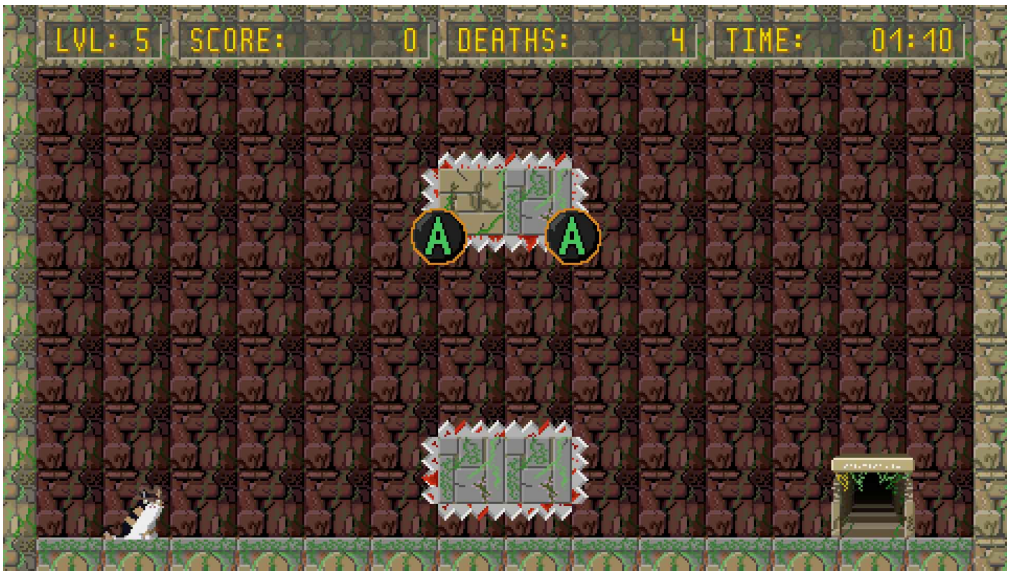
Nivel 3



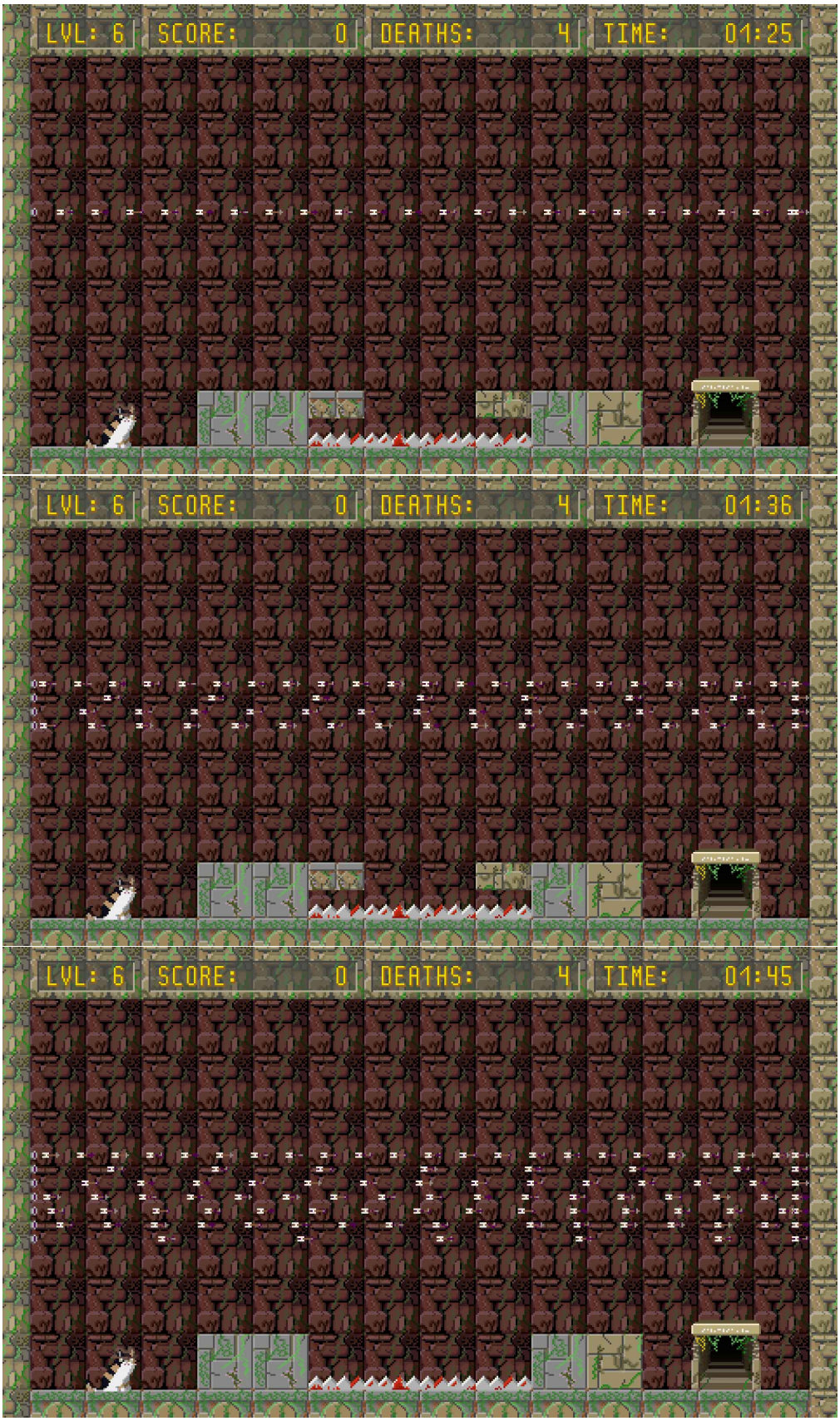
Nivel 4



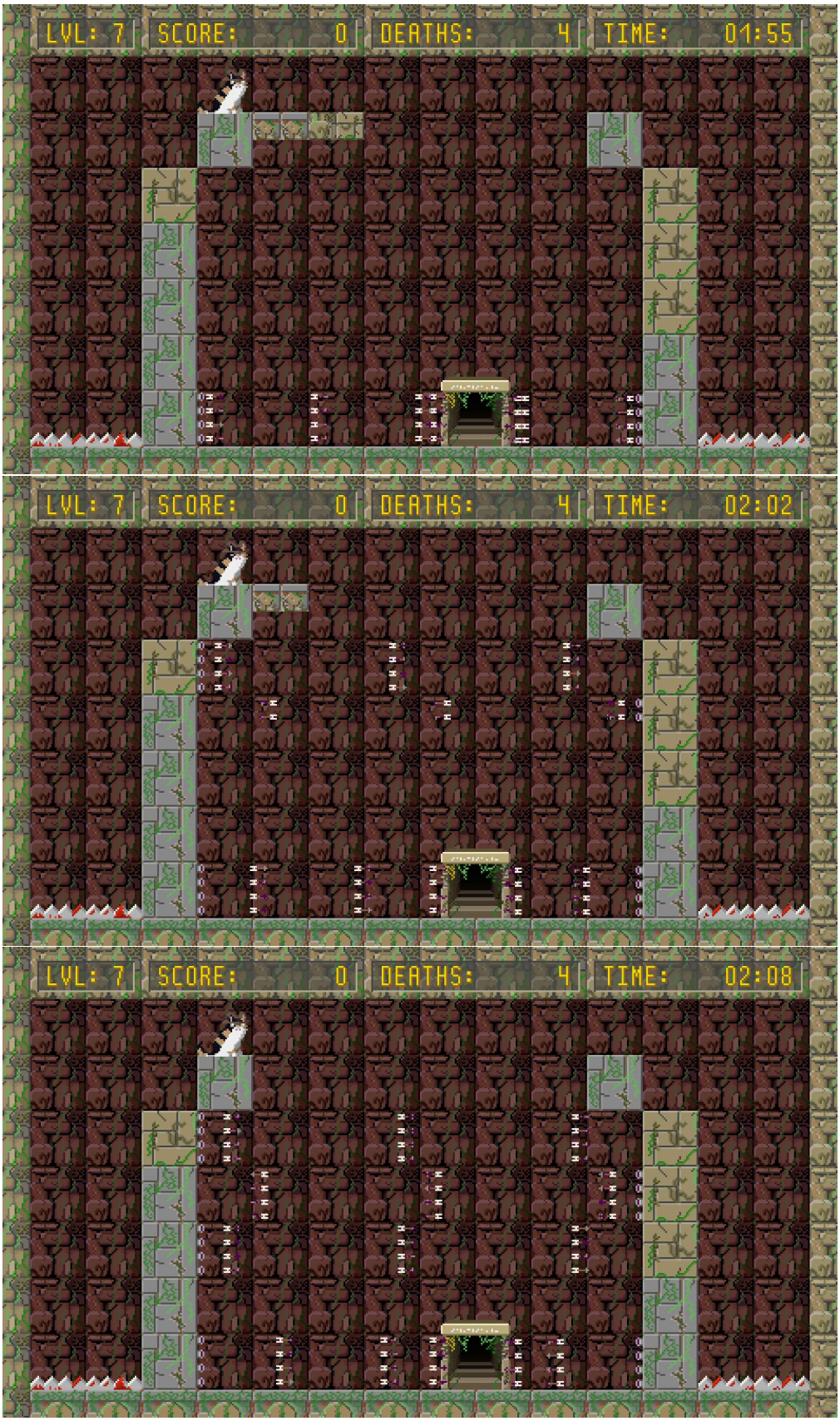
Nivel 5



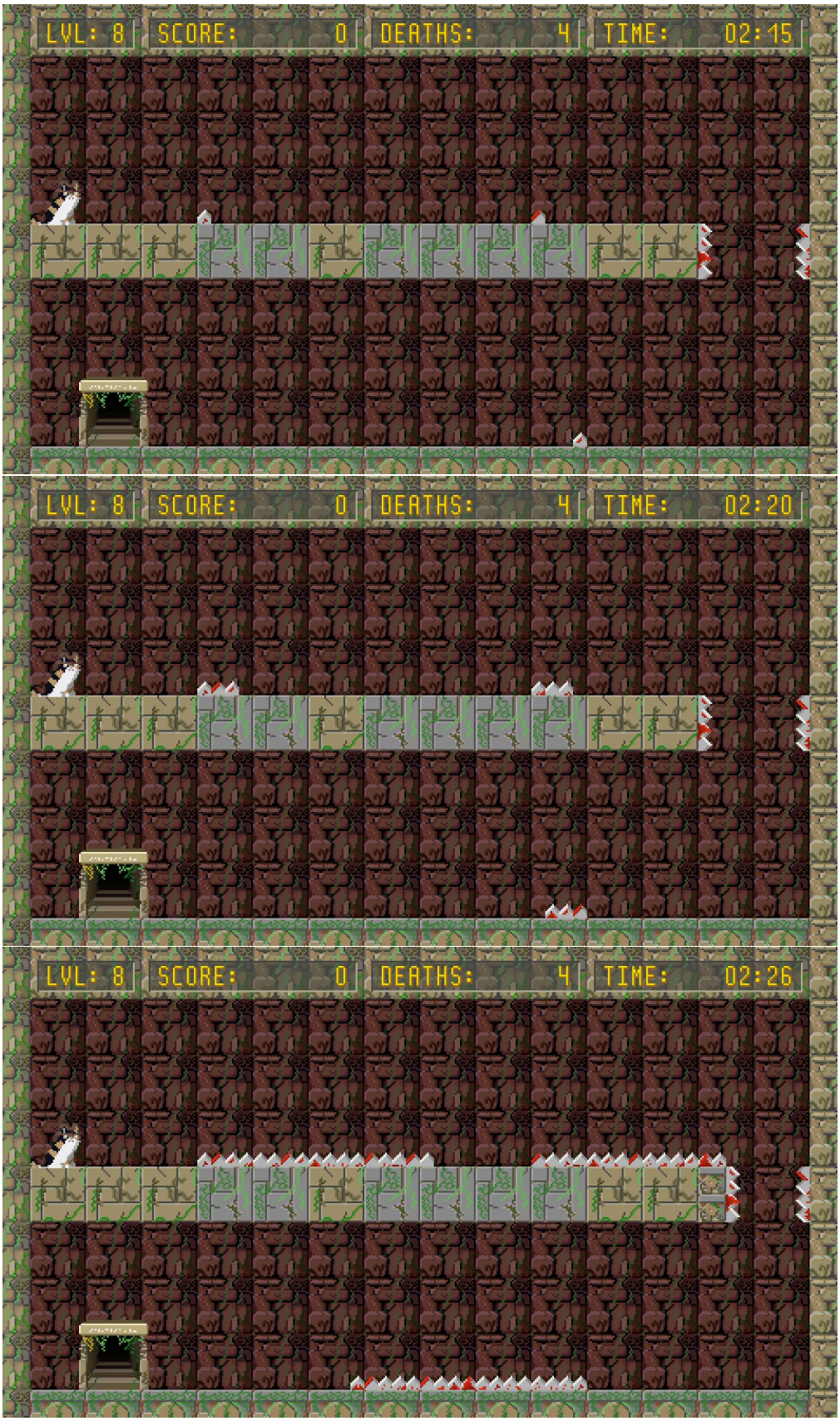
Nivel 6 - dificultad 0.0, 0.5 y 1.0



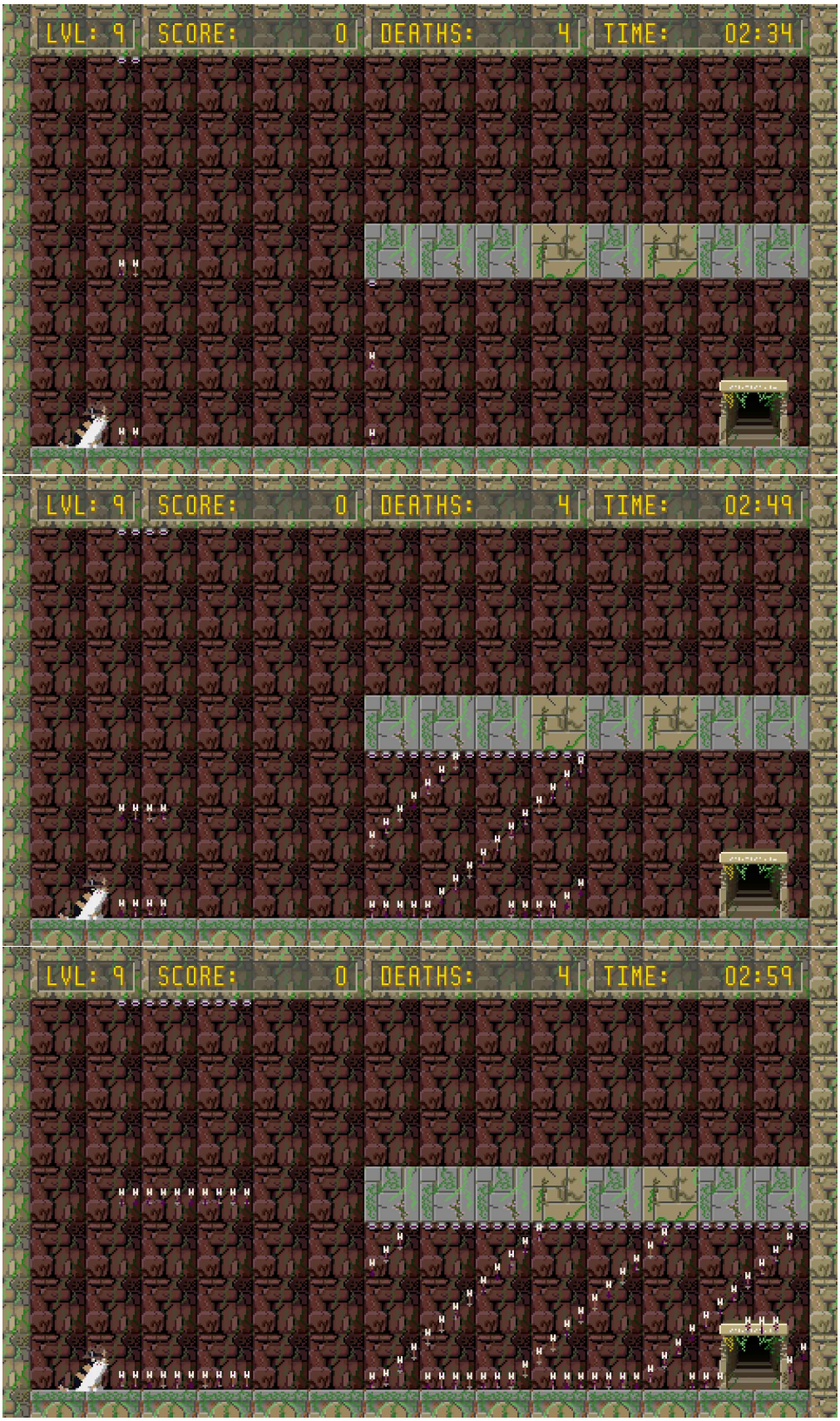
Nivel 7 - dificultad 0.0, 0.5 y 1.0



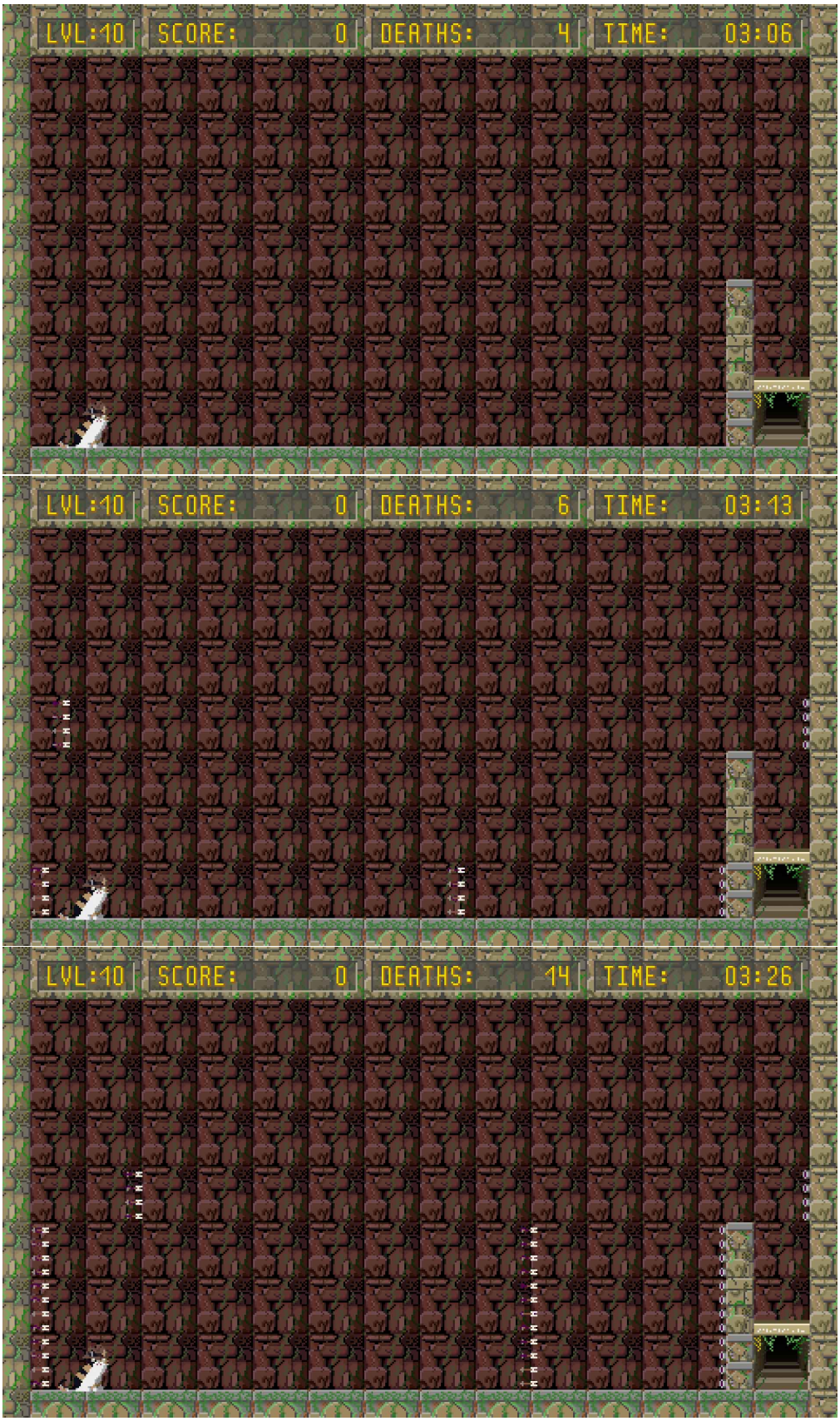
Nivel 8 - dificultad 0.0, 0.5 y 1.0



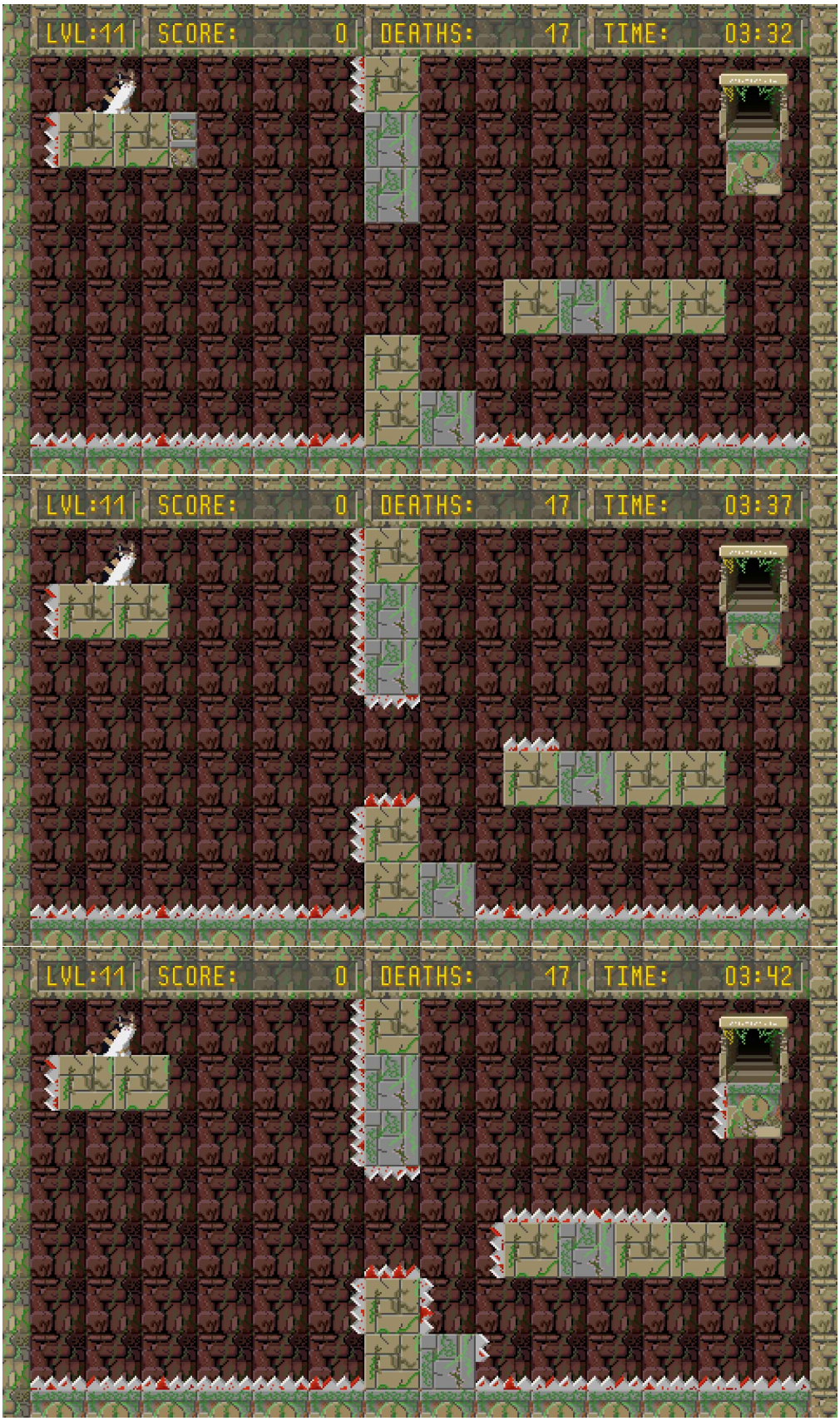
Nivel 9 - dificultad 0.0, 0.5 y 1.0



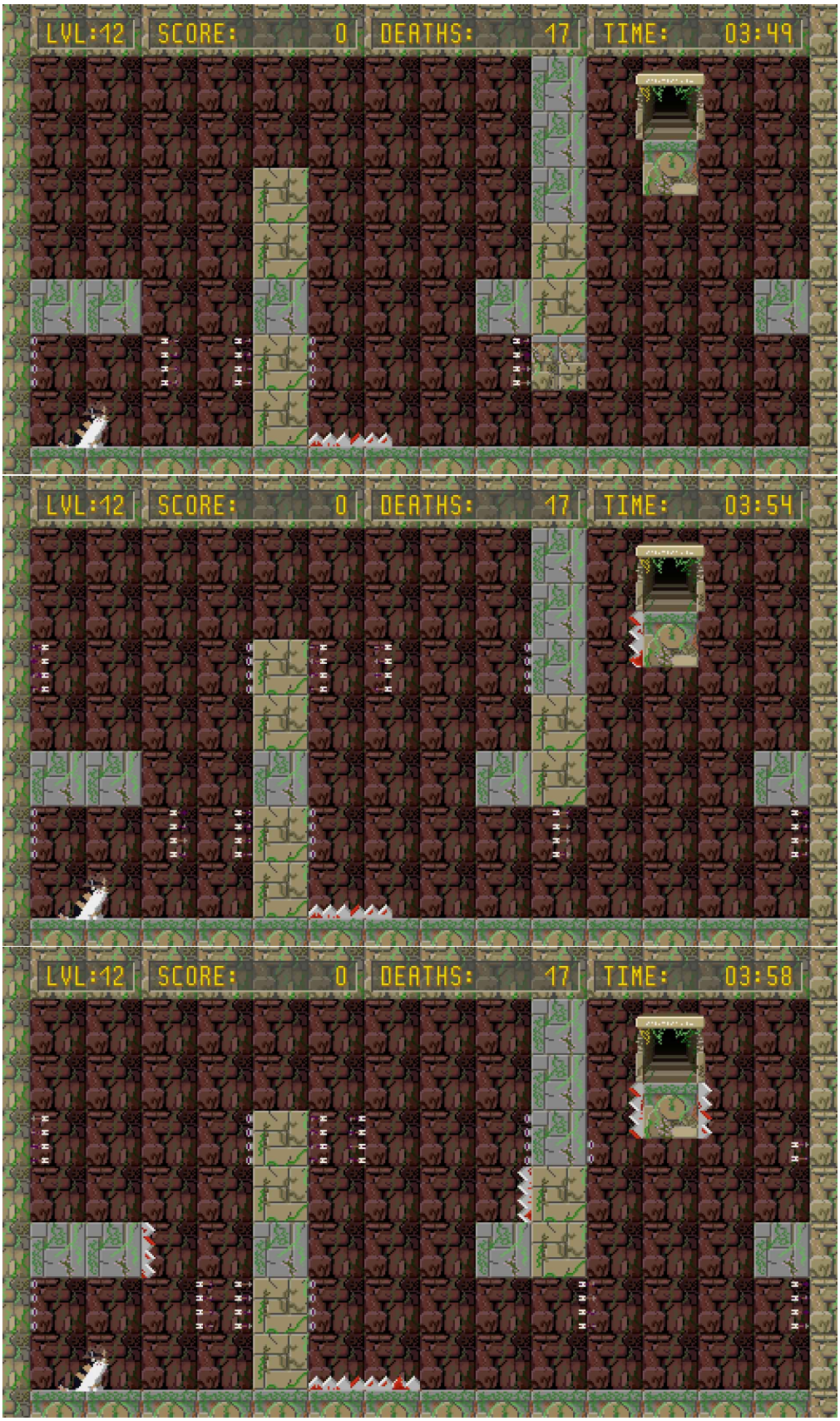
Nivel 10 - dificultad 0.0, 0.5 y 1.0



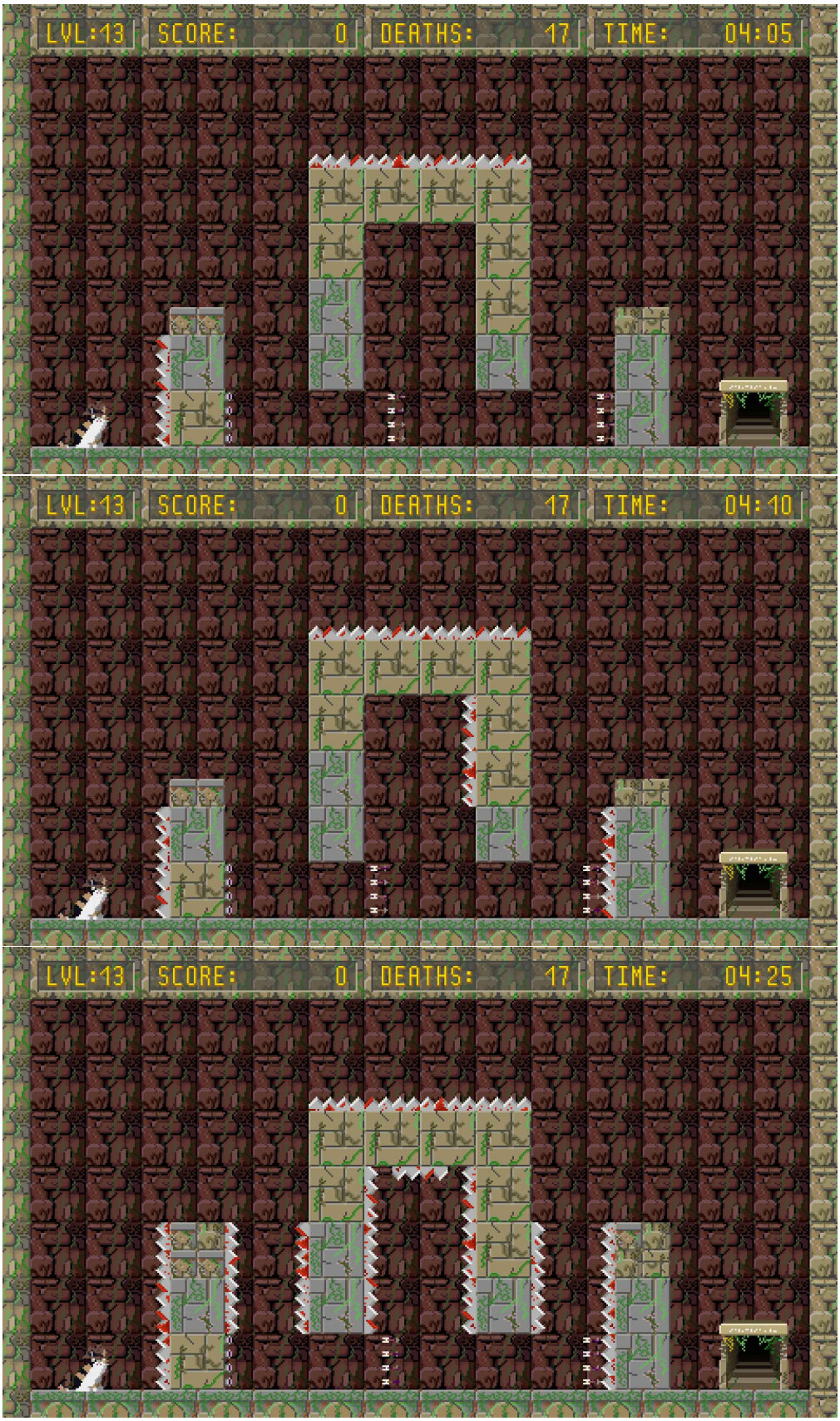
Nivel 11 - dificultad 0.0, 0.5 y 1.0



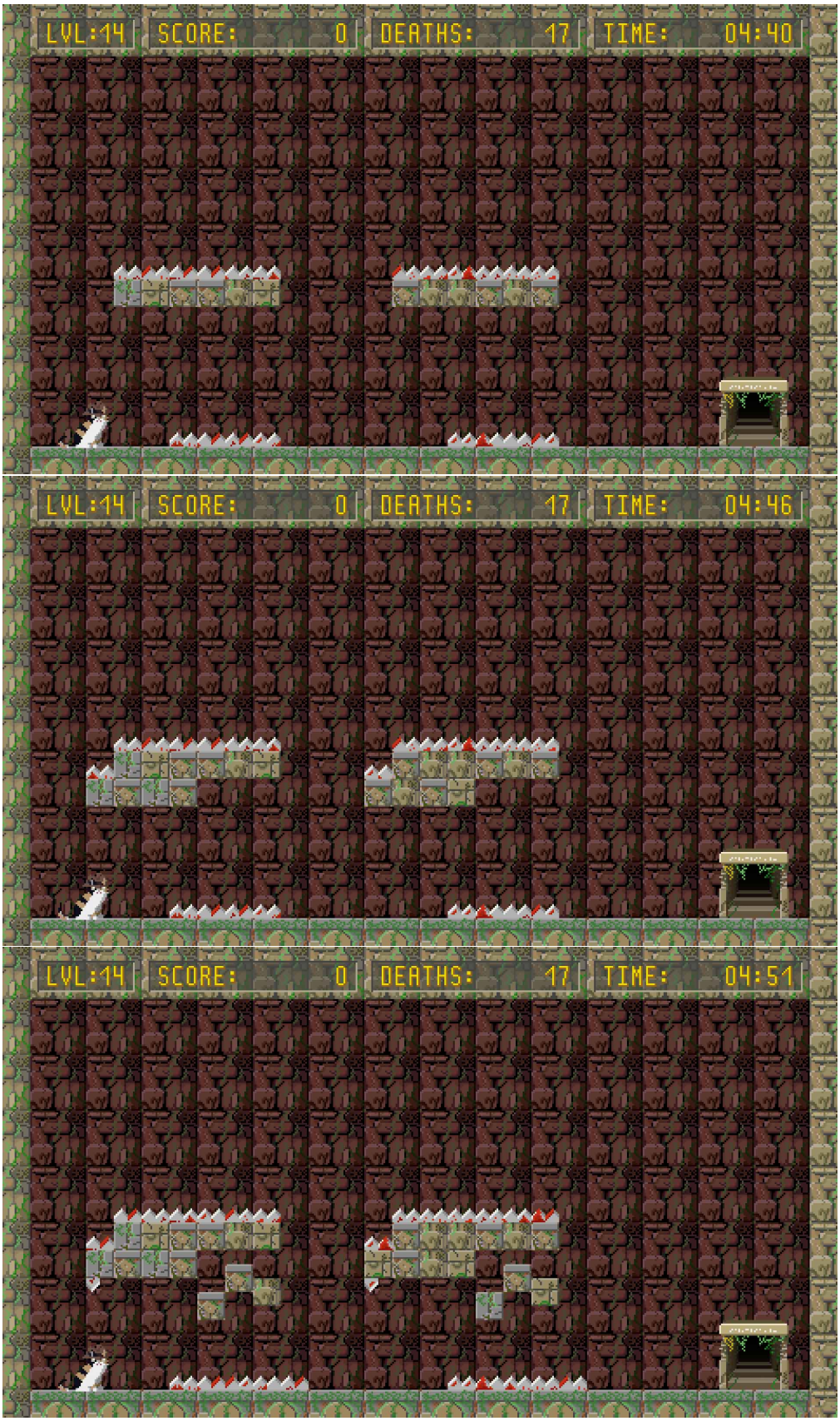
Nivel 12 - dificultad 0.0, 0.5 y 1.0



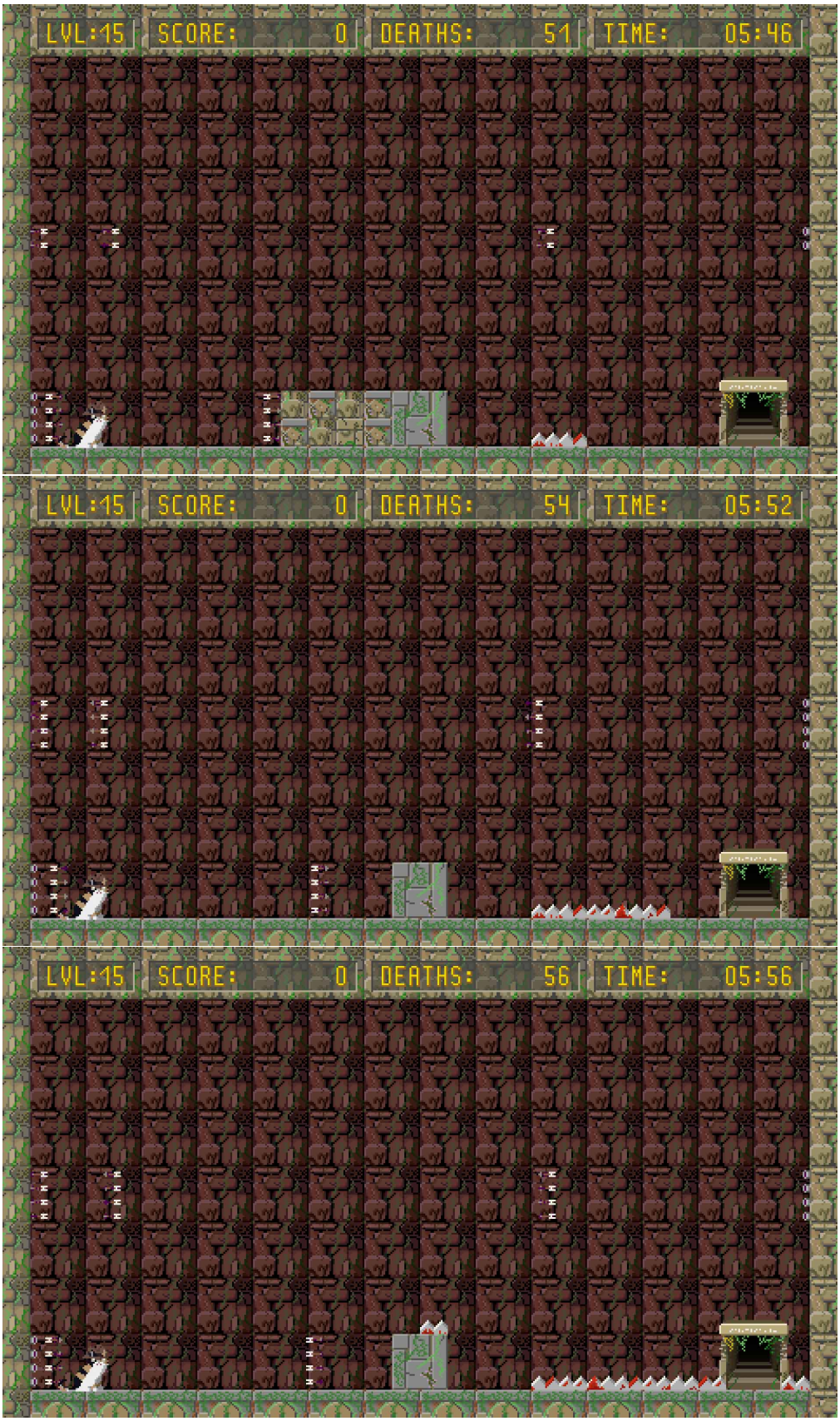
Nivel 13 - dificultad 0.0, 0.5 y 1.0



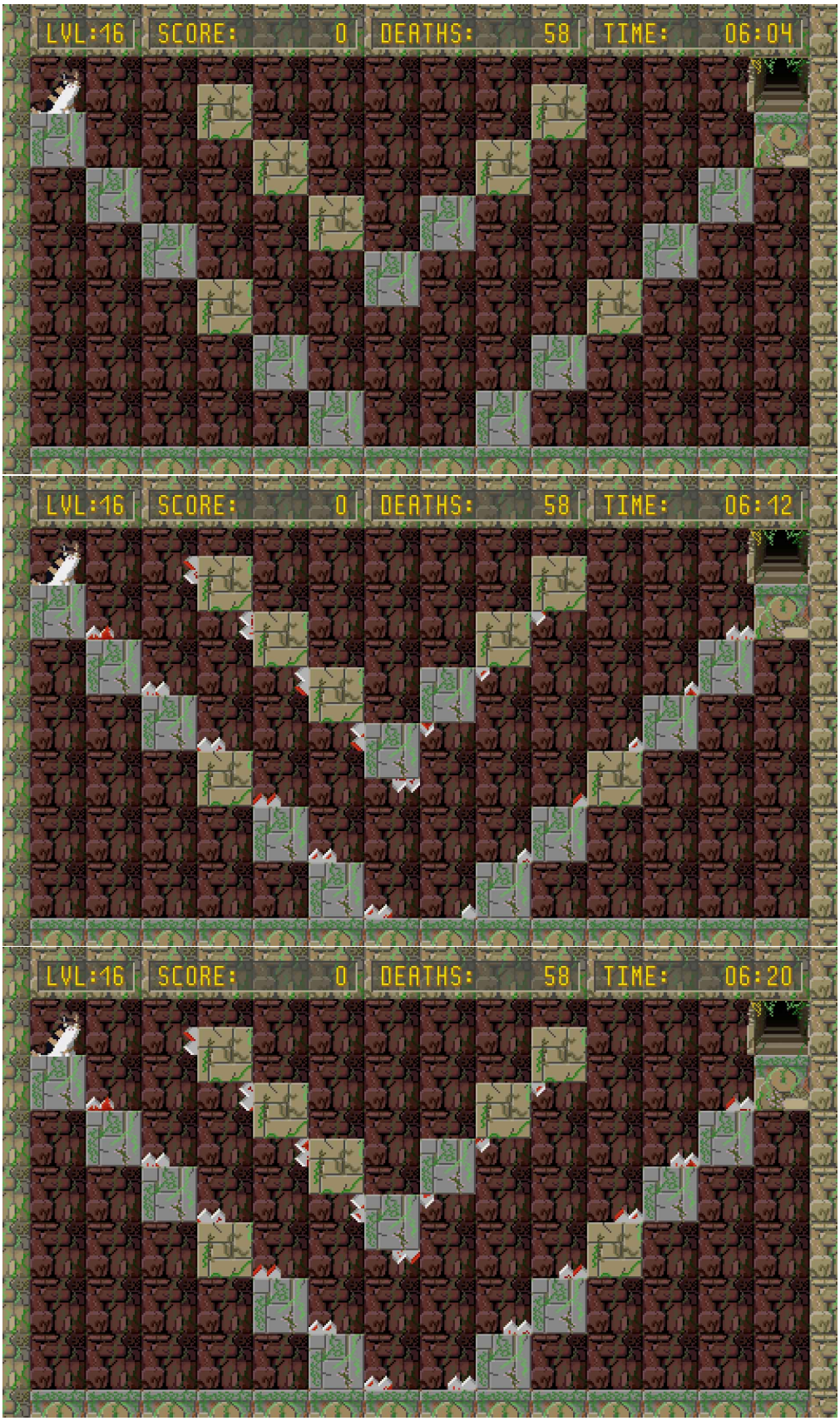
Nivel 14 - dificultad 0.0, 0.5 y 1.0



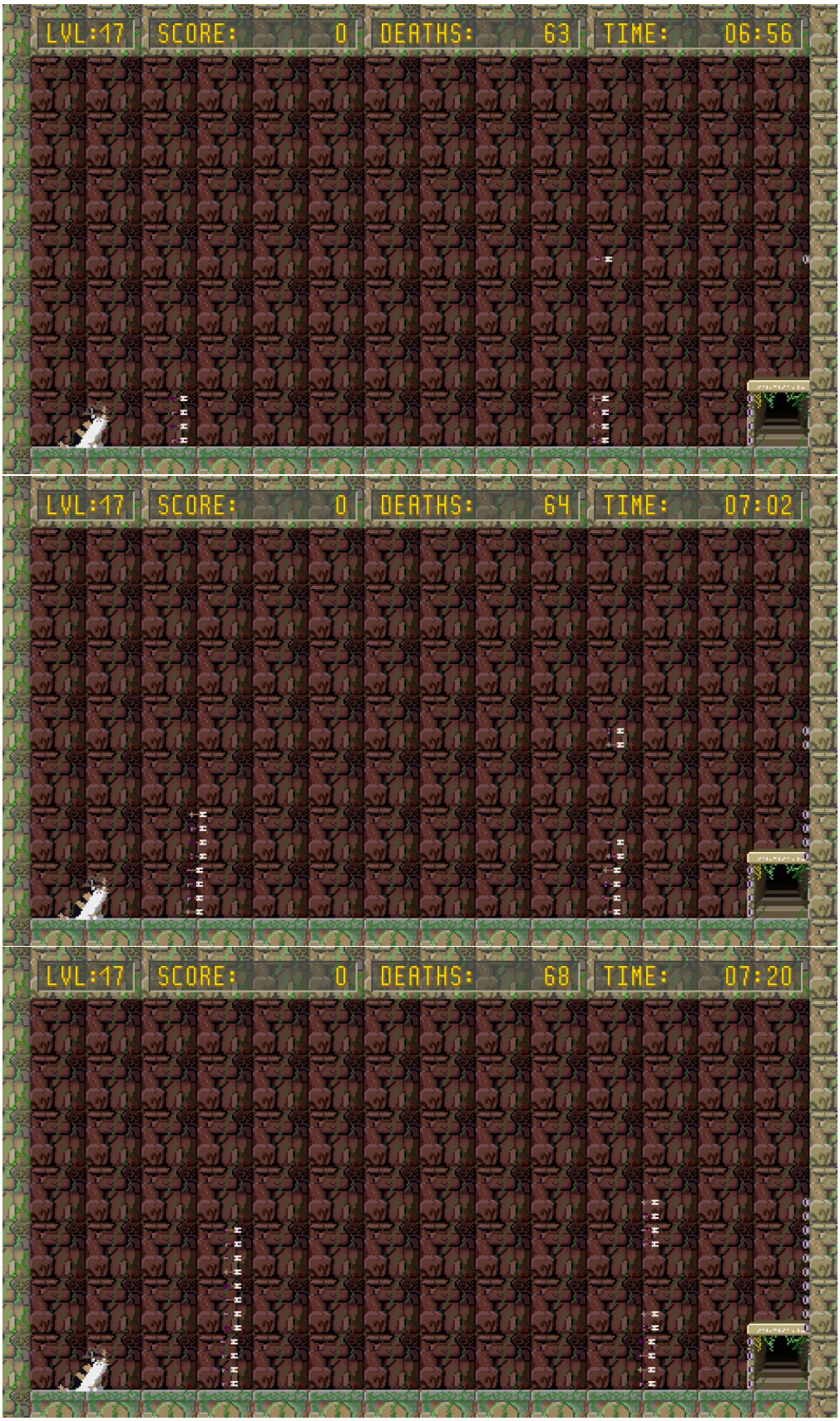
Nivel 15 - dificultad 0.0, 0.5 y 1.0



Nivel 16 - dificultad 0.0, 0.5 y 1.0



Nivel 17 - dificultad 0.0, 0.5 y 1.0



Apéndice C: Cuestionario de experiencia de juego

Sólo se presentan los módulos del cuestionario [16] utilizados dentro de la encuesta diseñada para esta memoria.

Módulo base

Para cada uno de los siguiente puntos indique cómo se sintió **DURANTE** el juego, usando la siguiente escala:

Para nada	Un poco	Más o menos	Bastante	Totalmente
0	1	2	3	4

1. Me sentí contento
2. Me sentí hábil
3. Estuve interesado en su historia
4. Pensé que fue divertido
5. Tuve toda mi atención puesta mientras jugaba
6. Me sentí feliz
7. Me puso de mal humor
8. Pensé en otras cosas mientras jugaba
9. Me pareció tedioso
10. Me sentí competente
11. Pensé que era difícil
12. Era estéticamente agradable
13. Olvidé todo a mi alrededor
14. Me sentí bien
15. Fui bueno en el juego
16. Me sentí aburrido
17. Me sentí exitoso
18. Me sentí imaginativo
19. Sentí que podía explorar las cosas
20. Lo disfruté
21. Fui rápido en alcanzar los objetivos del juego
22. Me sentí molesto
23. Me sentí presionado
24. Me sentí irritable
25. Perdí la noción del tiempo
26. Me sentí desafiado

27. Me pareció impresionante
28. Estuve profundamente concentrado en el juego
29. Me sentí frustrado
30. Se sintió como una experiencia enriquecedora
31. Perdí la conexión con el mundo exterior
32. Sentí la presión del tiempo
33. Tuve que esforzarme mucho

Módulo Post Juego

Para cada uno de los siguiente puntos indique cómo se sintió **DESPUÉS** del juego, usando la siguiente escala:

Para nada	Un poco	Más o menos	Bastante	Totalmente
0	1	2	3	4

1. Me sentí revitalizado
2. Me sentí mal
3. Me costó mucho volver a la realidad
4. Me sentí culpable
5. Se sintió como una victoria
6. Me pareció una pérdida de tiempo
7. Me sentí lleno de energía
8. Me sentí satisfecho
9. Me sentí desorientado
10. Me sentí exhausto
11. Sentí que podía haber hecho cosas más útiles
12. Me sentí empoderado
13. Me sentí cansado
14. Me sentí arrepentido
15. Me sentí avergonzado
16. Me sentí orgulloso
17. Tuve la sensación de haber regresado de un viaje