

## Treball final de grau

**Estudi: Grau en Disseny i Desenvolupament de Videojocs**

**Títol:** Desarrollo y análisis de videojuego con mecánica de cooperación online aleatoria

**Document:** Memòria

**Alumne:** Jaime Arancibia Soto

**Tutors:** Antonio Rodríguez Benítez i Marco González Núñez (Universidad de Talca)

**Departament:** Informàtica, Matemàtica Aplicada i Estadística

**Àrea:** Llenguatges i Sistemes Informàtics (LSI)

**Convocatòria (mes/any)** Setembre 2020



---

# Desarrollo de videojuego con mecánica de cooperación online aleatoria

---

Trabajo final de grado



GRAU EN DISSENY I DESENVOLUPAMENT DE VIDEOJOC  
INGENIERÍA EN DESARROLLO DE VIDEOJUEGOS Y REALIDAD VIRTUAL

Autor: Jaime Arancibia Soto

Tutor UdG: Antonio Rodríguez Benítez

Tutor UTalca: Marco González Núñez

# Contenido

1.	Introducción .....	9
1.1.	Tema .....	9
1.2.	Proceso.....	9
1.3.	Instrumentos de análisis de juegos cooperativos .....	9
1.4.	Uso del proyecto en el futuro .....	9
1.5.	Propósitos y objetivos del proyecto .....	9
1.6.	Propuesta inicial del videojuego.....	10
1.7.	Estructura del documento .....	10
2.	Marco teórico .....	12
2.1.	Desarrollo de videojuegos .....	12
2.1.1.	Diseño de alto nivel (Idea/Concepción) .....	12
2.1.2.	Diseño de nivel medio .....	17
2.1.3.	Modelos de negocios.....	18
2.1.4.	Público objetivo y perfil del jugador.....	19
2.1.5.	Publicación.....	22
2.1.6.	Mantenimiento.....	22
2.2.	Investigación sobre mecánicas cooperativas .....	22
2.2.1.	Interacción síncrona y asíncrona .....	23
2.2.2.	Juegos de suma no nula .....	24
2.2.3.	Cooperación y repartición de recompensas.....	25
3.	Estudio de viabilidad.....	26
3.1.	Recursos técnicos.....	26
3.1.1.	Hardware .....	26
3.1.2.	Software .....	27
3.2.	Recursos humanos.....	28
3.3.	Costo económico .....	29
3.3.1.	Coste de software de pago.....	29

3.3.2.	Costo de hardware .....	29
3.4.	Estudio de mercado .....	30
3.4.1.	Elección de modelos de negocios.....	30
3.4.2.	Competencia (estado del arte).....	30
3.4.3.	Cuadro comparativo .....	33
3.5.	Elección del perfil para el proyecto .....	34
3.6.	Cuadro de autovaloración .....	35
3.7.	Conclusión de viabilidad .....	35
4.	Metodología y plan de trabajo .....	36
4.1.	Planificación .....	36
4.2.	Producción y pruebas .....	37
4.3.	Metodologías de desarrollo ágil .....	37
4.4.	Abasto del proyecto.....	40
4.4.1.	Game Design Document (GDD) .....	40
4.4.2.	Tareas planificadas .....	40
5.	Marco de trabajo .....	44
5.1.	Referencia .....	44
5.2.	Motor de videojuego .....	46
5.2.1.	Game Object .....	47
5.2.2.	Componentes .....	48
5.2.3.	Assets.....	48
5.2.4.	Prefabs .....	48
5.3.	Servicios Externos .....	48
5.3.1.	PlayFab.....	48
5.3.2.	Unity Services .....	50
5.3.3.	DoTween (HoTween v2) .....	51
5.4.	Clasificación de edad .....	51
6.	Diseño del videojuego .....	53

6.1.	Estética.....	53
6.1.1.	Estilo de arte.....	53
6.1.2.	Iluminación y color .....	53
6.1.3.	Cámara.....	54
6.1.4.	Tilemap .....	55
6.1.5.	Animación.....	55
6.2.	Interfaz.....	55
6.2.1.	Inputs .....	55
6.2.2.	Outputs .....	55
6.2.3.	Usabilidad .....	55
6.2.4.	Flowchart .....	60
6.3.	Narrativa .....	61
6.4.	Audio.....	61
6.4.1.	Género e instrumentalización .....	61
6.4.2.	Formato del audio .....	62
6.4.3.	Adaptabilidad .....	62
6.4.4.	Efectos de sonido .....	62
6.4.5.	Consideraciones especiales .....	62
6.5.	Tecnología.....	62
6.5.1.	Creación de herramientas .....	62
2.1.2.	Integración con PlayFab .....	63
3.1.3.	Uso de Unity Ads .....	64
4.1.4.	Shaders .....	64
6.6.	Mecánicas .....	64
6.6.1.	Mecánicas centrales .....	64
2.2.4.	Mecánicas satelitales.....	68
6.7.	Personajes.....	69
6.7.1.	Personaje principal: Havoc .....	69

6.7.2.	Enemigos .....	70
7.	Implementación.....	73
7.1.	Estructura de ficheros.....	73
7.2.	Entorno de trabajo.....	74
7.2.1.	Grid .....	74
7.2.2.	UI.....	75
7.2.3.	Player .....	75
7.2.4.	Main Camera y Cinemachine Camera .....	76
7.2.5.	SoundManager .....	77
7.2.6.	GameManager .....	78
7.3.	Script de movimiento (GridMovement) .....	78
7.4.	Funcionamiento del GameManager .....	81
7.4.1.	Creación de niveles.....	81
7.4.2.	Reiniciar juego .....	82
7.4.3.	Muerte del personaje, fin de partida .....	83
7.5.	Sistema de misiones .....	84
7.5.1.	Aceptar misión.....	87
7.5.2.	Crear misión.....	89
7.6.	Extracción de datos.....	90
7.7.	Creación de enemigos.....	91
7.8.	Arte del videojuego.....	92
7.8.1.	Personaje principal .....	92
7.8.2.	Arte de enemigos y obstáculos .....	94
7.8.3.	Iluminación .....	94
7.9.	Enemigos.....	94
7.9.1.	Lanzador .....	94
7.9.2.	Slime y Volador.....	95
7.9.3.	Bastión .....	97

7.9.4.	Fantasma .....	99
7.9.5.	Estático .....	100
7.9.6.	Ninjarrior (adicional).....	101
7.9.7.	Pumpkummy (adicional).....	103
7.10.	Sonido.....	104
7.10.1.	Efectos de sonido.....	105
7.10.2.	Música.....	106
7.10.3.	Consideraciones especiales .....	107
7.11.	Navegación.....	107
7.11.1.	Gameplay central.....	107
7.11.2.	Configuración.....	108
7.11.3.	Lista de misiones.....	108
7.11.4.	Recompensas .....	108
7.11.5.	Ranking semanal .....	108
7.12.	PlayFab .....	108
7.12.1.	Sincronización de datos .....	108
7.12.2.	CloudScript.....	111
7.12.3.	Noticias .....	113
8.	Publicación.....	116
9.	Resultados .....	118
9.1.	Resultados del videojuego .....	118
9.2.	Datos recopilados .....	120
10.	Conclusiones.....	124
10.1.	Sobre el desarrollo del videojuego .....	124
10.2.	Análisis de datos recopilados.....	124
11.	Trabajo futuro.....	126
12.	Bibliografía.....	127





# 1. Introducción

## 1.1. Tema

Este trabajo tiene como tema principal de **desarrollo completo de un videojuego** para dispositivos móviles Android que incluye un instrumento para extraer datos sobre eventos y acciones dentro del videojuego a través de un sistema de negociación de recompensas. Sobre esto último, no es parte de la investigación el análisis concreto de los resultados obtenidos a partir de estos datos, ya que solo se propone el uso de videojuegos como instrumentos que podrían utilizarse en otros proyectos.

## 1.2. Proceso

Para completar el proyecto, se llevará a cabo el desarrollo de un videojuego, desde su concepción, diseño, producción y finalizando con su publicación. En las últimas secciones se analizará el trabajo hecho y cómo podría ser mejorado u optimizado.

## 1.3. Instrumentos de análisis de juegos cooperativos

Un videojuego, al conectarse con un servicio de back-end, podría llegar a utilizarse como instrumento que recopile información sobre las acciones y decisiones del jugador. Esto con el fin de analizar estos datos y optimizar la experiencia de ese mismo videojuego u otro diferente. Como punto de inicio, se puede desarrollar un videojuego que integre un sistema de recopilación de datos, pero cuyo objetivo sea probar su funcionamiento y no el análisis de los datos percibidos. Al ser un punto de partida, no se espera que el videojuego alcance un público masivo i que la masa de datos extraídos sea grande, pero al menos se espera que se logren recopilar datos en distintas situaciones que permitan entender el contexto en el que fueron creados y proponer maneras de relacionarlos o nuevos datos que podrían ser registrados.

## 1.4. Uso del proyecto en el futuro

Se espera que mediante el desarrollo de un videojuego se logre crear un instrumento que se pueda utilizar en otros proyectos de alcance masivo para obtener resultados que sean representativos.

## 1.5. Propósitos y objetivos del proyecto

Este proyecto tiene como fin el desarrollo de un videojuego aplicando los conocimientos teóricos y prácticos adquiridos y publicarlo para su uso por jugadores reales que puedan mantener un guardado de datos en línea e incluir una pequeña interacción entre estos. Adicionalmente, este videojuego contendrá un sistema para extraer, mas no analizar, los

datos de las acciones del jugador en una plataforma en línea accesible por el desarrollador. Así entonces, el videojuego puede usarse como referencia para otros proyectos en que se utilice el sistema de extracción de datos como instrumento para interpretar el comportamiento de los jugadores.

### *1.6. Propuesta inicial del videojuego*

En este proyecto se desarrollará **Havoc In The Dungeon**, un videojuego para Android en el que el jugador deberá sortear obstáculos para avanzar en un calabozo de longitud infinita y que permitirá recolectar monedas y otros objetos en un botín. Estos objetos deben enviarse a un laboratorio, por lo que el jugador debe asegurarse de llegar a un punto de conexión en el que guardar sus recompensas, de lo contrario perderá todo su progreso desde el último punto. Como alternativa, el jugador podrá solicitar ayuda a algún jugador aleatorio que podrá recuperar sus pertenencias mediante una misión de rescate, sin embargo, el otro jugador podría fallar o llevarse un gran porcentaje del botín. Como forma de equilibrar esta mecánica, si se decide llevar mayor parte del botín, la misión de rescate será más difícil, es a esta mecánica a la que se hace referencia cuando se habla de “**mecánica de cooperación online aleatoria**”.

### *1.7. Estructura del documento*

[En el capítulo 1](#) se realizó la introducción del proyecto, presentando el tema y los objetivos de la investigación.

[En el capítulo 2](#) se presenta el Marco Teórico, que consiste en la recopilación de información que fue investigada y que es de utilidad para comprender el desarrollo del proyecto.

[En el capítulo 3](#) se realiza un Estudio de Viabilidad en el que se describen los requisitos para poder desarrollar el proyecto.

[En el capítulo 4](#) se describe la metodología a utilizar y cómo se planificará el trabajo a realizar.

[En el capítulo 5](#) se presenta el marco de trabajo, el cual describe el funcionamiento de herramientas y servicios que se utilizarán para desarrollar el proyecto.

[En el capítulo 6](#) se detalla el diseño del videojuego que permitirá clarificar los componentes que se crearán y desarrollarán para él.

[En el capítulo 7](#) se describe el proceso de implementación del diseño del videojuego, lo que equivale al proceso de producción de este.

[En el capítulo 8](#) se presenta el proceso de publicación del videojuego y cómo se hizo llegar al público objetivo.

[En el capítulo 9](#) se encuentran los resultados del proyecto, tanto del videojuego en sí mismo como de su uso como instrumento para recopilar datos.

[En el capítulo 10](#) se realizan las conclusiones sobre el proyecto.

[En el capítulo 11](#) se describen los posibles pasos a seguir en el futuro para el videojuego desarrollado o el uso de este trabajo en nuevos proyectos.

Finalmente, [en el capítulo 12](#) se lista la bibliografía utilizada en este trabajo.

## 2. Marco teórico

En esta sección se desarrolla la teoría con la cual se fundamentará el resto del proyecto. Se divide en dos secciones: la materia respectiva al desarrollo de videojuegos y la investigación sobre mecánicas de cooperatividad.

### 2.1. Desarrollo de videojuegos

El proceso de creación de un videojuego, desde su concepción inicial hasta su publicación, pasa por distintas fases en las que participan diversas profesiones como la programación, arte, sonido, diseño, etc.

En esta subsección se describirán aquellos procesos y qué factores se toman en cuenta para su diseño.

Parte de la teoría se basa en el modelo MDA (Mechanics-Dynamics-Aesthetics), el cual es una herramienta para analizar los videojuegos desde el punto de vista del game design dividiéndolo en tres componentes: Mecánicas, Dinámicas y Estéticas (o emociones) [1] [2].

#### 2.1.1. Diseño de alto nivel (Idea/Concepción)

En esta etapa se establecen las reglas con las que contaremos para el resto del desarrollo, se describen las características del juego, requisitos y estrategias de desarrollo ágil.

##### 2.1.1.1. Pilares del Videojuego

Este paso es más orientado al diseño del videojuego como tal. Aquí son definidos los pilares del videojuego, los cuales son inamovibles y afectan drásticamente el resto del desarrollo.

Característica	Aplicación en Havoc In The Dungeon
¿Qué tipo de videojuego estamos produciendo?	Juego de acción tipo roguelike con mapa infinito.
¿Será en 2D o 3D?	2D, estilo pixel art.
¿Cuáles son algunas funcionalidades clave?	Explorar niveles, obstáculos fijos y móviles.
¿Quiénes son los personajes?	Un científico que debe destruir monstruos. Monstruos de mazmorras.

¿Dónde toma lugar el videojuego?	En una mazmorra oscura y peligrosa.
¿En qué plataformas estará disponible?	Android.
¿Qué tipo de persona jugaría al juego?	Jugadores casuales pero con buenas habilidades para superar los obstáculos.

Figura 1. Pilares del videojuego.

### 2.1.1.2. Elementos del videojuego

Se llaman elementos del videojuego a todos los componentes y partes que deben desarrollarse. La diferencia con los pilares del juego es que aquí se define qué es necesario crear o utilizar para que el juego tenga tales pilares. Los elementos se dividen en las siguientes categorías.

#### **Interfaz**

Este elemento describe qué dispositivos es necesario utilizar para jugar, que tipo de Interfaz de Usuario (UI) debe crearse.

#### **Mecánicas**

En este apartado se describen de manera general las reglas, objetivos y sistemas de juego que deben ser diseñados para el juego. Las mecánicas son subsistemas interactivos basados en reglas capaces de recibir entradas y reaccionar produciendo una salida.

#### **Audio**

Se describe qué tipo de música incluirá el juego, como se implementarán los efectos de sonido (SFX) y cuál será la estética sonora.

#### **Estética**

Se establece cuál es el estilo de arte que se usará y qué clase de animación debe implementarse. Además se indican decisiones específicas a utilizar en conjunto con los otros elementos. Por ejemplo, si es un juego rítmico, debe indicarse que ciertos objetos luzcan diferente según la música.

#### **Narrativa**

Se describe brevemente el conjunto de eventos, la temática del juego y la historia alrededor del videojuego.

## **Tecnología**

En este elemento se detalla el motor a utilizar o crear si es necesario. También el resto de herramienta de desarrollo que podrían ser necesarias así como el o los lenguajes de programación.

Para Havoc In The Dungeon, se establece el siguiente cuadro de elementos:

<b>Interfaz</b> <ul style="list-style-type: none"><li>- Flechas para moverse en cuatro direcciones.</li><li>- Información en la parte superior de la pantalla, fuera del gameplay.</li></ul>	<b>Estética</b> <ul style="list-style-type: none"><li>- Estilo pixel art (in-game y UI).</li><li>- Uso de iluminación 2D.</li><li>- Colores neones con fondos oscuros.</li></ul>
<b>Mecánicas</b> <ul style="list-style-type: none"><li>- Movimiento en cuatro direcciones.</li><li>- Generación de mapas de manera infinita.</li><li>- Puntos de guardado de recompensa.</li><li>- Sistema de misiones de rescate.</li></ul>	<b>Audio</b> <ul style="list-style-type: none"><li>- Sintetizador y cuerdas.</li><li>- Efectos de sonido lo-fi.</li></ul>
<b>Tecnología</b> <ul style="list-style-type: none"><li>- Unity Engine 2019.3.</li><li>- Azure PlayFab.</li><li>- Unity Services.</li></ul>	<b>Narrativa</b> <ul style="list-style-type: none"><li>- Explicada con texto e imagen solo al principio. No es una parte demasiado importante.</li></ul>

*Figura 2. Elementos del videojuego.*

### **2.1.1.3. Dinámicas de Juego**

Se denominan dinámicas de juego a aquellos comportamientos emergentes que se producen al interactuar con el juego. Son lo que media entre las reglas de juego y la experiencia del jugador. [3]

Algunas dinámicas populares son las recompensas, los rangos, logros, competencia, altruismo y comunicación.

Para Havoc In The Dungeon, se consideraron las siguientes dinámicas:

#### **1. Altruismo**

Esta permite compartir la moneda del juego con otros jugadores al repartir una recompensa por rescatarse entre sí.

## **2. Rangos**

Esta dinámica permite visibilizar las puntuaciones de los otros jugadores, en específico la máxima cantidad de pasos dados. Cada semana se reinicia la lista y se recompensa a los mejores puntajes.

### **2.1.1.4. Experiencias de Juego**

Las experiencias de juego son el conjunto de sensaciones y emociones que un jugador experimenta durante un videojuego, son un factor clave en la inmersión y el atractivo del juego. Algunas de estas experiencias son:

#### **1. Estimulación**

Es aquella experiencia causada por los elementos del videojuego que intentan llamar la atención del jugador y causar que este se sienta atraído al juego. Pueden ser causados por elementos gráficos o emociones específicas como miedo o adrenalina.

#### **2. Fantasía**

Esta experiencia se basa en la sensación de estar dentro del videojuego, tomando el rol del personaje o siendo afectado por el mundo que lo rodea.

#### **3. Narrativa**

Como su nombre lo indica, esta experiencia es causada por el conjunto de historias y eventos que se presentan a lo largo del juego y causan la inmersión del jugador gracias a su interés por saber qué ocurrirá y sentirse afectado por la trama.

#### **4. Desafío**

Ocurre cuando el jugador experimenta fuertes emociones al superar difíciles tareas y eventos del juego.

#### **5. Exploración**

Esta experiencia causa que el jugador sienta curiosidad por el mundo del videojuego, presentando lugares misteriosos o grandes donde el jugador pueda recorrer y descubrir nuevas cosas.

#### **6. Compañerismo**



Sucede cuando un jugador es capaz de ayudar a otro por un objetivo que puede o no ser común. Generalmente se presenta en videojuegos de más de un jugador donde hay mecánicas de equipo o puede darse gracias a la coordinación espontánea de dos o más jugadores.

### **7. Competencia**

Es la experiencia causada por situaciones en que uno o más jugadores deben enfrentarse a otros para ganar o superar al otro. Es vital en videojuegos competitivos masivos en línea.

### **8. Expresión**

La experiencia de expresión es aquella que sucede cuando un jugador es capaz de expresar su creatividad o genio dentro de un videojuego, permitiéndole crear cosas que antes no estaban o aportar en un proyecto ajeno.

### **9. Abnegación**

Esta experiencia se basa en la urgente necesidad de un jugador por volver a jugar o que ocurra algún evento. No se necesita que el jugador esté jugando activamente para que se experimente.

Un videojuego puede contener más de una experiencia de juego y según cada uno, puede jerarquizar y definir cuáles son más importantes.

Para el presente proyecto, se define la siguiente jerarquía de experiencias de juego:

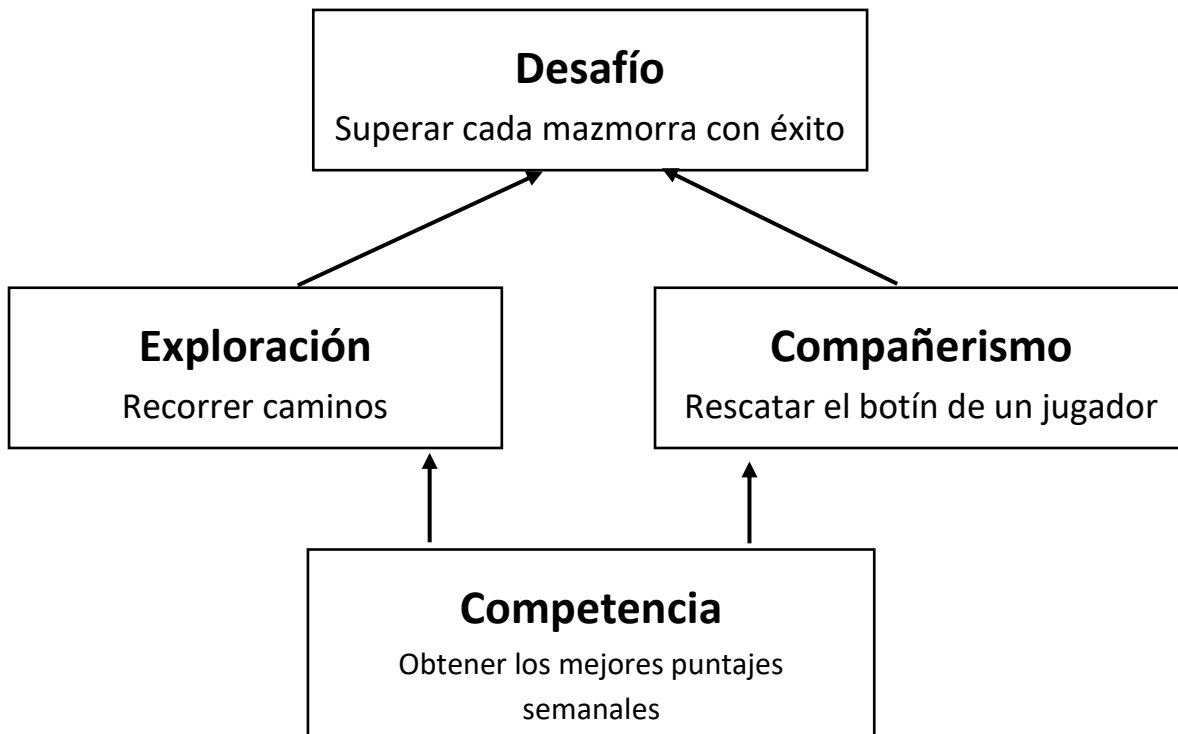


Figura 3. Jerarquía de Experiencias de Juego

### 2.1.2. Diseño de nivel medio

En esta etapa, que pertenece a la pre-producción, se define con más precisión cada punto, en el que cada elemento y dinámica son descritos para que el equipo desarrollador pueda planificar su trabajo y comenzar a producir.

La experiencia de juego no está bajo el control de los desarrolladores, ni siquiera del jugador de manera completamente voluntaria, ya que surge de las condiciones en que se juega y como se desenvuelven los eventos del videojuego, sin embargo, es posible inducir ciertos eventos y reglas que podrían producir un grupo de emociones deseables. Por ejemplo, un videojuego de terror contiene un nivel donde la música es muy tensa, hay poca iluminación y las mecánicas requieren de una máxima concentración, para que el jugador pueda disfrutar el juego es necesario que la inmersión sea muy alta, por lo que se adapta el contexto del juego a condiciones idóneas, siempre y cuando el jugador no tenga otros elementos distractores que no pueden ser controlados.

Hablamos de calidad de la experiencia cuando los elementos del videojuego se utilizan para generar experiencias en los jugadores al utilizarlos bajo ciertas condiciones. Un juego tiene buena calidad cuando un jugador, en condiciones estándar (sin elementos distractores), es capaz de experimentar la experiencia de juego deseada en la mayoría de los casos y en los momentos en que el desarrollador desee. De acuerdo con Williams, Yee y Caplan, los

jugadores que disfrutan de ciertos elementos de un videojuego, como la exploración o el roleplaying, no necesariamente encuentran que estos aspectos de los juegos son satisfactorios o convincentes [4]. Este efecto es causado por una mala calidad de experiencia.

Para diseñar la calidad de experiencia, se definen los siguientes puntos:

### Gameplay

Consiste en el conjunto de actividades que puede realizar el jugador y la respuesta de otras entidades dentro del contexto de juego ante estas o como cursos de acción autónomos.

### Contexto

El contexto del videojuego es el mundo en el que el jugador interactúa y percibe el gameplay.

### Playability

El concepto de playability se refiere a la facilidad con la que el videojuego permite entender y lograr los objetivos y reglas.

Para percibir estos conceptos, el jugador debe recibir cierta información, esta puede ser de dos tipos:

#### Información Funcional

Permite al jugador comprometerse con las actividades que tiene que realizar dentro del videojuego.

#### Información Estética

Permite al jugador sentirse inmerso en el contexto del videojuego, atrayendo y manteniendo su atención.

En la etapa de diseño de nivel medio, se vuelven a definir los elementos del videojuego (mecánicas, estética, sonido, narrativa, tecnología e interfaz) pero con un mayor detalle. En el presente documento se detalla esto en el apartado de Diseño de Videojuego.

### **2.1.3. Modelos de negocios**

Los videojuegos para dispositivos móviles no suelen utilizar un modelo de negocio de pago, sino que generan ingresos al monetizar aspectos dentro del mismo.

En general, los modelos de negocios han cambiado de vender elementos fijos con ventas únicas a productos basados en servicios, con commodities<sup>1</sup> virtuales, servicios de valor agregado y estrategias de anuncios [5].

Los principales modelos de negocio del mercado para videojuegos de dispositivos móviles son:

- **Monetización por anuncios:** Mediante este método, se recibe el pago de un servicio de publicidad que inserta anuncios en el videojuego. Estos anuncios pueden aparecer tanto de manera aleatoria o controladas por el jugador (recompensas por ver anuncios). Este método requiere una gran masa de usuarios o el diseño de sesiones largas de juego, ya que el cobro por anuncio visto es bastante bajo. Para distintos tipos de usuarios, es posible mostrar anuncios basados en datos demográficos, horarios, según el día de la semana y otros posibles parámetros. [6]
- **Compras dentro de la aplicación (IAP):** Este modelo permite a los jugadores comprar con dinero real objetos dentro del videojuego, como pueden ser divisas, ítems, vidas, etc. Este método requiere que el público esté dispuesto a pagar, por lo que el contenido debe ser tentador y mostrarse como exclusivo (aunque no lo sea), además se debe cuidar la mantención constante del sistema y actualizar periódicamente el contenido que se puede comprar. Es una estrategia arriesgada, considerando que requiere de una base de usuarios muy grande, ya que de acuerdo con [7], en promedio, solo el 0.5% de los jugadores realiza compras dentro de la aplicación y la mitad de las ganancias provienen del 10% de jugadores que hacen compras.
- **Aplicación pagada (pay-to-play):** Es uno de los modelos más clásicos. En este modelo el producto se adquiere mediante compra directa y única. Para optar a este método se debe considerar una gran campaña de marketing con el fin de alcanzar el mayor número de posibles consumidores y un Publisher se convierte en una figura clave para el éxito del producto. Podría ser una buena opción para títulos de grandes desarrolladores, pero si se toma en cuenta el tamaño del equipo desarrollador para el presente proyecto, se debe considerar otros modelos, que signifiquen una mejor posibilidad de monetizar el videojuego. [8]

#### 2.1.4. Público objetivo y perfil del jugador

Parte del éxito de un videojuego reside en que llegue a las manos de los jugadores correctos, es decir, que se elija un grupo objetivo de personas que sean potenciales consumidores del producto, ya sea por cuestiones de gusto personal (estética, mecánicas, género, temática, etc.) o adaptación al usuario (dificultad, lenguaje, plataforma, etc.). Es por ello por lo que

---

<sup>1</sup> Un commodity es un bien básico destinado a uso comercial, sin valor agregado o sin procesar.

durante todo el desarrollo se debe tener en cuenta quién lo jugará. Un error común es pensar en el público objetivo como si fuera uno mismo, dejando de lado a muchos jugadores que podrían tener una opinión y gustos diferentes a los propios.

Según [9] existen dos tipologías de distinción entre perfiles de jugadores, estas son la segmentación clásica y la que distingue entre jugadores ocasionales y “hardcore gamers”.

#### 2.1.4.1. Segmentación clásica

Esta toma en cuenta factores sociodemográficos, como la edad, el género o país del jugador. La complejidad reside en que a través de los años pueden cambiar los gustos de cada grupo, esto debido principalmente a la mutabilidad cultural y la constante globalización.

Para el 2020, hay un total de 214.4 millones de jugadores en Estados, teniendo como principal plataforma los teléfonos móviles [10]. Esto es una buena noticia considerando que en 2015 la cantidad de jugadores era de 155 millones [11].

Según el rango etario, distintos géneros de videojuegos pueden resultar más atractivos para cada grupo. Por ejemplo, los hombres de la generación millennial (entre 18 y 34 años) prefieren los juegos de Acción, Shooters y de Deportes, géneros que requieren de una respuesta más rápida del jugador y una gran concentración. En cambio, la generación Boomer (entre 55 y 64 años) prefiere juegos de cartas, puzzles y de mesa virtual, cuya principal característica es que no requieren de una gran agilidad, sino más bien de trabajo mental y formulación de estrategias.

En cuestiones de género, no existe una diferencia importante entre hombres y mujeres, ya que el porcentaje de jugadores es de 59% y 41% respectivamente.

#### 2.1.4.2. Distinción entre jugadores ocasionales y “hardcore gamers”

Dependiendo de la frecuencia e intensidad con la que juega una persona, se puede considerar como un jugador ocasional o como un hardcore gamer. La motivación para jugar suele ser distinta y también la plataforma en la que se juega. Esta distinción, sin embargo, es algo ambigua si no se subclasifican los tipos de jugadores. Para ello, consideraré los arquetipos de “entusiastas por los videojuegos”.

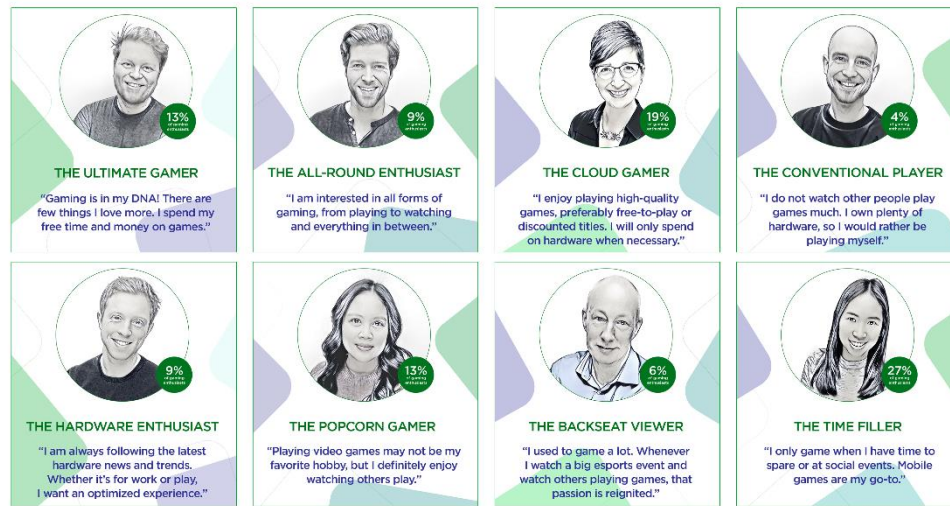
#### 2.1.4.3. Segmentación por arquetipo de persona

Según [12], existen 8 tipos de personas que se entusiasman por los videojuegos por diferentes motivos. Nótese que no solo se incluyen jugadores, sino que también se consideran a aquellos que disfrutan de los videojuegos por otros medios sin una interacción directa.



## THE NEW BREED OF GAME ENTHUSIASTS

AN OVERVIEW OF THE EIGHT PERSONAS THAT MAKE UP NEWZOO'S GAMER SEGMENTATION



© Copyright Newzoo 2019 | Newzoo's Gamer Segmentation | Source: Consumer Insights in 30 markets | [newzoo.com/consumer-insights](http://newzoo.com/consumer-insights)

Figura 4. Tipos de entusiastas por los videojuegos [12]

Dentro de los jugadores ocasionales o casuales, encontramos a los siguientes tipos de persona:

- El llenador de tiempo: Es quien juega durante sus tiempos libres, por lo que no suele jugar a títulos que requieran sesiones largas de juego y generalmente prefieren dispositivos móviles, debido a que permiten jugar donde y cuando quieran.

Los perfiles que se adaptan a los hardcore gamers son:

- El jugador definitivo: Es aquel que juega e invierte en videojuegos. Es su medio de entretenimiento principal.
- El jugador convencional: Es aquel que juega a menudo y que dispone de dispositivos para jugar a sus juegos preferidos.
- El entusiasta integral (all-around): Le gusta disfrutar los videojuegos tanto en su forma convencional como por otros medios.

Adicionalmente, se encuentran los tipos de persona que no juegan directamente a videojuegos:

- El jugador de popcorns: Se refiere a quienes prefieren ver a otras personas jugar, principalmente vía streaming.
- El espectador de asiento trasero: Es quien ya no juega o juega muy poco y ve eventos de videojuegos ya que le recuerdan a su época como jugador activo.

Existen jugadores que, independiente de si juegan mucho o no, tienen una alta fijación en la tecnología que usan, ya sea porque también la utilizan para otros fines o por una limitación técnica:

- Jugador en la nube: Prefiere jugar juegos de alta calidad y en lo posible gratuitos. No suelen invertir en hardware.
- El entusiasta del hardware: Al contrario que el anterior, este se preocupa mucho de los dispositivos en los que juega, ya que siempre desea la mejor experiencia de juego y suele usar el hardware tanto para jugar como para trabajar u otras actividades.

### 2.1.5. Publicación

Para hacer llegar el proyecto al público, es necesario que se publique en algún sitio accesible para la plataforma escogida, cumpliendo los requisitos que esta establezca. Adicionalmente se puede publicar contenido acerca del juego en otros sitios para llegar a más usuarios potenciales.

### 2.1.6. Mantenimiento

En esta etapa, en la que el videojuego ya está publicado y disponible para los jugadores, se procede a analizar los resultados y las opiniones de los usuarios.

Se realizan los cambios necesarios al videojuego para eliminar errores o mejorar la experiencia de juego.

Dependiendo del tipo de videojuego, se le puede agregar contenido de manera periódica para retener a los jugadores. Algunos ejemplos son la creación de nuevos niveles, modos de juego, ítems comprables dentro del juego, etc.

## 2.2. *Investigación sobre mecánicas cooperativas*

La adición de elementos cooperativos en títulos de videojuegos es una parte integral de muchos juegos y varios títulos, como *A Hat In Time* (Gears for Breakfast, 2017) o *Ghost of Tsushima* (Sucker Punch Productions, 2020) [13] han agregado un modo cooperativo después de su lanzamiento, siendo un componente que no era original del gameplay central. Se identifican distintos patrones de diseño de videojuegos cooperativos [14]:

- **Complementariedad:** Los jugadores tienen roles que se complementan el uno al otro.
- **Sinergias entre habilidades:** Permite a un tipo de personaje asistir o cambiar las habilidades de otro.
- **Habilidades que solo se pueden utilizar en el otro jugador:** Como la curación a los compañeros.

- **Objetivos compartidos:** Un grupo de jugadores deben cumplir con una única misión con un objetivo compartido.
- **Sinergias entre objetivos:** Los jugadores tienen objetivos diferentes pero que solo se pueden cumplir al jugar juntos.
- **Reglas especiales:** Los desarrolladores crean reglas para promover o facilitar la cooperación entre los equipos.

Todos estos patrones se pueden implementar en videojuegos con interacción síncrona sin mayor problema, ya que es cómo funciona la mayoría de los videojuegos multijugador y ya han sido probados en muchos títulos. Sin embargo, al crear un videojuego en el que los jugadores no necesariamente están jugando al mismo tiempo, es necesario adaptar las mecánicas de cooperación para que los jugadores puedan beneficiarse aunque jamás lleguen a interactuar directamente.

Los patrones que tiene que ver con las habilidades o estadísticas de los personajes, como la **complementariedad, sinergias de habilidades o habilidades que solo se pueden utilizar en otro jugador** sufren un cambio mayor, ya que necesitan que los personajes puedan cumplir los objetivos de la partida sin ayuda y que esta solo sirva como facilitador.

### 2.2.1. Interacción síncrona y asíncrona

Los videojuegos multijugador en tiempo real tienen algunos obstáculos para ser disfrutados en su completitud por usuarios que no tengan el tiempo suficiente para sesiones largas o muy demandantes. Además, en algunos casos es necesarios coordinar con otros jugadores para poder iniciar partidas o se necesita una conexión constante y estable para llevar a cabo una sesión de juego completa.

Una definición de juego síncrono es la siguiente:

*“Si la obtención de una victoria como parte de un bucle depende de las acciones de más de un jugador, ese bucle es síncrono. De lo contrario, es asíncrono” [15]*

En cambio, la interacción asíncrona permite que jugadores que no tengan la facilidad para jugar continuamente puedan disfrutar de un videojuego multijugador sin tener que darle dedicación total y pudiendo entrar y salir del juego en el momento que lo necesiten.

Para que esto funcione, es necesario que se apliquen ciertos requerimientos en el diseño del videojuego:

- No puede estar basado en turnos, ya que los jugadores pasarían la mayoría del tiempo esperando que el otro ejecute un movimiento, lo cual no es para nada divertido [16].



- El jugador debe poder entrar y salir de la partida en cualquier momento sin afectar negativamente la experiencia de los otros jugadores.
- La condición para terminar una partida no debe depender de las acciones de más de un jugador.

### 2.2.2. Juegos de suma no nula

En la teoría de juegos, se denomina juego de suma cero o nula a aquel en el que los participantes ganan utilidades equivalentes a lo que los otros pierden, por lo que al sumar las pérdidas y ganancias el resultado siempre es cero [17] [18].

Los juegos de suma no nula o no cero son lo contrario a los de suma cero, ya que los participantes pueden recibir beneficios o pérdidas en valores iguales o diferentes.

Un ejemplo clásico de problema de suma no nula es el “dilema del prisionero”, esta trata de dos prisioneros cuya condena depende de la confesión de cada uno tanto por separado como en conjunto.

Como no hay pruebas suficientes para condenarlos, se le da la opción a cada prisionero de confesar. En caso de hacerlo, sale libre solamente si el otro prisionero niega el crimen. Si ambos confiesan, la condena solo es de 6 años, mientras que si ambos lo niegan, como hay falta de pruebas la condena solo es de 1 año [19] [20].

	Tu confiesas	Tú lo niegas
El otro confiesa	Ambos son condenados a 6 años.	Eres condenado a 10 años y el otro es libre.
El otro lo niega	Eres libre y el otro es condenado a 10 años.	Ambos son condenados a 1 año.

*Figura 5. Dilema del prisionero.*

En este dilema, es posible maximizar el beneficio dependiendo de si ambos cooperan o prefieren jugar individualmente, sin embargo, ambos obtienen un mejor resultado si prefieren colaborar [20], aquí también pueden influir factores psicológicos, ya que en la realidad, los prisioneros podrían preferir no confesar a causa de las normas sociales o el miedo a las represalias [21].

Al aplicarlo a los videojuegos, se puede crear un sistema similar, en el que para obtener una ganancia, ambos jugadores, sin poder comunicarse, lleguen a un acuerdo para maximizar sus ganancias y con pocas probabilidades de resultar perjudicados.

### **2.2.3. Cooperación y repartición de recompensas**

Si el videojuego posee una mecánica de cooperación en la que los jugadores interactúan de manera asíncrona, no es posible generar una comunicación continua entre los jugadores durante una partida, ya que no se asegura que esta dependa de la presencia de los participantes.

Los jugadores pueden interactuar mediante los elementos del juego a través de una mecánica que les permita intercambiar objetos o ayudar al otro en caso de que haya perdido o necesite recuperar algún ítem perdido. En este último caso, se propone crear un sistema en el que los jugadores puedan salir beneficiados, obteniendo una recompensa o recuperando ítems; o perjudicados, no logrando obtener ninguna recompensa por ayudar al otro o perdiendo sus ítems permanentemente.

Este sistema, que utiliza un botín de ítems que se reparte entre los participantes puede utilizar un porcentaje de repartición decidido por los jugadores tanto de manera individual como grupal

Se realiza el siguiente cuestionamiento: ¿existe alguna diferencia dependiendo de quién decide el porcentaje de repartición? ¿preferiría un jugador no ofrecer recompensa al saber que el otro jugador podría quedarse con una porción muy grande es esta? ¿influye la cantidad de la recompensa en la decisión de ayudar al otro jugador?

### 3. Estudio de viabilidad

#### 3.1. Recursos técnicos

##### 3.1.1. Hardware

Debido a la elección de tecnologías a utilizar y/o desarrollar, se especifican los siguientes requerimientos mínimos de hardware para desarrollar el videojuego:

Tipo de dispositivo	Requisitos	Uso
Ordenador de sobremesa o portátil	<ul style="list-style-type: none"> <li>• Sistema Operativo Windows 7 o 10.</li> <li>• Tarjeta gráfica Nvidia GeForce GTX 1050 o equivalente compatible con DX10, DX11 y DX12.</li> <li>• 8GB de RAM.</li> </ul>	Desarrollo del videojuego. Programación. Creación de música y gráficos.
Dispositivo móvil	Sistema Operativo Android 4.4 o superior.	Ejecución del videojuego.

Figura 6. Requisitos mínimos de hardware para desarrollar el videojuego.

Para la realización de este proyecto se utilizarán los siguientes dispositivos:

Tipo de dispositivo	Nombre y detalles	Uso
Ordenador portátil	Lenovo Y520 <ul style="list-style-type: none"> <li>• Procesador Intel® Core™ i7 hasta 7.a generación</li> <li>• Memoria DDR4 8GB</li> <li>• NVIDIA GTX-1050</li> </ul>	En este dispositivo se desarrollará el videojuego mediante el software instalado en él.
Móvil	Huawei P10 Lite	En este dispositivo se realizarán la mayoría de las pruebas del proyecto de videojuego, ya que es la plataforma principal.

Tableta digitalizadora	Huion NEW1060 Plus	Mediante esta tableta se realizarán ilustraciones tanto para elementos dentro del videojuego como fuera.
------------------------	--------------------	--

Figura 7. Hardware utilizado en el proyecto

### 3.1.2. Software

El desarrollo del proyecto requiere el uso del siguiente tipo de software:

Tipo de software	Ejemplos
Motor de videojuegos.	Unity 3D, Godot Engine, Unreal Engine, Game Maker.
IDE.	Visual Studio, Visual Studio Code, Rider, MonoDevelop.
Edición fotográfica e ilustración.	Photoshop, MediBang, Krita, Paint Tool Sai.
Digital Audio Workstation (DAW).	Cakewalk, Reaper, FL Studio, Ableton Live.
Ilustración Pixel Art.	Aseprite, Pixelorama, Piskel.
Plataforma de Back-End.	PlayFab, Firebase, GameSparks.

Figura 8. Requisitos mínimos de software para desarrollar el videojuego.

Para la realización de este proyecto se hará uso del siguiente software:

Tipo de software	Nombre y detalles	Uso
Motor de videojuego.	Unity 3D (2019.3).	Desarrollo y exportación de videojuego.
Entorno de desarrollo.	Visual Studio Code.	Programación del código del videojuego.
Edición fotográfica.	Photoshop CC 2017.	Edición de ilustraciones.
Ilustración.	Paint Tool Sai 2.	Creación de ilustraciones para elementos fuera del gameplay principal, como lo pueden ser los

		elementos de publicidad o que se encuentren en las fichas de presentación de las tiendas donde sea publicado.
Digital Audio Workstation (DAW).	FL Studio 12.	Grabación y producción de música y efectos de sonido del videojuego.
Ilustración Pixel Art.	Aseprite.	Creación de imágenes para elementos del gameplay principal.
Planificación de proyectos.	Trello.	Herramienta para organizar y planificar el proyecto desde su documentación hasta procesos de diseño y programación.
Procesador de texto.	Microsoft Word.	Redacción de este y otros documentos de texto enriquecido.
Plataforma de Back-End.	Azure Playfab.	Administración de bases de datos, analíticas, guardado de partidas, IAP, etc.

Figura 9. Software utilizado en el proyecto

### 3.2. Recursos humanos

Tanto el desarrollo del videojuego como la recopilación y análisis de resultados serán procesos en los cuales solo trabajará una persona, sin embargo, la división del trabajo sigue siendo necesaria en los siguientes perfiles:

- Programador: Será el encargado de desarrollar el código del videojuego e implementar las herramientas necesarias (librerías, plugins, APIs, etc.). Este rol puede ser subdividido en tareas de análisis, investigación, diseño, testeo, documentación de código e implementación.
- Documentador: Redacta este documento, detallando todo el proceso que se llevó a cabo para finalizar el proyecto.
- Diseñador de videojuegos: Es quien piensa y estudia todos los aspectos del juego para que el producto cumpla sus objetivos de desarrollo y experiencia de juego. Suele estar en contacto con todas las áreas de la línea de producción.

- Artista y músico: Este perfil se involucra en la producción de los aspectos estéticos y musicales del videojuego, como su apartado gráfico, animaciones, efectos de sonido, música, etc.

### 3.3. Costo económico

En esta sección se realizará una estimación del costo económico que significaría realizar este proyecto. Se ha de tener en cuenta que la mayoría de las herramientas ya estaban adquiridas antes de comenzar el proyecto y que solo será desarrollado por la misma persona que escribe este documento, por lo tanto, no existirá un sueldo real.

#### 3.3.1. Coste de software de pago

Nombre	Precio
Photoshop CC	290,17 € (Plan anual)
Paint Tool Sai 2	46,54€
FL Studio 12	189€
Aseprite	15€
Licencia de Desarrollador Google Play	US\$25

#### 3.3.2. Costo de hardware

Nombre	Vida útil	Precio amortizado
Lenovo Y520	6	233€
Huawei P10 Lite	5	75€
Huion NEW1060 Plus	10	49,5€

### 3.4. Estudio de mercado

#### 3.4.1. Elección de modelos de negocios

Al ser un desarrollador independiente, y al este tratarse de un proyecto con fines principalmente académicos, se buscará utilizar un modelo de negocio que no requiera acuerdos con publishers, ni un pago excesivo de licencias. Un requisito es que la monetización del videojuego se realice de manera pasiva, es decir, se genere como parte del ciclo de gameplay y no mediante la intervención del desarrollador.

Según la información recabada, el modelo de negocio escogido para este proyecto es el de **monetización mediante anuncios**, dado que si bien no se espera una gran masa de usuarios, se puede llegar a obtener ingresos en el largo plazo. Adicionalmente, al no incluir compras dentro de la aplicación, no se hace necesario agregar contenido que pueda ser comprado, lo que reduce el tamaño del videojuego y lo convierte en un proyecto más abarcable en un corto plazo.

#### 3.4.2. Competencia (estado del arte)

Ya que será necesario realizar pruebas en múltiples usuarios, es demandante lograr un buen posicionamiento dentro de las plataformas en que el videojuego será publicado.

Es necesario entonces destacar ejemplos de videojuegos con características similares que hayan tenido éxito y también aquellos que han fracasado por diversas causas.

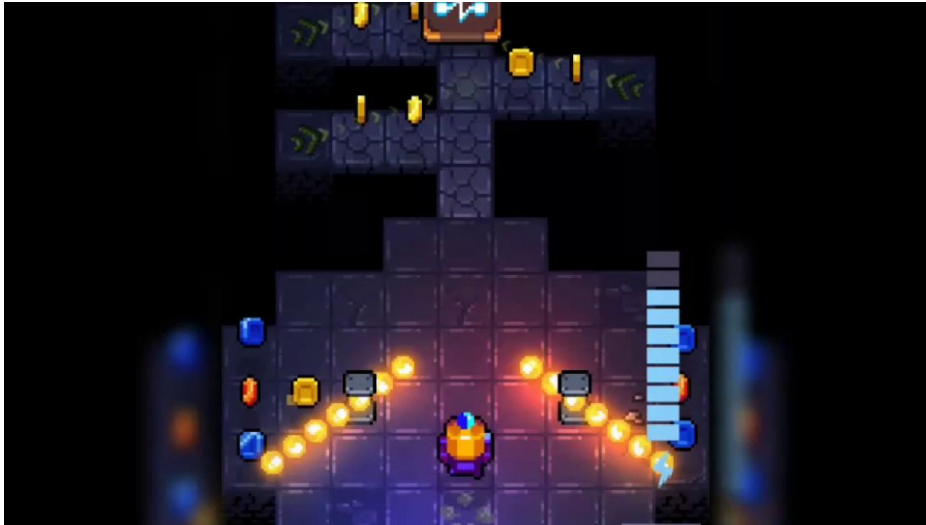
Los conceptos clave que se compararán con este videojuego serán:

Género/s	Roguelike,
Plataforma/s	Móvil, Web
Diseño de niveles	Procedural, Progresivo
Interacción entre jugadores	Indirecta
Tiempo promedio de sesión	Corto (5 minutos)

Figura 10. Conceptos clave para videojuegos similares

Utilizando motores de búsqueda combinando estas palabras clave, y seleccionando videojuegos a los que se podría agregar la mecánica de cooperatividad aleatoria, se obtuvieron los siguientes resultados destacables:

## ReDungeon (2017)



*Figura 11. Captura del videojuego ReDungeon*

Desarrollado por: Eneminds.

Publicado por: Nitrome.

Plataforma: Android

**ReDungeon** es un videojuego de género roguelike en el que el jugador debe avanzar en un mapa infinito compuesto por pequeños puzzles que se van añadiendo a medida que el personaje progresa verticalmente. Cada puzzle contiene trampas y enemigos que el jugador debe superar para conseguir llegar hasta el siguiente.

A nivel de mecánicas y arte, este es el principal referente para desarrollar Havoc In The Dungeon, sin embargo, este videojuego no cuenta con mecánicas de cooperación multijugador y difiere en la manera coleccionar objetos.



## Pokémon Mystery Dungeon (saga principal)



Figura 12. Captura de Pokémon Mystery Dungeon Red Team (GBA)

Desarrollado por: Spike Chunsoft.

Publicado por: Nintendo.

Plataformas: Game Boy Advance, Nintendo DS, Nintendo 3DS, Nintendo Switch.

**Pokémon Mystery Dungeon** (PMD) es una saga spin-off de Pokémon, cuyos primeros títulos fueron PMD: Red Team (GBA) y PMD: Blue Team (NDS). Su género es roguelike, RPG y aventura. En las fases de dungeons, el jugador recorre mapas generados proceduralmente derrotando a enemigos y recolectando objetos. Para superar cada nivel debe llegar a las escaleras que se encuentran dentro del mapa.

Este videojuego incluye la opción de pedir ayuda a otro jugador en caso de perder dentro de un nivel, el método es crear una clave de rescate en la que se ofrece una recompensa a quien acepte la misión.

## Pixel Dungeon



Figura 13. Captura de pantalla de Pixel Dungeon.

Desarrollado por: watabou.

Publicado por: watabou.

Plataformas: Android.

**Pixel Dungeon** es un videojuego para dispositivos móviles Android que entrega una experiencia de exploración en una mazmorra muy grande, consiste en un mapa que contiene diversos enemigos y objetos coleccionables. Se destaca por la inmensa cantidad de contenido y gran rejugarabilidad. Contiene un sistema de clases, evolución del personaje, medallas, rankings e inventario.

### 3.4.3. Cuadro comparativo

Utilizando la información sobre los videojuegos de referencia, es posible compararlos con Havoc In The Dungeon.

	Havoc In The Dungeon	ReDungeon	Pokémon Mystery Dungeon	Pixel Dungeon
Plataforma/s	Android, Web	Android	Nintendo (portátiles)	Android, Windows, macOS
Estilo Gráfico	Pixel Art	Pixel Art	Pixel Art	Pixel Art
Generación de niveles	Progresivo	Progresivo	Procedural	Procedural
Objetivo	Abrir habitaciones infinitamente	Avanzar infinitamente	Llegar al piso final	Mejorar al personaje y conseguir objetos
Objetos	Monedas y consumibles	Monedas	Consumibles	Monedas, consumibles, coleccionables y armas.
Obstáculos	Enemigos y trampas	Enemigos y trampas	Enemigos y trampas	Enemigos
Modelo de negocios	Anuncios	Anuncios e IAP	Pay-to-Play	IAP

Figura 14. Tabla comparativa de referencias de videojuegos.

### 3.5. Elección del perfil para el proyecto

Dado que Havoc In The Dungeon es un juego en el que pretendemos que las sesiones sean cortas y que se jueguen la mayor cantidad de partidas por sesión (y así aumentar las interacciones con el sistema de misiones de rescate), el tipo de persona objetivo será el **llenador de tiempo**. Considerando la naturaleza asíncrona del sistema de misiones planteado, esto permitiría que el rescate del jugador suceda aunque este no esté jugando en ese momento y pueda ver los resultados cuando vuelva a iniciar una sesión de juego.

### 3.6. Cuadro de autovaloración

En el siguiente cuadro se indica el peso asignado a 4 componentes del desarrollo de videojuegos en el presente proyecto.

Estética	25%
Narrativa	5%
Mecánicas	35%
Tecnología	35%

*Figura 15. Cuadro de autovaloración.*

La narrativa es un elemento que tendrá poca importancia en el videojuego, por lo que no se dará demasiado énfasis en ese componente.

El diseño y desarrollo del videojuego tendrá como componente principales las mecánicas y la tecnología. La estética, si bien es importante, tiene menor peso ya que no está en el campo de experticia del desarrollador.

### 3.7. Conclusión de viabilidad

Habiendo escogido el modelo de negocios y público objetivo se observa que existe un nicho de potenciales jugadores que podrían resultar interesados en el videojuego y descargarlo. Al utilizar el modelo de monetización por anuncios, se abre el espectro de jugadoras para aquellos que no podrían pagar por el producto.

Dadas las herramientas para desarrollar el videojuego, es posible terminarlo en un mediano plazo siempre y cuando el diseño de este sea correctamente definido desde el principio y se eviten cambios drásticos.

## 4. Metodología y plan de trabajo

En esta sección se describirán los procesos por los que se debe pasar para desarrollar tanto el videojuego como el análisis de resultados. Se describirá el plan de trabajo y el sistema de planificación de tareas y como se realiza la estimación de tiempo para ellas.

Usualmente, al desarrollar un videojuego dentro de un equipo de trabajo, se acuerdan estrategias para coordinar las tareas de cada miembro, que tipo de dependencia tienen, y cómo se comunicará el progreso de cada una de ellas.

En el caso de Havoc In The Dungeon, al ser un videojuego desarrollado por un único individuo, se hace omisión de prácticas como las reuniones de equipo, presentación de avances, validación de tareas, etc. Sin embargo, sigue siendo necesario contar con un sistema de planificación que permita organizarme y tener siempre en consideración lo que falta por hacer, las tareas que quedaron pendientes o anotaciones para tener en cuenta al momento de iniciar cada nueva tarea.

Generalmente, el desarrollo de un videojuego se divide en tres procesos: pre-producción, producción y post-producción. Cada una de estas etapas pasa por múltiples iteraciones antes de aprobarse, esto ya que al ser implementadas o prototipadas resulta que no cumplen con el objetivo inicial o no es compatible con el resto del juego.

A razón de que el presente proyecto se trata de un desarrollo a corto plazo, cada proceso debe realizarse con la menor cantidad de iteraciones posibles, por lo que he de estar muy seguro de cada decisión tomada sobre el videojuego, dejando para el final de la producción toda tarea de pulido que no influya de manera drástica en el producto. Para ello, se debe establecer una tabla o lista de requisitos mínimos sobre las funcionalidades del juego y el alcance que tendrán. Es importante mantener las expectativas a raya para no seguir un desarrollo muy ambicioso que no se pueda completar a tiempo.

### 4.1. *Planificación*

Esta fase sirve para planificar la metodología de trabajo tomando en cuenta el marco teórico y las decisiones de diseño iniciales, con el fin de ser óptimo en cuanto a uso de recursos, calidad del videojuego y tiempo necesario para el desarrollo. Para este paso se necesita realizar la asignación de las tareas, estimar cuánto tiempo se requerirá para cada una, establecer fechas clave (hitos) y tomar en cuenta en qué puntos ya no habrá retorno, es decir, no se harán cambios respecto a cierto elemento.

Como en todo proyecto, siempre pueden ocurrir imprevistos que retrasen u obstaculicen el desarrollo. Para ser eficiente, es necesario realizar la trazabilidad de las dependencias entre

tareas. Un buen método es el diagrama de Gantt, el cual es un instrumento o herramienta gráfica que ayuda a organizar las tareas y exponer su duración, dependencia y asignación a miembros del equipo.

#### ***4.2. Producción y pruebas***

Teniendo la documentación previa sobre diseño de nivel medio y siguiendo la planificación de tareas, se comienza la etapa de producción, en la que se desarrollan los elementos del videojuego. Usualmente se realizan reuniones periódicas entre los distintos departamentos de desarrollo para coordinar el resto del proyecto y comunicar los avances y problemas que se hayan tenido, sin embargo, al tratarse de un proyecto individual, se omite este paso.

Llegando al final del desarrollo, se ejecuta una serie de pruebas sobre el proyecto completo para cerciorarse de que no existan errores que interrumpan la ejecución del videojuego ni que interfieran con el gameplay. Además, se puede pulir algún elemento basado en la opinión del público tester.

Se debe planificar teniendo en cuenta que nunca se seguirá el calendario de tareas de manera exacta, esto porque tareas a las que se les asignó demasiado tiempo resultan ser más cortas de lo esperado, o el caso contrario en que una tarea que parece simple resulta ocupar un plazo mayor.

#### ***4.3. Metodologías de desarrollo ágil***

En un proyecto de desarrollo de software se refiere a una gama de enfoques de desarrollo ligero adaptados a:

- a) Acelerar el time-to-market <sup>2</sup> y la integración continua de los nuevos requisitos.
- b) Aumentar la productividad del desarrollo a la vez que se mantiene la calidad y flexibilidad del software.
- c) Agilizar la capacidad de respuesta a la vez que se disminuyen los gastos generales de desarrollo. [22]

El desarrollo ágil permite que el proceso sea iterativo e incremental, lo que influye en cómo se organizan las tareas y como se comunica el equipo.

---

<sup>2</sup> Time-to-market se refiere al tiempo necesario para que un producto esté disponible para ser lanzado al mercado.

Es necesario contextualizar los métodos de desarrollo al contexto del proyecto, esto ya que los métodos ágiles pueden fallar de varias maneras cuando se aplican “fuera de la caja”, es decir, con poca o ninguna adaptación al problema. [23]

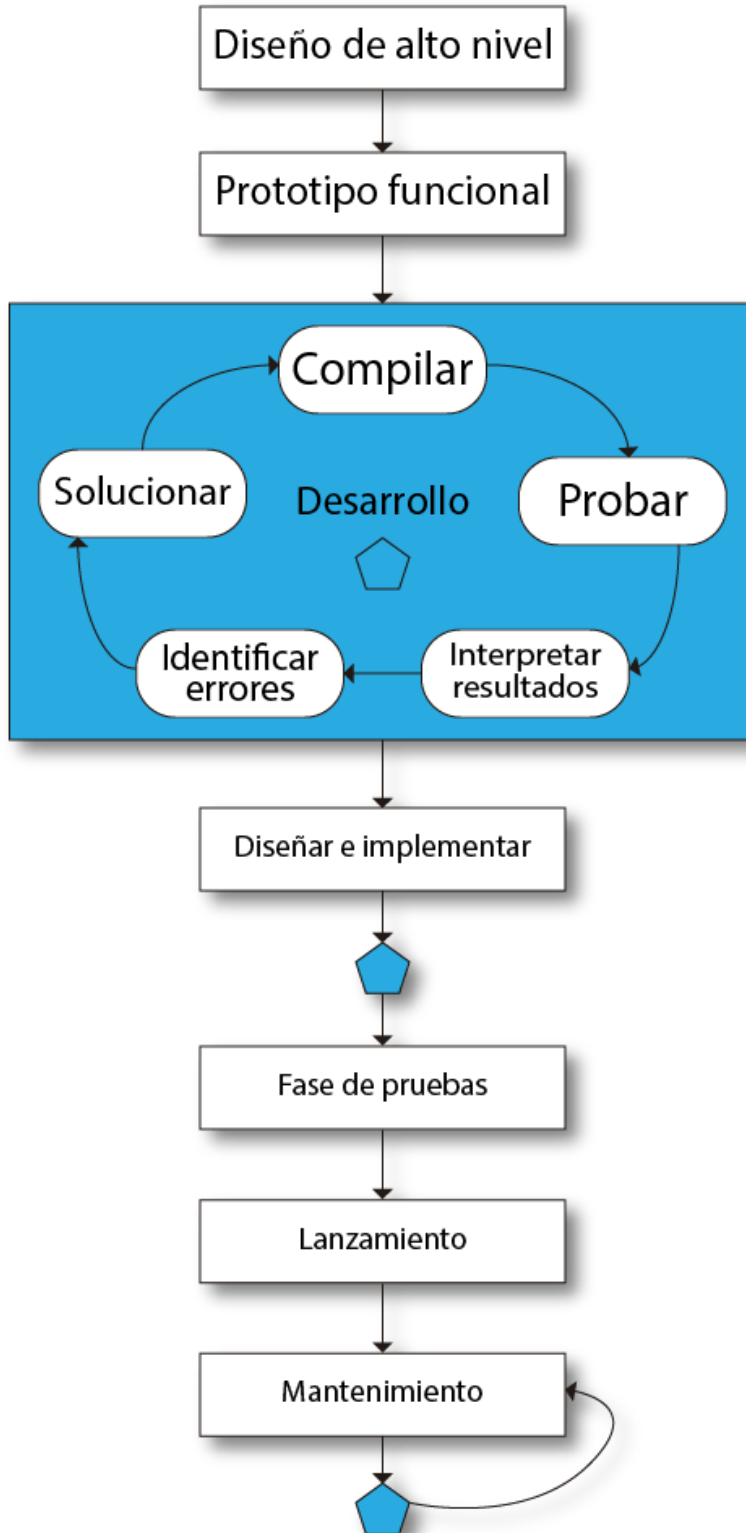


Figura 16. Modelo iterativo de desarrollo, adaptación del modelo de Will Luton [24]



#### 4.4. *Abasto del proyecto*

Para optimizar la planificación, se requiere definir qué cosas aparecerán en el videojuego. Para ello, durante el diseño de alto nivel, es necesario documentar cómo será el desarrollo. Esto nos permite revisar en cualquier momento si se está cumpliendo el plan de trabajo o si es necesario algún cambio. En proyectos con equipos grandes, es crucial mantener actualizado cualquier documento acerca de la planificación o la estructura del juego ya que cada miembro podrá estar informado sobre qué debe hacer y cuáles son los límites del proyecto.

##### 4.4.1. **Game Design Document (GDD)**

Este documento es el que detalla los personajes, niveles, mecánicas, vistas, menús y, en pocas palabras, el juego [25]. El GDD es creado por el equipo de desarrollo, que comprende a los diseñadores, artistas, programadores y miembros de otras áreas. Gracias a este documento, los miembros del equipo pueden conocer de qué trata el juego y cómo su trabajo está relacionado con él.

Suele escribirse en la etapa de pre-producción y de ser aprobado se complementa y expande.

Dada la naturaleza iterativa del desarrollo de videojuegos, el Game Design Document cambia constantemente, esto gracias a las nuevas visiones sobre el juego de parte del equipo o por ajustes que se le deban hacer a causa de problemas técnicos o requerimientos especiales.

##### 4.4.2. **Tareas planificadas**

Teniendo en cuenta el abasto del proyecto y la metodología de trabajo, se puede recurrir a la creación de un sistema de planificación de tareas.

Una forma de hacerlo es mediante el establecimiento de hitos, es decir, de metas a corto plazo en las que se espera que el proyecto esté en un estado específico, independiente de qué tareas fueron necesarias para ello. Este método permite mantener una visión clara de lo que se desea conseguir y cómo avanzará el proyecto a través del tiempo.

Sin embargo, no es dicotómico el usar el sistema de hitos con la planificación de tareas, ya que ambos pueden complementarse.

Para este proyecto, se utilizó el software **Trello**, el cual es un programa de organización de proyectos que utiliza el sistema de tarjetas Kanban<sup>3</sup>, el que permite visualizar el trabajo de manera gráfica y ordenar las tareas fácilmente.

---

<sup>3</sup> “Kanban” es una palabra japonesa que significa “señal visual”, el uso de tarjetas Kanban permite realizar un seguimiento de un elemento mientras circula por un flujo de trabajo.

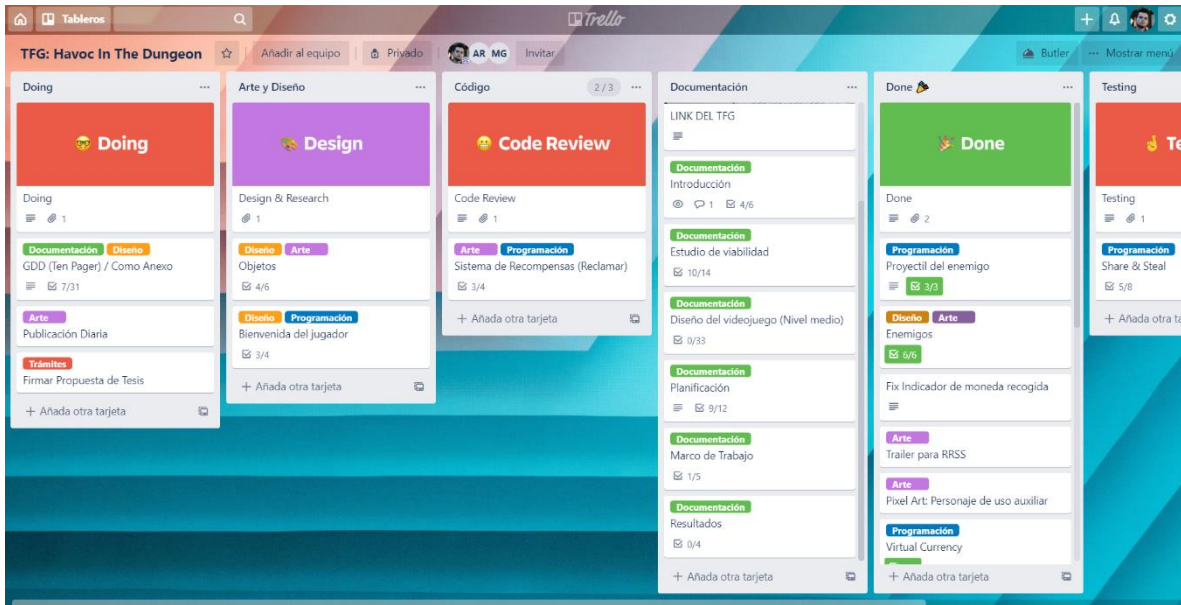


Figura 17. Captura de Trello en un momento del desarrollo.

Cada tarea se asigna a una lista de tarjetas, que puede ser de los siguientes tipos:

- a) Backlog: Pertenecen a un área de desarrollo (Diseño, Código, Arte, etc.), no se están realizando en este momento.
- b) Doing: Tareas que se están realizando en el momento.
- c) Testing: Tareas que se realizaron pero aún deben ser revisadas ya que podrían generar errores o necesitar refinamiento.
- d) Done: Tareas completadas, estas no vuelven a testing ni a backlog. Cada cierto tiempo, son eliminadas del tablero para ahorrar espacio visual.
- e) Documentación: Enumera la lista de elementos del presente documento. No se asigna a Doing ya que se desarrolla durante todo el proyecto.

A continuación, se enumeran tareas que son necesarias para el desarrollo del proyecto. Nótese que estas no se traducen de manera literal al tablero de Trello, ya que representan un proceso que puede ser subdividido en múltiples tareas. Además, solo se consideran tareas para el proceso de producción y post-producción.

Número	Tarea	Horas
1	Preparación del entorno de desarrollo en el motor. Instalar extensiones y ordenar carpetas.	5
2	Desarrollo del personaje principal. Movimiento, físicas y eventos.	8

3	Creación de herramienta para crear niveles.	5
4	Creación de herramientas del editor.	7
5	Desarrollo del GameManager <sup>4</sup> . Reglas del juego, eventos, referencias, etc.	30
6	Preparación del entorno Backend en PlayFab y vinculación con motor.	20
7	Sistema de guardado de datos locales y en la nube.	8
8	Desarrollo de la Interfaz de Usuario.	6
9	Desarrollo de enemigo u obstáculo (7).	21
10	Inserción de Anuncios dentro del juego.	3
11	Creación del sistema de llaves y puertas.	3
12	Desarrollo del sistema de misiones de rescate.	12
13	Desarrollo de menú principal.	8
14	Arte y animaciones de personajes y obstáculos (7).	14
15	Arte para nivel y decoraciones (10).	10
16	Música para el gameplay.	5
17	Generación de efectos de sonido (4).	5
18	Búsqueda e implementación de efectos de sonido adicionales gratuitos (4).	5
19	Creación de artworks para marketing (4).	8
20	Implementación de herramientas de análisis.	5

---

<sup>4</sup> El Game Manager es el componente de código donde se centralizan los elementos comunes del juego, como las reglas, puntaje, progreso y funcionalidades como guardar, perder, ganar, etc.

21	Prueba general previa al lanzamiento.	10
22	Generar recursos para Play Store.	5
23	Creación de música para el tráiler.	8
24	Publicar en Play Store (Ficha, clasificación, políticas, etc.).	6
25	Crear ficha en Itch.io.	3
26	Crear cuenta en Redes Sociales.	1
27	Publicación de lanzamiento.	1
28	Edición de tráiler para publicidad del juego.	6
	<b>TOTAL</b>	<b>228</b>

*Figura 18. Tabla de tareas del proyecto.*

En la tabla anterior, se consideran las tareas planificadas al principio, por lo que es posible que se realicen modificaciones en cuanto al contenido de las tareas como a las horas estimadas de trabajo.

## 5. Marco de trabajo

En esta sección se describe cómo se organizarán y utilizarán las herramientas escogidas en la metodología, además de explicar su funcionamiento. Adicionalmente, se contextualiza sobre las referencias y reglas a utilizar.

### 5.1. Referencia

Para desarrollar un videojuego nuevo, se puede partir desde varias bases. Puede surgir como una idea original de parte de una o más personas, ser una adaptación muy similar de un videojuego ya existente, o una reinención de sus mecánicas. En el caso de Havoc In The Dungeon, se siguió el último método, ya que el gameplay y arte son similares a un videojuego ya existente, sin embargo se añadieron nuevas mecánicas y reglas de juego, las cuales le darán su identidad respecto a otros juegos en el mercado y su referencia principal.

A continuación, se explica el funcionamiento de una partida de *ReDungeon*, el juego de referencia.

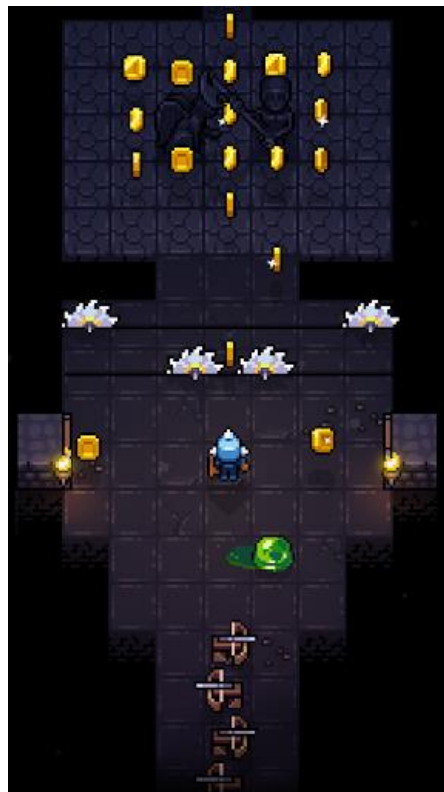


Figura 19. Captura de pantalla de una partida de *ReDungeon*

Es un videojuego de un jugador para dispositivos móviles, en el que el jugador debe avanzar verticalmente por mazmorras que contienen trampas, enemigos y monedas. Cada mazmorra es un nivel que se añade a un mapa que se genera de manera infinita. Cabe decir que no se

generan proceduralmente, sino que cada mazmorra es creada por el equipo desarrollador, ya que necesitan diseñarse de acuerdo con la dificultad del juego y deben ser superables por el jugador.

Si el personaje toca un enemigo o elemento dañino, como flechas, fuego, púas, etc.) o cae al vacío la partida se acaba y el jugador podrá comenzar una nueva o, si está disponible, ver un anuncio para continuar la partida actual. Las monedas coleccionadas durante la partida se guardarán en el inventario del jugador.



Figura 20. Captura de algunos personajes jugables disponibles en ReDungeon.

Las monedas coleccionadas se utilizan para comprar o mejorar personajes dentro de la tienda del videojuego. Cada personaje posee una habilidad única que le da una ventaja dentro de cada partida. Algunas de estas habilidades son:

- Coleccionar monedas a la distancia.
- Ser inmune al fuego.
- Poseer un escudo de uso limitado.
- Saltar las baldosas vacías.

El videojuego desarrollado en el presente proyecto tomará como referencia el gameplay dentro de la partida, utilizando el sistema de enemigos y trampas. Para coleccionar las monedas, el jugador deberá tocar un punto de guardado dentro de una mazmorra que aparece cada cierta distancia. Este factor implica que el jugador puede perder el botín de monedas que haya recolectado en una partida si pierde antes de guardar.

Para que el jugador tenga la oportunidad de recuperar el botín perdido, se implementará el sistema de **misiones de rescate**, este consiste en que un jugador puede solicitar la ayuda de otro para que recupere sus monedas a cambio de una repartición de estas.

El jugador que acepte la misión deberá recorrer una cierta cantidad de distancia dentro del nivel. Esta distancia se ajusta según el porcentaje de repartición de recompensa y cantidad de monedas totales.

El uso de monedas quedará pendiente y se implementará después de la fecha de lanzamiento, ya que requiere la generación de mucho contenido (personajes u objetos) y no habría espacio temporal para realizarlo dentro del tiempo de desarrollo previsto.

## 5.2. Motor de videojuego

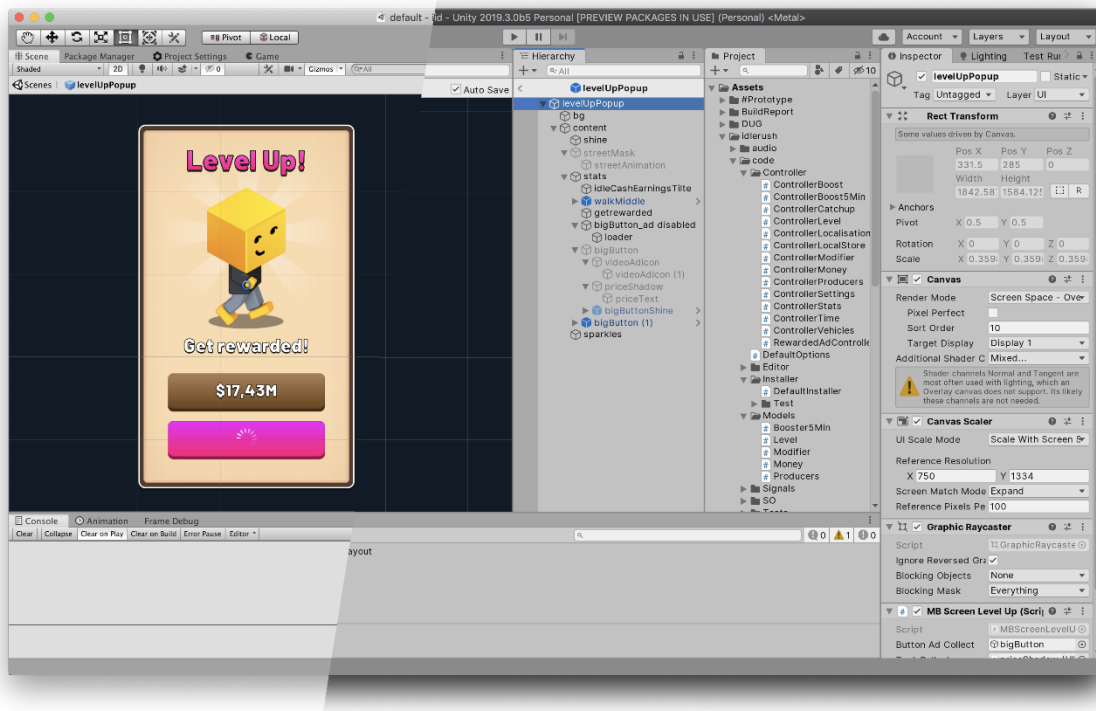


Figura 21. Captura del motor Unity 3D versión 2019.3

Para el desarrollo completo de Havoc In The Dungeon, se utilizó el motor de videojuegos Unity 3D, en su versión 2019.3, que era la versión estable más actual al momento de comenzar.

La elección de este motor sobre otros es debida a los siguientes motivos:

- Junto con Unreal Engine, es el motor más utilizado a escala global por los desarrolladores de videojuegos, sobre todo en el caso del desarrollo indie. Las empresas grandes suelen tener sus motores privativos que no son revelados al público.
- Es versátil, permite desarrollar todo tipo de videojuegos, tanto 2D como 3D e incluso en Realidad Virtual. Con él se pueden crear videojuegos de muchos géneros.

- Permite exportar a múltiples plataformas, incluyendo PC, Mac, Web, Android, iOS, Xbox, etc.
- Cuenta con una licencia personal gratuita que es bastante completa, tiene algunas limitaciones como la aparición de su logo en el splashscreen<sup>5</sup> del juego, lo que no es un impedimento muy grande para desarrollar.
- Permite crear herramientas dentro del mismo motor, las cuales ayudan a los desarrolladores a agilizar distintos procesos y automatizar algunas funciones.
- Es el motor en el que más experiencia previa tengo y he utilizado para crear la mayoría de mis videojuegos anteriores.
- Usa el lenguaje de programación C#, en el que tengo mucha experiencia.
- Cuenta con una comunidad inmensa y muy activa. Esto proporciona una gran facilidad para encontrar ayuda en caso de que se tenga alguna duda o problema con el motor, ya que es muy probable que alguien ya haya preguntado algo similar en sus foros o existan tutoriales que abarquen el tema.
- La Asset Store tiene una gran cantidad de contenido gratuito que puede ayudar a agilizar u optimizar los videojuegos desarrollados en el motor.

Es prudente aclarar algunos conceptos básicos que se utilizan constantemente dentro de Unity y que pueden ser diferentes a otros motores de videojuegos.

### **5.2.1. Game Object**

Un GameObject es un objeto que forma parte de una escena. Por sí mismo no tiene ninguna funcionalidad, para ello se le pueden agregar componentes.

---

<sup>5</sup> Un splashscreen es la imagen o recurso gráfico que se muestra al inicializar una aplicación en algún dispositivo.



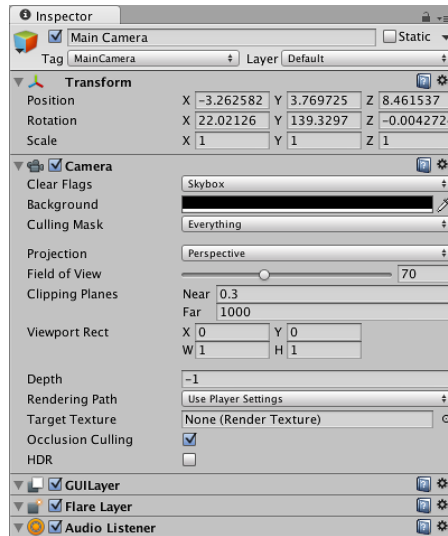


Figura 22. Detalles de un GameObject en el inspector.

## 5.2.2. Componentes

Representan un script que se adjunta a un GameObject, estos proporcionan las funcionalidades a cada objeto. Se pueden crear scripts propios, esos heredarán de la clase MonoBehaviour.

## 5.2.3. Assets

Son los recursos que se usan para los elementos del juego. Estos pueden ser archivos de audio, imágenes, texto, scripts, etc.

## 5.2.4. Prefabs

Representan un GameObject que ha sido guardado dentro de la carpeta de Assets. Estos pueden instanciarse en una escena y mantener sus propiedades.

## 5.3. Servicios Externos

En este apartado se describirán los servicios externos al motor de videojuego que servirán para distintos procesos del videojuego que requieren acceder o modificar datos de un servidor.

### 5.3.1. PlayFab

Azure PlayFab es un conjunto de servicios para operar y analizar videojuegos en tiempo real desde un panel de administración. Es utilizado por videojuegos como *Rainbow Six Siege* (Ubisoft) y *Minecraft* (Mojang). Puede utilizarse en diversas plataformas y tecnologías, entre

ellas, Unity. Al integrarse en un videojuego se pueden habilitar diversas funciones, las que se utilizaron para Havoc In The Dungeon son las siguientes.

#### 5.3.1.1. Identidad y datos multiplataforma

Este servicio permite que un jugador conserve sus datos independientemente de la plataforma en la que esté jugando. Esto gracias a los servidores que proporciona y la base de datos optimizada para almacenar y operar los datos de los jugadores.

#### 5.3.1.2. Estadísticas y Leaderboards

Se puede llevar un registro de estadísticas de los jugadores y generar una lista con estos datos en un orden determinado por el desarrollador. Se pueden automatizar procesos para realizar acciones sobre los jugadores que obtengan ciertas posiciones.

#### 5.3.1.3. Gestión de contenido

Es posible agregar y actualizar contenido del videojuego de manera remota gracias a este servicio.

#### 5.3.1.4. Automatización

El servicio de automatización permite configurar operaciones personalizadas y reaccionar a eventos en tiempo real. Gracias a la tecnología CloudScript es posible escribir código Javascript que se comunicará con el videojuego mediante un sistema de llamada y respuesta asíncrona.

#### 5.3.1.5. Experimentos

Los experimentos son un sistema que separa a grupos de usuarios para probar cambios en el videojuego y compararlos entre los grupos. Se utilizará para diferenciar los tipos de misiones que utilizarán los jugadores.

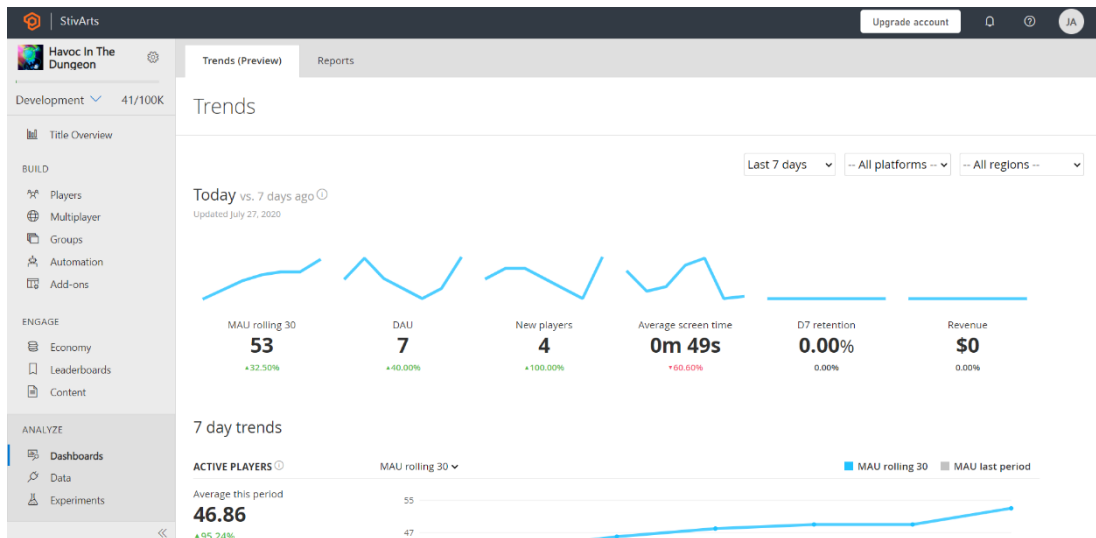


Figura 23. Captura del panel de tendencias en PlayFab.

### 5.3.2. Unity Services

El motor de videojuegos Unity 3D permite a sus usuarios acceder a ciertos servicios en la nube para agregar funcionalidades al proyecto para gestionar, atraer y retener usuarios, monetización y analizar datos.

De todos estos servicios, se utilizaron los siguientes.

#### 5.3.2.1. Unity Collab

Este servicio permite mantener un control de versiones del proyecto en un servidor. Gracias a esto se pueden guardar los cambios realizados en la nube y compartir el trabajo con el equipo de desarrollo. Internamente utiliza Git.

#### 5.3.2.2. Unity Ads

Unity permite integrar un sistema de monetización mediante anuncios dentro del videojuego. Es muy fácil de utilizar, lo que ayuda a agilizar su implementación.

#### 5.3.2.3. Unity Analytics

Mediante Analytics, es posible registrar los eventos del juego y mantener un registro de estadísticas sobre el proyecto, como los usuarios activos, tiempo de juego y eventos personalizados. Estos últimos se utilizarán para la parte de análisis de este trabajo.

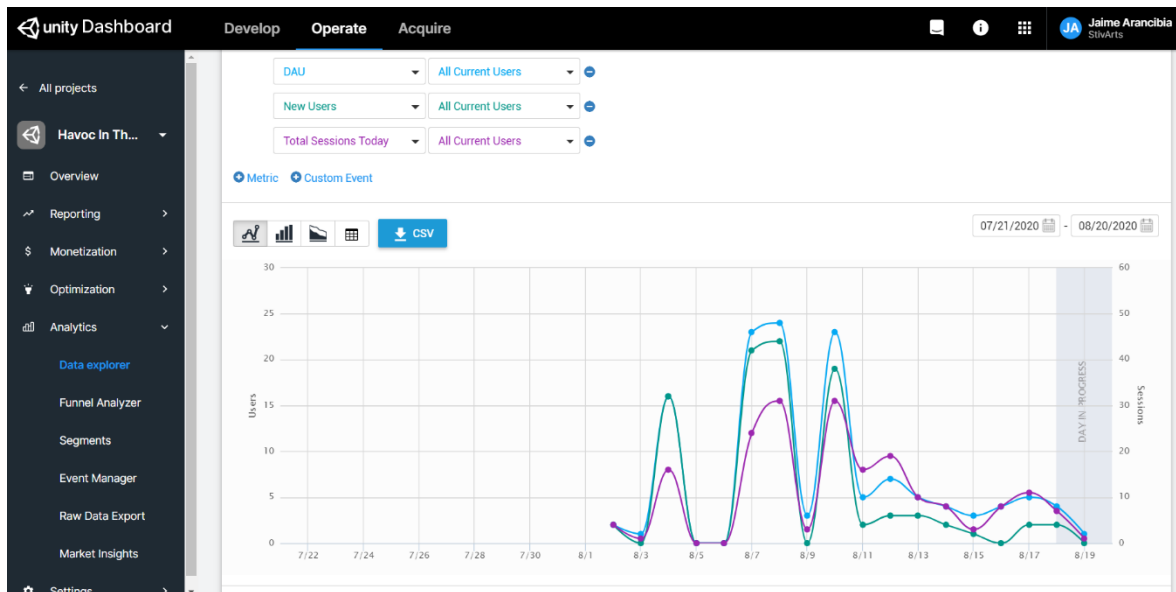


Figura 24. Captura del panel de administración de Unity.

### 5.3.3. DoTween (HoTween v2)

DoTween es un paquete descargable desde la Asset Store de Unity que permite crear interpolaciones fácilmente y a casi cualquier componente.

Gracias a este plugin se logra reducir el tiempo de desarrollo ya que en pocas líneas se crean funciones que normalmente tendrían una extensión mucho mayor.

Para el proyecto se utilizó en interpolaciones de movimiento, color, interfaz de usuario, texto, físicas, variables independientes, etc.



Figura 25. Imagen de presentación de DoTween en la Asset Store.

### 5.4. Clasificación de edad

Con el objetivo de regular el acceso al contenido por parte de público no apto, existen sistemas de clasificación de edad para videojuegos. Los más usados son ESRB y PEGI.

Sirven para advertir sobre el contenido que hay en el videojuego, como el grado de violencia, lenguaje utilizado, cuestiones sexuales, religiosas o culturales.

ESRB (USA)	PEGI (EU)	RARS (Russia)	ACB (Australia)	USK (Germany)
				
				
				
				
				
				

Figura 26. Clasificaciones de edad según países. Fuente: kaspersky.es

Un mismo juego puede tener diferentes clasificaciones según la zona y sistema, por ejemplo, el sistema PEGI tiene una clasificación para mayores de 12 años, en cambio ESRB tiene como más cercano el Everyone 10+, para mayores de 10 años.

## 6. Diseño del videojuego

A continuación se presenta el detalle del diseño de videojuego de nivel medio. Se omiten las secciones que hayan sido presentadas previamente, como el público objetivo, referencias y estudio de mercado.

### 6.1. Estética

#### 6.1.1. Estilo de arte



Figura 27. Distintos estilos de arte de videojuegos. Fuente: gamerpros.co

En el apartado de estética, se decide utilizar el estilo de arte pixel art, ya que representa un menor costo y puede desarrollarse y actualizarse de manera más rápida. Igualmente es necesario buscar las correctas referencias y ser metódico para obtener el mejor resultado posible.

#### 6.1.2. Iluminación y color

Se aprovechará el sistema de iluminación 2D para agregar luces puntuales y globales en la escena y objetos específicos. Cada color debe tener un significado para orientar al jugador.

El mundo del videojuego será oscuro, misterioso y peligroso. Para representar estos conceptos se recurre a la teoría de psicología del color.

Color	Significados	Aplicación
Rojo	Peligro, acción, energía, pasión.	Los elementos que sean peligrosos deben ser iluminados con una luz roja.

	Verde	Crecimiento, salud, generosidad.	Los elementos que tengan efecto positivo, como el punto de guardado, deben ser iluminados con este color.
	Azul	Calma, paz, confianza.	Se utilizará para elementos que no causen daño ni interactúen con el personaje.
	Morado	Espiritualidad, poder, frustración.	Este color se usará para ambientar la escena. No debe ser propio de ningún objeto específico excepto el escenario.
	Amarillo	Felicidad, positivismo, riqueza.	Se debe usar para elementos coleccionables que aumenten estadísticas del jugador.

### 6.1.3. Cámara

El videojuego se presentará en una perspectiva elevada o top-down, la cual es común en videojuegos de rol o de acción en 2D ya que permite ver los objetos del escenario en una misma escala sin que se interpongan entre ellos. La proyección será ortogonal, por lo que no existirá una deformación de los objetos según la distancia que haya con la cámara.

La cámara debe seguir al jugador, sin embargo debe hacerse de manera suave e implementando una zona en la que el jugador puede moverse sin que la cámara lo haga.

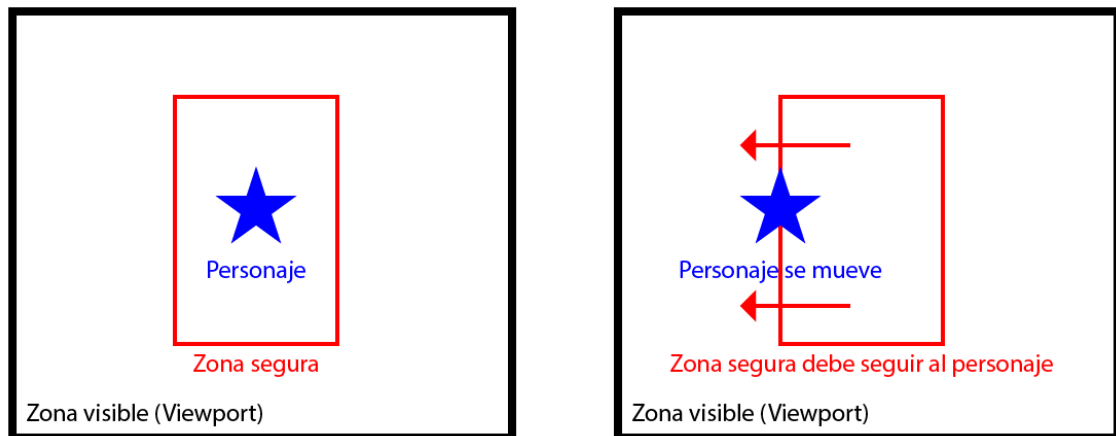


Figura 28. Diagrama del funcionamiento de la cámara.

Adicionalmente, la cámara debe tener un ligero movimiento continuo en forma de vibración, este ayuda a que si el jugador no se mueve, el nivel siga teniendo dinámica además de generar la sensación de que el mundo está vivo y en acción.

#### 6.1.4. Tilemap

Para dibujar el mapa, se utilizará un sistema de tiles<sup>6</sup>, por lo que es necesario crear un tileset con los elementos básicos del mapa. La mayoría de los objetos del mapa deberán tener el mismo tamaño o caber dentro de un tile, incluido el personaje.

La resolución de un tile es de 24x24 pixeles, la cual es adecuada para que se alcancen a distinguir los detalles sin ser demasiado costoso de producir.

#### 6.1.5. Animación

La animación de los personajes y objetos será mediante la técnica tradicional cuadro a cuadro. Ya que el estilo es pixel, es posible animar sin alterar demasiado el dibujo y con gran rapidez.



Figura 29. Animación de un enemigo del videojuego.

### 6.2. Interfaz

#### 6.2.1. Inputs

Para generar acciones dentro del juego se utilizarán botones en la interfaz gráfica. Ya que la plataforma de destino es dispositivos móviles, se debe considerar que la entrada es mediante pantalla touch y debe adaptarse a la resolución del dispositivo.

#### 6.2.2. Outputs

Solo de ser necesario, el dispositivo debe vibrar para alertar al jugador sobre algún evento del juego. La vibración no debe ser larga ni demasiado frecuente.

#### 6.2.3. Usabilidad

Para cumplir con un buen estándar en cuanto a usabilidad de interfaz gráfica, se recurre a los 10 principios de usabilidad de Jakob Nielsen [26] aplicados a los videojuegos.

---

<sup>6</sup> Los tiles consisten en construir el mundo del juego o el mapa de niveles a partir de imágenes pequeñas de forma regular llamadas tiles, los mapas de tiles (tilemaps) también se pueden asignar a una cuadrícula lógica, que se puede usar de otras formas dentro de la lógica del juego. [27]



### 6.2.3.1. Visibilidad del estado del sistema

El sistema siempre debería informar a los jugadores qué está ocurriendo con una buena retroalimentación.

Para ello, toda la información vital que tenga relación con el estado del juego o del jugador deberá ser visible durante la partida, en un lugar que no interfiera demasiado con el gameplay.



*Figura 30. Diagrama de posicionamiento de interfaz gráfica. El foco de la acción es donde la mayoría de los eventos ocurren dentro de la pantalla.*

### 6.2.3.2. Correspondencia entre el sistema y el mundo real

Se debe comunicar al jugador en su lenguaje, con conceptos y palabras que pueda entender y que le sean familiares.

Los objetos tendrán una forma similar a lo que representan en la vida real. Por ejemplo, los cofres deben tener los colores y elementos básicos de uno real a pesar de que puedan tener algunas diferencias menores.



Figura 31. Representación de un cofre en un sprite<sup>7</sup> del juego. Se mantienen los colores, forma y elementos como la cerradura.

### 6.2.3.3. Control de usuario y libertad

El jugador debe tener la opción de cancelar sus acciones en caso de que no hayan sido intencionadas o simplemente desee cambiarlas. Esta funcionalidad debe ser visible y descubrible por el usuario.

Para las funcionalidades del juego y distintos menús, debe existir un elemento gráfico que le permita volver atrás libremente.



Figura 32. Panel con la opción de volver atrás (flecha verde).

### 6.2.3.4. Consistencia y estándares

Si hay cosas que significan lo mismo, entonces deberían tener una presentación visual similar. El jugador no debería esforzarse en intuir qué elementos, botones, palabras o situaciones son equivalentes.

En el videojuego debe existir un estándar respecto al uso de color para representar los modos de juego o secciones. Para lo relacionado al gameplay central, debe usarse la paleta principal. Otros modos pueden ser:

- Misiones
- Recompensas

---

<sup>7</sup> Un Sprite es un archivo de imagen que es integrado en una escena, frecuentemente en videojuegos 2D.

- Errores

La forma de los botones de confirmación siempre debe ser la misma, aunque su color varíe según el modo o sección del juego.



Figura 33. Distintos colores de un mismo botón.

### 6.2.3.5. Prevención de errores

Las acciones que sean críticas para la partida deben ser confirmadas por el jugador, ya que podrían ser ejecutadas por error y perjudicar al usuario.

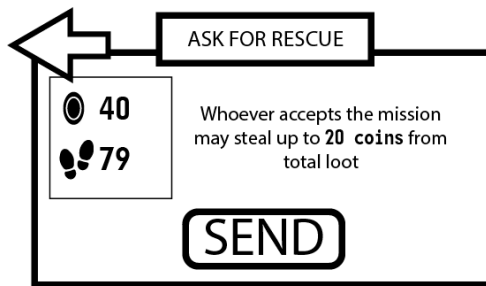


Figura 34. Antes de enviar una misión se pide una confirmación del jugador y se le advierte de lo que ocurrirá.

### 6.2.3.6. Reconocer en vez de recordar

Si un jugador desea hacer una acción en el momento que pueda ser necesario, el videojuego debe hacer visible las instrucciones o botones relacionados a esta.

Dentro de la partida, donde el jugador está concentrado en el gameplay, se deben presentar todas las opciones de movimiento, información de la partida, opción de pausa y retroalimentación acerca de los eventos del juego. No tendría sentido que se presente la opción de actualizar la aplicación en esa pantalla. En cambio, en el menú principal deberían ser visibles todas las opciones para acceder al control del juego, es decir, configuraciones y submenús donde el jugador interactúa y que no son parte del gameplay central.

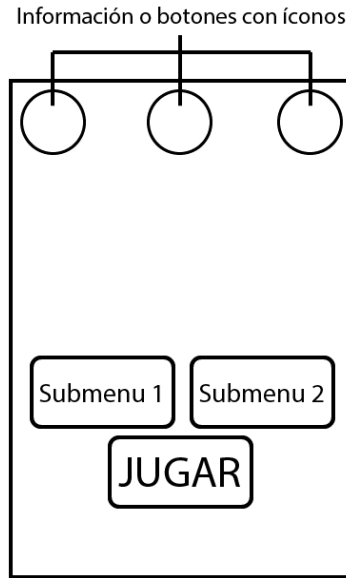


Figura 35. Diagrama del menú principal.

#### 6.2.3.7. Flexibilidad y eficiencia de uso

El usuario novato debe tener a la mano todas las opciones, sin embargo, a medida que juega más partidas, debería ser capaz de aumentar su eficacia. Si bien el juego no tiene demasiadas funcionalidades que puedan ser utilizadas bajo distintos niveles de experiencia, idealmente ninguna opción debería ser invisible al jugador.

#### 6.2.3.8. Diseño estático y minimalista

No se debe presentar información irrelevante al jugador que lo pueda distraer o que no sea necesaria. Las opciones relacionadas a una sección específica solo deberían mostrarse cuando se está navegando en ella.

#### 6.2.3.9. Ayudar a los jugadores a reconocer, diagnosticar y recuperarse de los errores

En caso de que exista un error dentro del juego, debe presentarse un diálogo en el que se explique en el lenguaje del jugador qué es lo que ocurrió y qué puede hacerse para solucionarse. Por ejemplo, si el jugador pierde sus monedas, se debe hacer visible la cantidad de monedas perdidas y si tiene la opción de recuperarlas.

En caso de errores relacionados a la conexión con el servidor, estos deben ser, solo si es importante para el jugador, mostrados mediante un Toast<sup>8</sup>.

#### 6.2.3.10. Ayuda y documentación

Aunque es importante que el jugador pueda comprender el videojuego sin ayuda externa, es bueno en términos de usabilidad que exista documentación a la que se pueda acceder.

Para este proyecto debe utilizarse un portal donde los usuarios puedan escribir sus dudas o retroalimentación. Este portal puede ser tanto la tienda de aplicaciones, página del juego y/o redes sociales.

#### 6.2.4. Flowchart

En la siguiente figura se visualiza un esquema de interfaz gráfica que indica cómo se relacionan las distintas pantallas del juego.

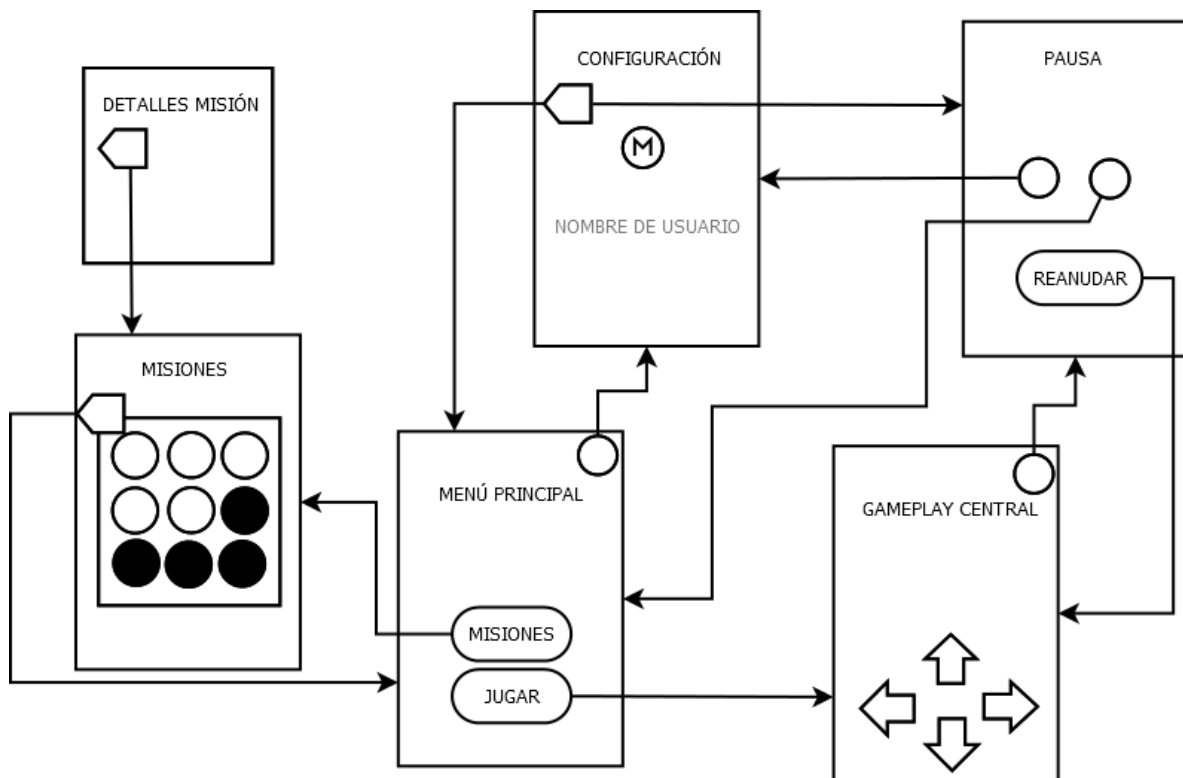


Figura 36. Flowchart del videojuego.

<sup>8</sup> Un Toast es un mensaje que se muestra en pantalla durante pocos segundos al usuario. Es nativo de ciertos sistemas operativos de dispositivos móviles.

### 6.3. Narrativa

A continuación, se presenta un cuadro en el que se responden las 6 preguntas básicas sobre la narrativa de un videojuego.

¿Qué?	Se escaparon unos monstruos de un laboratorio.
¿Por qué?	Porque hubo una ruptura dimensional.
¿Cómo?	Al abrir un portal que estaba mal sellado se escapó un ser que liberó al resto.
¿Cuándo?	En el pasado (años 80-90) con tecnología alternativa.
¿Quién?	Un grupo de científicos de seres extra dimensionales.
¿Dónde?	El Havoc Industries, un laboratorio que experimenta con seres de otra dimensión para crear soluciones médicas.

Figura 37. Tabla con las seis W's de la narrativa.

Mediante esta tabla, que sirve como punto de partida, es posible construir la historia y plot<sup>9</sup> del videojuego. Algunas de las seis preguntas sólo ocurren en la mente del jugador, es decir, no se explica explícitamente en el videojuego.

En Havoc In The Dungeon la historia no es muy importante, ya que se trata de un juego en el que cada partida es diferente y cuya duración depende del jugador. Se planea crear un plot que evolucionará según las actualizaciones del juego, es decir, en cada actualización importante, se agregarán elementos al juego que tendrán una pequeña historia asociada y, en la mente del jugador, se creará una historia sobre el videojuego.

### 6.4. Audio

#### 6.4.1. Género e instrumentalización

La música del videojuego debe utilizar sonidos de sintetizadores y armonizaciones exóticas, idealmente en modo locrio o frigio. Como referencia debe utilizarse la música de la serie Stranger Things o la saga Blade Runner, que utilizan este tipo de instrumentalización para evocar un ambiente extraño y misterioso.

---

<sup>9</sup> El plot es la secuencia de eventos dentro de una historia que afectan otros eventos a través del principio de causa y efecto.

#### **6.4.2. Formato del audio**

El audio del videojuego debe ser mediante archivos que contengan una pista de audio, es decir, formatos estándares. Esto excluye los formatos de interface digital para instrumentos musicales, como MIDI.

#### **6.4.3. Adaptabilidad**

Debe escucharse una pista de audio principal durante el gameplay, que esté preparada para escucharse en bucle. Paralelamente se reproducirán pistas silenciadas que utilizarán el mismo tempo y que servirán para inducir ambientes según los eventos del juego, momento en que aumentarán su volumen. Por ejemplo, en una situación de tensión se activará el sonido de una pista de percusión.

#### **6.4.4. Efectos de sonido**

Los efectos de sonido serán creados para el juego o se harán modificaciones a archivos gratuitos disponibles en internet. Los efectos de sonido deben tener un efecto lo-fi, lo que ayuda a reforzar el concepto de ambiente extraño y del pasado.

#### **6.4.5. Consideraciones especiales**

Ya que el videojuego se utilizará desde dispositivos móviles, es importante considerar que las frecuencias más bajas no serán tan audibles, por lo que todo el audio que necesite ser escuchado como retroalimentación de acciones o eventos importantes deben utilizar notas de octavas altas o en su defecto ecualizarse para aumentar los decibelios de la zona por sobre los 1000Hz.

### *6.5. Tecnología*

#### **6.5.1. Creación de herramientas**

Para agilizar el desarrollo del proyecto, será necesario crear las siguientes herramientas o plugins:

##### **3.1.1.1. Creador de niveles**

En conjunto con el sistema de Tilemaps de Unity, se debe crear un componente de editor que permita fijar los límites del mapa, los puntos importantes, los datos del evento si es el caso y un sistema para activar o desactivar la aparición de mapas con un rango de probabilidad.

### 3.1.1.2. Ajuste gráfico de movimientos

Para los comportamientos de enemigos u obstáculos que requieran el movimiento del objeto, se debe agregar una representación visual de los puntos de inicio y término, con el fin de visualizar rápidamente el trayecto dentro del editor de niveles.

### 3.1.1.3. Pestaña de menú con herramientas para el desarrollador.

En la barra de herramientas, se deberá agregar un menú que permita generar acciones de ayuda para el desarrollador, estas acciones serán, como mínimo, las siguientes.

- Eliminar datos guardados.
- Acceder al listado de niveles.
- Acceder al editor de niveles.
- Acceso directo a carpetas del proyecto.

### 3.1.1.4. Íconos en la jerarquía

Esta herramienta permite asignar un ícono a los componentes creados, los que serán vistos dentro de la jerarquía del editor, ayudando a identificar rápidamente los GameObjects dentro de Unity.

## 2.1.2. Integración con PlayFab

El videojuego debe inicializar su conexión al servidor de PlayFab al comenzar la ejecución. Si la conexión es exitosa, se debe invocar a todos los métodos que necesiten que se haya iniciado la sesión. En caso contrario, se deben desactivar todas las funciones que requieran conexión al servicio.

Para el guardado de datos, se debe crear una copia local en formato JSON que contenga el inventario y progreso del jugador. Cuando exista conexión, se guardará tanto local como remotamente. Si hay problemas de conectividad, se guardarán los datos de manera local y se sincronizarán nuevamente cuando se restaure la conexión.

Dentro de PlayFab, se deben mantener separados los datos altamente dinámicos con los que no cambien o requieran conexión permanente, como puede ser el nombre de usuario, recompensas o récords. Esto con el fin de que se evite la sobreescritura de datos.

Se debe inicializar un experimento A/B que tenga una asignación del 50% de usuarios para cada variante, con el fin de mantener a los dos tipos de usuarios en cantidades iguales cuya aplicación tendrá algunas diferencias.



### **3.1.3. Uso de Unity Ads**

Los anuncios deben mostrarse cuando un jugador pierda y haya avanzado un mínimo de 20 pasos, esto con el fin de que un usuario novato, que pierda más seguido, no vea demasiados anuncios y por tanto no se agobie.

### **4.1.4. Shaders**

Para algunos elementos de UI que deban llamar la atención del jugador, se debe crear un shader que provoque un movimiento constante.

Algunos enemigos podrían requerir un shader especial para ser representados de mejor forma. Por ejemplo, un fantasma debería tener un efecto de distorsión.

## **6.6. *Mecánicas***

### **6.6.1. Mecánicas centrales**

En este apartado se enumeran los sistemas de juego que forman parte del gameplay central y que contienen sus mecánicas.

#### **2.2.3.1. Sistema de juego: Movimiento del jugador**

Este sistema contempla todas las mecánicas que influyen en el movimiento del personaje principal controlado por el jugador.

El personaje comienza en el estado estático. Cuando el jugador toca una flecha en la pantalla, comienza la animación de movimiento. Al finalizar la animación, se realiza el "Landing", este proceso contempla la evaluación de las condiciones del jugador y del lugar hacia el que se movió. Si en el lugar existe una baldosa, el personaje volverá al estado estático, de lo contrario caerá si la partida se acabará.

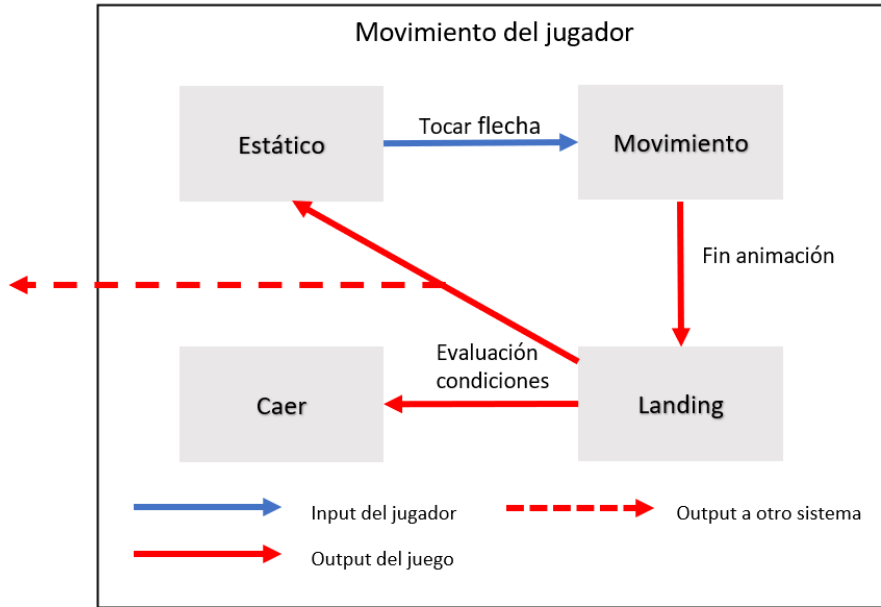


Figura 38. Sistema de juego: Movimiento del jugador.

### 2.2.3.2. Sistema de juego: Accionadores (trampas)

Este sistema define cómo funciona un accionador, que es una baldosa que al ser pisada por el personaje desata un evento que sirve como habilidad para el jugador.

Cuando se vuelve activo, se crean objetos en la escena, que pueden tener distintas propiedades. Algunos tienen una duración extendida, por lo que entran en un loop hasta que expire y finalice su efecto. Los objetos instanciados por un accionador podrían provocar efectos en otros sistemas.

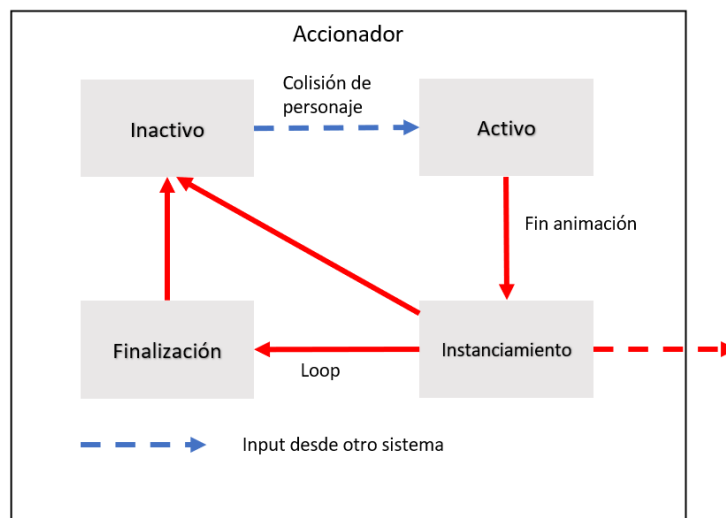


Figura 39. Sistema de juego: Accionadores.

### 2.2.3.3. Sistema de misiones

El sistema de misiones, al que se hace referencia cuando se menciona la mecánica de cooperación online aleatoria, se refiere a un mecánica en la que un jugador, tras haber perdido, es capaz de solicitar la ayuda de otro jugador para recuperar el botín recolectado durante su partida. El otro jugador, puede acceder a una lista de misiones disponibles y aceptar la que vea más conveniente. El punto clave, es que uno de los dos jugadores puede elegir un **porcentaje de repartición el recompensas**, esto para que completar una misión tenga beneficios para ambas partes.

Para completar una misión, el jugador deberá avanzar una cantidad de pasos definida según las características de la misión escogida. Si el jugador pierde antes de alcanzar este umbral, la misión se dará por fallida y el botín se perderá.

¿Quién decide este porcentaje? Ambas opciones son posibles de implementar, y por ello se realizará un test A/B en el que un grupo de jugadores solo podrá elegir el porcentaje de repartición si solicita una misión y el otro solo si la acepta.

Los sistemas serán llamados **Share** (quien pide ayuda comparte parte de su botín) y **Steal** (quien acepta la misión puede robar parte de las recompensas).

Para equilibrar el juego, si un jugador quiere retener más parte del botín, los pasos necesarios para cumplir la misión serán mayores.

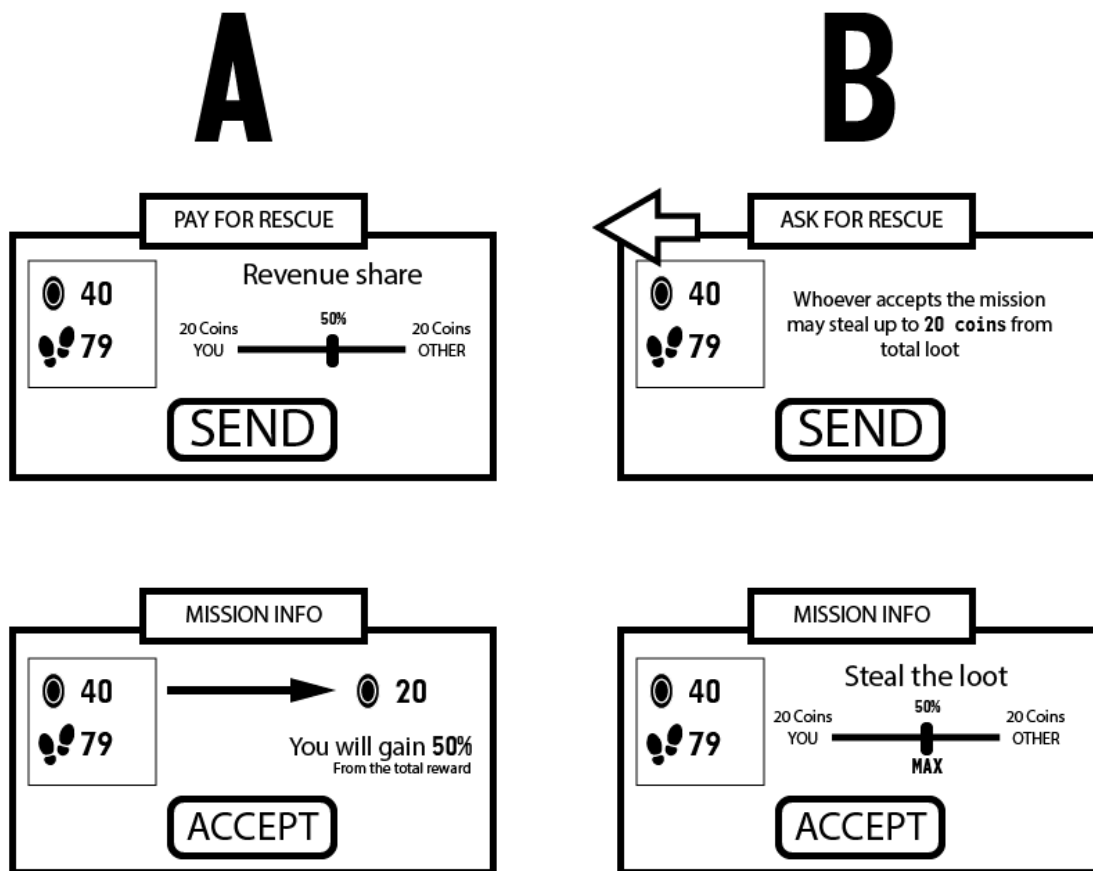


Figura 40. Diagrama con las diferencias entre el sistema de misiones A (Share) y el B (Steal).

#### 2.2.3.4. Sistema de juego: Crear misión

Cuando un jugador pierde, tiene la posibilidad de solicitar la ayuda de otro jugador, publicando una misión en un tablero general.

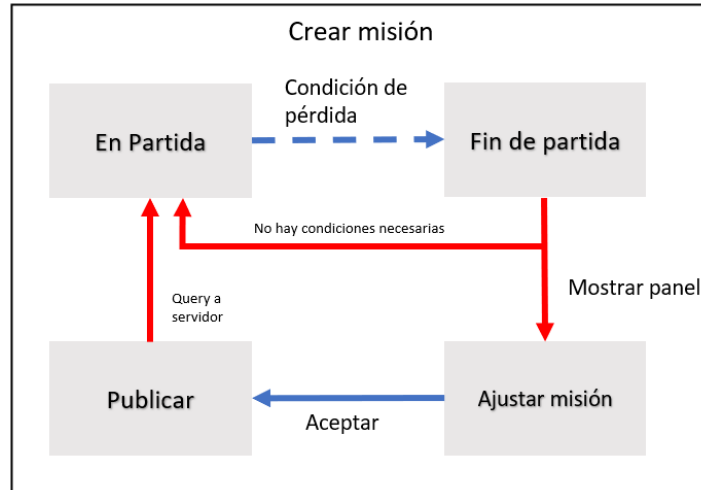


Figura 41. Sistema de juego: Crear misión.

### 2.2.3.5. Sistema de juego: Aceptar misión

Un jugador, antes de iniciar una partida, puede abrir el panel de misiones, y si hay alguna disponible, puede ver su detalle y aceptarla.

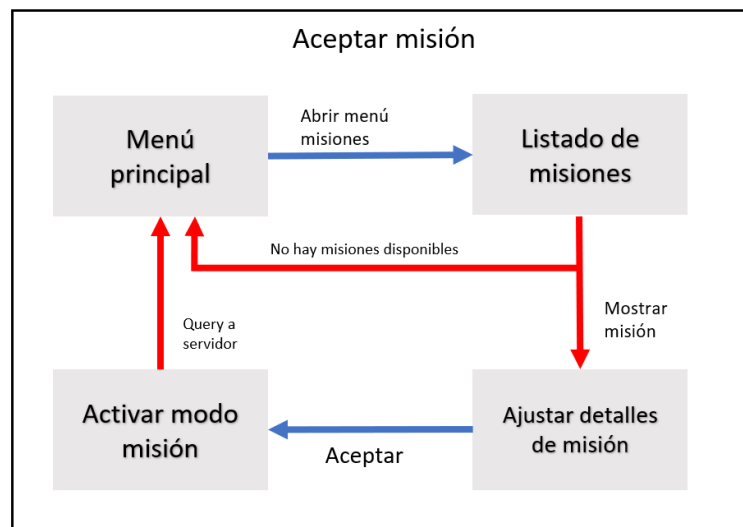


Figura 42. Sistema de juego: Aceptar misión.

## 2.2.4. Mecánicas satelitales

Las mecánicas satelitales son aquellas que complementan y/o mejoran a las mecánicas ya existentes.

### 2.2.4.1. Mecánicas de realce

Estas mecánicas son aquellas que tienen el propósito de realzar otras mecánicas, renovándolas o complementando su funcionamiento, ejemplo de esto son los power-ups.

Para Havoc In The Dungeon no se contemplan mecánicas de realce en su versión inicial.

#### 2.2.4.2. Mecánicas alternativas

Son aquellas que permiten al jugador realizar una acción de otras maneras pero con el mismo o similar resultado.

<b>Uso de llaves y puertas</b>	Al tocar una puerta, si el jugador tiene una llave del mismo color de esta, podrá romperla y saltarse parte de una mazmorra.
<b>Accionadores</b>	Se puede considerar a algunos accionadores como mecánica alternativa si el jugador puede superar el nivel sin utilizarlos.
<b>Portales</b>	Si un jugador toca un portal, será trasladado hacia el otro lado de este. En algunas mazmorras se puede usar como alternativa, a cambio de perder recompensas.

#### 2.2.4.3. Mecánicas de oposición

Estas mecánicas tienen como propósito obstaculizar el progreso del jugador.

<b>Eventos de mazmorra</b>	Son mazmorras en las que el jugador debe cumplir un objetivo para poder superarlas ya que estarán bloqueadas por una puerta.
<b>Enemigos y obstáculos</b>	Aunque forman parte del gameplay central, los enemigos y obstáculos también se pueden considerar como mecánicas de oposición, ya que impiden el paso libre por una mazmorra.

### 6.7. Personajes

En esta sección se describirán las características estéticas, narrativas y funcionales de los personajes del videojuego.

#### 6.7.1. Personaje principal: Havoc

El personaje principal, que será controlado por el jugador, cuenta con la siguiente historia:

“Es un científico que entra a la mazmorra bajo el laboratorio de Havoc Laboratories para sellar la entrada de los monstruos, pero debe enfrentarse a temibles enemigos y trampas, y no sabe qué tan profundo es el lugar, tendrá que salir con vida de su misión.”

Estéticamente, el personaje debe destacar sobre los enemigos y el mundo que lo rodea, por lo que su traje debe usar un color muy contrastante con las mazmorras y ser más brillante que

el resto. Su rostro no debe ser visto, esto tanto por temas de clasificación de edad (ningún personaje es explícitamente humano) como para captar el interés del usuario.

Posee la mecánica de movimiento y puede utilizar los accionadores para combatir a los enemigos.

## 6.7.2. Enemigos

En esta sección se listarán los enemigos que habitarán las mazmorras, se describirán sus habilidades y su apariencia. Para identificarlos, se usan nombres clave, que no necesariamente son idénticos a lo que podría visualizarse en el videojuego.

### 6.7.2.1. Lanzador

Este enemigo es capaz de lanzar un objeto a una distancia corta de forma periódica. El objetivo es que el jugador debe esquivar el objeto o calcular el tiempo adecuado para atravesarlo

Estéticamente, es un ser con capucha y máscara similar a la de los doctores de la edad media durante la peste negra, pudiendo incluso ser realmente un cuervo.



*Figura 43. Referencia para enemigo lanzador.*

### 6.7.2.2. Slime

El slime es un enemigo clásico de los videojuegos de rol, se trata de una especie de gelatina que se mueve lentamente en un patrón predefinido en vertical u horizontal.

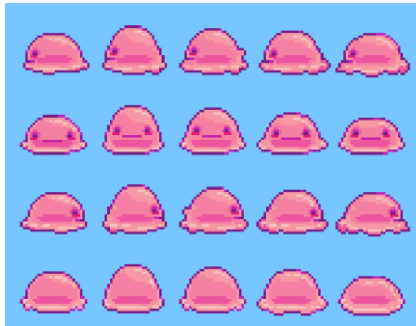


Figura 44. Spritesheet de referencia del Slime. Fuente: GameDev Market.

### 6.7.2.3. Volador

El enemigo volador, al igual que el slime, se mueve en un patrón definido, pero a una mayor velocidad y con la posibilidad de hacerlo de manera diagonal y sobrevolar baldosas vacías. Estéticamente debe lucir como un bicho que pueda volar, esta decisión puede cambiar durante la implementación.

### 6.7.2.4. Bastión

El bastión es un enemigo que se queda estático y cada 2 segundos escanea su alrededor (en 8 direcciones) para encontrar al personaje principal. Si encuentra al personaje, se activará una mira que le indicará al jugador donde se moverá el enemigo. Después de unos segundos (por ajustar) el bastión se trasladará a esa posición.

En el apartado estético, este debe ser un tótem de piedra con ojos que se ilumina cada vez que escanea su alrededor.

### 6.7.2.5. Fantasma

El enemigo fantasma es capaz de flotar en el mapa en un movimiento circular constante a una velocidad que permita que el jugador pueda pasar por donde estuvo o estará el fantasma sin colisionar con él.

Debe lucir como un espectro o fantasma semitransparente que deje una estela al moverse.

Será necesario crear un shader especial que simule una ligera distorsión que varía con el tiempo.

### 6.7.2.6. Estático

El enemigo estático no realiza ningún movimiento, por lo que el jugador solo debe evitar tocarlo. En una mazmorra donde se implemente este personaje, es necesario que se encuentre en lugares estrechos o que haya varias instancias de este, con el fin de que el jugador juegue de manera cautelosa.



No hay una definición estética para este personaje, ya que su funcionalidad al ser tan básica podría implementarse en más de un tipo de enemigo.

No se descarta la adición de nuevos tipos de enemigos durante la implementación o después del lanzamiento.

## 7. Implementación

En esta sección se describe el proceso de implementación del diseño del videojuego en el proyecto de software, incluyendo tanto el desarrollo del videojuego como su posterior publicación.

Considérese el videojuego como una demo, ya que solo se desarrollaron los aspectos clave definidos en el videojuego y no el total de funcionalidades que se implementarán en una versión final después de presentarse este trabajo. Dada la naturaleza del género del videojuego escogido, es necesario agregar contenido de manera constante para mantener activos a los jugadores, por lo que el producto publicado podría tener diferencias con lo escrito en esta sección al momento de su lectura.

En algunas subsecciones se adjuntará parte del código del videojuego. No necesariamente se incluirá el texto del archivo completo, sino que solo las partes que sean de mayor importancia, evitando así la repetición innecesaria de contenido muy similar.

### 7.1. Estructura de ficheros

Al tratarse de un videojuego desarrollado en Unity, por defecto se crean carpetas que contienen los paquetes básicos para la ejecución del proyecto. La mayoría del contenido creado para el proyecto estará dentro de la carpeta Assets.

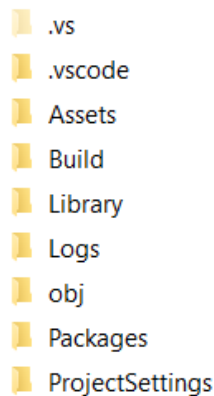


Figura 45. Estructura de carpetas por defecto.

En la carpeta Assets se crearon otras subcarpetas para organizar de mejor manera los ficheros del juego, por ejemplo, todo el código estará ubicado en la carpeta Scripts, a excepción de aquel que se use para herramientas del editor, en cuyo caso serán contenidos en la carpeta Editor. Adicionalmente, algunos plugins al ser instalados crean carpetas adicionales.

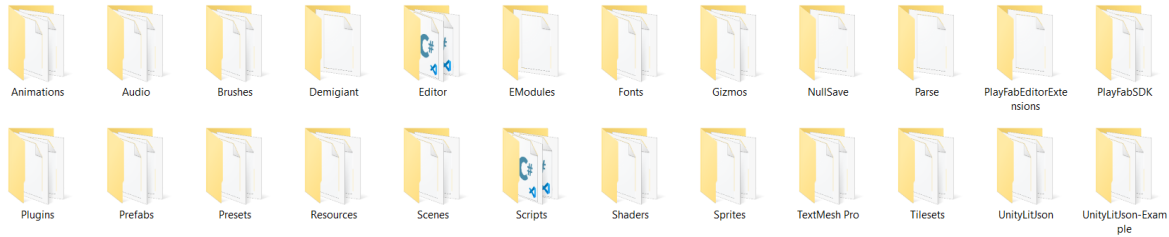


Figura 46. Estructura de carpetas dentro de Assets.

## 7.2. Entorno de trabajo

Todo el videojuego se ejecuta desde una sola escena, la cual contiene tanto el gameplay central como las interfaces de usuario de los distintos menús que hay en el juego. En la siguiente figura se observa la jerarquía de objetos del videojuego sin ejecutarse.

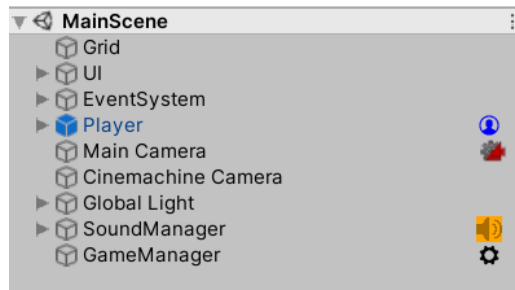


Figura 47. Vista de la jerarquía de la escena principal.

Los elementos principales de la escena son:

### 7.2.1. Grid

Este objeto contiene los niveles (mazmorras) que se irán agregando dinámicamente a la escena. Cada nivel consiste en un GameObject con el componente Tilemap, el cual requiere un componente Grid como padre para ser visualizado.

En la siguiente figura se observa la visualización de cuadrícula al seleccionar el objeto Grid.

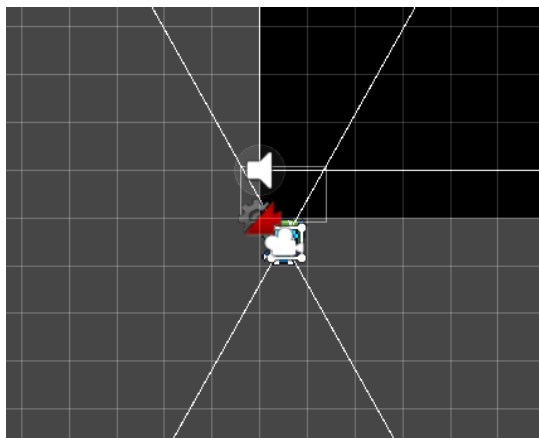


Figura 48. Escena con el Grid seleccionado.

### 7.2.2. UI

Este objeto contiene el componente *Canvas*, que es el área donde los objetos de interfaz gráfica se visualizarán y sirve como objeto padre de estos.

UI tiene como objetos hijos a distintos *GameObjects* con el componente *Panel* que sirven para diferenciar conjuntos de interfaces gráficas (menús).

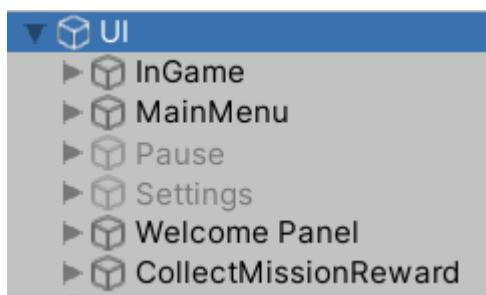


Figura 49. Jerarquía del editor con el objeto UI abierto.

Por defecto, la UI utiliza una resolución de 720x1280 píxeles (9:16). A partir de esta se disponen en escena todos los elementos de UI que luego se adaptan a otras resoluciones.

### 7.2.3. Player

En la escena siempre se encuentra el objeto Player, que contiene los siguientes componentes:

- *GridMovement*: Es el script para el movimiento del jugador
- *Collider2D*: Representa la forma física del objeto.
- *RigidBody2D*: Representa el cuerpo físico del objeto, permite la detección de colisiones y simulación de físicas.
- *SpriteRenderer*: Permite mostrar un Sprite en la escena.
- *Animator*: Permite activar y organizar un conjunto de animaciones del objeto.

En cada partida, este objeto comienza en la posición (0,-1,0) y avanza en los ejes x e y a lo largo de la partida.

#### 7.2.4. Main Camera y Cinemachine Camera

Estos *GameObjects* representan la cámara del videojuego. Son objetos separados ya que el paquete Cinemachine de Unity utiliza un sistema de “brain” que evalúa los parámetros asignados a la cámara, como la velocidad de seguimiento, vibración, zona segura, etc., que se encuentran como objeto separado de Main Camera.

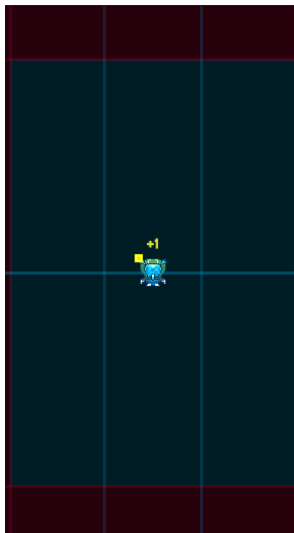


Figura 50. Visualización de la escena al seleccionar Cinemachine.

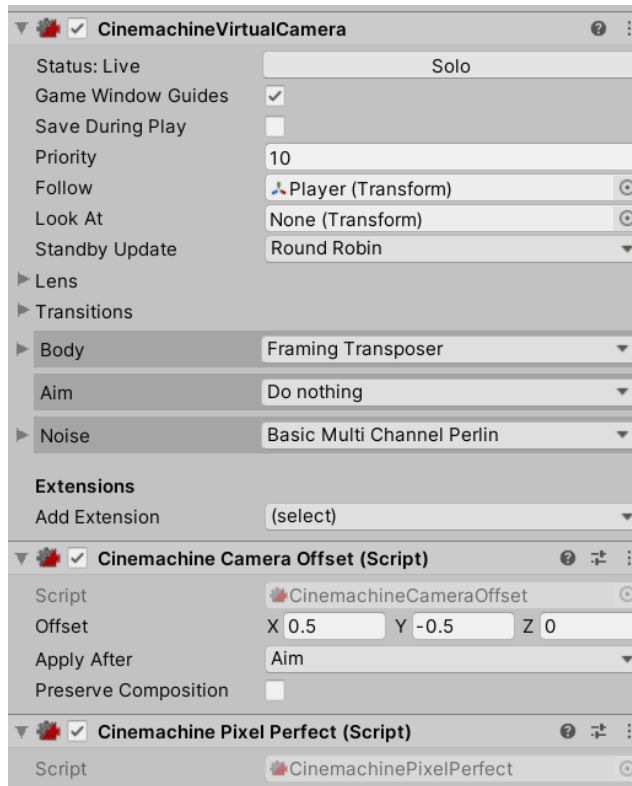


Figura 51. Parámetros asignados a la VirtualCamera.

## 7.2.5. SoundManager

Es un objeto que solo contiene un componente *SoundManager* el cual es un script que contiene referencias a varios *AudioSource* (componente para reproducción de sonido) y que sirve para reproducir los efectos de sonido desde cualquier parte del código del juego.



Figura 52. Componente SoundManager en el inspector.

## 7.2.6. GameManager

Este objeto contiene el componente GameManager, por lo que es uno de los más importantes en la escena. En él se pueden asignar parámetros muy importantes como la referencia al jugador, a los niveles, los datos de misiones, el inventario, etc.

Contiene algunas variables que solo están disponibles en el editor y que sirven para forzar un tipo de rescate independiente de la asignación que se haya hecho al jugador en el experimento A/B de PlayFab.

Además, al objeto se le agregó un componente PlayFabLogin que ejecuta el inicio de sesión del jugador en el servidor PlayFab. Este script envía la señal de inicio de sesión exitoso o fallido al resto de los scripts que lo necesiten.

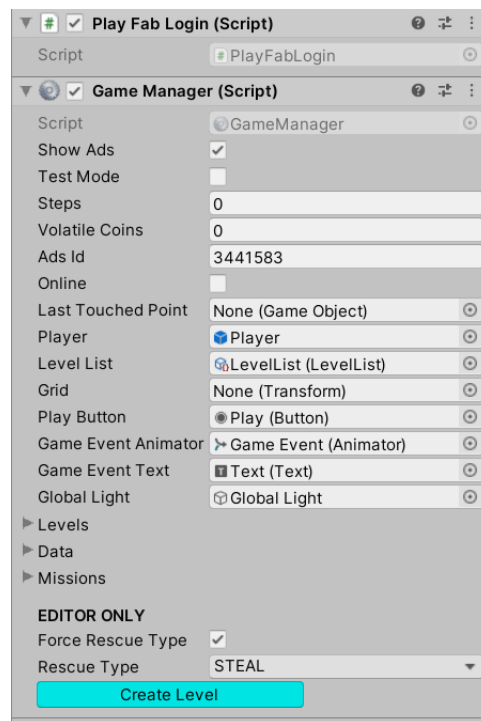


Figura 53. Visualización de GameManager en el inspector.

## 7.3. Script de movimiento (GridMovement)

El Script *GridMovement* permite que un objeto realice un movimiento basado en unidades fijas y detecta si hay o no baldosas de un tilemap colisionando con el objeto.

El método *Move* tiene varias firmas distintas, pero todas ejecutan *Move(Vector2)* en su cuerpo. Este método inicializa una animación que traslada al personaje en la cantidad dada de unidades siempre y cuando haya una partida en juego.

Al finalizar la animación de movimiento, se ejecuta el método *CheckFall()* el cual analiza si existe una baldosa en el tilemap actual, en caso de que no la haya, se ejecutan los métodos asociados a la caída, como *Die()* del GameManager el que a su vez invoca a la acción *onDie*, la cual tiene listeners<sup>10</sup> en varios scripts.

---

<sup>10</sup> Un listener es un método o instrucción que se ejecuta cuando se invoca un evento independiente de su procedencia.



```

public void Move(Vector2 movement)
{
    if (canMove)
    {
        GetComponent<Rigidbody2D>().velocity = Vector2.zero;
        this.movement = movement;
        GetComponent<Rigidbody2D>().DOComplete();
        if (movement.x != 0)
        {
            GetComponent<Rigidbody2D>().DOMoveX(Mathf.Round(transform.position.x) +
            movement.x, speed).OnComplete(FixPosition);
        }
        else
        {
            GetComponent<Rigidbody2D>().DOMoveY(Mathf.Round(transform.position.y) +
            movement.y, speed).OnComplete(FixPosition);
        }
        lastPosition = transform.position;
        GameManager.Instance.UpdateScore(lastPosition.y + movement.y);
        Invoke("CheckFall", 0.02f);
    }
}

void CheckFall()
{
    Tilemap map = GameManager.Instance.lastTouchedPoint.GetComponentInParent<Tilemap>();
    Vector3Int positionCheck = map.WorldToCell(lastPosition + (Vector3)movement +
    new Vector3(0.5f, -0.5f, 0));
    bool hasFallen = !map.HasTile(positionCheck);
    Vector2 prevMovement = movement / 2;
    if (hasFallen)
    {
        CancelMovement();
        GetComponent<Rigidbody2D>().MovePosition(GetComponent<Rigidbody2D>().position +
        prevMovement);
        SoundManager.Instance.falling.Play();
        GameManager.Instance.Die();
        GetComponent<Collider2D>().enabled = false;
        GetComponent<SpriteRenderer>().DOFade(0, 0.6f).OnComplete(() =>
        {
            GetComponent<Rigidbody2D>().velocity = Vector2.zero;
            GetComponent<Rigidbody2D>().gravityScale = 0;
        });
        GetComponent<Rigidbody2D>().gravityScale = 1;
    }
}
}

```

## 7.4. Funcionamiento del GameManager

El GameManager es el script más importante del juego, ya que indica los estados del juego y de los objetos dentro de la escena.

Dentro de los parámetros de la clase, están los UnityAction, que representan eventos que al ser invocados serán escuchados por listeners en el resto de los scripts, de esta manera es posible dividir el código en diferentes clases que tengan funciones más específicas y no interfieran con la funcionalidad original del GameManager.

El GameManager mantiene constantemente una referencia al personaje principal y al mapa en el que se está jugando.

### 7.4.1. Creación de niveles

Cada vez que el jugador tiene una diferencia de 20 pasos con el nivel activo más antiguo, este se elimina y se crea un nuevo nivel arriba del que fue creado más recientemente. GameManager mantiene una lista con la referencia a los archivos de nivel llamada "levelList". Aleatoriamente, se elige un nivel dentro de los que esté en la lista y este será elegido si cumple con las siguientes condiciones:

- Su variable interna activeLevel es verdadera.
- Su variable probability es menor a una variable aleatoria entre 1 y 100.
- Aún no se han alcanzado los pasos mínimos (stepsForSave) para que aparezca el nivel de guardado. En tal caso se reemplazará el nivel elegido por el nivel "Save".

```

public void CreateNewLevel()
{
    GameObject newLevel;
    if (levels.Count > 0)
        lastLevelPoint = levels[levels.Count - 1].GetComponent<GameLevel>().endPoint.position;

    if (steps - stepsSinceLastSave > stepsForSave)
    {
        newLevel = Instantiate(levellist.saveLevel, lastLevelPoint + Vector3.up, Quaternion.identity, grid.transform);
        stepsSinceLastSave = steps;
        stepsForSave += 12;
    }
    else
    {
        LevelData level;
        do
        {
            level = levellist.levels[UnityEngine.Random.Range(0, levellist.levels.Count)];
        } while (!level.activeLevel || level.probability < UnityEngine.Random.Range(0, 99));
        newLevel = Instantiate(level.levelPrefab, lastLevelPoint + Vector3.up, Quaternion.identity, grid.transform);
    }
    newLevel.transform.Translate(newLevel.transform.position - newLevel.GetComponent<GameLevel>().startPoint.position);
    levels.Add(newLevel);
    newLevel.GetComponent<TilemapRenderer>().sortingOrder = totalLevels;
    totalLevels--;
}

```

Figura 55. Método *CreateNewLevel()* del *GameManager*.

#### 7.4.2. Reiniciar juego

Para limpiar el progreso de niveles de una partida, se utiliza el método *ResetGame()*, este reinicia los parámetros fundamentales (pasos, niveles superados, pasos desde el último punto de guardado, etc.) a su valor por defecto. Reestablece el movimiento del jugador, elimina los tilemaps de la partida anterior de la escena y crea un nuevo nivel de partida.

```

public void ResetGame()
{
    //Reset parameters
    stepsForSave = 60;
    playing = true;
    showAds = true;
    steps = 0;
    totalLevels = 0;
    volatileCoins = 0;
    stepsSinceLastSave = 0;

    player.GetComponent<GridMovement>().canMove = true;

    onStep.Invoke();
    GameObject[] levelObjects = GameObject.FindGameObjectsWithTag("Level");
    for (int i = 0; i < levelObjects.Length; i++)
    {
        DestroyImmediate(levelObjects[i]);
    }
    levels = new List<GameObject>();
    lastLevelPoint = Vector3.zero;
    player.transform.position = Vector3.up;

    CreateNewLevel();

    lastTouchedPoint = levels[0].GetComponent<GameLevel>().startPoint.gameObject;
    onReset.Invoke();
}

```

Figura 56. Método ResetGame() del GameManager.

### 7.4.3. Muerte del personaje, fin de partida

El método Die() del GameManager se ejecuta al caer por una baldosa vacía o chocar con un enemigo u obstáculo. Si hay una misión activa sin completar, esta se dará por perdida y se desactivará. Si el jugador ya ha dado 20 pasos y hay un anuncio disponible, este se mostrará.

```

internal void Die()
{
    if (playing)
    {
        if (IsMissionActive())
        {
            if (steps < missions.currentMision.steps)
                missions.LoseMission();
        }
        ShakeCamera(5, 25, 0.6f);
        if (Advertisement.IsReady(videoad) && showAds && online)
        {
            if (UnityEngine.Random.Range(0, 100) < 50 && steps > 20)
            {
                showAds = false;
                Invoke("ShowAd", 0.5f);
            }
            else
            {
                DieEvent();
            }
        }
        else
        {
            DieEvent();
        }
        globallight.GetComponent<Animator>().SetTrigger("Die");
        playing = false;
    }
}

void DieEvent()
{
    missions.CheckCollectReward();
    onDie.Invoke();
    missions.GetAllMisions();
}

```

Figura 57. Métodos Die() y DieEvent() del GameManager.

## 7.5. Sistema de misiones

Para diferenciar los dos sistemas de misiones, se creó un experimento A/B en PlayFab. El grupo A utiliza el sistema “Share” en el que quien pide la misión decide el porcentaje de repartición y el grupo B utiliza el sistema “Steal” que funciona de manera contraria: quien acepta la misión elige el porcentaje de repartición.

Se creó un script separado del GameManager para la gestión del sistema de misiones. Una Misión es una clase que contiene el ID único del jugador que solicita la ayuda, los pasos necesarios para cumplir el objetivo y la recompensa para el jugador. La clase *MissionManager*. Cada vez que se inicia sesión, mediante el método *GetAllMissions()* se consulta la Leaderboard “Missions\_Share” o “Missions\_Request” (dependiendo del tipo de jugador en el experimento A/B) de PlayFab y se guardan todas entradas con valor positivo, esto ya que no se pueden eliminar las entradas de una Leaderboard.

Se prefirió utilizar el Leaderboard en vez de consultar a todos los jugadores existentes ya que el procesamiento de esa operación sería demasiado extenso y podrían ocurrir problemas con la sincronización de los datos.

Para cada entrada positiva, se realiza una consulta de los datos del jugador que pide la misión, en específico del parámetro “MissionRequest” el cual contiene la información de la misión.

### Leaderboard







Rank	Name	Value
 0	Ares502	47
 1	Rorschach	0
 2	pc game	0
 3	Paxn	0
 4	noobmaster69	0
 5	tamam_studios	0

Figura 58. Leaderboard utilizado para las misiones de rescate en modo "Share".

#### PLAYER DATA

[X REMOVE](#) [TALL DISPLAY](#)

<input type="checkbox"/>	Key	Value	Permissions
<input type="checkbox"/>	MatchList	{ "coins":1372,"keysInventory":{"key_gold","key_gold","key_gold","key_g... <a href="#">JSON</a>	Public
<input type="checkbox"/>	MissionRequest	{ "coins":12,"steps":47,"sharedCoins":3} <a href="#">JSON</a>	Private
<input type="checkbox"/>			Public

Figura 59. Datos de un jugador en PlayFab.

```

public void GetAllMisions()
{
    Debug.Log("Obteniendo misiones de " + GetStatisticName());
    PlayFabClientAPI.GetLeaderboard(new GetLeaderboardRequest
    { StatisticName = GetStatisticName(), StartPosition = 0, MaxResultsCount = 10 },
    (GetLeaderboardResult r) =>
    {
        allMisions.Clear();
        List<PlayerLeaderboardEntry> missions = r.Leaderboard;
        foreach (PlayerLeaderboardEntry item in missions)
        {
            if (item.PlayFabId != PlayerPrefs.GetString("UserId"))
            {
                Mission m = new Mission();
                m.userId = item.PlayFabId;
                m.steps = item.StatValue;
                m.displayName = item.DisplayName;
                if (m.steps > 0)
                {
                    GetMissionData(m.userId, m);
                    allMisions.Add(m);
                }
            }
        }
        GameManager.Instance.onGetMissions.Invoke();
    }, null);
}
}

```

Figura 60. Método GetAllMisions() del MissionManager.

La fórmula para calcular la cantidad de pasos según el porcentaje de repartición es la siguiente:

$$pasos = pasos\ originales - \left( \frac{pasos\ originales}{1.5} * \frac{50 - porcentaje\ repartición}{100} \right)$$

Esta fórmula permite una relación lineal entre los pasos a cumplir con un porcentaje de 0 y uno de 100, resultando en un porcentaje de pasos a cumplir respecto a los pasos originales de un 67% y 133%, respectivamente. Los pasos originales se obtienen al utilizar un porcentaje de repartición de 50%.

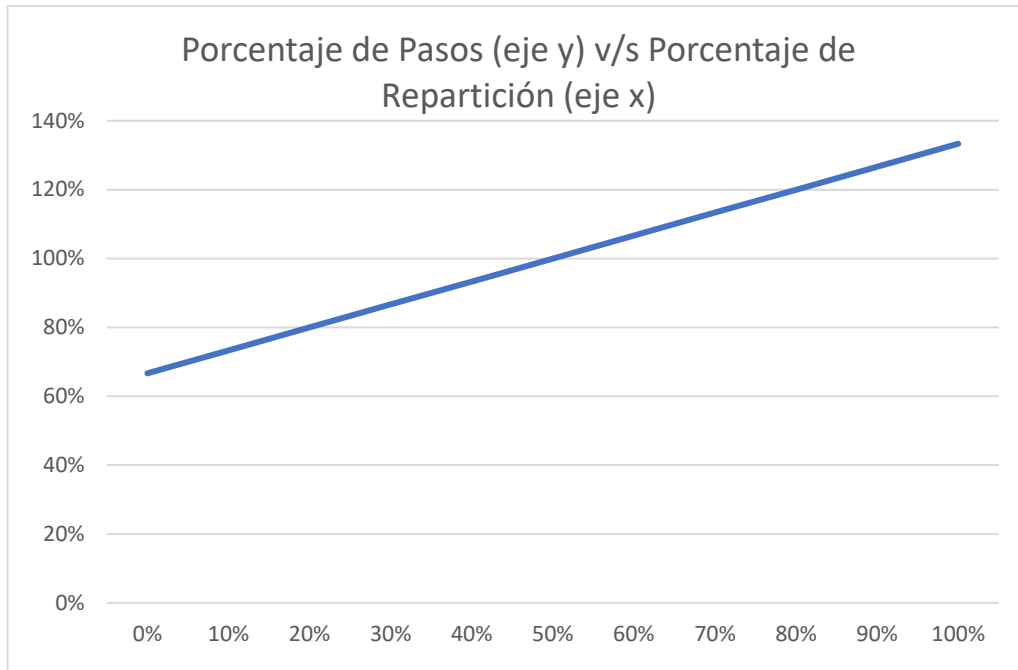


Figura 61. Gráfico de relación entre porcentaje de pasos a cumplir respecto a los originales v/s porcentaje de repartición escogido.

### 7.5.1. Aceptar misión

Al aceptar una misión, el parámetro *CurrentMission* se iguala a la misión escogida que estaba en la lista de misiones disponibles. Luego, en la partida, cada vez que el jugador de un paso se comprobará si ya se han realizado los necesarios para cumplir la misión. En caso de cumplir los objetivos de la misión, se notificará al jugador de la recompensa obtenida y se volverá al modo normal de juego. Si el jugador pierde mientras se está realizando una misión, esta se desactivará, es decir, volverá a la lista de misiones disponibles mientras que la misión actual será nula.





Figura 62. Lista de misiones disponibles dentro del menú Missions.



Figura 63. Partida del juego mientras se realiza una misión. En la parte superior se observan los pasos necesarios para cumplir el objetivo.

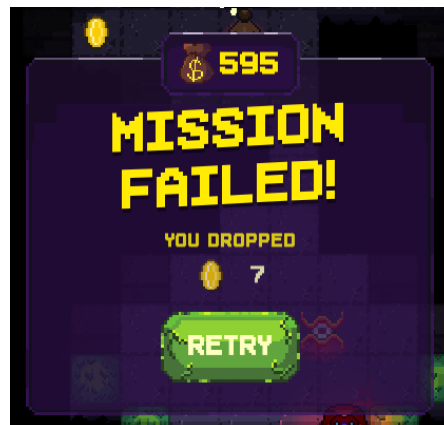


Figura 64. Panel de Misión Fallida.

Dependiendo del tipo de rescate (Share o Steal) el panel de ajuste de misión tendrá variaciones. En el caso de Share, quien acepta la misión solo puede ver el porcentaje de repartición y decidir si aceptar o no la misión.

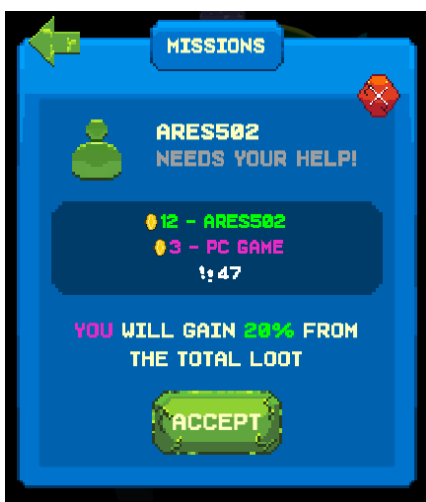


Figura 65. Panel de ajuste de misión. Modo Share.

En el modo Steal, el jugador podrá ajustar el porcentaje de ganancia, con un máximo de un 50% y luego aceptar la misión.

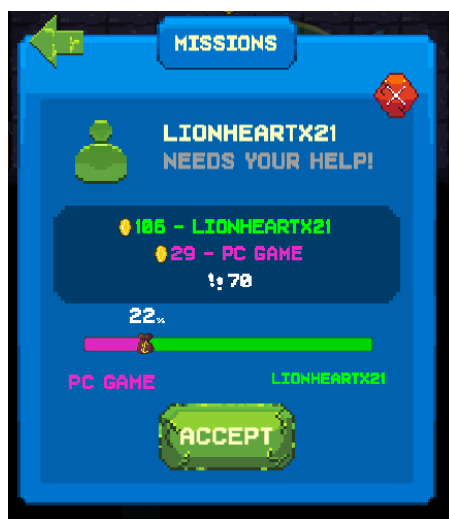


Figura 66. Panel de ajuste de misión. Modo Steal.

### 7.5.2. Crear misión

Cuando un jugador pierde, si no ha solicitado una misión y tiene conexión a internet, se le ofrecerá la opción de pedir ayuda mediante una misión de rescate. Dependiendo del tipo de rescate, el jugador podrá o no decidir el porcentaje de repartición del botín.



Figura 67. Opción de pedir ayuda.

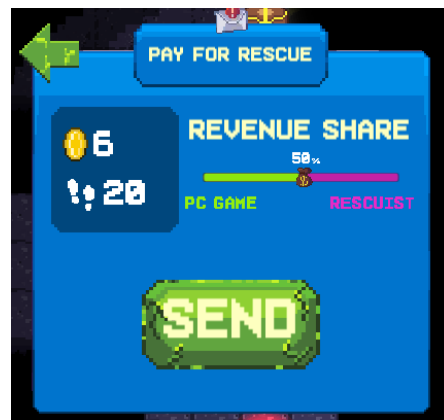


Figura 68. Panel de ajuste de petición de misión. Modo Share.

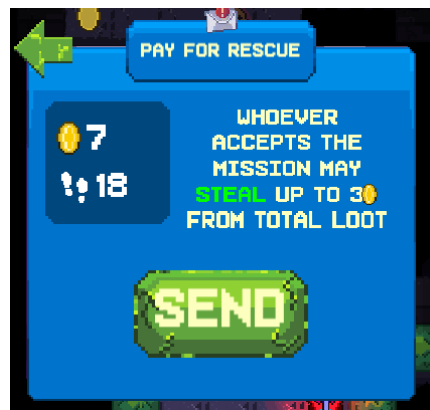


Figura 69. Panel de ajuste de petición de misión. Modo Steal.

## 7.6. Extracción de datos

Utilizando Unity Analytics, fue posible generar eventos personalizados que son enviados al servicio en la nube al aceptar, solicitar o rechazar una misión. Para esto, se utiliza el método `Analytics.CustomEvent()` al cual se le pueden agregar distintos parámetros personalizados.

```

public void AcceptMission()
{
    mission.isActive = true;
    GameManager.Instance.missions.currentMision = mission;
    GameManager.Instance.playButton.onClick.Invoke();
    float rate = (float)mission.sharedCoins / (mission.sharedCoins +
        mission.coins);
    float elapsed = Time.time - time;
    Analytics.CustomEvent(GameManager.Instance.missions.GetStatisticName() +
        "_Accept", new Dictionary<string, object>
    {
        { "shared_coins", mission.sharedCoins },
        { "coins", mission.coins },
        { "rate", rate },
        { "steps", mission.steps },
        { "time_elapsed", elapsed }
    });
}

```

Figura 70. Método AcceptMission() del componente MissionManager.

Luego, es posible observar estos datos en la página de administración del proyecto Unity Dashboard en línea.

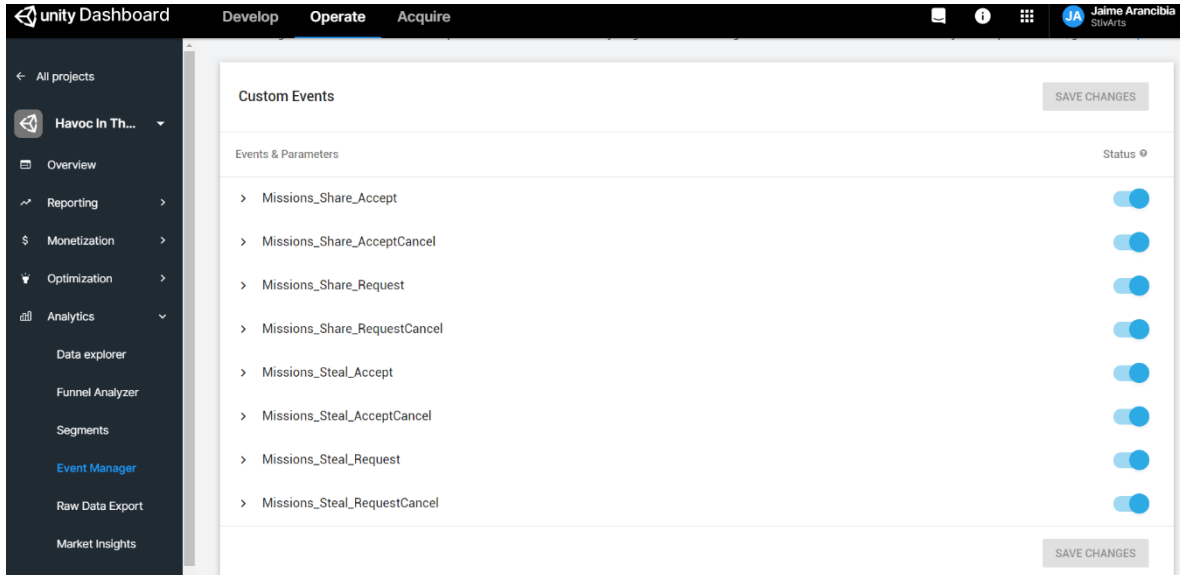


Figura 71. Captura de Unity Dashboard en la pestaña Event Manager.

## 7.7. Creación de enemigos

Ya que los enemigos comparten muchas características en común, se decidió crear una clase padre “Enemy” de la cual heredan todos los enemigos. Esta clase contiene

## 7.8. Arte del videojuego

### 7.8.1. Personaje principal

El diseño del personaje principal pasó por varias iteraciones, esto ya que a medida que el juego evolucionaba, algunos elementos artísticos se volvían obsoletos. En un principio, el personaje tendría el aspecto del enemigo lanzador, sin embargo no resultó ser un diseño atractivo para el usuario.

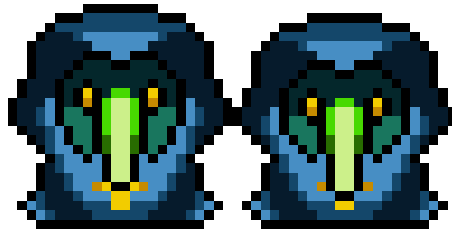


Figura 72. Diseño inicial del personaje principal.

En la siguiente iteración, se procuró crear un personaje con el que el jugador pudiese empatizar, por lo que debía tener un aspecto más antropomorfo.



Figura 73. Segunda versión del personaje principal.

El problema de esta versión es que sus proporciones le hacían ver demasiado pequeño respecto al tamaño de una baldosa, esto causaba que las colisiones con los enemigos o las paredes se viesen irreales y costaba más identificar al personaje dentro de la escena.

Finalmente, se desarrolló una tercera versión que mejorara los defectos de la anterior. Este diseño posee una nueva proporción entre su cabeza y cuerpo, lo que le hacía ver más grande y se podía entender mejor los detalles del personaje.



Figura 74. Versión final del personaje principal.

Todos los sprites fueron desarrollados con el software Aseprite. Para crear las animaciones se hizo uso de las funciones internas del programa, como las capas, rotación inteligente, papel cebolla<sup>11</sup>, etc.

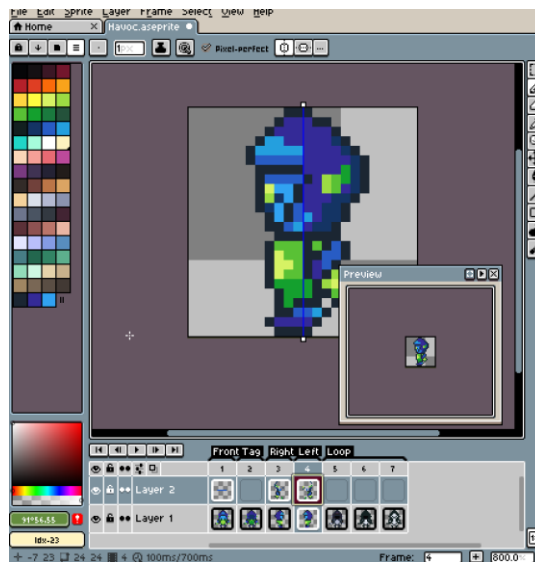


Figura 75. Captura de pantalla de Aseprite.

Una vez que el tercer diseño se definió como el final, se procedió a realizar distintos trabajos de artwork para uso en la tienda de aplicaciones y portales donde se haga publicidad acerca del juego.



Figura 76. Artwork de Havoc para su uso en el marketing del juego.

---

<sup>11</sup> El papel cebolla, en el ámbito de ilustración digital, es una herramienta que permite visualizar los fotogramas de una animación mediante capas superpuestas unas con otras.

### 7.8.2. Arte de enemigos y obstáculos

Para los enemigos del videojuego o los obstáculos se realizó un proceso similar al del personaje principal, aunque con menos iteraciones, ya que no es necesario buscar la empatía del jugador, sino que se comprenda su mecánica y sea identificable como objeto peligroso.

En general, el tiempo necesario para crear el arte de un enemigo no fue más de 3 horas ya que primero se definieron las mecánicas del enemigo y por tanto ya se contaba con una idea de cómo luciría. Aquellos enemigos que tienen más animaciones fueron los que requirieron más tiempo.

### 7.8.3. Iluminación

Se instaló el paquete Lightweight Rendering Pipeline al proyecto de Unity, lo que permite utilizar un sistema de iluminación en 2D. La iluminación del juego está basada completamente en este sistema, ya que el jugador posee una iluminación de Sprite que solo permite visualizar los elementos en un rango a su alrededor, así mismo, algunos objetos, como los enemigos o elementos decorativos, emiten su propia luz, lo que es una ayuda visual para el jugador al permitir identificar fácilmente el tipo de objeto que aparece en pantalla.

## 7.9. Enemigos

En esta sección se describe la implementación de los enemigos diseñados en el punto 6.7.1 en concreto la manera en que se programaron sus habilidades.

### 7.9.1. Lanzador

El enemigo lanzador, llamado internamente “Wizard” consiste en un objeto con el componente *En\_Wizard* que cada cierto período de tiempo es capaz de instanciar un objeto *Gun*, el cual se mueve en una dirección y velocidad constante definidas por el desarrollador.

En cuanto a su uso en el editor, es posible ajustar el intervalo de disparo, la dirección en la que dispara y el prefab que se instanciará.

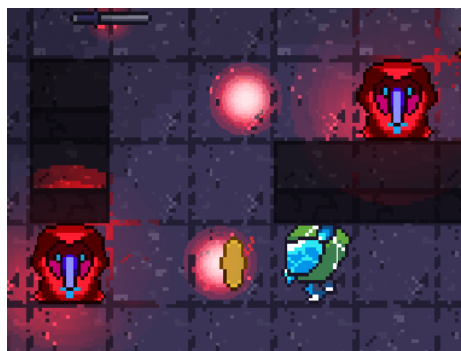


Figura 77. Enemigos lanzadores en la partida.

```

public class En_Wizard : Enemy
{
    private void OnDrawGizmos()
    {
        Gizmos.DrawIcon(transform.position + new Vector3(0.5f, -0.5f) +
            (Vector3)DirectionStatic.GetDirection(gunDirection), "Arrow" +
            gunDirection.ToString());
    }
    public GameObject gunPrefab;
    public int interval = 2;
    public Direction gunDirection = Direction.Right;
    private Vector2 gunVectorDirection;

    private void Awake()
    {
        gunVectorDirection = DirectionStatic.GetDirection(gunDirection);
    }
    void Start()
    {
        InvokeRepeating("Shoot", 0, interval);
    }

    void Shoot()
    {
        if (gunPrefab != null && GameManager.Instance.playing)
        {
            Gun gun = Instantiate(gunPrefab, transform.position +
                new Vector3(0.5f, -0.5f), Quaternion.identity).GetComponent<Gun>();
            gun.StartMovement(gunVectorDirection);
        }
    }
}

```

Figura 78. Código del componente *En\_Wizard*.

### 7.9.2. Slime y Volador

Tanto el enemigo Slime como el Volador poseen características muy similares, ya que ambos consisten en un enemigo que se mueve en un patrón definido de manera constante.

Por ello, ambos tipos de enemigos poseen el mismo código y se diferencian solo en el aspecto gráfico y en la configuración de velocidad de cada uno.

Consisten en un objeto con el componente *Enemy* y un componente *LoopMovement* el cual permite elegir la distancia que se recorrerá en los ejes horizontal y vertical y la velocidad en que lo harán. En el caso del Slime, la velocidad es menor y el patrón de movimiento siempre



se hará dentro de baldosas no vacías. El enemigo volador posee una velocidad mayor y puede moverse por las baldosas vacías.



Figura 79. Enemigo Volador en la partida.



Figura 80. Mazmorra repleta de enemigos Slime.

```
public class LoopMovement : MonoBehaviour
{
    private void OnDrawGizmosSelected()
    {
        Gizmos.DrawIcon(transform.position +
            (Vector3Int)movement, "ArrowDown");
    }
    public Vector2Int movement;
    public float time = 3;
    void Start()
    {
        transform.DOMove(transform.position +
            (Vector3Int)movement, time).SetLoops(-1, LoopType.Yoyo).SetEase(Ease.Linear);
    }
}
```

Figura 81. Código del componente LoopMovement.

### 7.9.3. Bastión

El enemigo de tipo bastión posee un componente llamado *En\_Bastion*, el cual invoca de manera periódica el método *Scan()* que consiste en llamar al método *OverlapBox()* de las físicas 2D de Unity. Si se detecta al jugador dentro del rango de consulta, se instancia un objeto que le indicará al jugador dónde se moverá el personaje y pasados 2 segundos hará la traslación hacia la ubicación marcada.



Figura 82. Enemigo Bastión a punto de moverse hacia la ubicación del personaje principal.

```

void Scan()
{
    sprite.DOColor(Color.white, 0.3f);
    animator.Play("SearchingBastion");
    if (Vector2.Distance(transform.position,
        GameManager.Instance.player.transform.position) < 5)
    {
        radar.Play();
    }
    Collider2D col = Physics2D.OverlapBox(transform.position,
        Vector2.one * 3, 0, playerLayer);
    if (col != null)
    {
        if (col.CompareTag("Player"))
        {
            GameObject aim = Instantiate(aimPrefab, col.transform.position +
                new Vector3(0.5f, -0.5f, 0),
                Quaternion.identity, transform.parent);
            aim.transform.DOScale(0.1f, 2f).OnComplete(() =>
            {
                GetComponent<Collider2D>().enabled = false;
                Destroy(aim);
                MoveToAim(aim.transform.position);
            });
            return;
        }
    }
    sprite.DOColor(Color.black, 0.3f).SetDelay(0.65f);
}

void MoveToAim(Vector3 pos)
{
    slide.Play();
    transform.DOMove(pos, 0.3f).OnComplete(() =>
    {
        GetComponent<Collider2D>().enabled = true;
        sprite.DOColor(Color.white, 0.3f);
    });
}

```

Figura 83. Métodos Scan() y MoveToAim() del componente En\_Bastion.

#### 7.9.4. Fantasma

El enemigo de tipo fantasma consiste en un objeto con el componente *Enemy* y un componente *CircleMovement*, se decidió separar la lógica de estos dos componentes ya que podría reutilizarse para enemigos que se desarrollen en el futuro.

*CircleMovement* genera un movimiento circular en bucle a una velocidad, distancia del centro y dirección definida por el desarrollador.



Figura 84. Enemigo Fantasma dentro de una partida, rodeando al jugador.

```

public class CircleMovement : MonoBehaviour
{
    public float radius = 10f;
    public float speed = 1f;
    public bool offsetIsCenter = true;
    public Vector3 offset;
    private Vector3 initialPosition;
    public bool randomDirection = false;

    void Start()
    {
        initialPosition = transform.position;
        if (offsetIsCenter)
        {
            offset = transform.position;
        }
        if (randomDirection)
        {
            speed *= Random.Range(0, 100) < 50 ? 1 : -1;
        }
    }

    void Update()
    {
        transform.position = new Vector3(
            (radius * Mathf.Cos(Time.time * speed)) + offset.x,
            (radius * Mathf.Sin(Time.time * speed)) + offset.y,
            offset.z);
    }
}

```

Figura 85. Código del componente CircleMovement.

### 7.9.5. Estático

El enemigo estático fue muy simple en cuanto a su implementación. Consiste solamente en un GameObject con el componente Enemy (y el resto de los componentes necesarios para que este funcione) y que provoca el fin de la partida si el jugador colisiona con él. Luego, se cambia el Sprite para diferenciar a los enemigos.



Figura 86. Sprite del enemigo Llave maldita.

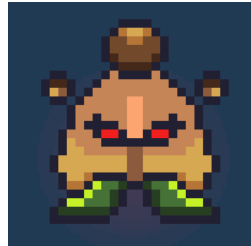


Figura 87. Sprite del enemigo Uglyton.

### 7.9.6. Ninjarrior (adicional)

Posterior al lanzamiento, se agregó el enemigo Ninjarrior, que no estaba en el diseño inicial del videojuego. Su habilidad consiste en lo siguiente:

Si el jugador pasa por su lado, el enemigo activará una alerta por un segundo antes de embestir hacia la dirección donde se encuentra el personaje principal. El objetivo de crear este enemigo es que el jugador deba moverse rápidamente y estar atento antes que sea embestido.



Figura 88. Mazmorra repleta de enemigos Ninjarrior.

```

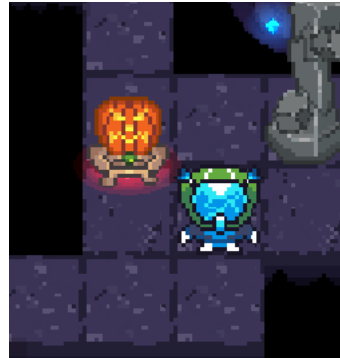
public class En_Ninja : Enemy
{
    public int distance;
    public LayerMask playerLayer;
    public SpriteRenderer kanji;
    public AudioSource clang;
    public AudioSource attack;
    bool moving = false;
    void Start()
    {
        animator = GetComponent<Animator>();
        GameManager.Instance.onStep += InvokeSeek;
        kanji.color = Color.clear;
    }
    void InvokeSeek()
    {
        Invoke("Seek", 0.1f);
    }
    void Seek()
    {
        if (!moving)
        {
            RaycastHit2D ray = Physics2D.Raycast(transform.position,
            Vector2.right, distance, playerLayer);
            if (ray.collider != null)
            {
                Attack();
            }
        }
    }
    void Attack()
    {
        moving = true;
        clang.Play();
        CancelInvoke("Seek");
        kanji.color = Color.white;
        kanji.transform.localScale = Vector3.one * 0.2f;
        kanji.transform.DOScale(Vector3.one, 0.3f);
        transform.DOMoveX(transform.position.x +
        distance, 0.45f).SetEase(Ease.InCubic).SetDelay(0.5f).OnStart(() =>
        {
            attack.PlayScheduled(0.1f);
            animator.Play("Ninja Run");
            kanji.DOFade(0, 0.2f);
            kanji.transform.DOScale(Vector3.one * 0.2f, 0.2f);
        }).OnComplete(() =>
        {
            moving = false;
            animator.Play("Ninja Idle");
            distance *= -1;
            GetComponent<SpriteRenderer>().flipX = distance < 0;
        });
    }
    private void OnDestroy()
    {
        GameManager.Instance.onStep -= InvokeSeek;
    }
}

```

Figura 89. Código del componente En\_Ninja.

### 7.9.7. Pumpkummy (adicional)

En una actualización del juego, se agregó el enemigo Pumpkummy, cuya habilidad consiste en moverse verticalmente hacia un punto y luego reaparecer en su posición inicial luego de unos pocos segundos.



*Figura 90. Enemigo Pumpkummy en la partida.*



```

public class En_Pumkummy : Enemy
{
    [Range(-5, 5)]
    public int distance = -2;
    [Range(0.1f, 3)]
    public float duration = 1;
    Vector2 initialPosition;
    public GameObject particle;

    void Start()
    {
        initialPosition = transform.position;
        animator = GetComponent<Animator>();
        Move();
    }
    public void Move()
    {
        animator.Play("Run");
        GetComponent<Collider2D>().enabled = true;
        transform.DOMoveY(transform.position.y +
            distance, duration).OnComplete(() =>
        {
            animator.Play("Disappear");

            GetComponent<Collider2D>().enabled = false;
            Invoke("Grow", 0.8f);
        });
    }
    public void Grow()
    {
        transform.position = initialPosition;
        animator.Play("Grow");
    }
}

```

Figura 91. Código del componente En\_Pumpkummy.

## 7.10. Sonido

La mayor parte del sonido del videojuego es creación propia, a excepción de algunos archivos que se han tomado de librerías públicas gratuitas y que se han modificado para adaptarse al videojuego.

El software utilizado para la creación de sonido fue exclusivamente el DAW<sup>12</sup> FL Studio 20®.

Hay un total de 20 efectos de sonido y 2 temas musicales incorporados dentro del juego.

### 7.10.1. Efectos de sonido

A continuación se listan los efectos creados, su propósito y autoría.

*CP = Creación propia.*

*RF = Modificación de sonido royalty free.*

*H = Híbrido entre CP y RF.*

Nombre de efecto	Utilización en el juego (Objeto en el que se usa)	Autoría
BastionScan	Habilidad de escanear al personaje principal (Enemigo Bastión).	CP
BastionSlide	Movimiento del enemigo hacia el personaje principal (Enemigo Bastión).	RF
Coin	Recoger moneda (Moneda y Cofre).	CP
Death	Muerte del personaje principal (Personaje principal).	CP
DoorBreak	Abrir puerta con llave (Puerta).	RF
ElectricZap	Colisión de personaje con bola de energía (Bola de energía)	H
EventAccomplished	Al completar un evento de juego (SoundManager)	CP
Falling	Al caer por una baldosa vacía (Personaje principal)	CP
FireBlast	Función del actuador de fuego (Actuador Fuego)	H
Grass	Romper una baldosa de césped (Baldosa césped)	RF
Gun	Función del actuador de blast (Actuador Blast)	RF
MenuButton	Presionar un botón (UI)	CP

<sup>12</sup> Abreviatura para Digital Audio Workstation.

NinjaAttack	Movimiento del enemigo Ninjarrior (Enemigo Ninjarrior)	H
OnCollected	Recoger una llave (Llave)	CP
Portal	Colisión con portal (Portal)	CP
Radar	Radar del enemigo bastión (Enemigo Bastión)	CP
Save	Tocar el tótem de guardado (Totem de guardado)	CP
Steps	Dar un paso (Personaje principal)	CP
SwordClang	Pre-movimiento del enemigo Ninjarrior (Enemigo Ninjarrior)	RF
Error	Error del juego (SoundManager)	CP

Figura 92. Lista de efectos de sonido creados.

### 7.10.2. Música

Respecto a la creación de música, se generaron dos archivos diferentes, sin embargo, ambos son parte de un mismo proceso de producción.



Figura 93. Captura del proceso de producción de la música del juego.

Se desarrolló una pista de música que puede ser reproducida en bucle sin que esto sea muy evidente. Existe una pista de percusiones que acompaña a una melodía y armonía principal que fue exportada en un archivo diferente.

Durante la partida del juego se escucha de manera permanente y en bucle la pista sin las percusiones. Cuando se activa un evento de juego, la pista de percusiones se activa y suena de manera sincronizada con la música principal del juego.

### **7.10.3. Consideraciones especiales**

Debido a que durante todo el juego se escucha una pista de audio constantemente, es necesario que, mientras el jugador no esté en una partida del gameplay central, se aplique un efecto de low-pass<sup>13</sup> a la pista para reducir su presencia.

## **7.11. Navegación**

En esta sección, se describe como se implementó el flowchart del videojuego, indicando cómo acceder a las distintas secciones del videojuego.

La aplicación inicializa con un splashscreen que muestra un artwork del videojuego. Una vez que ha cargado, se accede al menú principal del videojuego, desde aquí es posible acceder a los siguientes menús:

### **7.11.1. Gameplay central**

En esta pantalla se ejecutan las partidas del videojuego. Incluye los botones de movimiento y el botón para acceder al menú de pausa.

Incluye un contador de pasos avanzados en la partida y el listado de llaves obtenidas

#### **7.11.1.1. Menú de pausa**

El menú de pausa consiste en un panel que se muestra sobre la pantalla de gameplay central. Contiene un contador de pasos, monedas y estado de la misión (si es que hay una activa). Además, incluye un botón para terminar la partida y volver al menú principal y otro para acceder al menú de configuración.

---

<sup>13</sup> Una ecualización low-pass es aquella que solo deja pasar a las frecuencias más bajas.

### **7.11.2. Configuración**

Consiste en un panel que contiene un editor del nombre de usuario y un toggle<sup>14</sup> para activar o desactivar el sonido del juego.

### **7.11.3. Lista de misiones**

El panel de misiones (ver Figura 63) abre un cuadro donde, en forma de cuadrícula, se muestran todas las misiones disponibles para aceptar. Al seleccionar una misión, se abre un sub-panel para ajustar los detalles de la misión y/o aceptarla (ver Figura 65. Panel de ajuste de misión. Modo Share.).

### **7.11.4. Recompensas**

El panel de recompensas es un panel donde se muestra un listado en forma de cuadrícula con todos los ítems que el jugador tiene disponible para reclamar. Si no hay ítems disponibles, se visualiza un texto que indica al jugador la situación.

### **7.11.5. Ranking semanal**

El ranking semanal es un panel en el que se muestra un listado ordenado con las clasificaciones de mayor número de pasos avanzados por los jugadores. Se incluye un indicador del rango alcanzado por el jugador que hace la consulta.

## **7.12. PlayFab**

En este apartado se describe cómo fue implementado el servicio de Azure PlayFab dentro del proyecto, exceptuando las partes que se hayan mencionado en apartados anteriores.

### **7.12.1. Sincronización de datos**

Como el videojuego requiere el guardado de los datos del jugador, en específico el inventario, se implementó un sistema de sincronización con el servidor de PlayFab que verifica los datos que se guardan en el juego tanto en tiempo real como los que se hayan almacenado previamente de manera local en el dispositivo del jugador.

Si bien PlayFab dispone de un sistema de inventario, este resulta demasiado complejo para un juego que no requiere la conexión constante del videojuego a internet. Por tanto, se decidió utilizar guardar los datos en el Player Data del jugador como un JSON. Esto último permite que el contenido guardado sea idéntico tanto local como remotamente (una cadena de caracteres).

---

<sup>14</sup> Un toggle es un componente de interfaz gráfica que al ser accionado cambia entre dos estados.

Dentro del videojuego, existe un script llamado *DataManager* que es capaz de almacenar los datos del jugador y comunicarlos al GameManager. Estos datos pueden ser convertidos fácilmente en una cadena de caracteres en formato JSON o al contrario, convertir una cadena de caracteres recibida (desde el servidor) en los datos del jugador.

Por temas de seguridad, los datos del jugador solo pueden ser modificados desde el cliente que inicia sesión con su identificador de jugador único. Ningún jugador puede modificar los datos de otro.

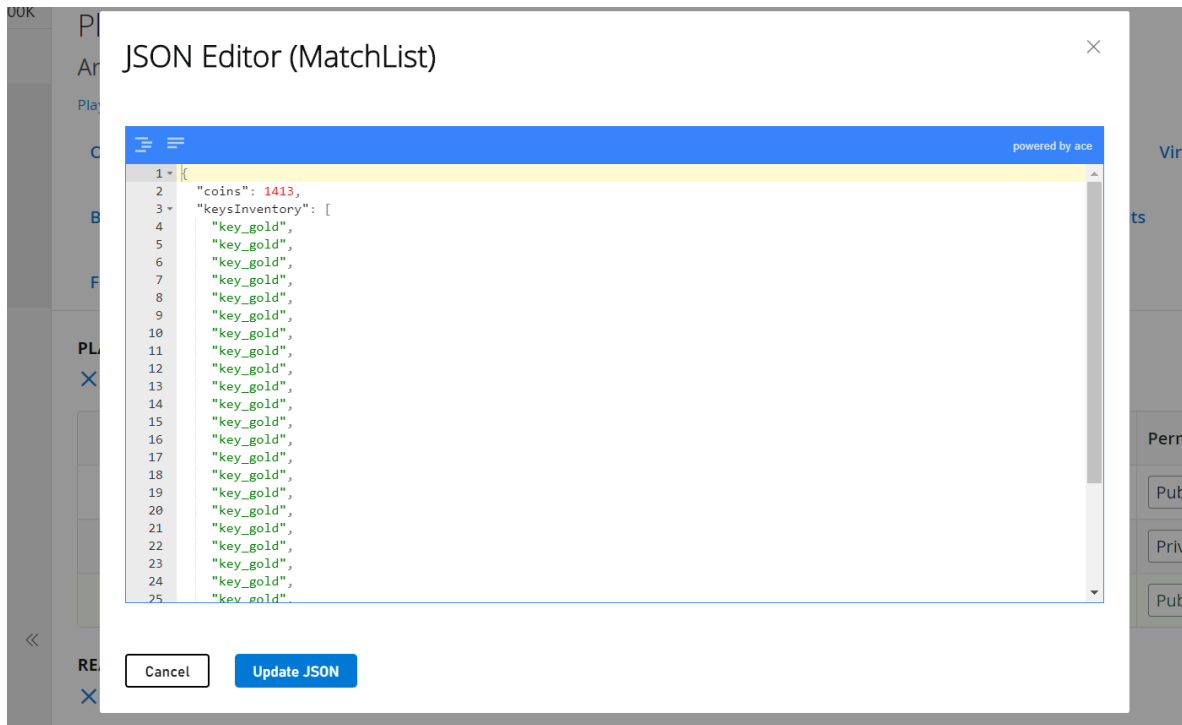


Figura 94. Editor de JSON en los datos del jugador (PlayFab).

```

[Serializable]
public class PlayerData
{
    public int coins;
    public List<string> keysInventory = new List<string>();
}
[Serializable]
public class DataManager
{
    public PlayerData data;
    public void GetUserData(bool online = true)
    {
        if (online)
        {
            GetUserDataRequest request = new GetUserDataRequest();

            PlayFabClientAPI.GetUserData(request, (result) =>
            {
                Debug.Log("Got user data:");
                if ((result.Data == null) || (result.Data.Count == 0))
                {
                    Debug.Log("No user data available");
                }
                else
                {
                    string jsonData = result.Data["MatchList"].Value;
                    data = JsonUtility.FromJson<PlayerData>(jsonData);
                }
            }, (error) =>
            {
                Debug.Log("Got error retrieving user data:");
                Debug.Log(error.ErrorMessage);

                string jsonData = PlayerPrefs.GetString("playerData");
                data = JsonUtility.FromJson<PlayerData>(jsonData);
            });
        }
        else
        {
            string jsonData = PlayerPrefs.GetString("playerData");
            data = JsonUtility.FromJson<PlayerData>(jsonData);
        }
    }
}

```

```

public void UpdateData()
{
    string json = JsonUtility.ToJson(data);
    Debug.Log(json);

    Dictionary<string, string> playerData = new Dictionary<string, string>();
    playerData.Add("MatchList", json);
    UpdateUserDataRequest rq = new UpdateUserDataRequest();
    rq.Data = playerData;
    rq.Permission = UserDataPermission.Public;
    try
    {
        PlayFabClientAPI.UpdateUserData(rq, OnAddDataSuccess, OnAddDataError);
    }
    catch (Exception e)
    {
        Debug.Log(e);
    }
    PlayerPrefs.SetString("playerData", json);
}
void OnAddDataSuccess(UpdateUserDataResult _thisResult)
{
    GameManager.Instance.onUpdateData.Invoke();
}
void OnAddDataError(PlayFabError _thisErrorResult)
{
    Debug.LogWarning("Got an error: " + _thisErrorResult.ErrorMessage);
}
}

```

Figura 95. Código del componente DataManager.

### 7.12.2. CloudScript

La mayoría de los métodos que hacen uso del servicio de PlayFab lo hacen mediante la ejecución de CloudScript, por lo que es posible modificar la manera en que se procesan distintas funciones del juego desde la plataforma de administración de PlayFab, fuera del motor del juego.

En el siguiente código se observa que al ejecutar un CloudScript, se envía un conjunto de datos y se espera una respuesta de manera asíncrona, la cual puede ser recibida como un JSON.



```

public class UI_VersionPanel : MonoBehaviour
{
    CanvasGroup canvasGroup;
    void Start()
    {
        canvasGroup = GetComponent<CanvasGroup>();
        canvasGroup.alpha = 0;
        GameManager.Instance.onLogin += () =>
        {
            PlayFabClientAPI.ExecuteCloudScript(new ExecuteCloudScriptRequest()
            {
                FunctionName = "getVersion",
                GeneratePlayStreamEvent = true
            }, resultCallback =>
            {
                LitJson.JsonData jsonData = MissionManager.ResultToJSON(resultCallback);
                if (jsonData["version"] != null)
                {
                    string version = (string)jsonData["version"];
                    if (!Application.version.Equals(version))
                    {
                        canvasGroup.DOFade(1, 0.4f);
                    }
                }
            }, errorCallback => { });
        }
    }
}

```

Figura 96. Código del componente UI\_VersionPanel.

```

90
91 handlers.completeMission = function (args,context){
92     var request = {
93         PlayFabId: args.PlayerId, Statistics: [{
94             StatisticName: args.type,
95             Value: 0
96         }]
97     };
98     server.UpdatePlayerStatistics(request);
99
100
101     server.UpdateUserData({
102         PlayFabId: args.PlayerId,
103         Data: {
104             MissionRequest: "",
105             MissionResult: args.coins
106         }
107     });
108 }
109
110
111 handlers.updateStepsRecord = function (args,context){
112     var request = {
113         PlayFabId: currentPlayerId, Statistics: [{
114             StatisticName: "StepsRecord",
115             Value: args.steps
116         }]
117     };
118     server.UpdatePlayerStatistics(request);
119 }
120
121
122
123 handlers.getVersion = function(args,context){
124     var data = server.GetTitleData({
125         Keys:["Version"]
126     });
127     gameversion = data.Data["Version"];
128     return{version : gameversion};
129 }

```

Figura 97. Parte de una revisión del CloudScript del videojuego en PlayFab.

### 7.12.3. Noticias

Se utilizó el sistema de noticias de PlayFab para comunicar al jugador sobre el contenido de la actualización más reciente. Se planea utilizar también este servicio para informar sobre nuevos eventos del juego, como nuevos modos, personajes, ofertas en la tienda etc.



Figura 98. Captura del menú principal con una noticia sobre la actualización más reciente.

#### TITLE NEWS CONTENT

Timestamp (UTC) \*


August 23, 2020 3:14 AM  

Status \*

Published 

#### TITLE AND BODY

Language \*

English 

Title \*

The Pumpkummy Update

Body \*

 JSON

```
{
  "body": "This hybrid between a pumpkin and a
mummy is waiting for you...
This and much more in update <color=yellow>0.40!
</color>",
  "url": "http://sorgardteam.com/wp-
content/uploads/2020/08/Pumpkummy.jpg"
}
```

Figura 99. Creación de una noticia en el portal de administración de PlayFab.

## 8. Publicación

Luego de 6 meses de desarrollo, se publicó la primera versión de Havoc In The Dungeon en la Play Store. El proceso, si bien estuvo exento de complicaciones, requirió la creación de diversos archivos como las capturas de pantalla (que además son editadas para lucir mejor en la ficha de Play Store), las políticas de privacidad, el ícono, un video promocional y la descripción larga y corta en tres idiomas diferentes.

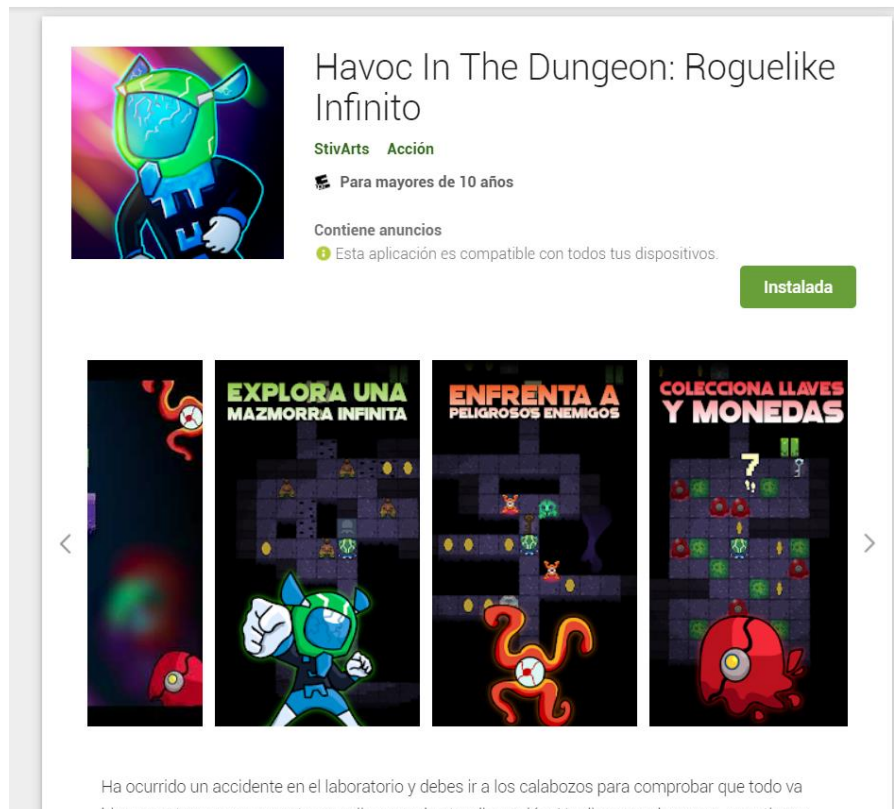


Figura 100. Captura de la ficha de Play Store del videojuego.

El tiempo que tardó la aprobación de la aplicación fue de 5 días, en los que el equipo técnico de Google revisó la aplicación y la ficha que se publicaría.

Inicialmente, se publicó el juego en una Beta Abierta, ya que así podría recibir los comentarios de los jugadores para mejorar el juego sin afectar la valoración general del juego. Luego de dos versiones en las que se arreglaron pequeños errores, se lanzó oficialmente la versión de producción del videojuego.

La clasificación de edad se establece de acuerdo con un extenso cuestionario de la consola de Google Play, dependiendo del sistema de clasificación, se obtuvieron distintos ratings.



Figura 101. Clasificaciones de contenido del videojuego.

Adicionalmente, se creó una página del juego en la plataforma Itch.io, muy popular en lo que videojuegos indie respecta, esto para atraer al público que estuviese en búsqueda de videojuegos de desarrolladores independientes.

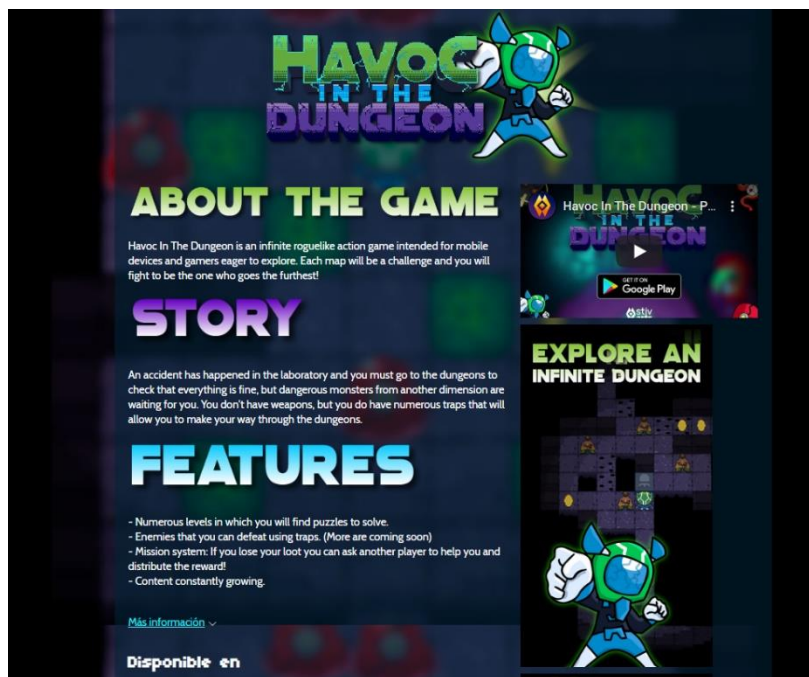


Figura 102. Captura de la página de Havoc In The Dungeon en Itch.io.

## 9. Resultados

En esta sección se comentarán los resultados del proyecto, tanto a los del videojuego en la tienda de aplicaciones como la recopilación de datos generados en el sistema de misiones.

### 9.1. Resultados del videojuego

El videojuego fue publicado el día 7 de agosto de 2020 en su versión Beta, hasta el día 31 de agosto se crearon 9 actualizaciones en Play Store.

Cuenta con 13 usuario activos, es decir, usuarios que instalaron la aplicación y han encendido su dispositivo móvil en los últimos 7 días.

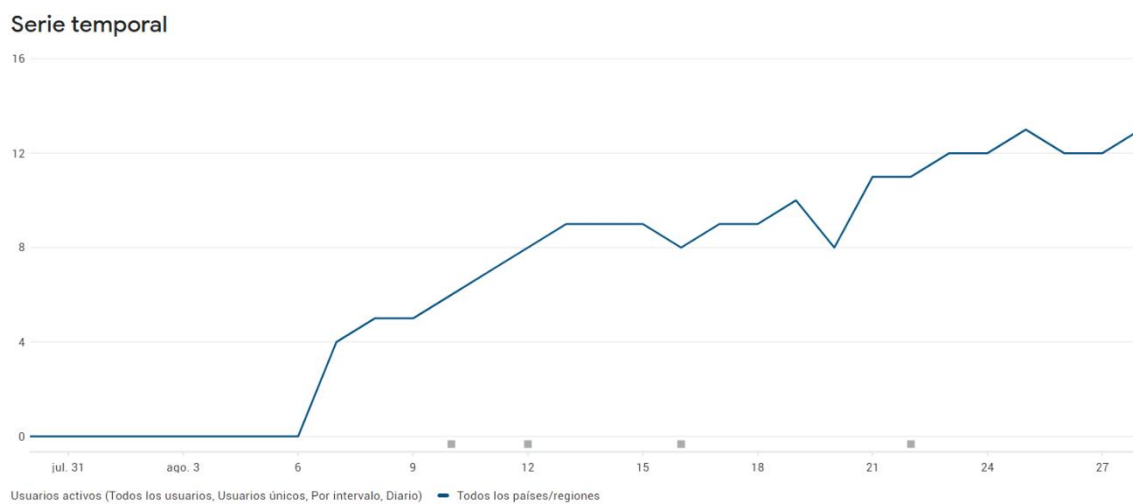


Figura 103. Gráfico de usuarios activos.

El peak de adquisición de usuarios fue el día del lanzamiento, con 10 usuarios, sin embargo, ese mismo día hubo una pérdida de 6 usuarios.

### Serie temporal

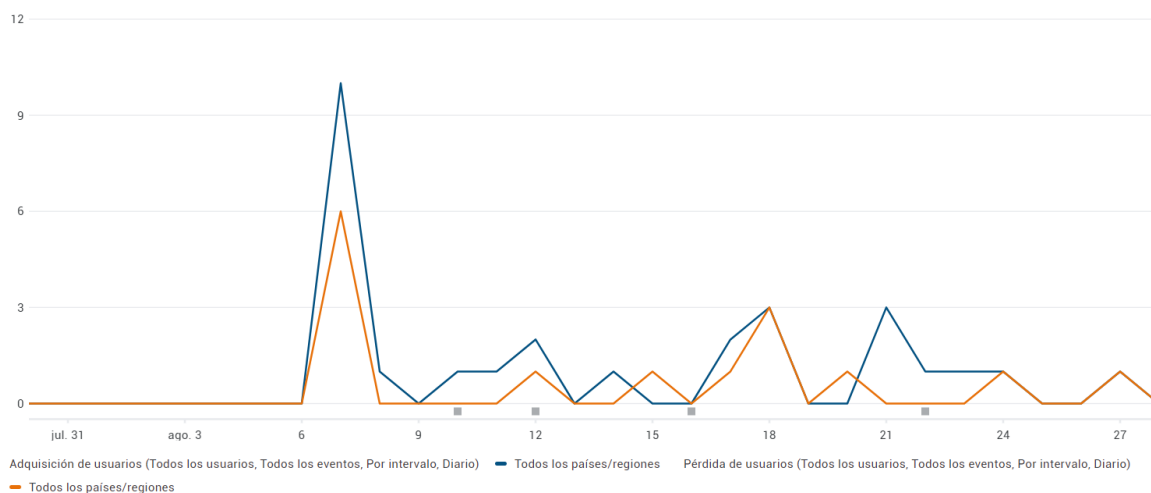


Figura 104. Gráfico de adquisición de usuarios v/s pérdida de usuarios.

En cuanto al rendimiento en la página del juego en Play Store se observa un total de 86 visitantes, es decir, usuarios que vieron la página sin tener la aplicación instalada en ninguno de sus dispositivos. De estos, 21 usuarios instalaron la aplicación, lo que implica una tasa de conversión de 24,42%.

### Análisis de Play Store

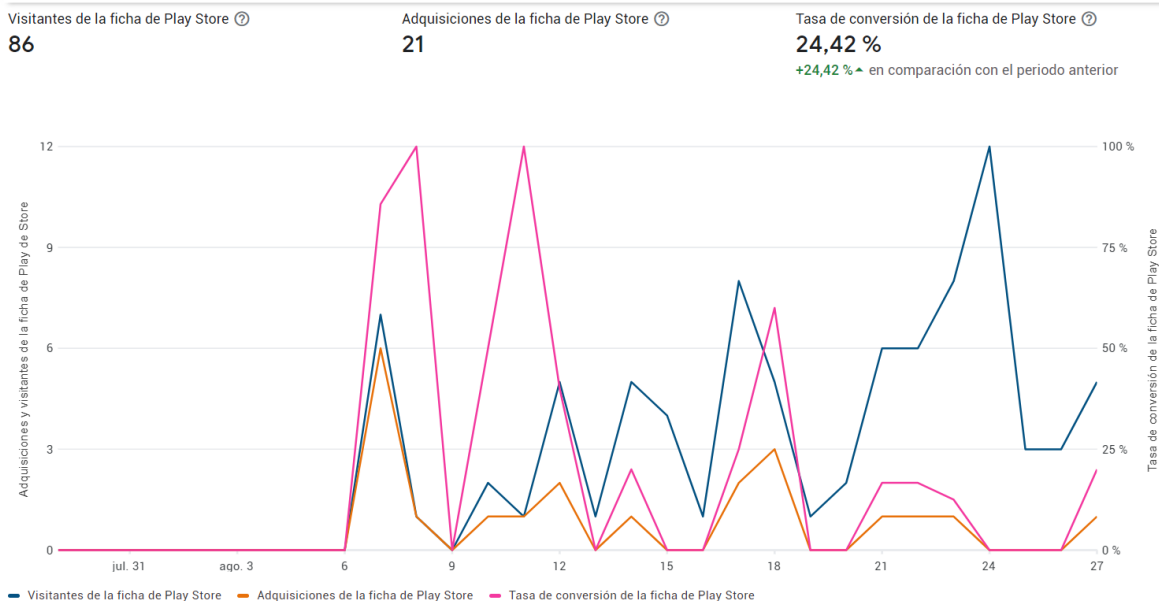


Figura 105. Gráfico de rendimiento de la ficha de Play Store.

En cuanto a los datos estadísticos obtenidos desde PlayFab, se observa lo siguiente:



La duración promedio de tiempo del jugador viendo la ejecución del juego (screentime) es de 1 minuto con 25 segundos.

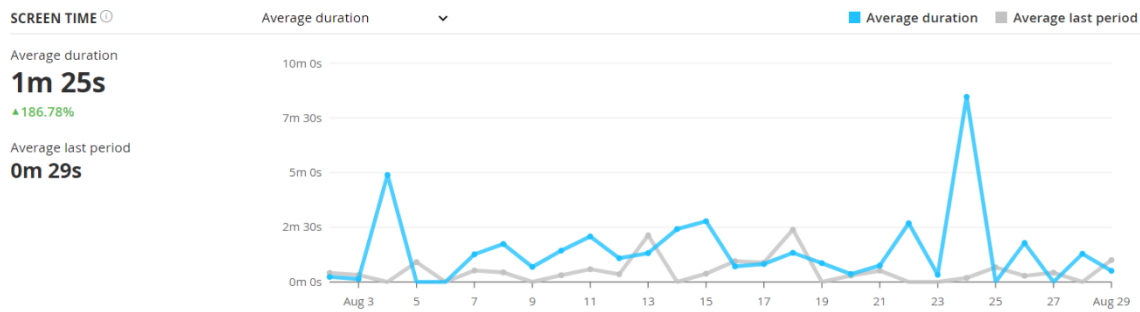


Figura 106. Duración promedio de Screentime.

Se inició sesión 422 veces en las primeras 4 semanas del juego, con un total de 61 usuarios únicos. Es necesario mencionar que muchos de estos usuarios representan los dispositivos que Google utiliza para probar la aplicación antes de aprobarla y no son jugadores reales ni activos del videojuego.

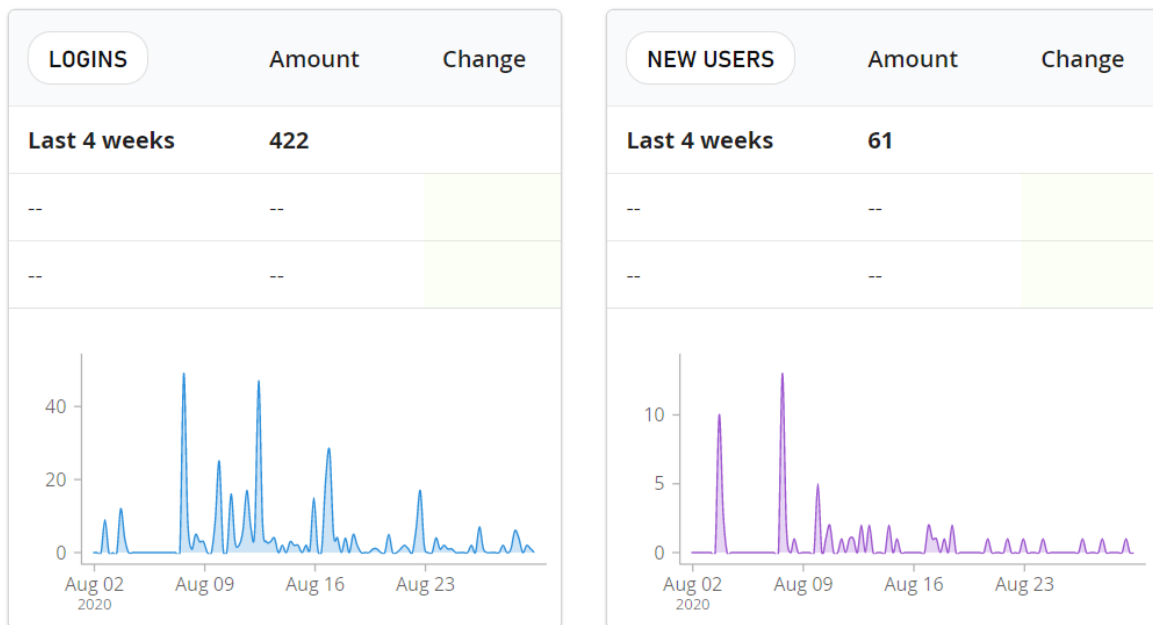


Figura 107. Cantidad de inicios de sesión y usuarios nuevos en 4 semanas.

## 9.2. Datos recopilados

Se presentan los datos que fueron recopilados desde el servicio de Unity Analytics sobre los eventos del sistema de misiones.

Estos eventos son los siguientes:

Modo Share	Modo Steal
Missions_Share_Accept	Missions_Steal_Accept
Missions_Share_AcceptCancel	Missions_Steal_AcceptCancel
Missions_Share_Request	Missions_Steal_Request
Missions_Share_RequestCancel	Missions_Steal_RequestCancel

Figura 108. Eventos enviados a Unity Analytics.

Los eventos guardan datos, los que describen la situación en la que se encontraba el jugador al momento de activarse el evento. Los datos guardados son los siguientes:

time_elapsed	Tiempo transcurrido desde que se visualizan los datos de la misión.
steps	Pasos necesarios para cumplir la misión.
shared_coins	Monedas que obtendrá el jugador que hará el rescate.
coins	Monedas que obtendrá el jugador que será rescatado.
rate	Porcentaje de repartición de monedas $\frac{\text{coins}}{\text{coins} + \text{shared\_coins}}$ .

Figura 109. Datos guardados en cada evento.

Se han elegido estos datos ya que representan el contexto del juego al momento de aceptar o cancelar una misión.

En el caso de *time\_elapse*, se podría evaluar en qué casos el jugador puede tardar más o menos en tomar una decisión, esto al vincularse directamente con *rate* o con *steps*.

El parámetro *rate* permite obtener la relación entre monedas que recibirá quien solicita la misión y quien la cumple, tanto si fue ajustado por un jugador u otro. Este parámetro se puede comparar con la cantidad de pasos (*steps*) que representarían la dificultad de la misión o con el total de monedas a repartir. En el primer caso se podría averiguar una tendencia de los jugadores a tomar más o menos parte de la recompensa según el esfuerzo que deban realizar y al analizar esos datos ajustarlos para maximizar la interacción o algún parámetro del juego que los desarrolladores deseen modificar. En el segundo caso, se podría evaluar qué tanto valoran los jugadores la recompensa recibida y si al ser cantidades muy bajas o grandes prefieran quedarse con una cantidad de esta.

El significado literal de los parámetros no es necesario que se traduzcan a otro videojuego que pueda ocupar un sistema similar, ya que pueden existir diferentes tipos de recompensa, dificultad y/o riesgo.

A continuación, se presenta un ejemplo de cómo se podrían relacionar ciertos datos.

Para el evento Missions\_Share\_Accept, que es enviado por un jugador que acepta una misión en el modo Share, se presentan los siguientes datos:

Desviación estándar	6.65	11.82	1.72	0.13	4.03	5.56
Promedio	8.38	22.86	4.57	0.54	6.71	11.29
	time_elapsed	steps	shared_coins	rate	coins	monedas totales
	4.45	15	2	0.33	1	3
	3.01	47	3	0.40	2	5
	11.19	26	4	0.60	6	10
	21.52	20	7	0.59	10	17
	9.82	17	5	0.62	8	13
	6.02	24	6	0.57	8	14
	2.65	11	5	0.71	12	17

Figura 110. Datos recopilados del evento Missions\_Share\_Accept

Sería posible analizar si existe una relación entre el total de monedas en juego con el ratio de repartición.

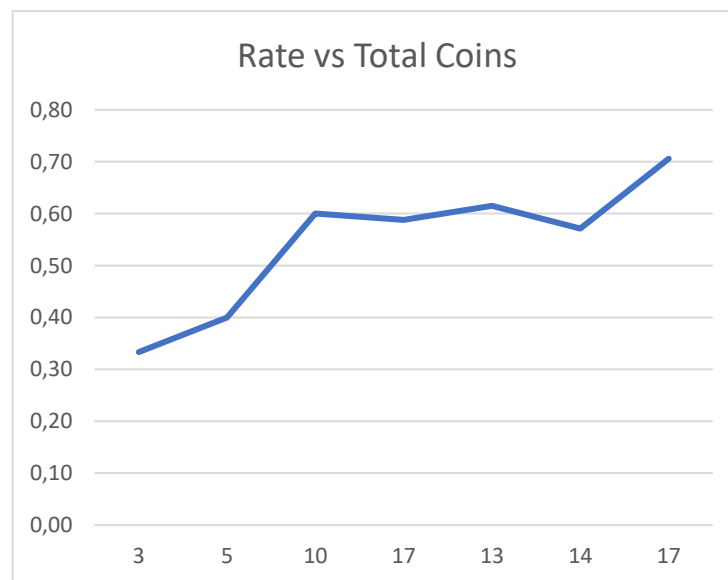


Figura 111. Gráfico Rate v/s Monedas Totales.

Por la cantidad de datos recopilados, no es posible llegar a conclusiones válidas, sin embargo, en un proyecto que logre extraer una masa de datos, sería posible observar si existe una tendencia o si hay rangos en los que los jugadores se comporten de manera diferente.

Otra observación interesante sería comparar estos datos con los del evento `Mission_Steal_Accept` ya que en el modo `Steal` el usuario que decide el valor `rate` es quien acepta la misión.

Desviación estándar	98.38	31.35	9.68	0.10	8.82	13.16
Promedio	86.84	50.86	10.57	0.39	11.86	22.43
	time_elapsed	steps	shared_coins	rate	coins	monedas totales
	7.5	25	1	0.2	2	3
3	133.	61	5	0.4	6	11
	8.8	10	8	0.4	9	17
1	138.	49	1	0.3	1	21
	31.0	45	3	0.4	2	31
	19.9	57	2	0.4	8	36
3	269.	10	1	0.2	1	38

Figura 112. Datos del evento `Missions_Steal_Accept`.

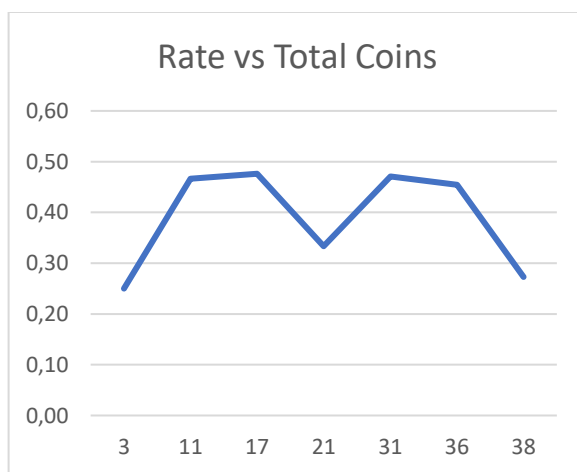


Figura 113. Gráfico de `Rate vs Total Coins` en `Missions_Steal_Accept`.

## 10. Conclusiones

### *10.1. Sobre el desarrollo del videojuego*

El producto final cumple con el diseño inicial del juego, se lograron desarrollar todas las mecánicas que estaban planificadas e incluso se crearon nuevas durante la parte final del proceso. Gracias a la manera en que se estructura el videojuego, el videojuego es altamente expandible, lo que permite agregar nuevo contenido constantemente sin tener que iterar demasiado en procesos anteriores ni tener que modificar demasiados aspectos del juego.

En los procesos de ideación y pre-producción se realizó la aplicación de la teoría y práctica aprendida durante los cursos de la carrera universitaria y se complementaron con los conocimientos adquiridos durante la investigación realizada en este trabajo.

Generalmente, existe un espacio de tiempo entre que el producto finaliza su desarrollo y es lanzado al público, pero en este caso, como era necesario recopilar resultados, se lanzó el producto apenas se terminó la producción.

Una estrategia para haber alcanzado una mayor masa de usuarios es haber invertido tiempo en la aplicación de un buen ASO<sup>15</sup>, sin embargo, los datos recopilados posiblemente hubieran sido aún menores al retrasar el lanzamiento del producto.

### *10.2. Análisis de datos recopilados*

Como fue previsto, la cantidad de datos recopilados es muy baja, sin embargo, si se ha logrado extraer información del jugador respecto al sistema de misiones en un experimento A/B.

Del total de la masa de usuarios (60) que realmente jugaron, solo un 23,3% (14) utilizó el sistema de misiones, por lo que es necesario hacer de este sistema algo más visible y entendible por el usuario para que aproveche sus beneficios, además, ya que las monedas no tienen una utilidad real dentro del juego, los usuarios podrían perder interés en recuperarlas, pero esto es solo una observación especulativa. No es posible obtener resultados conclusivos, sin embargo, si se aplicara este método como instrumento en un videojuego con un alcance más masivo, sería posible relacionar los datos para analizar el comportamiento de los jugadores y automatizar el juego para maximizar las ganancias y/o interacciones.

Dada la manera en que se construyó el juego, es posible agregar nuevos eventos fácilmente en todo tipo de interacciones del jugador y sistemas de juego, por ejemplo, se podría registrar si un jugador que aceptó una misión logró completarla y vincular el evento con un índice de

---

<sup>15</sup> Abreviatura de App Store Optimization, el cual es un proceso para optimizar la ficha de una aplicación logrando un mayor alcance orgánico de usuarios.

éxito según otros factores como el total de pasos, lista de mazmorras que superó, tiempo, etc. Estos nuevos eventos podrían ayudar a reajustar valores para un mejor balance del juego, maximización de compras dentro de la aplicación, detectar errores en el diseño de nivel, etc. Es por ello por lo que se realizó este trabajo, para crear un videojuego que además contara con un instrumento que permitiera recopilar información importante para ajustar el diseño del videojuego y que en otros proyectos pueda ser utilizado como referencia independiente de los sistemas de juego que se diseñen.

## 11. Trabajo futuro

Havoc In The Dungeon es un videojuego que aún tiene una gran cantidad de contenido que ofrecer, por tanto, se planea seguir trabajando en él para mejorarlo y llegar a más público potencial.

Como primer paso, se suspenderá el sistema de misiones, ya que para funcionar correctamente es necesario que haya muchos usuarios diarios activos. A cambio de este sistema, se creará la tienda, en ella se podrán adquirir nuevos personajes que se diferenciarán según su aspecto y habilidades únicas. Esto sirve como incentivo para coleccionar monedas que hasta la fecha no tenían un uso dentro del juego.

Además, para fomentar la rejugabilidad, se agregará un sistema de medallas que premie a los jugadores por cumplir ciertos objetivos dentro del juego, como alcanzar una cantidad de pasos, derrotar un número grande de enemigos o sobrevivir en la mazmorra durante varios minutos. Estas medallas se exhibirán en el perfil del jugador, que podrá ser compartido con los otros mediante la implementación de un sistema que permita agregar amigos y una función para compartir el progreso en redes sociales.

Respecto a las mazmorras y enemigos, se agregará más del doble del contenido que hay actualmente, incluyendo nuevas mecánicas para el jugador y los enemigos.

Una vez que el juego esté listo para una gran actualización, se publicará no solo en Play Store, sino que, en otras tiendas de aplicaciones como App Store, Galaxy Store, AppGallery, etc.

Si se logra un éxito moderado, se considerará llevar el videojuego más allá de los dispositivos móviles y crear una versión para otras plataformas, o incluso crear un nuevo juego a modo de secuela. Si no hay buenos resultados, se aplicará un sistema automatizado que mantenga el juego activo sin la atención completa del desarrollador para poder crear nuevos juegos en el futuro.

## 12. Bibliografía

- [1] R. Hunicke, M. Leblanc y R. Zubek, «MDA: A Formal Approach to Game Design and Game Research,» 2004.
- [2] T. Abbott, «MDA Framework- Unconnected Connectivity,» Gamasutra, 12 Diciembre 2010. [En línea]. Available: [https://www.gamasutra.com/blogs/TuckerAbbott/20101212/88611/MDA\\_Framework\\_Unconnected\\_Connectivity.php](https://www.gamasutra.com/blogs/TuckerAbbott/20101212/88611/MDA_Framework_Unconnected_Connectivity.php). [Último acceso: 2020 Agosto 2020].
- [3] Tokio School, «TokioSchool.com,» [En línea]. Available: <https://www.tokioschool.com/noticias/que-son-dinamicas-videojuegos/>. [Último acceso: 11 Junio 2020].
- [4] D. Williams, N. Yee y S. E. Caplan, «Who plays, how much, and why? Debunking the stereotypical gamer profile,» *Journal of Computer-Mediated Communication*, vol. 13, nº 4, p. 993–1018, 2008.
- [5] F. Waldner y M. Zsifkovits, «Are service-based business models of the video game industry blueprints for the music industry?,» *International Journal of Services Economics and Management*, vol. 5, nº 1, pp. 5-20, 2013.
- [6] J. Turner, A. Scheller-Wolf y S. Tayur, «Scheduling of Dynamic In-Game Advertising,» *OPERATIONS RESEARCH*, vol. 59, nº 1, pp. 1-16, 2011.
- [7] T. Rayna y L. Striukova, «Few to Many': Change of Business Model Paradigm in the Video Game Industry,» *Communications & Strategies*, nº 94, pp. 61-81,154-155, 2014.
- [8] P. Chantepie, L. Michaud, L. Simon y P. Zackariasson, «Introduction The rebound of videogame industry,» *Communications & Strategies*, nº 94, pp. 9-15,151-152, 2014.
- [9] J. Antón, «EL MARKETING EN LOS VIDEOJUEGOS: ANÁLISIS DEL SECTOR, ACEPTACIÓN DE LOS GAMERS Y BENEFICIOS DEL USO DE VIDEOJUEGOS COMO MEDIO DE DIFUSIÓN DE CAMPAÑAS DE MARKETING,» Madrid, 2014.
- [10] The Entertainment Software Association (ESA), 2020 ESSENTIAL FACTS About the Computer and Video Game Industry, 2020.



- [11] Entertainment Software Association (ESA), 2015 ESSENTIAL FACTS ABOUT THE COMPUTER AND VIDEO GAME INDUSTRY, 2015.
- [12] Newzoo, 2019. [En línea]. Available: <https://newzoo.com/insights/articles/women-account-for-46-of-all-game-enthusiasts-watching-game-video-content-and-esports-has-changed-how-women-and-men-alike-engage-with-games/>.
- [13] S. Keane, «cnet,» 17 Agosto 2020. [En línea]. Available: <https://www.cnet.com/news/ghost-of-tsushima-will-add-co-op-multiplayer-for-free/?ftag=COS-05-10aaa1e>. [Último acceso: 18 Agosto 2020].
- [14] J. B. Rocha, S. Mascarenhas y R. Prada, «Game mechanics for cooperative games,» *ZON Digital Games*, pp. 72-80, 2008.
- [15] What Games Are, «Synchronous or Asynchronous Gameplay [Definitions],» 6 Agosto 2011. [En línea]. Available: <https://www.whatgamesare.com/2011/08/synchronous-or-asynchronous-definitions.html>. [Último acceso: 19 Agosto 2020].
- [16] Lightspeed, «Games 2.0: Asynchronous gaming,» 17 Agosto 2020. [En línea]. Available: <https://medium.com/@lightspeedvp/games-2-0-asynchronous-gaming-6285ac1e12d3>. [Último acceso: 18 Agosto 2020].
- [17] J. v. Neumann y O. Morgenstern, *Theory of Games and Economic Behavior*, Princeton: Princeton University Press , 1944.
- [18] R. Fiani, «Teoria Dos Jogos,» Elsevier Editora, 2006, p. 35.
- [19] W. Poundstone, *Prisoner's Dilemma: John Von Neumann, Game Theory and the Puzzle of the Bomb*, Nueva York: Doubleday, 1992.
- [20] R. Axelrod, *La evolución de la cooperación : el dilema del prisionero y la teoría de juegos*, Alianza Editorial, S.A., 1986.
- [21] D. G. Ross, «Using cooperative game theory to contribute to strategy research,» *Strat Mgmt J.*, vol. 39, nº 11, pp. 2859-2876, 2018.
- [22] V. Devedžić y S. R. Milenković, «Teaching Agile Software Development: A Case Study,» *IEEE Transactions on Education*, vol. 54, nº 2, pp. 273-278, 2011.

- [23] P. Kruchten, «Contextualizing agile software development,» *Journal of Software: Evolution and Process*, vol. 24, nº 4, pp. 351-361, 2013.
- [24] W. Luton, «Gamastura,» 15 Octubre 2009. [En línea]. Available: [https://www.gamasutra.com/view/feature/132554/making\\_better\\_games\\_through\\_.php](https://www.gamasutra.com/view/feature/132554/making_better_games_through_.php). [Último acceso: 15 julio 2020].
- [25] E. Bethke, *Game Development and Production*, Texas: Wordware Publishing, 2003.
- [26] M. R. Jakob Nielsen, «Heuristic evaluation of user interfaces,» de *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, Seattle, Washington, USA, 1990.
- [27] Mozilla Developer Network, «MDN Web Ddocs,» 12 Agosto 2020. [En línea]. Available: <https://developer.mozilla.org/en-US/docs/Games/Techniques/Tilemaps>.