UNIVERSIDAD CATÓLICA DE CUENCA

UNIVERSIDAD CATÓLICA DE CUENCA

Comunidad Educativa al Servicio del Pueblo

FACULTAD DE INFORMÁTICA, CIENCIAS DE LA COMPUTACIÓN E INNOVACIÓN TECNOLÓGICA

CARRERA DE REALIDAD VIRTUAL Y VIDEOJUEGOS

Desarrollo de la base técnica mediante programación de sistemas y mecánicas en caja blanca para Inkfall

PROYECTO DE INTEGRACIÓN CURRICULAR PREVIO A LA OBTENCIÓN DEL TÍTULO DE INGENIERO EN REALIDAD VIRTUAL Y VIDEOJUEGOS

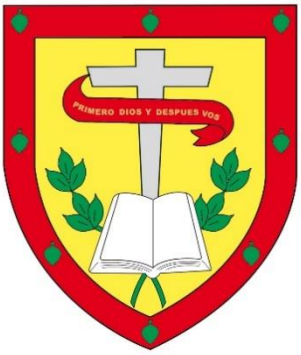
AUTOR: JOAN ALEXIS DUTÁN TORRES

DIRECTOR: ING. JOHN LUIS ESTRADA GUAYTA, M.S.

CUENCA - ECUADOR

2026

DIOS, PATRIA, CULTURA Y DESARROLLO



UNIVERSIDAD CATÓLICA DE CUENCA

Comunidad Educativa al Servicio del Pueblo

FACULTAD DE INFORMÁTICA, CIENCIAS DE LA COMPUTACIÓN E INNOVACIÓN TECNOLÓGICA

CARRERA DE REALIDAD VIRTUAL Y VIDEOJUEGOS

Desarrollo de la base técnica mediante programación de sistemas y mecánicas en caja blanca para Inkfall

PROYECTO DE INTEGRACIÓN CURRICULAR PREVIO A LA OBTENCIÓN DEL TÍTULO DE INGENIERO EN REALIDAD VIRTUAL Y VIDEOJUEGOS

AUTOR: JOAN ALEXIS DUTÁN TORRES

DIRECTOR: ING. JOHN LUIS ESTRADA GUAYTA, M.S.

CUENCA – ECUADOR

2026

DIOS, PATRIA, CULTURA Y DESARROLLO

Universidad Católica de Cuenca

DECLARATORIA DE AUTORÍA Y RESPONSABILIDAD

Declaratoria de Autoría y Responsabilidad

Joan Alexis Dután Torres portador(a) de la cédula de ciudadanía Nº **1726333055**. Declaro ser el autor de la obra: **"Desarrollo de la base técnica mediante programación de sistemas y mecánicas en caja blanca para Inkfall."**, sobre la cual me hago responsable sobre las opiniones, versiones e ideas expresadas. Declaro que la misma ha sido elaborada respetando los derechos de propiedad intelectual de terceros y eximo a la Universidad Católica de Cuenca sobre cualquier reclamación que pudiera existir al respecto. Declaro finalmente que mi obra ha sido realizada cumpliendo con todos los requisitos legales, éticos y bioéticos de investigación, que la misma no incumple con la normativa nacional e internacional en el área específica de investigación, sobre la que también me responsabilizo y eximo a la Universidad Católica de Cuenca de toda reclamación al respecto.

Cuenca, **31 de agosto de 2026**

F: .....

Joan Alexis Dután Torres

C.I. **1726333055**

Universidad Católica de Cuenca

CERTIFICADO

Certificado

Certifico que la presente Propuesta Tecnológica fue desarrollado por **Joan Alexis Dután Torres** portador(a) de la cedula de ciudadanía N.º **1726333055** con el tema "**Desarrollo de la base técnica mediante programación de sistemas y mecánicas en caja blanca para Inkfall**.", bajo mi supervisión.

Cuenca, **31 de agosto de 2026**

F: .....

Ing. John Luis Estrada Guayta, M.S.

Docente - Tutor

Dedicatoria

A mi padre, quien por más de diez años tuvo la paciencia para esperar que encuentre mi vocación.

Agradecimiento

Agradezco al ingeniero John Estrada por su excelente labor como maestro.

Resumen

El presente trabajo tuvo como objetivo desarrollar la base técnica del videojuego Inkfall mediante la programación de sistemas y mecánicas en un entorno de caja blanca utilizando el motor Unity. La investigación surgió de la necesidad de evaluar tempranamente la jugabilidad antes de incorporar los recursos artísticos definitivos, debido a que el desarrollo simultáneo de componentes técnicos y visuales puede dificultar la identificación de problemas relacionados con la interacción, la progresión y la dificultad. Se realizó un estudio aplicado, con enfoque cuantitativo y alcance descriptivo. El prototipo se construyó mediante la implementación progresiva de sistemas de control del jugador, interacción con el entorno, combate, cambio de armas, inteligencia artificial, rompecabezas y progresión dinámica de la dificultad, integrados en dos escenarios funcionales mediante una arquitectura modular. La validación se efectuó con participantes pertenecientes al público objetivo, utilizando un cuestionario estructurado con escala de Likert para evaluar la jugabilidad, la respuesta de los sistemas y la experiencia de uso. Los resultados permitieron identificar dificultades en las mecánicas de movimiento, el cambio de armas y el comportamiento de la cámara. A partir de estas observaciones, se realizaron ajustes en la respuesta de los controles, la transición entre armas y el seguimiento visual del jugador. Las pruebas posteriores comprobaron el funcionamiento de las correcciones y su integración en los escenarios. Se concluye que el prototipo consolidó una base técnica modular y funcional para incorporar posteriormente recursos gráficos, sonoros y narrativos.

Palabras Clave: *Caja blanca, mecánicas de juego, Unity, pruebas de usuario, arquitectura modular.*

Abstract

This study aimed to develop the technical foundation of the Inkfall video game by programming systems and gameplay mechanics within a white-box environment using the Unity game engine. The research arose from the need to evaluate gameplay at an early stage before incorporating the final artistic assets, since the simultaneous development of technical and visual components can hinder the identification of issues related to player interaction, progression, and difficulty. An applied study was conducted using a quantitative approach with a descriptive scope. The prototype was developed through the progressive implementation of player control systems, environmental interaction, combat, weapon switching, artificial intelligence, puzzles, and dynamic difficulty progression, integrated into two functional levels through a modular architecture. Validation was carried out with participants from the target audience using a structured Likert-scale questionnaire to assess gameplay, system responsiveness, and user experience. The results identified difficulties in movement mechanics, weapon switching, and camera behavior. Based on these findings, adjustments were made to control responsiveness, weapon switching, and player camera tracking. Subsequent testing confirmed the effectiveness of these improvements and their successful integration into the levels. It is concluded that the prototype established a modular and functional technical foundation for the subsequent incorporation of graphics, audio, and narrative assets.

Keywords: *White-box, gameplay mechanics, Unity, user testing, modular architecture*

Índice de contenidos

Índice de tablas

Índice de ilustraciones

Índice de gráficos

Introducción

El desarrollo de un videojuego involucra múltiples disciplinas, entre ellas programación, diseño de mecánicas, modelado tridimensional, diseño de niveles, interfaces, audio y narrativa. Debido a esta diversidad de componentes, resulta necesario disponer de una base técnica que permita implementar, integrar y evaluar las mecánicas principales antes de incorporar los recursos artísticos definitivos. En este contexto, el uso de prototipos en caja blanca constituye una estrategia ampliamente utilizada para validar el funcionamiento de los sistemas del juego, detectar problemas de interacción y realizar ajustes durante las primeras etapas del desarrollo.

El presente trabajo tiene como propósito desarrollar la base técnica del videojuego Inkfall mediante la programación de sistemas y mecánicas implementados en un entorno de caja blanca[1] utilizando el motor Unity[2]. Para ello, se desarrollan los sistemas que conforman el funcionamiento principal del prototipo, incluyendo el control del jugador, los sistemas de combate e interacción, la inteligencia artificial de los enemigos, las mecánicas específicas de cada escenario y un sistema de transformación dinámica basada en intentos fallidos. La implementación se realiza mediante una arquitectura modular que facilita la integración de los diferentes componentes y permite futuras ampliaciones durante el desarrollo del videojuego.

---

[1] técnica de desarrollo que permite probar la lógica y funcionamiento interno de un videojuego antes de implementar sus elementos visuales finales.

[2] motor de desarrollo utilizado para crear videojuegos y experiencias interactivas en 2D, 3D y realidad virtual.

El prototipo desarrollado constituye una base funcional sobre la cual podrán incorporarse posteriormente los recursos gráficos, sonoros y narrativos definitivos del proyecto. De esta manera, la validación temprana de los sistemas implementados permite reducir riesgos durante las etapas posteriores de desarrollo y proporciona una estructura técnica organizada para la evolución del videojuego.

A continuación, se presenta una breve descripción de los capítulos que conforman este trabajo:

Capítulo I. Marco referencial: En este capítulo se presenta el contexto de la investigación, la problemática que motiva el desarrollo del proyecto, la formulación del problema, la justificación, los objetivos generales y específicos, el alcance, las delimitaciones, la metodología empleada y los beneficiarios del proyecto.

Capítulo II. Marco teórico: En este capítulo se exponen los fundamentos conceptuales, metodológicos y tecnológicos que sustentan la investigación. Se analizan los principios del prototipado en caja blanca, el desarrollo de videojuegos, las mecánicas de juego, la arquitectura de software, la transformación dinámica basada en intentos fallidos, las herramientas utilizadas y las tecnologías empleadas para la implementación del prototipo.

Capítulo III. Manual de usuario: En este capítulo se describe el funcionamiento del prototipo desde la perspectiva del usuario. Se presentan los requisitos para la ejecución, el proceso de instalación, la configuración inicial, los controles, las mecánicas principales y el modo de utilización de los dos niveles implementados.

Capítulo IV. Manual del programador: En este capítulo se documenta la implementación técnica de los sistemas desarrollados. Se describen la estructura del proyecto, la arquitectura utilizada, la programación de los sistemas principales, la implementación de las mecánicas de ambos niveles, la configuración del sistema de

transformación dinámica basada en intentos fallidos y los principales componentes que conforman la base técnica del videojuego Inkfall.

Capítulo I

1. Marco Referencial

1.1. Contextualización del problema

En el desarrollo de videojuegos, la construcción de una base técnica funcional constituye una etapa fundamental para validar mecánicas, sistemas de interacción y comportamiento del juego antes de iniciar la producción artística. Durante esta fase, los desarrolladores implementan escenarios, inteligencia artificial, sistemas de combate, interacción con objetos, progresión y demás componentes que permiten comprobar la viabilidad del diseño propuesto. Tradicionalmente, los mecanismos de dificultad se implementan mediante la modificación de parámetros cuantificables, como el incremento de la resistencia o el daño de los enemigos, la reducción de recursos disponibles o la aplicación de penalizaciones al jugador. Este enfoque responde a modelos tradicionales de diseño centrados en el equilibrio de sistemas, lo que limita el potencial del medio interactivo al reducir la experiencia del fracaso a un obstáculo funcional, en lugar de integrarlo como un recurso dentro de la lógica del juego [1].

Desde una perspectiva técnica, esta limitación también se refleja en la forma en que se diseñan e implementan los sistemas de videojuegos. En muchos casos, estos sistemas se estructuran a partir de parámetros ajustables que priorizan la modificación de valores sobre cambios en el comportamiento del entorno o en las reglas del juego [2]. Esto dificulta la implementación de propuestas que buscan alterar la lógica del sistema mediante modificaciones en las mecánicas, el escenario o la progresión del juego como respuesta al desempeño del jugador [3].

En este contexto, la construcción de una base técnica funcional constituye una etapa relevante para el desarrollo de videojuegos, debido a que permite comprobar de manera temprana el funcionamiento de los diferentes sistemas que conforman la experiencia jugable. La implementación de componentes como el control del jugador, la interacción con el entorno, el combate, la inteligencia artificial, los rompecabezas y los sistemas de progresión requiere verificar su comportamiento individual y su integración con el resto del juego antes de incorporar los recursos artísticos definitivos. El prototipado permite concentrar inicialmente los esfuerzos en la lógica y el funcionamiento del sistema, facilitando la identificación de problemas y la realización de ajustes durante las primeras etapas del desarrollo [4], [5].

Desde esta perspectiva, el desarrollo en caja blanca constituye una alternativa para representar de manera preliminar los escenarios y componentes del videojuego mediante geometría y recursos visuales básicos, priorizando la programación y el funcionamiento de las mecánicas sobre la presentación visual. Esta estrategia facilita la modificación de los sistemas y escenarios durante las etapas iniciales, permitiendo comprobar la interacción entre las mecánicas y el entorno antes de avanzar hacia una producción artística de mayor complejidad [5]. Por ello, resulta necesario establecer una base técnica que permita implementar, integrar y comprobar progresivamente los principales sistemas de un videojuego dentro de un prototipo funcional.

En este escenario se desarrolla Inkfall, un videojuego que requiere la implementación e integración de diferentes sistemas y mecánicas para conformar una experiencia jugable funcional. La problemática se relaciona con la necesidad de disponer de una base técnica que permita programar y comprobar progresivamente estos componentes mediante un prototipo en caja blanca, incluyendo el control del jugador, la interacción con el entorno, el combate,

el cambio de armas, la inteligencia artificial, los rompecabezas y la progresión de la dificultad. En consecuencia, se plantea el desarrollo de una base técnica mediante la programación de sistemas y mecánicas en caja blanca, con el propósito de establecer una estructura funcional que permita comprobar la integración de los principales componentes del videojuego antes de incorporar los recursos gráficos, sonoros y narrativos definitivos.

1.2. Planteamiento del problema

El desarrollo de un videojuego requiere la integración coordinada de múltiples sistemas que permiten construir la experiencia de juego, entre ellos el control del jugador, la interacción con el entorno, el combate, la inteligencia artificial, los sistemas de progresión y las mecánicas específicas de cada escenario. Cuando estos componentes no son implementados y validados sobre una base técnica organizada durante las primeras etapas del desarrollo, resulta difícil comprobar su funcionamiento conjunto y detectar oportunamente problemas relacionados con la interacción, la progresión o el comportamiento del juego.

Esta situación se origina debido a que, en muchos proyectos, la implementación de los sistemas técnicos se realiza simultáneamente con el desarrollo de los recursos gráficos, sonoros y narrativos. Como consecuencia, la validación de las mecánicas suele depender de escenarios visuales aún incompletos, lo que dificulta aislar problemas propios de la programación y retrasa la identificación de errores relacionados con la lógica del juego. Asimismo, la ausencia de una arquitectura modular limita la posibilidad de incorporar nuevas funcionalidades o modificar las existentes sin afectar otros componentes del sistema.

En el caso del videojuego Inkfall, esta problemática se manifiesta en la necesidad de disponer de una base técnica que permita implementar, integrar y comprobar

progresivamente los principales sistemas del videojuego mediante un prototipo en caja blanca. La ausencia de esta base impediría verificar el funcionamiento conjunto de las mecánicas implementadas, dificultaría la evaluación temprana de la experiencia de juego y aumentaría el riesgo de trasladar errores técnicos a etapas posteriores del desarrollo, donde las modificaciones resultan más complejas y costosas.

Por esta razón, resulta necesario desarrollar una base técnica modular que permita implementar y validar los principales sistemas y mecánicas del videojuego antes de incorporar los recursos artísticos definitivos. Esta base debe facilitar la integración de los diferentes componentes, permitir la evaluación de su funcionamiento mediante prototipos jugables y proporcionar una arquitectura escalable que sirva como fundamento para las siguientes fases del desarrollo de Inkfall.

1.3. Formulación del problema

Con base en el problema, se formula la siguiente pregunta de investigación:

¿De qué manera el desarrollo de la base técnica mediante la programación de sistemas y mecánicas en caja blanca permite implementar y comprobar el funcionamiento de los principales componentes jugables del videojuego Inkfall?

1.4. Justificación

En la actualidad, la implementación de sistemas de dificultad en los videojuegos continúa basándose, en gran medida, en la modificación de parámetros numéricos como la vida de los enemigos, el daño infligido o la disponibilidad de recursos. Aunque estas estrategias permiten ajustar el nivel de desafío, ofrecen un margen limitado para explorar enfoques en los que la dificultad se construya a partir de cambios en las mecánicas y en el

entorno de juego. En este contexto, resulta pertinente investigar alternativas que permitan implementar y validar este tipo de sistemas desde las primeras etapas del desarrollo.

Desde una perspectiva académica, la presente investigación aporta al campo del desarrollo de videojuegos mediante el diseño e implementación de un prototipo funcional en caja blanca que incorpora un sistema de transformación dinámica basada en intentos fallidos. La investigación toma como base el desarrollo iterativo de prototipos, permitiendo validar la implementación de las mecánicas propuestas antes de la incorporación de elementos artísticos o de producción, práctica ampliamente recomendada dentro del diseño de videojuegos [4].

En el ámbito técnico, el desarrollo del prototipo permite analizar la viabilidad de una arquitectura modular para la implementación de un sistema de transformación dinámica basada en intentos fallidos dentro de Unity. La separación de responsabilidades entre los diferentes componentes del sistema facilita la organización del código, la incorporación de nuevas mecánicas y la evolución del prototipo durante las distintas etapas del desarrollo. Asimismo, el empleo de un prototipo en caja blanca favorece la identificación temprana de problemas de diseño y de implementación, reduciendo la complejidad asociada al desarrollo de un videojuego completo [5].

En este sentido, el desarrollo del prototipo funcional de Inkfall constituye una contribución al estudio de sistemas de transformación dinámica basada en intentos fallidos aplicados a videojuegos, al proporcionar una base técnica que permite demostrar la viabilidad de este enfoque mediante una implementación funcional. De esta manera, la investigación se justifica por su aporte al conocimiento sobre el diseño e implementación de mecánicas adaptativas, así como por ofrecer una referencia para futuros proyectos que busquen

desarrollar sistemas de transformación dinámica basada en intentos fallidos desde las primeras fases de producción.

1.5. Objetivos

1.5.1. Objetivo General

Desarrollar y validar la base técnica modular del videojuego Inkfall mediante la programación de sistemas y mecánicas en caja blanca utilizando el motor Unity, integrando el control del jugador, la interacción con el entorno, el combate, el comportamiento de enemigos y la progresión dinámica basada en intentos fallidos, con el fin de construir un prototipo funcional.

1.5.2. Objetivos Específicos

- Diseñar los diagramas técnicos del sistema de juego y la lógica del sistema de progresión basado en intentos fallidos, definiendo la estructura de los módulos, los flujos de interacción y las condiciones de activación y transición entre los distintos niveles de dificultad.
- Programar las mecánicas fundamentales del juego en caja blanca, incluyendo movimiento del jugador, sistemas de interacción, combate y comportamientos de enemigos, asegurando que sean funcionales, testeables y desacoplados de elementos visuales finales.
- Implementar prototipos jugables de dos escenarios, correspondientes a un nivel de tipo shooter[3] en un entorno de laberinto y un nivel de evasión basado en gestión de

[3] género de videojuego centrado en el combate mediante armas a distancia, donde el jugador debe disparar a enemigos u objetivos.

recursos (encendedor), permitiendo validar distintas dinámicas de juego dentro de una misma arquitectura.

- Estructurar una arquitectura modular y escalable, que permita la integración posterior de arte, animaciones, sonido y otros componentes sin modificar la lógica base del sistema.
- Evaluar el funcionamiento de las mecánicas y sistemas implementados mediante pruebas con participantes pertenecientes al público objetivo, con el propósito de identificar dificultades de interacción y aspectos susceptibles de mejora.

1.6. Alcance

El presente trabajo contempla el desarrollo integral de la base técnica del videojuego mediante la programación de sistemas y mecánicas en un entorno de caja blanca. Esto incluye la implementación de la lógica del jugador, sistemas de movimiento, interacción con el entorno, mecánicas de juego específicas, comportamiento de enemigos y el sistema de progresión basado en muertes. Asimismo, se desarrollarán prototipos funcionales de dos escenarios jugables: un nivel de tipo shooter en un entorno de laberinto y un nivel centrado en la evasión y gestión de recursos, donde la mecánica principal gira en torno al mantenimiento de una fuente de luz.

El enfoque del trabajo está orientado a la construcción de sistemas funcionales, coherentes y testeables, priorizando la lógica y el comportamiento del juego sobre su presentación visual. En este sentido, se desarrollarán estructuras modulares que permitan organizar las distintas mecánicas de manera escalable, facilitando su posterior integración con otros componentes del proyecto.

No se incluye dentro del alcance la implementación visual final, el diseño artístico, modelado 3D, animaciones, efectos visuales, interfaz de usuario ni el pulido estético del juego. Tampoco se contempla la producción completa de todos los niveles planteados en la propuesta original, limitándose a la construcción funcional de dos escenarios como casos de prueba para validar las mecánicas desarrolladas. El objetivo es establecer una base técnica sólida que pueda ser utilizada posteriormente por otros miembros del equipo para la integración y finalización del producto.

1.7. Delimitaciones

Tabla 1

Delimitaciones

Tipo de delimitación	Descripción del límite
Espacial	La investigación se desarrolla en el entorno de desarrollo Unity y se orienta a la construcción de la base técnica del videojuego Inkfall, sin considerar procesos externos de producción, publicación o comercialización del videojuego.
Temporal	El desarrollo de la investigación y del prototipo funcional se realiza durante el período correspondiente al trabajo de titulación, considerando únicamente las actividades ejecutadas dentro de dicho intervalo de tiempo.

Tipo de delimitación	Descripción del límite
Tecnológica	La propuesta se implementa utilizando el motor Unity y el lenguaje de programación C#, complementados con Blender para la elaboración de escenarios en caja blanca y con assets de terceros para funcionalidades base como el controlador del jugador y la inteligencia artificial.
Funcional	El prototipo comprende la implementación de los principales sistemas de programación del videojuego, incluyendo movimiento del jugador, interacción con el entorno, combate, inteligencia artificial, rompecabezas, sistema de transformación dinámica basada en intentos fallidos, progresión y diseño de niveles en caja blanca. No incluye el desarrollo de modelos tridimensionales finales, animaciones personalizadas, efectos visuales avanzados, optimización para distribución comercial ni contenidos narrativos completos.
Poblacional	La validación del prototipo se realiza con una muestra de veinte participantes de entre 18 y 24 años, pertenecientes al público objetivo del videojuego, mediante pruebas de uso y aplicación de un cuestionario con escala de Likert.

Tipo de delimitación	Descripción del límite
Metodológica	La investigación corresponde a un estudio aplicado con enfoque cuantitativo, centrado en el desarrollo, implementación y evaluación de la base técnica del videojuego. No contempla la comparación con otros videojuegos ni estudios experimentales orientados a demostrar relaciones de causalidad.
Económica	El desarrollo se realiza empleando principalmente herramientas disponibles para fines académicos, recursos propios y assets previamente adquiridos o con licencias compatibles, sin considerar costos asociados a la producción artística completa o a la comercialización del videojuego.

1.8. Metodología

La presente investigación corresponde a un estudio aplicado con enfoque cuantitativo, orientado al desarrollo y validación de la base técnica del videojuego Inkfall mediante programación en caja blanca utilizando el motor Unity. El trabajo se centra en la implementación de los principales sistemas del videojuego, priorizando la construcción de la lógica de programación y la interacción entre sus componentes sobre el desarrollo de elementos gráficos o audiovisuales definitivos.

Se adopta un diseño no experimental de alcance descriptivo, debido a que no se manipulan variables independientes ni se establecen grupos de comparación. La evaluación del prototipo se orienta a describir el funcionamiento de las mecánicas implementadas y la

percepción de los usuarios durante su interacción con el videojuego. Para ello, se consideran como aspectos de evaluación el funcionamiento de los sistemas implementados, la claridad de las interacciones y la percepción del desafío generado por las mecánicas del prototipo.

Estos aspectos se examinan mediante pruebas de uso y la aplicación de un cuestionario estructurado con escala de Likert a los participantes. La implementación se realizó de manera progresiva, integrando los sistemas de forma modular hasta conformar un prototipo funcional, lo que permitió realizar ajustes sobre los componentes desarrollados durante el proceso de construcción.

1.8.1. Procedimiento

El desarrollo se divide en fases secuenciales, donde cada una construye sobre la anterior hasta lograr un prototipo jugable completo en caja blanca.

1.8.1.1. Implementación del sistema base del jugador

- En esta fase se desarrolla el controlador principal del jugador, incluyendo movimiento, rotación y control de cámara.
- Se implementa un sistema de desplazamiento en primera persona que permita caminar, correr y girar dentro del entorno, asegurando una respuesta fluida y precisa a las entradas del usuario. Además, se establece un sistema básico de colisiones que permita interactuar correctamente con el entorno.

Resultado obtenido: Un controlador funcional que permita al jugador desplazarse libremente dentro de un entorno de prueba, con comportamiento estable y sin errores de colisión.

1.8.1.2. Construcción de escenarios en caja blanca

- Se desarrollan los escenarios básicos de los dos niveles del juego utilizando geometría simple.
- El primer escenario corresponde a un laberinto tipo shooter, compuesto por pasillos, paredes y zonas abiertas. El segundo escenario corresponde a un entorno cerrado tipo casa o edificio, diseñado para la mecánica de evasión basada en la luz.
- Se definen rutas, espacios de interacción y zonas clave donde se desarrollarán las mecánicas principales.

1.8.1.3. Implementación del sistema de interacción

- Se desarrolla un sistema que permita al jugador interactuar con objetos del entorno, como recoger elementos, activar eventos o manipular objetos.
- Este sistema se basa en detección por proximidad o raycast[4], permitiendo identificar objetos interactivos y ejecutar acciones específicas.

1.8.1.4. Implementación del sistema de combate (nivel shooter)

- Se desarrolla el sistema de disparo para el nivel tipo shooter, incluyendo la generación de proyectiles, detección de impactos y respuesta de los enemigos.
- Se implementa la lógica de armas, control de disparo y retroalimentación básica (impactos, eliminación de enemigos).

[4] técnica que lanza una línea invisible para detectar objetos o colisiones en una dirección determinada.

1.8.1.5. Implementación del comportamiento de enemigos

- Se programan los comportamientos básicos de los enemigos para ambos niveles.
- En el nivel shooter, los enemigos deben detectar al jugador, perseguirlo y atacarlo. En el nivel de evasión, el enemigo actúa como una amenaza que responde a condiciones específicas, como la ausencia de luz o la cercanía al jugador.
- Se pueden utilizar máquinas de estado para controlar comportamientos como patrulla, persecución y ataque.

1.8.1.6. Implementación del sistema del encendedor (nivel de evasión)

- Se desarrolla la mecánica central del segundo nivel, basada en la gestión de una fuente de luz.
- Se implementa un sistema donde el encendedor puede apagarse debido a acciones del jugador o eventos del entorno. Además, se incorpora una mecánica de reencendido mediante interacción controlada.
- El sistema incluye condiciones de riesgo cuando el jugador permanece en la oscuridad por demasiado tiempo.

1.8.1.7. Implementación del sistema de progresión basado en muertes

- Se desarrolla un sistema que modifica el estado del juego en función de la cantidad de veces que el jugador ha fallado.
- Este sistema altera condiciones del entorno, comportamiento de enemigos o reglas de interacción, permitiendo generar variaciones en la experiencia sin depender únicamente de cambios numéricos.

1.8.1.8. Integración del loop de juego

- Se integran todos los sistemas desarrollados dentro de un loop funcional que incluya inicio, desarrollo, fallo y reinicio del juego.
- Se asegura que el ciclo de juego sea coherente y que todos los sistemas interactúen correctamente entre sí.

1.8.1.9. Pruebas y refinamiento

Una vez implementados los sistemas y funcionalidades del prototipo, se realizaron pruebas funcionales orientadas a verificar su correcto funcionamiento técnico. Estas pruebas permitieron comprobar que cada sistema respondiera de acuerdo con los requisitos establecidos y que la interacción entre sus diferentes componentes produjera los resultados esperados.

Para la validación se establecieron condiciones iniciales y entradas específicas para cada prueba. Posteriormente, se comparó el comportamiento obtenido con el resultado esperado, registrando el cumplimiento de cada caso. Cuando se identificaron errores o comportamientos diferentes a los previstos, se realizaron ajustes en la programación, configuración de componentes o parámetros del sistema, y posteriormente se repitieron las pruebas para verificar la corrección implementada.

La validación funcional se organizó mediante una matriz de pruebas que permitió documentar sistemáticamente los sistemas evaluados, las condiciones iniciales, las entradas utilizadas, los resultados esperados y obtenidos, así como el estado final de cumplimiento. Esta matriz permitió complementar la evaluación de percepción realizada con los participantes mediante una verificación objetiva del funcionamiento de las funcionalidades implementadas.

Tabla 2

Matriz de pruebas de sistemas principales del juego

Sistema	Condición inicial	Entrada	Resultado esperado	Resultado obtenido	Estado
Movimiento	Jugador en posición inicial	Entrada de movimiento mediante teclado	El jugador se desplaza en la dirección indicada	El jugador se desplazó correctamente en la dirección indicada	Cumple
Salud	Jugador con vida completa	Recibir un ataque enemigo	Los puntos de vida disminuyen según el daño establecido	Los puntos de vida disminuyeron correctamente	Cumple
Muerte	Jugador sin vida	Recibir daño que reduzca la vida a cero	Se activa el evento de muerte y se ejecuta el proceso correspondiente	El evento de muerte se ejecutó correctamente	Cumple
Dificultad	Contador de muertes en 0	El jugador muere	El contador deathCount aumenta en uno	El contador aumentó correctamente	Cumple
Dificultad	Perfil configurado para determinada cantidad de muertes	Alcanzar la cantidad de muertes establecida	Se ejecuta el método Apply() del perfil correspondiente	El perfil se activó y ejecutó su lógica	Cumple

Sistema	Condición inicial	Entrada	Resultado esperado	Resultado obtenido	Estado
Enemigos	Spawner configurado con un prefab	Activación del punto de aparición	Se genera el enemigo asignado	El enemigo correspondiente fue generado	Cumple
Reinicio de nivel	Jugador muerto	Activación del proceso de reinicio	La escena se reinicia y el jugador vuelve al estado inicial	La escena se renició correctamente	Cumple
Interfaz de vida	Jugador con vida máxima	Recibir daño	La interfaz refleja la nueva cantidad de vida	La interfaz se actualizó correctamente	Cumple

1.8.2. Validación del prototipo

La validación del prototipo se realizó mediante sesiones de prueba con veinte participantes de entre 18 y 24 años, pertenecientes al público objetivo del videojuego. Durante estas sesiones, los participantes interactuaron con el prototipo funcional y experimentaron las mecánicas implementadas en los dos escenarios desarrollados. Posteriormente, se aplicó un cuestionario estructurado con escala de Likert, compuesto por cinco afirmaciones orientadas a evaluar la percepción de los principales sistemas y mecánicas implementados en el prototipo.

El instrumento se estructuró en cinco dimensiones de evaluación, correspondientes al control del jugador, sistema de combate y cambio de armas, comportamiento de la cámara, mecánica de gestión de la vela e inteligencia artificial de los enemigos. Cada dimensión fue evaluada mediante una afirmación relacionada directamente con el comportamiento

observable del sistema durante la interacción con el prototipo. El cuestionario utilizó una escala de valoración de cinco puntos, donde 1 corresponde a “Totalmente en desacuerdo”, 2 a “En desacuerdo”, 3 a “Ni de acuerdo ni en desacuerdo”, 4 a “De acuerdo” y 5 a “Totalmente de acuerdo”. El cuestionario completo utilizado durante las sesiones de validación se presenta en los anexos de la investigación.

Para el análisis de la información se contabilizaron las respuestas obtenidas para cada una de las cinco opciones de la escala en cada afirmación. Posteriormente, los resultados se organizaron mediante porcentajes para identificar la tendencia de valoración de los participantes en cada dimensión evaluada. Esta información permitió determinar los sistemas que presentaban una valoración menor en comparación con los demás aspectos del prototipo y que, por tanto, requerían revisión o ajustes técnicos.

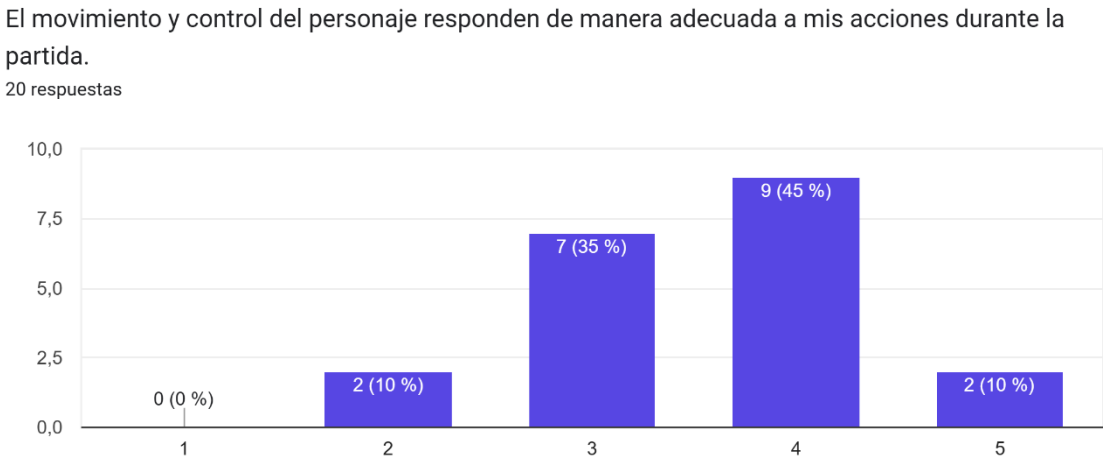
La información recopilada permitió identificar aspectos susceptibles de mejora principalmente relacionados con las mecánicas de movimiento, el cambio de armas y el comportamiento de la cámara. A partir de estas observaciones se realizaron modificaciones orientadas a mejorar la respuesta de los controles, la transición entre armas y el seguimiento visual del jugador.

1.8.2.1. Resultados de la encuesta

Para la evaluación de las funcionalidades del prototipo se empleó una escala de Likert de cinco puntos. El valor 1 corresponde a «Totalmente en desacuerdo», el valor 2 a «En desacuerdo», el valor 3 a «Ni de acuerdo ni en desacuerdo», el valor 4 a «De acuerdo» y el valor 5 a «Totalmente de acuerdo». Esta escala permitió cuantificar el nivel de percepción de los participantes respecto al funcionamiento de las diferentes mecánicas y sistemas evaluados.

Figura 1

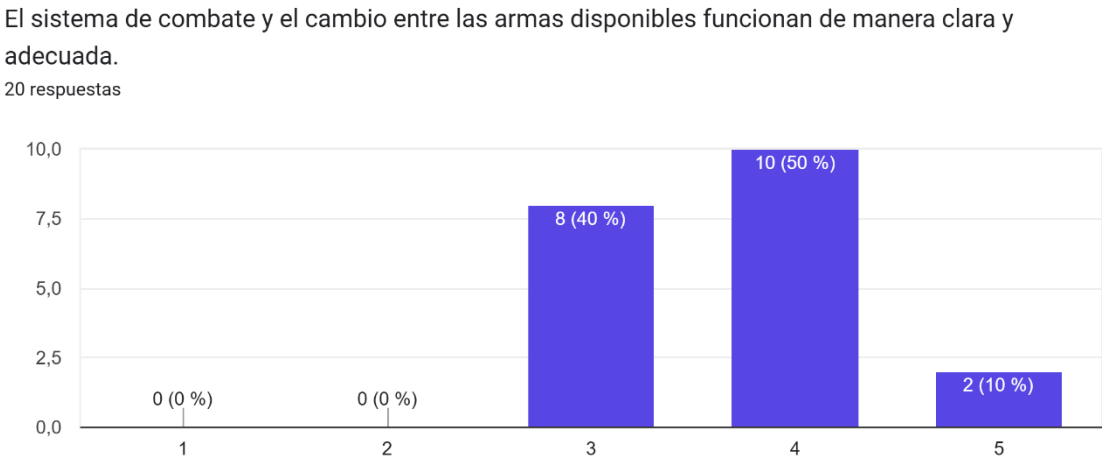
Primera pregunta de la encuesta



La mayor proporción corresponde a "De acuerdo", con un 45 %. Si se agrupan las respuestas favorables (4 y 5), se obtiene un 55 %, mientras que el 35 % de los participantes mantuvo una posición neutral. Esto indica una percepción generalmente favorable del sistema de movimiento y control, aunque existe margen para continuar mejorando su respuesta.

Figura 2

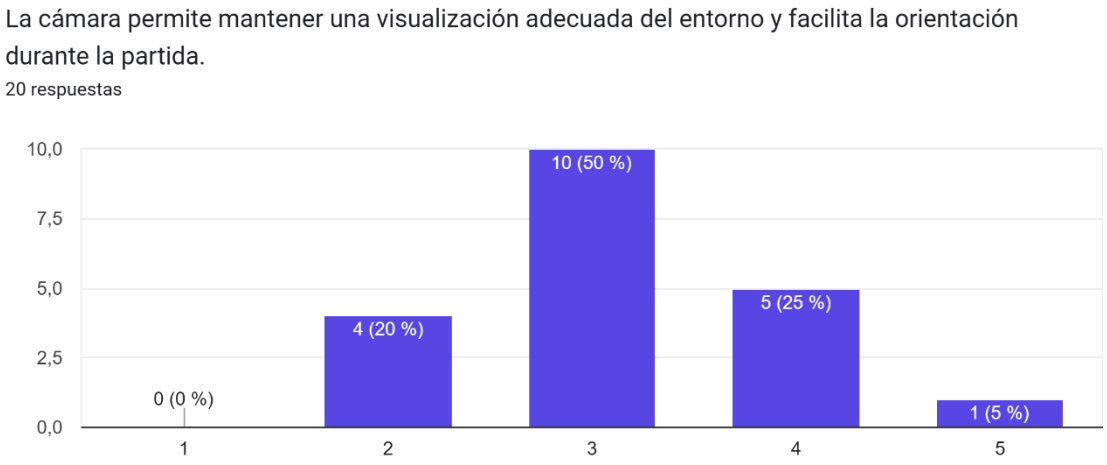
Segunda pregunta de la encuesta



El 60 % de los participantes presentó una valoración favorable del sistema de combate y cambio de armas, correspondiente a las categorías "De acuerdo" y "Totalmente de acuerdo". El 40 % restante presentó una valoración neutral y no se registraron respuestas desfavorables.

Figura 3

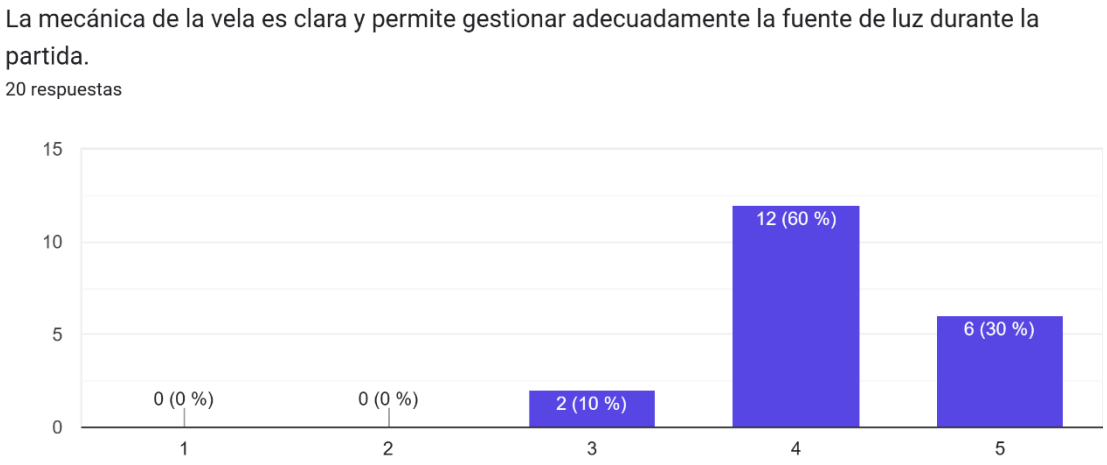
Tercera pregunta de la encuesta



En esta dimensión se observa una valoración más heterogénea. El 50 % de los participantes seleccionó una respuesta neutral, mientras que el 30 % presentó una valoración favorable y el 20 % una valoración desfavorable. Este resultado evidencia que el comportamiento de la cámara constituyó uno de los aspectos con mayor oportunidad de mejora dentro del prototipo.

Figura 4

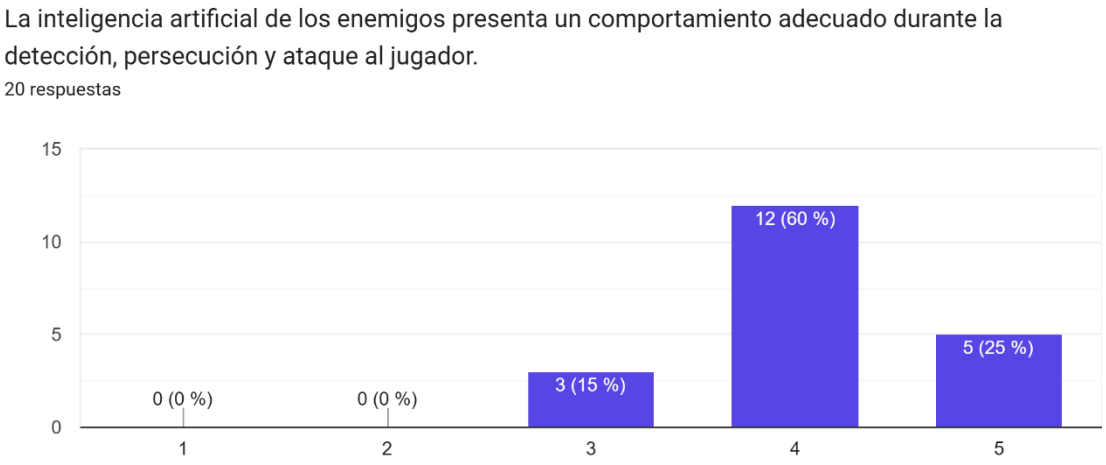
Cuarta pregunta de la encuesta



La mecánica de la vela obtuvo una valoración favorable elevada. El 90 % de los participantes seleccionó las categorías "De acuerdo" o "Totalmente de acuerdo", mientras que únicamente el 10 % presentó una valoración neutral. No se registraron respuestas desfavorables.

Figura 5

Quinta pregunta de la encuesta



La inteligencia artificial presentó una valoración favorable del 85 %, correspondiente a las respuestas "De acuerdo" y "Totalmente de acuerdo". El 15 % restante presentó una valoración neutral y no se registraron respuestas desfavorables.

1.8.2.2. Análisis general de la validación

Los resultados obtenidos muestran una percepción predominantemente favorable respecto al funcionamiento de las mecánicas evaluadas. La mecánica de la vela presentó el mayor nivel de aceptación, con un 90 % de respuestas favorables, seguida por el comportamiento de la inteligencia artificial de los enemigos, con un 85 %. En ambos casos no se registraron respuestas correspondientes a las categorías de desacuerdo.

El sistema de combate y cambio de armas obtuvo un 60 % de respuestas favorables, mientras que el sistema de movimiento y control alcanzó un 55 %. Aunque en ambos casos predominó una valoración positiva, se observó una proporción considerable de respuestas neutrales, correspondiente al 40 % y 35 %, respectivamente. Estos resultados permitieron

identificar oportunidades de mejora relacionadas con la respuesta de los controles y la interacción con las armas.

La cámara presentó el resultado menos favorable entre las dimensiones evaluadas. El 50 % de los participantes seleccionó una valoración neutral, el 30 % presentó una valoración favorable y el 20 % una valoración desfavorable. Este resultado permitió identificar el comportamiento de la cámara como uno de los principales aspectos susceptibles de refinamiento dentro del prototipo.

En términos generales, los resultados muestran que las funcionalidades evaluadas presentaron una aceptación mayoritariamente positiva por parte de los participantes. No obstante, la presencia de respuestas neutrales y desfavorables permitió identificar elementos específicos que requerían ajustes. Estos resultados fueron utilizados como referencia para el refinamiento del prototipo, particularmente en los sistemas de movimiento, combate y cámara.

1.9. Resultados obtenidos

Como resultado del presente trabajo se obtuvo una base técnica modular y funcional para el videojuego Inkfall, desarrollada mediante la programación de sistemas y mecánicas en un entorno de caja blanca utilizando el motor Unity. El prototipo integra correctamente los principales componentes del videojuego, incluyendo el control del jugador, la interacción con el entorno, el sistema de combate, el cambio de armas, la inteligencia artificial de los enemigos, los rompecabezas, la gestión de recursos y el sistema de transformación dinámica basada en intentos fallidos, implementados en dos escenarios jugables que permitan comprobar el funcionamiento conjunto de las mecánicas desarrolladas.

Asimismo, la arquitectura implementada facilita la integración posterior de recursos gráficos, sonoros y narrativos sin requerir modificaciones en la lógica principal del sistema, proporcionando una estructura modular y escalable para la continuidad del desarrollo del videojuego. Finalmente, las pruebas realizadas con participantes pertenecientes al público objetivo permitieron identificar aspectos susceptibles de mejora en la jugabilidad y en el comportamiento de los sistemas implementados, contribuyendo a validar el funcionamiento del prototipo y a realizar los ajustes necesarios antes de avanzar hacia etapas posteriores del desarrollo.

1.10. Beneficiarios

1.10.1. Directos

Los beneficiarios directos corresponden al equipo de desarrollo del videojuego Inkfall, al disponer de una base técnica funcional que integra los principales sistemas de juego implementados mediante programación en caja blanca. Esta base permite contar con una estructura funcional sobre la cual continuar el desarrollo del videojuego, facilitando la incorporación posterior de recursos gráficos, audiovisuales y narrativos. Asimismo, los jugadores que participaron en la validación del prototipo se benefician indirectamente de la evaluación y mejora de la experiencia de juego a partir de sus observaciones y valoraciones.

1.10.2. Indirectos

Los beneficiarios indirectos incluyen a estudiantes y desarrolladores interesados en el desarrollo de videojuegos, quienes pueden tomar como referencia el proceso de implementación y validación de sistemas de juego mediante prototipado en caja blanca. De igual manera, la investigación puede contribuir al ámbito académico al documentar la implementación de sistemas como inteligencia artificial, interacción, combate, progresión y

transformación dinámica basada en intentos fallidos dentro de un prototipo funcional desarrollado en Unity, sirviendo como referencia para futuros proyectos relacionados con la programación y prototipado de videojuegos.

Capítulo II

2. Marco Teórico

2.1. Antecedentes de investigación o soluciones relacionadas

La Tabla 3 describe los aportes realizados al proyecto por diferentes autores.

Tabla 3

Antecedentes de investigación

Autor/fuente	Título	Aporte al proyecto
Zohaib(2018)	Dynamic Difficulty Adjustment (DDA) in Computer Games: A Review	Aporta fundamentos para el desarrollo del sistema de transformación dinámica basada en intentos fallidos de Inkfall, particularmente para la implementación de modificaciones progresivas en las mecánicas, el comportamiento de los enemigos y las condiciones del entorno en función del desempeño del jugador [6].
Ang y Mitchell (2017)	Comparing Effects of Dynamic Difficulty Adjustment Systems on Video Game Experience	Sirve como referencia para la validación del sistema de dificultad implementado en Inkfall, destacando la importancia de evaluar mediante usuarios cómo los mecanismos de ajuste dinámico influyen en la percepción del desafío y en la experiencia de juego [7].

Autor/fuente	Título	Aporte al proyecto
Borg et al. (2020)	Video Game Development in a Rush: A Survey of the Global Game Jam Participants	Sustenta la importancia del desarrollo de prototipos funcionales y de la realización de pruebas durante las primeras etapas de producción. Esto se relaciona con la programación en caja blanca utilizada en Inkfall para construir y validar la funcionalidad del juego antes de incorporar los elementos visuales definitivos [8].
Marklund et al. (2019)	What Empirically Based Research Tells Us About Game Development	Aporta una perspectiva sobre el desarrollo de videojuegos como un proceso técnico compuesto por múltiples actividades y componentes. Su aporte se relaciona con la implementación progresiva y modular de los sistemas de Inkfall, integrando componentes como el jugador, interacción, combate, inteligencia artificial y progresión [9].
Khalifa et al. (2019)	Intentional Computational Level Design	Aporta una referencia sobre la relación entre el diseño de niveles y las mecánicas de juego. En Inkfall, este enfoque se refleja en la construcción de escenarios en caja blanca y en la distribución de enemigos, armas, trampas, puntos de aparición y rutas de progresión para facilitar la integración y validación de las mecánicas implementadas [10].

2.2. Base teórica

El desarrollo de videojuegos comprende un proceso de creación en el que se integran componentes de programación, diseño de mecánicas, interacción y construcción de escenarios para generar experiencias digitales interactivas. Dentro de este proceso, el prototipado permite comprobar de manera temprana las ideas y mecánicas planteadas antes de avanzar hacia una implementación completa. El prototipado y el diseño iterativo constituyen prácticas relevantes dentro del proceso de creación de videojuegos, ya que permiten experimentar, evaluar y refinar las ideas durante el desarrollo [11]. En este proyecto, estos principios se aplican mediante la construcción de un prototipo funcional en caja blanca, utilizando geometría y recursos visuales provisionales para concentrar el desarrollo en la programación y validación de las mecánicas de Inkfall.

La programación modular constituye otro fundamento para la construcción de sistemas interactivos, debido a que permite organizar la funcionalidad de un proyecto mediante componentes con responsabilidades específicas que pueden integrarse entre sí. Esta forma de estructuración facilita el desarrollo progresivo y la incorporación de diferentes sistemas dentro de un proyecto de mayor complejidad. En el desarrollo de videojuegos, la inteligencia artificial también puede estructurarse mediante componentes independientes relacionados con percepción, toma de decisiones y acciones, permitiendo construir comportamientos complejos a partir de módulos especializados [12]. En Inkfall, este principio se refleja en la implementación independiente de sistemas como el control del jugador, interacción, combate, armas, trampas, inteligencia artificial y progresión, los cuales posteriormente se integran para conformar el funcionamiento general del prototipo.

El diseño de niveles y la inteligencia artificial son componentes estrechamente relacionados con la experiencia de juego, debido a que determinan cómo el jugador se

desplaza, toma decisiones e interactúa con los desafíos presentes en el entorno. El diseño de niveles implica organizar espacialmente los elementos del juego para establecer rutas, obstáculos, desafíos y oportunidades de interacción, mientras que la inteligencia artificial permite generar comportamientos dinámicos en los personajes no controlados directamente por el usuario [13]. En el presente proyecto, estos fundamentos se aplican mediante la construcción de dos escenarios en caja blanca y la distribución de enemigos, armas, trampas, puntos de aparición y elementos necesarios para la progresión. De esta manera, la programación de los sistemas se valida directamente dentro de los espacios diseñados, comprobando la interacción entre las mecánicas y el entorno de juego.

Finalmente, la transformación dinámica basada en intentos fallidos constituye uno de los fundamentos principales de la propuesta, debido a que permite modificar las condiciones de desafío de un videojuego durante su ejecución. La literatura especializada identifica la adaptación de dificultad como un área orientada a ajustar el desafío y la experiencia de juego de acuerdo con diferentes características del jugador y del desarrollo de la partida [14]. Asimismo, las investigaciones sobre Dynamic Difficulty Adjustment muestran que las estrategias de adaptación pueden modificar uno o varios componentes del juego y emplear diferentes variables para determinar cuándo y cómo realizar dichos cambios [15]. En Inkfall, este concepto se implementa mediante un sistema basado en las muertes del jugador, donde cada fracaso activa modificaciones progresivas que afectan las mecánicas, los escenarios, los enemigos, las trampas y otros elementos del juego. Esta implementación busca explorar una alternativa en la que el fracaso no represente únicamente una penalización, sino que provoque transformaciones en las condiciones de la partida y en la experiencia del jugador.

2.3. Base metodológica

Para el desarrollo de la propuesta se consideraron diferentes enfoques metodológicos aplicables a la creación de software y videojuegos, entre ellos metodologías ágiles, enfoques iterativos e incrementales y procesos tradicionales de desarrollo secuencial. Las metodologías ágiles permiten organizar el desarrollo mediante ciclos cortos de trabajo y adaptación continua, mientras que los enfoques iterativos e incrementales permiten construir progresivamente una solución mediante la incorporación y refinamiento sucesivo de sus componentes [16]. Para el desarrollo de videojuegos, estos enfoques resultan especialmente pertinentes debido a la necesidad de probar continuamente las mecánicas y realizar ajustes a partir de los resultados obtenidos durante el desarrollo [11].

En este proyecto se adopta un enfoque aplicado, con alcance descriptivo y un proceso de desarrollo iterativo basado en el prototipado en caja blanca. Esta elección responde a la necesidad de construir progresivamente los sistemas funcionales de Inkfall, comenzando por los componentes base y avanzando hacia su integración dentro de un prototipo jugable. El proceso permite implementar, probar y refinar de manera progresiva sistemas como el jugador, interacción, combate, inteligencia artificial, diseño de niveles y transformación dinámica basada en intentos fallidos, priorizando la validación de su funcionamiento antes de incorporar los elementos visuales definitivos. De esta manera, el enfoque metodológico seleccionado permite concentrar los esfuerzos en la programación y en la comprobación de la lógica del videojuego, reduciendo la dependencia de recursos artísticos durante las primeras etapas del desarrollo.

2.4. Base tecnológica

La propuesta tecnológica se desarrolla utilizando Unity como motor de desarrollo y C# como lenguaje principal de programación. Unity proporciona un entorno integrado para la creación de videojuegos y experiencias interactivas, incorporando herramientas para la gestión de escenas, objetos, físicas, animaciones, iluminación, audio y ejecución multiplataforma [17]. En el presente proyecto, Unity constituye el entorno principal donde se integran los diferentes sistemas programados y los recursos provenientes de los assets utilizados. C# se emplea para desarrollar la lógica específica del videojuego mediante scripts que controlan componentes como el jugador, las armas, las interacciones, las trampas, los enemigos y el sistema de transformación dinámica basada en intentos fallidos.

La construcción de los escenarios se realiza mediante herramientas de modelado 3D, principalmente Blender, utilizado para elaborar la geometría básica de los espacios que conforman el prototipo en caja blanca. Esta herramienta permite crear y modificar modelos tridimensionales que posteriormente pueden ser importados a Unity para formar parte de las escenas del videojuego [18]. Para complementar el desarrollo, se emplean assets especializados que proporcionan funcionalidades previamente desarrolladas. Entre ellos se utiliza un sistema de disparos en primera persona para componentes como el movimiento del jugador, control de cámara, cambio de armas e interacción con elementos del entorno; un sistema de inteligencia artificial para implementar comportamientos de los enemigos; y recursos adicionales para sistemas específicos del segundo nivel. Estos componentes son integrados y complementados mediante scripts desarrollados específicamente para el proyecto.

El proyecto también utiliza componentes tecnológicos orientados a la interacción y al comportamiento de los sistemas de juego. Unity proporciona herramientas como el sistema

de físicas, colliders y triggers, utilizados para detectar interacciones entre el jugador y elementos del entorno, como armas, trampas, zonas de activación y objetos interactivos. Asimismo, el sistema de navegación mediante NavMesh permite establecer desplazamiento y búsqueda de rutas para personajes controlados por inteligencia artificial. En el segundo nivel, estos componentes se combinan con sistemas provenientes de los assets utilizados para controlar el comportamiento del enemigo y establecer respuestas específicas ante las acciones del jugador, como la interacción con la fuente de luz y las zonas que modifican temporalmente el comportamiento del enemigo.

Finalmente, la solución utiliza el sistema de prefabs de Unity como mecanismo para organizar y reutilizar los elementos funcionales que conforman el prototipo. Los prefabs permiten almacenar configuraciones de GameObjects y sus componentes para reutilizarlos en diferentes partes de una escena o proyecto, facilitando la construcción de elementos como armas, trampas, enemigos, objetos interactivos y escenarios [17]. En el desarrollo de Inkfall, este sistema se emplea para estructurar los elementos que componen los niveles y permitir su distribución dentro de los escenarios de caja blanca. La combinación de Unity, C#, Blender, los sistemas proporcionados por los assets y las herramientas nativas del motor permite construir una base funcional en la que los diferentes componentes programados se integran para conformar el prototipo jugable. La Tabla 3 resume las diferentes herramientas utilizadas.

Tabla 4

Base tecnológica

	Categoría	Herramienta
	Motor de desarrollo	Unity 6000.3.10f1
	Lenguaje de programación	C#
	Entorno de desarrollo	Visual Studio 2022
	Render Pipeline	URP
	Sistema de entrada	Unity New Input System
	Sistema operativo de desarrollo	Windows x64
	Fecha de congelamiento	31 de julio del 2026

2.5. Modelos, estándares y arquitecturas

La arquitectura de un videojuego puede organizarse mediante diferentes modelos de diseño de software, dependiendo de las características y necesidades del proyecto. En el desarrollo con motores como Unity, es común emplear una arquitectura basada en componentes, donde las entidades del juego se construyen mediante la combinación de componentes que encapsulan diferentes comportamientos y responsabilidades. Este enfoque favorece la reutilización y composición de funcionalidades, permitiendo desarrollar sistemas complejos a partir de elementos independientes [17]. Asimismo, la programación orientada a objetos permite estructurar el software mediante clases y objetos que representan entidades y comportamientos, facilitando la organización y mantenimiento del código [19].

Los patrones de diseño constituyen soluciones generales y reutilizables para problemas recurrentes en el diseño de software. En el desarrollo de videojuegos, estos principios pueden combinarse con arquitecturas basadas en componentes y modelos de programación orientada a objetos para estructurar sistemas independientes y facilitar su integración. Además, el uso de estándares y buenas prácticas de ingeniería de software

contribuye a mejorar características como la mantenibilidad, reutilización y organización de las soluciones desarrolladas [20].

2.6. Herramientas, lenguajes, frameworks o librerías

El desarrollo del prototipo se fundamenta principalmente en el motor de videojuegos Unity, utilizado como entorno para la construcción, integración y ejecución de la propuesta. Unity proporciona herramientas para la creación de escenas, gestión de GameObjects, físicas, animaciones, audio, iluminación, navegación e interacción, permitiendo centralizar el desarrollo de los diferentes sistemas que conforman un videojuego. Como lenguaje de programación se utiliza C#, empleado para desarrollar la lógica y los comportamientos específicos que complementan las funcionalidades proporcionadas por el motor y los recursos externos integrados al proyecto [17].

Para el desarrollo y modelado de los elementos utilizados durante la etapa de prototipado en caja blanca se emplea Blender, una herramienta de creación de contenido 3D que permite realizar modelado, edición y preparación de recursos tridimensionales para su posterior integración en motores de videojuegos. El uso de esta herramienta permite construir representaciones geométricas simplificadas de los espacios y elementos necesarios para validar la navegación y el funcionamiento de las mecánicas antes de incorporar los recursos visuales definitivos [18]. El código fuente se desarrolla y edita mediante un entorno de desarrollo compatible con C#, complementando el flujo de trabajo proporcionado por Unity.

El proyecto también integra frameworks, librerías y assets especializados que amplían las capacidades disponibles en el motor. Entre estos recursos se incluyen sistemas para el desarrollo de videojuegos en primera persona, herramientas de inteligencia artificial y navegación de enemigos, sistemas de interacción y recursos orientados a la implementación

de eventos y mecánicas específicas. Estos componentes permiten aprovechar funcionalidades previamente desarrolladas y complementarlas mediante programación propia, reduciendo el tiempo necesario para construir desde cero sistemas comunes y permitiendo concentrar el desarrollo en las funcionalidades específicas de la propuesta. La integración de estos recursos se realiza dentro del entorno de Unity, donde sus componentes pueden combinarse con scripts desarrollados en C# para conformar el funcionamiento general del prototipo.

2.7. Marco conceptual

Los videojuegos representan una de las formas de software interactivo con mayor crecimiento en las últimas décadas, integrando disciplinas como la informática, el diseño, el arte, la narrativa y la inteligencia artificial para construir experiencias en las que el usuario participa activamente. A diferencia de otros medios digitales, un videojuego no presenta una secuencia de contenidos completamente predeterminada, sino que responde continuamente a las acciones del jugador, generando una interacción constante entre el usuario y el sistema.

Desde una perspectiva de ingeniería, un videojuego puede entenderse como un conjunto de sistemas interconectados que trabajan de manera coordinada para procesar información, actualizar el estado del entorno virtual y proporcionar una respuesta inmediata a las acciones del jugador. Esta estructura hace que el desarrollo de videojuegos implique no solo aspectos visuales o narrativos, sino también el diseño de arquitecturas de software capaces de integrar múltiples componentes de forma organizada y eficiente.

Para comprender la forma en que un videojuego funciona internamente, es necesario analizar algunos de sus elementos principales, entre los que se encuentran las mecánicas de juego, las dinámicas que emergen durante la interacción, la jugabilidad, la comunicación

entre el jugador y el entorno, así como el ciclo de ejecución responsable del funcionamiento continuo del sistema.

2.7.1. Concepto de videojuego

Un videojuego puede definirse como un sistema informático interactivo diseñado para proporcionar una experiencia mediante la interacción entre un jugador y un entorno virtual regido por un conjunto de reglas. Su funcionamiento depende de la integración de diversos componentes de software que procesan continuamente las acciones del usuario y generan respuestas acordes con el estado actual del juego.

A diferencia de otros sistemas interactivos, un videojuego se caracteriza por mantener una comunicación constante entre el jugador y el entorno virtual. Cada acción realizada por el usuario modifica, en mayor o menor medida, el estado del sistema, el cual responde mediante cambios en el escenario, en los personajes, en los objetos o en las condiciones de juego. Este intercambio continuo convierte al videojuego en un sistema dinámico cuya evolución depende tanto de las reglas programadas como de las decisiones tomadas por el jugador.

Desde el punto de vista del desarrollo, un videojuego está conformado por diversos sistemas especializados, tales como el control del jugador, la inteligencia artificial, la gestión de físicas, la interacción con el entorno, el sistema de eventos y la interfaz de usuario. Aunque cada uno cumple una función específica, todos deben operar de manera coordinada para garantizar el correcto funcionamiento de la aplicación y ofrecer una experiencia interactiva coherente.

2.7.2. Mecánicas de juego

Las mecánicas de juego corresponden al conjunto de reglas y procedimientos que determinan las acciones disponibles para el jugador y las respuestas que el sistema genera ante dichas acciones. Estas mecánicas establecen la estructura básica del videojuego, definiendo aspectos como el movimiento, el combate, la interacción con objetos, la resolución de desafíos o cualquier otra acción que forme parte de la experiencia interactiva [21].

Las mecánicas representan uno de los elementos centrales del diseño de videojuegos, ya que constituyen el medio mediante el cual el jugador puede interactuar con el entorno virtual. Cada mecánica define una posibilidad de acción y establece las condiciones bajo las cuales dicha acción produce determinados resultados. De esta manera, las reglas implementadas por el sistema delimitan las opciones disponibles para el usuario y determinan el comportamiento esperado del juego.

En el contexto de la presente investigación, las mecánicas adquieren una especial relevancia debido a que el sistema de transformación dinámica basada en intentos fallidos modifica precisamente algunos de estos comportamientos conforme progresa la partida. En lugar de limitarse a incrementar valores numéricos, el prototipo desarrollado busca alterar determinadas condiciones del juego mediante cambios en las propias mecánicas implementadas.

2.7.3. Dinámicas de juego

Las dinámicas de juego representan los comportamientos que surgen como consecuencia de la interacción entre las mecánicas implementadas y las decisiones adoptadas por el jugador durante la partida. Mientras las mecánicas describen las reglas del sistema, las

dinámicas corresponden a la manera en que dichas reglas se manifiestan durante la experiencia de juego [22].

Las dinámicas no son programadas como eventos individuales, sino que emergen naturalmente a partir de la combinación de múltiples mecánicas funcionando de manera simultánea. Esto explica por qué dos jugadores pueden experimentar situaciones diferentes aun enfrentándose al mismo escenario, ya que sus decisiones producen secuencias distintas de interacción con el sistema.

En investigaciones relacionadas con sistemas adaptativos, mencionadas en la Tabla 3, las dinámicas poseen una importancia considerable debido a que permiten modificar el comportamiento general del videojuego sin alterar necesariamente su estructura básica. Esta característica resulta especialmente pertinente para el desarrollo de Inkfall, donde la dificultad evoluciona mediante cambios progresivos en el funcionamiento del sistema conforme aumenta el número de muertes del jugador.

2.7.4. Jugabilidad

La jugabilidad, también conocida como gameplay, hace referencia a la calidad de la experiencia interactiva generada por la relación entre el jugador y el videojuego. Este concepto comprende aspectos como la claridad de las reglas, la precisión de los controles, la capacidad de respuesta del sistema y el equilibrio existente entre el nivel de desafío y las habilidades requeridas para superar los distintos objetivos [23].

La jugabilidad no depende exclusivamente de una mecánica específica, sino del funcionamiento conjunto de todos los sistemas que conforman el videojuego. Cuando estos componentes interactúan de forma adecuada, el jugador percibe una experiencia coherente y

comprensible que favorece el aprendizaje de las reglas y el desarrollo progresivo de nuevas estrategias.

En el caso de esta investigación, la jugabilidad constituye un aspecto relevante porque el sistema de transformación dinámica basada en intentos fallidos modifica determinadas condiciones del juego durante la partida. En consecuencia, resulta necesario que estos cambios mantengan la coherencia del funcionamiento general del prototipo y no afecten negativamente la comprensión de las mecánicas implementadas.

2.7.5. Interacción jugador-entorno

La interacción jugador-entorno comprende el conjunto de acciones mediante las cuales el usuario puede influir sobre el mundo virtual y recibir respuestas por parte del sistema. Estas interacciones incluyen actividades como desplazarse por el escenario, manipular objetos, activar mecanismos, utilizar herramientas o enfrentarse a diferentes obstáculos presentes durante la partida [24].

La interacción constituye uno de los elementos que diferencian a los videojuegos de otros medios digitales, ya que el usuario participa activamente en la evolución del entorno virtual. Cada acción realizada modifica el estado del sistema y produce nuevas condiciones que influyen en el desarrollo de la experiencia.

En el prototipo desarrollado para esta investigación, la interacción con el entorno permite que el jugador active diferentes mecánicas dentro de los escenarios en caja blanca, facilitando la validación del comportamiento de los distintos sistemas implementados y de las modificaciones producidas por la transformación dinámica basada en intentos fallidos.

2.7.6. Feedback del jugador

El feedback del jugador corresponde a la información que el sistema proporciona como respuesta a las acciones realizadas por el usuario. Esta retroalimentación puede presentarse mediante elementos visuales, auditivos o de otra naturaleza, permitiendo que el jugador comprenda las consecuencias de sus decisiones y adapte su comportamiento durante la partida [25].

La retroalimentación constituye un mecanismo indispensable para la comunicación entre el jugador y el videojuego. Gracias a ella, el usuario puede interpretar el estado del sistema, reconocer el éxito o fracaso de sus acciones y comprender el funcionamiento de las diferentes mecánicas implementadas.

En el desarrollo de un prototipo funcional en caja blanca, el feedback mantiene su importancia incluso cuando no existen recursos gráficos definitivos. Indicadores simples como cambios de estado, mensajes, barras de vida o modificaciones del entorno permiten verificar el correcto funcionamiento de los sistemas antes de incorporar los elementos artísticos del videojuego.

2.7.7. Loop de juego

El game loop, o bucle de juego, constituye la estructura responsable de controlar la ejecución continua del videojuego mientras este permanece en funcionamiento. Durante cada iteración del ciclo, el sistema procesa las entradas realizadas por el jugador, actualiza el estado de todos los componentes del juego y genera la salida correspondiente en pantalla [26].

Este proceso se ejecuta de forma repetitiva cientos de veces por segundo, garantizando que todos los sistemas del videojuego funcionen de manera sincronizada.

Gracias al game loop, las acciones del jugador producen respuestas inmediatas y el entorno virtual mantiene un comportamiento continuo y consistente durante toda la experiencia.

En la presente investigación, el game loop representa la base sobre la cual operan los distintos sistemas implementados en el prototipo, permitiendo que las mecánicas, la inteligencia artificial, la interacción con el entorno y el sistema de transformación dinámica basada en intentos fallidos actualicen su comportamiento de manera coordinada en cada ciclo de ejecución del videojuego.

2.7.8. Diseño de videojuegos

El diseño de videojuegos constituye el proceso mediante el cual se definen las reglas, mecánicas, dinámicas y objetivos que conformarán la experiencia interactiva del jugador. A diferencia de otras áreas del desarrollo de software, el diseño de videojuegos no se limita a establecer requisitos funcionales, sino que busca construir sistemas capaces de generar experiencias entretenidas, coherentes y desafiantes mediante la interacción continua entre el usuario y el entorno virtual.

Durante el proceso de diseño se establecen los elementos que determinan el comportamiento general del videojuego, considerando aspectos como la forma en que el jugador interactúa con el sistema, la progresión del desafío, la organización de las mecánicas y la relación existente entre los diferentes componentes del juego. Como resultado, el diseño se convierte en una etapa que guía el desarrollo técnico y facilita la construcción de sistemas capaces de responder de manera coherente a las acciones del jugador.

2.7.9. Diseño centrado en el jugador

El diseño centrado en el jugador es un enfoque que orienta el desarrollo del videojuego hacia las necesidades, habilidades y expectativas del usuario, procurando que las

decisiones de diseño favorezcan una experiencia de interacción clara, comprensible y progresiva [27].

Este enfoque plantea que las mecánicas, reglas y objetivos del videojuego deben construirse considerando la forma en que serán percibidos por el jugador, permitiendo que el aprendizaje ocurra de manera gradual y que los desafíos evolucionen conforme aumentan las habilidades adquiridas durante la partida. De esta manera, el diseño deja de centrarse únicamente en la implementación técnica y comienza a considerar la manera en que el usuario experimenta el funcionamiento del sistema.

En la presente investigación, este enfoque resulta pertinente debido a que el sistema de transformación dinámica basada en intentos fallidos modifica las condiciones del juego conforme avanza la partida. En consecuencia, dichas modificaciones deben implementarse procurando mantener una progresión comprensible y coherente dentro del prototipo desarrollado.

2.7.10. Flujo

El flujo corresponde a un estado de concentración e involucramiento en el que el jugador mantiene un equilibrio entre el nivel de desafío presentado por el videojuego y sus propias habilidades para superarlo [28].

2.7.11. Inmersión

La inmersión hace referencia al grado en que el jugador se siente involucrado dentro del entorno virtual, percibiendo el mundo del videojuego como un espacio coherente en el que sus acciones poseen significado [29].

2.7.12. Agencia del jugador

La agencia del jugador corresponde a la capacidad que posee el usuario para tomar decisiones significativas dentro del videojuego y observar las consecuencias derivadas de dichas acciones [30].

2.7.13. Diseño sistémico

El diseño sistémico consiste en organizar un videojuego como un conjunto de sistemas interrelacionados que interactúan entre sí para generar comportamientos más complejos que los producidos por cada componente de manera individual [31].

Este enfoque permite construir videojuegos en los que las mecánicas, la inteligencia artificial, la interacción con el entorno y otros sistemas mantienen relaciones constantes, produciendo situaciones emergentes durante la experiencia de juego. Como consecuencia, el comportamiento general del videojuego no depende exclusivamente de reglas aisladas, sino de la interacción existente entre todos sus componentes.

La implementación del prototipo desarrollado en esta investigación adopta este enfoque, ya que el sistema de transformación dinámica basada en intentos fallidos modifica distintos elementos del juego sin alterar el funcionamiento independiente de cada uno de ellos.

2.7.14. Curva de dificultad

La curva de dificultad representa la progresión del nivel de desafío que experimenta el jugador a lo largo del videojuego [32].

Su diseño busca mantener un incremento gradual de la complejidad, permitiendo que el jugador desarrolle nuevas habilidades antes de enfrentarse a retos más exigentes. Una

curva adecuadamente diseñada evita cambios bruscos que puedan generar frustración o disminuir el interés durante la experiencia.

En la presente investigación, este concepto sirve como base para comprender la forma en que el sistema de transformación dinámica basada en intentos fallidos modifica progresivamente las condiciones del prototipo conforme aumenta el número de intentos fallidos del jugador.

2.7.15. Narrativa emergente

La narrativa emergente corresponde a las historias que surgen de manera natural como resultado de la interacción entre el jugador y los diferentes sistemas del videojuego, sin depender completamente de una secuencia narrativa predefinida [33].

2.7.16. Sistemas de dificultad en videojuegos

La dificultad constituye uno de los elementos más importantes dentro del diseño de videojuegos, ya que determina el nivel de desafío que experimenta el jugador durante la interacción con el sistema. Un diseño adecuado de la dificultad permite mantener el interés del usuario, favoreciendo un equilibrio entre las habilidades que desarrolla y los retos que debe superar a lo largo de la experiencia.

Tradicionalmente, los videojuegos han implementado sistemas de dificultad basados en configuraciones estáticas seleccionadas antes de iniciar la partida. Sin embargo, el crecimiento de los sistemas adaptativos ha permitido desarrollar nuevos enfoques capaces de modificar las condiciones del juego durante su ejecución, respondiendo al comportamiento del jugador y a su progreso dentro de la experiencia. Estos enfoques han dado origen a los sistemas de transformación dinámica basada en intentos fallidos, los cuales representan el principal objeto de estudio de la presente investigación.

2.7.17. Sistema de dificultad

La transformación dinámica basada en intentos fallidos es una técnica de diseño de videojuegos que consiste en modificar automáticamente el nivel de desafío de un juego de acuerdo con el comportamiento, desempeño o progreso del jugador durante la partida [34]. Su objetivo consiste en mantener una experiencia equilibrada mediante la adaptación continua de diferentes elementos del sistema, evitando que el jugador experimente niveles excesivos de frustración o aburrimiento.

A diferencia de los sistemas tradicionales, la transformación dinámica basada en intentos fallidos no depende de una configuración fija establecida antes de iniciar el juego. En su lugar, el propio sistema analiza determinadas variables relacionadas con el comportamiento del jugador y, a partir de ellas, realiza modificaciones sobre distintos componentes del videojuego. Estas modificaciones pueden afectar aspectos como la inteligencia artificial de los enemigos, la frecuencia de aparición de obstáculos, la distribución de recursos, el diseño del escenario o las reglas que gobiernan las mecánicas implementadas.

En la presente investigación, la transformación dinámica basada en intentos fallidos se implementa mediante un sistema de adaptación basado en el número acumulado de muertes del jugador, utilizado como variable de entrada para determinar el estado de dificultad de la partida. Cada muerte representa un intento fallido y provoca la activación progresiva de perfiles de dificultad previamente definidos, los cuales modifican diferentes condiciones del juego. La función de adaptación se estructura mediante umbrales asociados a la cantidad de muertes, de manera que el sistema aplica cambios acumulativos sobre las mecánicas, el comportamiento de los enemigos y las condiciones del entorno. La dirección de estos ajustes no se limita exclusivamente al incremento de valores numéricos, sino que

busca reconfigurar progresivamente las condiciones de la experiencia mediante la incorporación de nuevos obstáculos, cambios en el comportamiento de los enemigos y modificaciones en las reglas de interacción. De esta manera, el sistema utiliza los intentos fallidos como indicador del progreso de la partida para generar una transformación progresiva del desafío, constituyendo uno de los sistemas implementados dentro de la base técnica del videojuego Inkfall.

2.7.18. Prototipado en videojuegos

El desarrollo de videojuegos es un proceso iterativo que involucra el diseño, implementación y evaluación continua de diferentes sistemas antes de obtener un producto final. Debido a la complejidad que caracteriza a este tipo de proyectos, resulta poco recomendable desarrollar directamente una versión completa del videojuego sin haber validado previamente el funcionamiento de sus principales mecánicas. Por esta razón, el uso de prototipos constituye una práctica ampliamente utilizada dentro de la industria y la investigación, ya que permite comprobar el comportamiento de los sistemas desde las primeras etapas del desarrollo.

Los prototipos facilitan la identificación temprana de errores de diseño, problemas de implementación y oportunidades de mejora, reduciendo el costo asociado a realizar modificaciones en fases avanzadas del proyecto. Asimismo, permiten centrar el esfuerzo de desarrollo en la validación de la lógica del juego antes de incorporar elementos gráficos, sonoros o narrativos que incrementan la complejidad del producto.

2.7.19. Whiteboxing

El whiteboxing, también conocido como desarrollo en caja blanca, es una técnica utilizada durante las primeras etapas de creación de videojuegos que consiste en construir

versiones preliminares de los escenarios y sistemas empleando geometría simple y recursos visuales básicos. Su propósito principal consiste en validar el funcionamiento de las mecánicas, la distribución de los espacios, la interacción con el entorno y el comportamiento general del juego antes de iniciar la producción de los elementos artísticos definitivos [32].

A diferencia de un prototipo orientado a presentar el aspecto visual del videojuego, el whiteboxing prioriza el funcionamiento interno del sistema. En esta etapa, aspectos como modelos tridimensionales, texturas, iluminación, animaciones o efectos visuales son reemplazados por elementos simples cuya única finalidad consiste en representar el comportamiento esperado de cada componente. De esta manera, el equipo de desarrollo puede concentrarse en comprobar que las mecánicas implementadas funcionan correctamente y que la estructura general del videojuego responde a los objetivos de diseño planteados.

Otra ventaja del desarrollo en caja blanca consiste en la facilidad para realizar modificaciones durante el proceso de implementación. Debido a que los escenarios y sistemas aún no dependen de recursos gráficos complejos, es posible ajustar rápidamente la disposición de los niveles, modificar las mecánicas o replantear el comportamiento de los distintos sistemas sin afectar otras áreas del proyecto. Esta flexibilidad favorece la experimentación y permite refinar el diseño antes de avanzar hacia etapas de producción con mayor inversión de tiempo y recursos.

En la presente investigación, el desarrollo del videojuego Inkfall se limita a la construcción de un prototipo funcional en caja blanca. En consecuencia, el objetivo principal no consiste en producir un videojuego terminado, sino en implementar y validar el funcionamiento de un sistema de transformación dinámica basada en intentos fallidos

mediante escenarios construidos con geometría básica y sistemas completamente funcionales.

2.7.20. Prototipado iterativo

El prototipado iterativo es una metodología de desarrollo basada en la construcción progresiva de versiones funcionales de un sistema, las cuales son evaluadas y refinadas mediante ciclos sucesivos de implementación, prueba y mejora [35]. Cada iteración permite incorporar nuevos componentes, corregir problemas detectados durante las evaluaciones anteriores y validar el comportamiento del sistema antes de continuar con etapas posteriores del desarrollo.

Este enfoque reconoce que resulta difícil prever todas las necesidades de un proyecto desde sus primeras fases, especialmente cuando se desarrollan sistemas interactivos como los videojuegos. Por esta razón, el proceso iterativo favorece la experimentación y permite introducir modificaciones conforme se obtienen nuevos resultados durante la implementación o las pruebas realizadas con los usuarios.

En el desarrollo de videojuegos, el prototipado iterativo facilita la evaluación de mecánicas, sistemas de inteligencia artificial, niveles, interfaces y demás componentes antes de consolidar una versión definitiva del producto. Cada ciclo de desarrollo proporciona información que permite mejorar el diseño, optimizar la arquitectura del software y reducir la probabilidad de errores durante etapas posteriores.

La presente investigación adopta este enfoque metodológico mediante el desarrollo progresivo del prototipo de Inkfall. Durante su construcción, las diferentes mecánicas, sistemas de interacción y el sistema de transformación dinámica basada en intentos fallidos

son implementados, evaluados y ajustados de forma continua hasta obtener una versión funcional capaz de demostrar la viabilidad técnica de la propuesta planteada.

2.7.21. Arquitectura de software

La arquitectura de software constituye la estructura sobre la cual se organizan los diferentes componentes que conforman un sistema informático. En el desarrollo de videojuegos, una arquitectura adecuada permite distribuir las responsabilidades entre distintos módulos, facilitando la implementación de nuevas funcionalidades, el mantenimiento del código y la evolución del proyecto conforme aumentan sus requerimientos.

A diferencia de aplicaciones convencionales, los videojuegos integran múltiples sistemas que deben ejecutarse de manera simultánea, como el control del jugador, la inteligencia artificial, la gestión del entorno, las interfaces de usuario y los sistemas de interacción. Debido a esta complejidad, resulta necesario adoptar estrategias de organización que permitan mantener la independencia entre los distintos componentes y reducir las dependencias innecesarias durante el desarrollo.

2.7.22. Arquitectura modular

La arquitectura modular es un enfoque de diseño de software que consiste en dividir un sistema en componentes independientes, cada uno con responsabilidades claramente definidas [19]. Esta organización permite que cada módulo sea desarrollado, probado y mantenido de forma individual, reduciendo la complejidad del sistema y favoreciendo la reutilización de código.

En el desarrollo de videojuegos, este enfoque facilita la implementación de sistemas especializados, como el control del jugador, la inteligencia artificial, la interacción con el

entorno o la gestión de interfaces, sin que las modificaciones realizadas en uno de ellos afecten directamente al resto de componentes. Como consecuencia, el desarrollo se vuelve más organizado y permite incorporar nuevas funcionalidades de manera progresiva.

La presente investigación adopta una arquitectura modular para estructurar el prototipo funcional de Inkfall, permitiendo implementar los diferentes sistemas del videojuego de forma independiente y facilitando la integración del sistema de transformación dinámica basada en intentos fallidos con el resto de componentes desarrollados.

2.7.23. Arquitectura desacoplada

La arquitectura desacoplada corresponde a un enfoque de diseño en el que los diferentes componentes del software minimizan las dependencias directas entre sí, permitiendo que cada uno pueda evolucionar sin afectar significativamente el funcionamiento de los demás [36].

El desacoplamiento favorece la mantenibilidad del sistema al reducir la cantidad de modificaciones necesarias cuando un componente cambia su comportamiento. Además, facilita la reutilización de módulos en otros proyectos y mejora la organización general del código al establecer responsabilidades claramente diferenciadas.

En videojuegos, este principio resulta especialmente importante debido a la constante interacción entre numerosos sistemas. Una arquitectura desacoplada permite que elementos como la inteligencia artificial, los sistemas de interacción o las mecánicas de juego intercambien información sin depender directamente de la implementación interna de otros módulos.

2.7.24. Sistema de eventos

Un sistema de eventos es un mecanismo de comunicación mediante el cual diferentes componentes del software pueden intercambiar información sin establecer dependencias directas entre ellos [37]. Su funcionamiento se basa en la emisión de eventos por parte de un componente y la recepción de dichos eventos por aquellos módulos interesados en responder a una determinada acción o cambio de estado.

La utilización de sistemas de eventos favorece el desacoplamiento entre los distintos componentes del videojuego, ya que elimina la necesidad de establecer referencias permanentes entre ellos. Como resultado, la arquitectura se vuelve más flexible y facilita la incorporación de nuevos comportamientos sin modificar la estructura existente.

Dentro del desarrollo de videojuegos, este tipo de comunicación resulta especialmente útil para coordinar acciones entre múltiples sistemas, permitiendo responder de forma organizada a eventos como la derrota de un enemigo, la interacción con un objeto, la finalización de una misión o cualquier otra situación relevante durante la ejecución del juego.

2.7.25. Sistema de estados

Un sistema de estados es un modelo de organización del comportamiento que permite representar el funcionamiento de una entidad mediante un conjunto de estados claramente definidos y las transiciones existentes entre ellos [38].

Cada estado representa una condición específica del sistema, mientras que las transiciones establecen las reglas que determinan cuándo debe cambiarse de un estado a otro. Este enfoque facilita el control del comportamiento de diferentes elementos del videojuego,

evitando estructuras de programación excesivamente complejas y mejorando la claridad del código.

Los sistemas de estados son ampliamente utilizados para controlar el comportamiento de personajes, enemigos, interfaces y otros componentes que presentan diferentes modos de funcionamiento durante la ejecución del videojuego, permitiendo administrar de forma organizada los cambios de comportamiento que ocurren durante la partida.

2.7.26. Escalabilidad de sistemas

La escalabilidad de sistemas hace referencia a la capacidad que posee un software para incorporar nuevas funcionalidades o aumentar su complejidad sin comprometer su funcionamiento ni su estabilidad [39].

En el desarrollo de videojuegos, la escalabilidad permite ampliar progresivamente el proyecto mediante la incorporación de nuevas mecánicas, escenarios, enemigos o sistemas sin necesidad de modificar completamente la arquitectura existente. Esto favorece tanto el mantenimiento del código como la evolución del videojuego a lo largo de su desarrollo.

En la presente investigación, la escalabilidad representa un aspecto importante debido a que el prototipo desarrollado constituye únicamente la base técnica del videojuego Inkfall. En consecuencia, la arquitectura implementada busca facilitar la incorporación futura de nuevos niveles, mecánicas y recursos audiovisuales, manteniendo la organización del sistema y permitiendo que el proyecto continúe evolucionando sin requerir cambios estructurales significativos.

Los conceptos abordados en este apartado proporcionan los fundamentos necesarios para comprender la organización interna del prototipo desarrollado en la presente investigación. La adopción de una arquitectura modular, desacoplada y escalable,

complementada mediante sistemas de eventos y sistemas de estados, permite construir una base técnica organizada que facilita la implementación y evolución del sistema de transformación dinámica basada en intentos fallidos propuesto para el videojuego Inkfall.

Capítulo III

3. Manual de usuario

3.1. Introducción

El presente Manual de Usuario tiene como finalidad proporcionar la información necesaria para la correcta instalación, ejecución y utilización del prototipo del videojuego Inkfall. Este documento está dirigido a cualquier usuario que desee interactuar con el sistema, ofreciendo una guía clara sobre los requisitos del software, los controles, las mecánicas principales y el funcionamiento general del juego.

Inkfall es un videojuego narrativo desarrollado en el motor Unity como parte de un proyecto académico. El prototipo está compuesto por dos niveles que presentan diferentes mecánicas de juego y un sistema de transformación dinámica basada en intentos fallidos que modifica determinadas condiciones de la partida en función del desempeño del jugador.

Este manual describe paso a paso el proceso para ejecutar el videojuego, conocer la interfaz, utilizar los controles y comprender las principales mecánicas implementadas.

3.2. Requisitos del sistema

Tabla 5

Requisitos del sistema

Componente	Requisitos
Sistema Operativo	Windows 10 de 64 bits
Procesador	Intel Core i5 de 8va generación o AMD Ryzen 5 equivalente
Memoria RAM	8 GB
Tarjeta gráfica	NVIDIA GTX 1650 o AMD Radeon RX 570, con 4 GB de VRAM
Almacenamiento	8 GB disponibles

3.3. Instalación

Para ejecutar el videojuego Inkfall es necesario contar con un equipo que cumpla con los requisitos mínimos del sistema y seguir el procedimiento de instalación descrito a continuación.

3.3.1. Descarga del juego

Obtenga el archivo comprimido que contiene el videojuego desde el medio proporcionado por los desarrolladores (memoria USB, enlace de descarga o repositorio institucional).

3.3.2. Descompresión de los archivos

Ubique el archivo descargado y descomprímalo en la carpeta de su preferencia utilizando el explorador de archivos de Windows o cualquier software de descompresión compatible.

3.3.3. Ejecución del juego

Abra la carpeta del proyecto y ejecute el archivo denominado **Inkfall.exe** para iniciar el videojuego.

3.3.4. Inicio del videojuego

Una vez ejecutado el archivo, se mostrará la pantalla principal del videojuego, desde la cual el usuario podrá acceder al menú principal y comenzar la experiencia de juego.

3.4. Interfaz del juego

La interfaz de usuario de Inkfall fue diseñada para ofrecer una navegación clara e intuitiva desde el inicio del videojuego. Su estructura permite acceder de forma organizada a las diferentes funciones disponibles antes de comenzar la partida, manteniendo una distribución uniforme de los elementos visuales y una identidad gráfica coherente con la propuesta artística del proyecto.

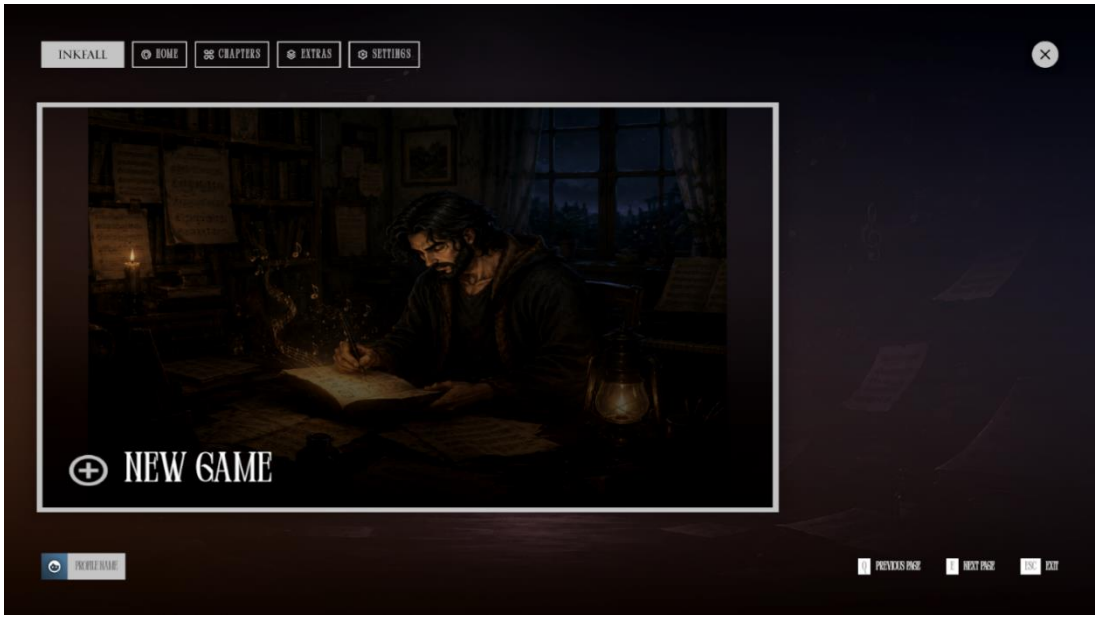
Al ejecutar el videojuego, el usuario visualizará una pantalla de bienvenida con el logotipo de Inkfall y el mensaje "Press Any Key to Start". Para acceder al menú principal, únicamente deberá presionar cualquier tecla del teclado.

3.4.1. Menú principal

El menú principal constituye el centro de navegación del videojuego y reúne las principales opciones disponibles para el usuario. La interfaz está organizada mediante pestañas ubicadas en la parte superior, permitiendo cambiar entre las diferentes secciones sin abandonar el menú principal.

Ilustración 1

Menú principal

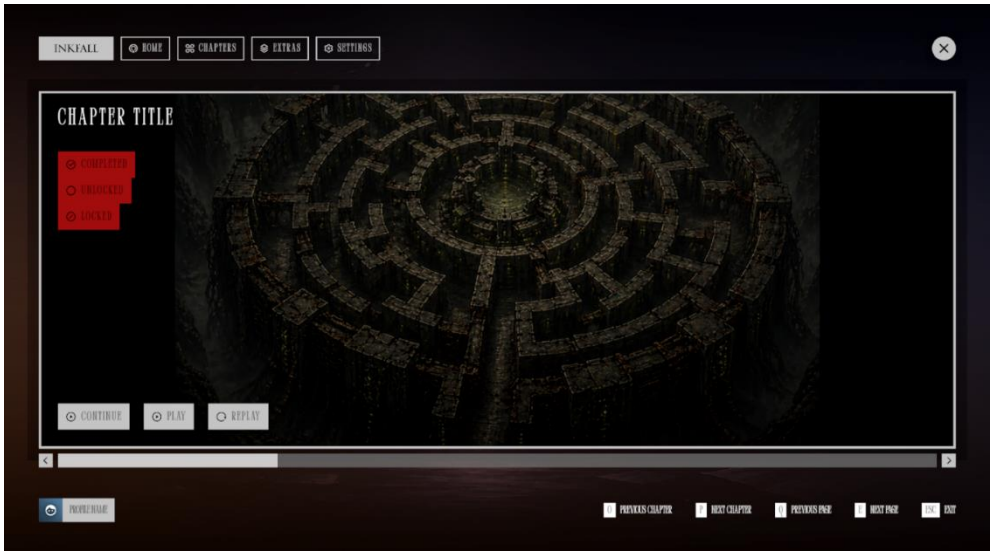


La pestaña Home presenta la opción New Game, desde donde el usuario puede iniciar una nueva partida y acceder al contenido del videojuego. Además, en la parte inferior se muestran los controles disponibles para desplazarse entre las páginas del menú o salir de la aplicación.

La pestaña Chapters permite visualizar los capítulos disponibles dentro del prototipo. Cada capítulo incluye una imagen representativa, una breve descripción y las opciones para iniciar o repetir el nivel correspondiente.

Ilustración 2

Pestaña chapters



La pestaña extras contiene información complementaria del videojuego, como los créditos.

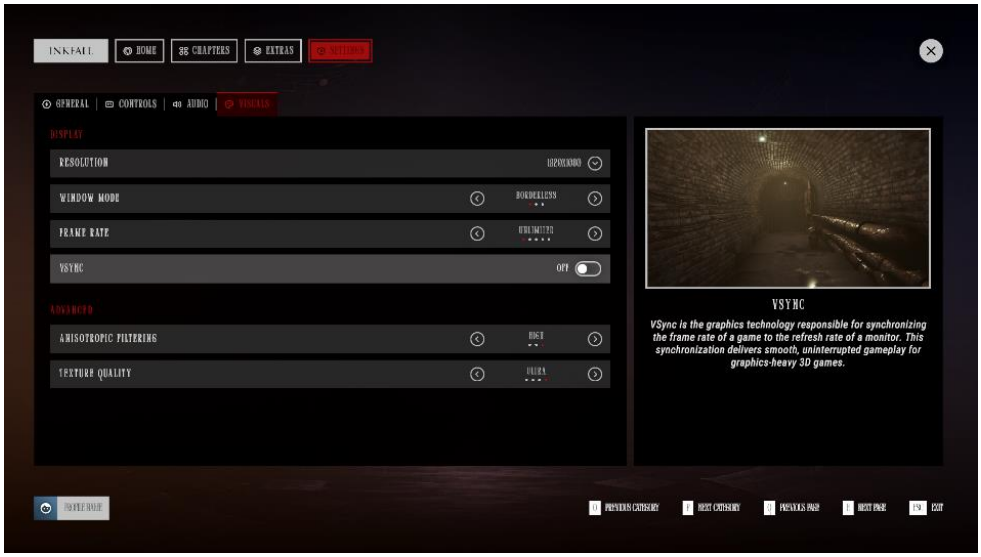
Ilustración 3

Pestaña Extras

Finalmente, la pestaña Settings permite modificar diferentes parámetros del videojuego. Dentro de esta sección el usuario puede acceder a opciones generales, configuración de controles, audio, aspectos visuales y funciones de accesibilidad, adaptando la experiencia de juego de acuerdo con sus preferencias.

Ilustración 4

Pestaña settings



3.4.2. Pantalla del juego

Durante la partida se mostrará la interfaz principal (HUD), la cual proporciona al jugador la información necesaria para el desarrollo del juego. Su diseño mantiene la identidad visual de Inkfall, utilizando elementos gráficos inspirados en el logotipo del videojuego.

El indicador de vida está representado por el círculo presente en el logotipo, el cual refleja el estado de salud del personaje durante la partida. En el centro se encuentra una gota que indica el nivel de dificultad seleccionado, permitiendo al jugador identificar esta información de forma rápida sin afectar la experiencia visual.

En el Capítulo 1, la interfaz incorpora información adicional relacionada con el sistema de combate, mostrando el arma equipada, el inventario de armas y la cantidad de munición disponible. Estos elementos permiten al jugador administrar sus recursos durante los enfrentamientos.

Por otra parte, en el Capítulo 2, la interfaz se adapta a la mecánica propia de ese escenario. Debido a que la jugabilidad está basada en el uso de un encendedor como elemento principal, los indicadores relacionados con las armas son reemplazados por la información necesaria para esta mecánica, manteniendo una interfaz clara y acorde con la experiencia del jugador.

3.5. Controles

Tabla 6

Controles de juego

Acción	Tecla
Mover	WASD
Cámara	Mouse
Disparar	Click izquierdo
Apuntar	Click derecho
Saltar	Espacio
Interactuar	E
Recargar	R
Habilidad especial	F
Cambiar arma	Rueda del mouse
Abrir puerta	Click izquierdo
Recargar encendedor	R

3.6. Cómo jugar

Al iniciar Inkfall, el jugador tomará el control del protagonista en 2 escenarios distintos. El objetivo es explorar los escenarios, interactuar con los elementos del entorno y superar los desafíos planteados en cada nivel. A medida que el jugador progresa, encontrará mecánicas diferentes que requieren combinar exploración, observación y habilidad para completar cada sección del prototipo.

3.6.1. Inicio

Al ejecutar el juego se mostrará el menú principal, desde donde el usuario podrá iniciar una nueva partida o seleccionar uno de los 2 niveles. Una vez seleccionada la opción

correspondiente, el juego cargará el nivel y el jugador aparecerá en el punto inicial del escenario.

Durante la partida, el jugador podrá desplazarse libremente utilizando los controles establecidos e interactuar con los diferentes elementos presentes en el entorno para avanzar en la experiencia.

3.6.2. Primer nivel

El primer nivel está orientado a las mecánicas de combate y exploración. El jugador deberá recorrer un laberinto mientras enfrenta enemigos controlados por inteligencia artificial.

Para avanzar es necesario:

- Explorar el escenario para encontrar la ruta correcta.
- Utilizar el arma para eliminar a los enemigos que bloquean el camino.
- Evitar recibir demasiado daño durante los enfrentamientos.
- Continuar avanzando hasta alcanzar la salida del nivel.

Si la vida del jugador llega a cero, la partida finalizará y el juego se reiniciará aplicando el sistema de transformación dinámica basada en intentos fallidos.

3.6.3. Segundo nivel

El segundo nivel cambia completamente la dinámica del juego. En este escenario, la supervivencia depende del manejo adecuado de un encendedor que actúa como la principal fuente de luz del jugador, Con ello, la experiencia adopta una dinámica de terror y evasión.

Durante el recorrido, el encendedor puede apagarse debido a movimientos bruscos ya sea de la cámara o del personaje. Cuando esto ocurre, el jugador deberá encenderlo lo antes posible para evitar permanecer en la oscuridad durante un tiempo prolongado.

Mientras explora el escenario, el jugador deberá recorrer las habitaciones, interactuar con los objetos necesarios y encontrar la ruta y la llave que le permita completar el nivel evitando ser alcanzado por la amenaza presente en el escenario.

3.6.4. Sistema de dificultad

Inkfall incorpora un sistema de transformación dinámica basada en intentos fallidos que aumenta progresivamente el comportamiento del juego conforme el jugador acumula intentos fallidos.

Cada vez que el jugador muere, el sistema registra el intento y aplica cambios en determinados elementos del juego. Estos cambios pueden afectar el comportamiento de los enemigos, las condiciones del escenario o las mecánicas disponibles, generando una experiencia diferente en cada nuevo intento.

Este sistema forma parte de la propuesta principal del prototipo y busca demostrar una alternativa a los modelos tradicionales de dificultad basados únicamente en el incremento de parámetros numéricos o en la disminución de la dificultad por fracaso para hacer más fácil el nivel.

3.7. Créditos

Tabla 7.

Créditos

Integrante	Rol
Joan Dután	Programador principal
Alejandro Ortiz	Diseño y arte visual
Juan José Bacuilima	Modelado 3D y animación
Alexander Abad	Integración de sistemas

Capítulo IV

4. Manual del programador

4.1. Arquitectura general del sistema

El prototipo de Inkfall fue desarrollado utilizando el motor Unity, empleando una arquitectura basada en componentes, donde cada objeto de la escena está conformado por distintos componentes encargados de administrar funcionalidades específicas. Este enfoque permitió integrar recursos propios con assets de terceros, facilitando la implementación de las mecánicas del videojuego y reduciendo el tiempo de desarrollo de sistemas ya consolidados.

Para la construcción del prototipo se utilizaron diferentes recursos externos como base para determinados sistemas. El controlador del jugador del primer nivel se implementó a partir de un asset de disparos en primera persona, el cual proporciona funcionalidades como el movimiento del personaje, el control de la cámara, el sistema de disparo y el cambio de armas. De manera similar, el comportamiento de los enemigos de ambos niveles se desarrolló utilizando assets especializados en inteligencia artificial, los cuales fueron adaptados para responder a las necesidades específicas del proyecto.

Sobre esta base se implementaron los sistemas propios de Inkfall, los cuales constituyen el principal aporte del desarrollo. Entre ellos se encuentran el sistema de recolección de armas, las habilidades especiales asociadas a cada arma, el sistema de trampas, el sistema de transformación dinámica basada en intentos fallidos, la mecánica del encendedor, el sistema de llaves, las puertas con interacción, el rompecabezas del piano, el sistema de palancas y el diseño completo de ambos niveles en un entorno de caja blanca.

Con el propósito de facilitar la integración de elementos artísticos durante futuras etapas del desarrollo, todos los objetos interactivos fueron estructurados como prefabs independientes. Esto incluye armas, enemigos, interfaces de usuario, elementos interactivos y componentes del escenario, permitiendo que los modelos temporales utilizados durante el prototipado puedan sustituirse posteriormente por recursos gráficos definitivos sin modificar la lógica implementada.

Las siguientes secciones describen de manera individual la implementación de cada uno de los niveles que conforman el prototipo, detallando los sistemas desarrollados, los assets utilizados y la integración de las diferentes mecánicas dentro de cada escena.

4.2. Tecnologías, motores y herramientas utilizadas

El desarrollo de la base técnica de Inkfall se realizó utilizando Unity como motor de desarrollo y C# como lenguaje principal de programación. La implementación de las mecánicas, sistemas de interacción, inteligencia artificial, progresión dinámica y demás componentes funcionales se llevó a cabo mediante scripts desarrollados específicamente para el proyecto. Asimismo, la construcción de los escenarios en caja blanca se realizó utilizando Blender para el modelado de la geometría básica, mientras que Visual Studio se empleó como entorno de desarrollo para la programación y depuración del código.

Adicionalmente, el proyecto hace uso de herramientas y componentes nativos de Unity, como el sistema de físicas, el sistema de entrada (New Input System), la navegación mediante NavMesh, el Universal Render Pipeline (URP) y el sistema de prefabs, además de assets especializados que proporcionan funcionalidades base para determinados sistemas del videojuego y que posteriormente fueron adaptadas mediante programación propia. La Tabla 3 presenta un resumen de las principales tecnologías, motores y herramientas utilizadas

durante el desarrollo del prototipo, incluyendo las versiones empleadas y la fecha de congelamiento del entorno de desarrollo.

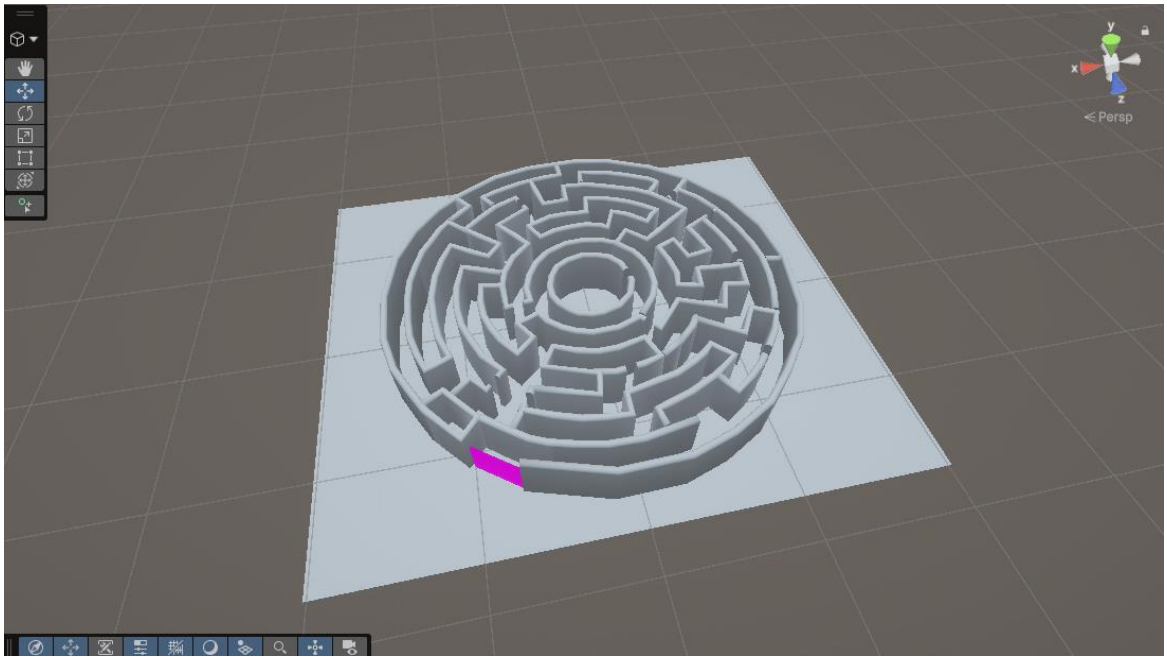
4.3. Desarrollo de sistemas principales: nivel shooter

El primer nivel del prototipo corresponde a un escenario de combate en primera persona desarrollado completamente en un entorno de caja blanca. Su propósito es implementar y validar las mecánicas base del videojuego, incluyendo el desplazamiento del jugador, el sistema de combate, la inteligencia artificial de los enemigos, las trampas del escenario y el sistema de transformación dinámica basada en intentos fallidos.

Para la construcción del escenario se diseñó un laberinto utilizando Blender, empleando únicamente geometría básica con el objetivo de priorizar la implementación de las mecánicas sobre los elementos visuales. Una vez finalizado el modelado, el escenario fue importado a Unity, donde se configuraron los componentes físicos necesarios para permitir la navegación del jugador y el desplazamiento de los enemigos mediante navegación automática.

Ilustración 5

Modelo base del laberinto

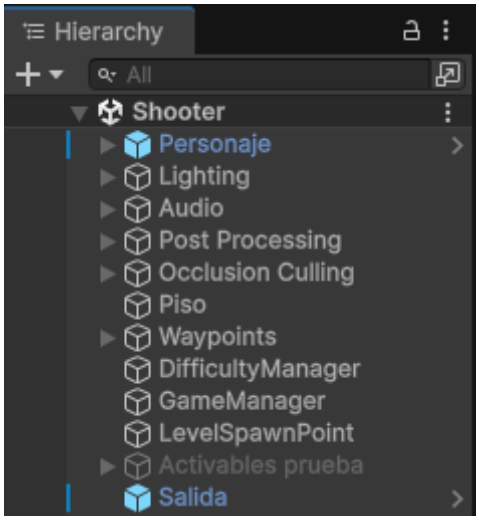


Como base para el desarrollo del jugador se utilizó un asset de disparos en primera persona, encargado de administrar el movimiento, la cámara, el sistema de disparo y el cambio de armas. Sobre esta estructura se implementaron los sistemas propios del proyecto, incorporando nuevas mecánicas y adaptando el comportamiento del prototipo a las necesidades de Inkfall.

La escena fue organizada mediante distintos GameObjects responsables de controlar el flujo general del nivel, incluyendo el jugador, los enemigos, los puntos de aparición, las armas disponibles, las trampas, la salida del nivel y el sistema de transformación dinámica basada en intentos fallidos. Esta estructura permitió mantener separados los diferentes sistemas y facilitar su integración dentro del escenario.

Ilustración 6

Organización de la jerarquía del primer nivel



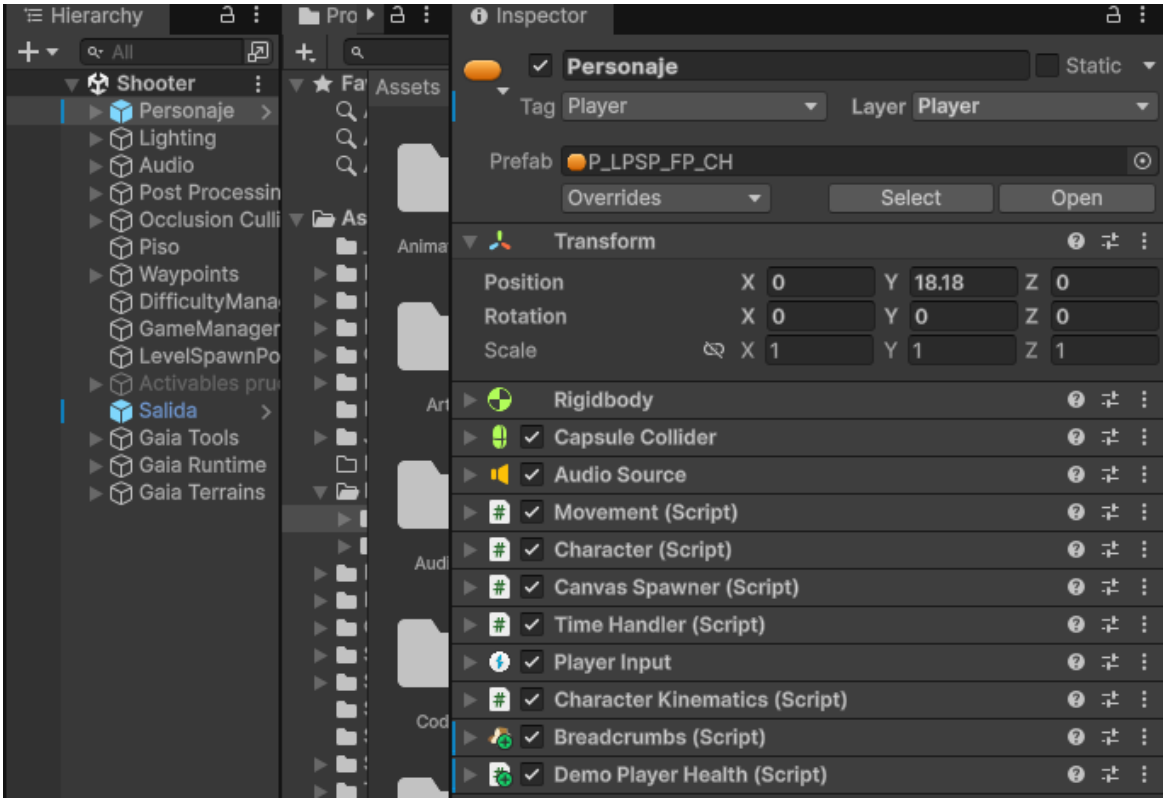
4.3.1. Sistema del jugador

El sistema del jugador fue implementado utilizando como base el asset “Low Poly Shooter Pack”, el cual proporciona la estructura principal para el control del personaje dentro del escenario. Este asset incorpora funcionalidades como el desplazamiento del jugador, el control de la cámara, el sistema de disparo, el cambio de armas y la administración básica de los componentes necesarios para un videojuego de disparos en primera persona.

Dentro de la escena, el jugador se encuentra representado por un GameObject principal que agrupa todos los componentes necesarios para su funcionamiento. Entre estos se encuentran el controlador de movimiento, la cámara principal, el administrador de armas, el sistema de interacción y los componentes relacionados con la detección de colisiones. Esta organización permitió integrar nuevas funcionalidades sin modificar la estructura general del controlador proporcionado por el asset.

Ilustración 7

Vista en inspector del gameObject Personaje



Durante el desarrollo del prototipo se realizaron adaptaciones sobre este controlador para permitir la integración de las mecánicas propias de Inkfall. Entre ellas se encuentran el sistema de recolección de armas, la incorporación de habilidades especiales para cada arma, la comunicación con el sistema de transformación dinámica basada en intentos fallidos y la interacción con los diferentes elementos presentes dentro del escenario.

La utilización de este controlador permitió centrar el desarrollo en la implementación de nuevas mecánicas sin necesidad de construir desde cero funcionalidades ya consolidadas, aprovechando una base estable sobre la cual se integraron el resto de sistemas del videojuego.

Ilustración 8

Vista principal del jugador en caja blanca

4.3.2. Sistema de armas

El sistema de armas fue desarrollado sobre la estructura proporcionada por el controlador en primera persona, ampliando las funcionalidades del asset para adaptarlo a las necesidades del prototipo. Además del sistema de disparo incluido en la base, se implementó un mecanismo propio para la recolección de armas distribuidas dentro del escenario, permitiendo que el jugador obtenga nuevo equipamiento conforme avanza por el nivel.

Cada arma se encuentra representada mediante un prefab independiente ubicado estratégicamente dentro del laberinto. Para su recolección se desarrolló un sistema de interacción que detecta cuando el jugador entra en contacto con el arma disponible. Una vez realizada la interacción, el sistema reemplaza el arma actualmente equipada, actualiza el modelo visible para el jugador y habilita automáticamente las funcionalidades correspondientes al nuevo armamento. Este comportamiento fue implementado mediante

scripts propios que gestionan el intercambio del arma y la sincronización con el controlador del jugador.

Ilustración 9

Prefab de escopeta

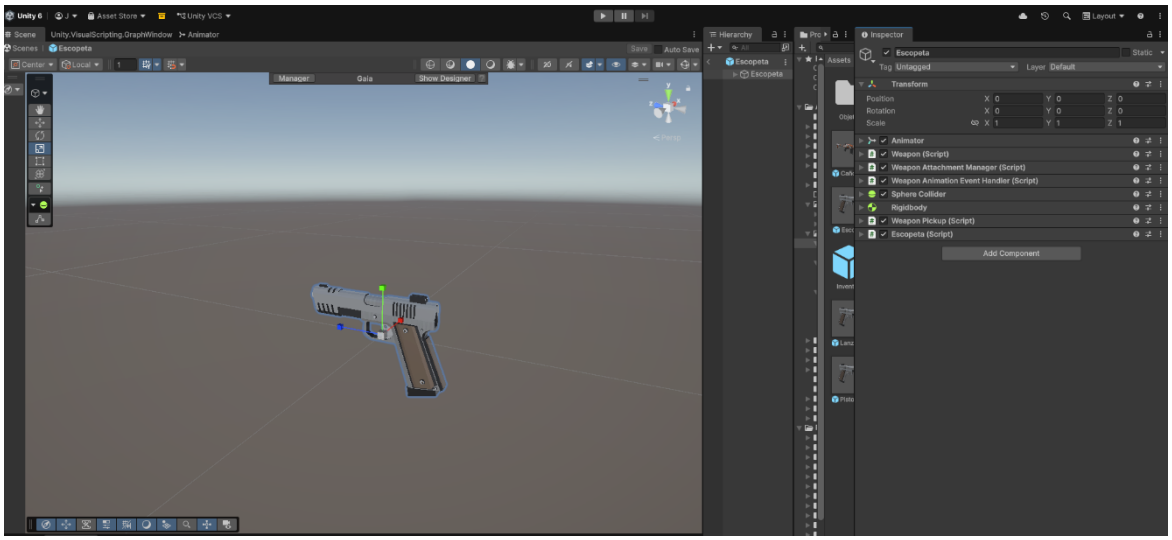


Ilustración 10

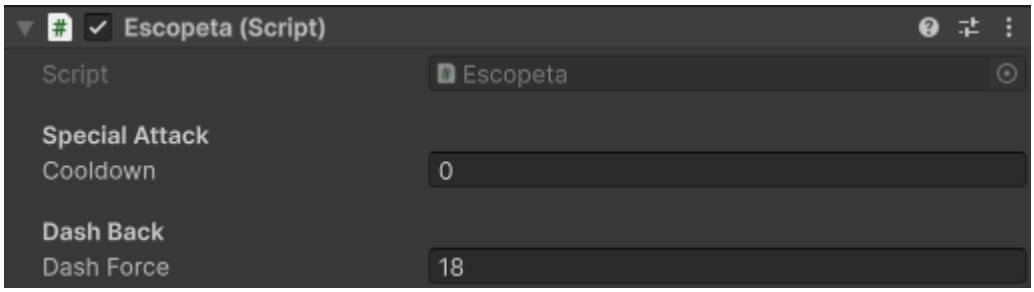
Sistema de recolección y cambio de armas

```
11      void Update()
12      {
13          if (!playerInRange)
14              return;
15
16          // Si el arma ya está dentro de un inventario, no hacer nada
17          if (GetComponentInParent<Inventory>() != null)
18              return;
19
20          if (Input.GetKeyDown(KeyCode.E))
21          {
22              if (inventory != null)
23              {
24                  inventory.SwapWeapon(gameObject);
25
26                  // evitar doble activación
27                  playerInRange = false;
28              }
29          }
30      }
31
```

Como complemento al sistema de recolección, cada arma incorpora una habilidad especial diseñada para ofrecer diferentes alternativas durante el combate. Estas habilidades fueron implementadas de manera independiente, permitiendo que cada una posea una lógica de funcionamiento específica sin afectar el comportamiento del resto del sistema.

Ilustración 11

Funcionamiento del script Escopeta



Las habilidades implementadas son las siguientes:

Dash: permite realizar un desplazamiento rápido hacia atrás, facilitando la evasión de ataques y el reposicionamiento durante el combate.

Ilustración 12

Método ExecuteSpecial

```
protected override void ExecuteSpecial()
{
    Transform player = transform.root;

    Rigidbody rb =
        player.GetComponent<Rigidbody>();

    if (rb != null)
    {
        // mantener velocidad vertical
        rb.linearVelocity = new Vector3(
            0f,
            rb.linearVelocity.y,
            0f
        );

        // dash hacia atr♦s
        rb.AddForce(
            -player.forward * dashForce,
            ForceMode.Impulse
        );
    }

    Debug.Log("Shotgun Dash Back!");
}
}
}
```

Teletransporte: desplaza instantáneamente al jugador hacia una posición aleatoria ubicada dentro de un radio previamente definido, permitiendo escapar de situaciones de riesgo o modificar la estrategia de combate.

Ilustración 13

Método de teletransporte aleatorio

```
protected override void ExecuteSpecial()
{
    Transform player = transform.root;

    // Direcci♦n base
    Vector3 direction = player.forward;

    // Variaci♦n aleatoria
    direction += new Vector3(
        Random.Range(-0.2f, 0.2f),
        0f,
        Random.Range(-0.2f, 0.2f)
    );

    direction.Normalize();

    Vector3 targetPosition;

    RaycastHit hit;

    // Detectar pared
    if (Physics.Raycast(
        player.position,
        direction,
        out hit,
        teleportDistance,
        collisionLayers))
    {
        targetPosition =
            hit.point -
            direction * wallOffset;
    }
    else
    {
        targetPosition =
            player.position +
            direction *
            teleportDistance;
    }
```

Gancho: lanza un proyectil que, al impactar sobre un enemigo, atrae al jugador hacia la posición del objetivo, reduciendo rápidamente la distancia entre ambos e incorporando una nueva alternativa de movilidad dentro del escenario.

Ilustración 14

Método de funcionamiento del proyectil de gancho

```
18      private void OnCollisionEnter(Collision collision)
19      {
20          if (hooked)
21              return;
22
23          if (collision.collider.CompareTag("Enemy"))
24          {
25              hooked = true;
26
27              GameObject player =
28                  GameObject.FindGameObjectWithTag("Player");
29
30              StartCoroutine(
31                  PullPlayer(
32                      player.transform,
33                      collision.transform
34                  )
35              );
36          }
37          else
38          {
39              Destroy(gameObject);
40          }
41      }
```

Proyectil explosivo: dispara un proyectil que, al colisionar, genera una explosión capaz de aplicar una fuerza de impulso tanto al jugador como a los enemigos cercanos. Esta mecánica puede utilizarse tanto para controlar grupos de enemigos como para impulsar al jugador hacia otras zonas del escenario.

Ilustración 15

Método de proyectil explosivo

```
protected override void ExecuteSpecial()
{
    // Crear proyectil
    Transform player =
transform.root;

    Vector3 spawnPos =
        player.position +
        player.forward * 1.5f +
        Vector3.up * 1.5f;

    GameObject projectile =
        Instantiate(
            projectilePrefab,
            spawnPos,
            player.rotation
        );

    // Collider proyectil
    Collider projectileCol =
        projectile.GetComponent<Collider>();

    // Collider player
    Collider playerCol =
        transform.root.GetComponent<Collider>();

    // Ignorar colisi*n
    if (projectileCol != null &&
        playerCol != null)
    {
        Physics.IgnoreCollision(
            projectileCol,
            playerCol
        );
    }

    // Rigidbody
    Rigidbody rb =
        projectile.GetComponent<Rigidbody>();

    if (rb != null)
    {
        rb.linearVelocity =
            player.forward * shootForce;
    }

    Debug.Log("Shockwave Shot!");
}
}
```

Cada una de estas habilidades fue implementada mediante scripts independientes asociados a las diferentes armas, permitiendo mantener una arquitectura modular y facilitando su integración con el sistema general de combate.

Ilustración 16

Vista de proyectil explosivo en escena

Con el propósito de mantener una estructura organizada durante el desarrollo, todas las armas fueron almacenadas como prefabs reutilizables, separando los modelos tridimensionales, los parámetros configurables y la lógica de programación. Esta organización permitió que los elementos visuales utilizados durante el prototipado pudieran reemplazarse posteriormente sin modificar el funcionamiento de las mecánicas implementadas.

La integración entre el controlador del jugador, el sistema de recolección y las habilidades especiales permitió construir un sistema de armas flexible, capaz de modificar dinámicamente las capacidades del jugador conforme avanza por el escenario y proporcionando diferentes formas de afrontar los encuentros con los enemigos.

4.3.3. Sistema de enemigos e inteligencia artificial

El comportamiento de los enemigos del primer nivel fue implementado utilizando el asset “Breadcrumb AI”, el cual proporciona las funcionalidades necesarias para la navegación autónoma dentro del escenario y la interacción con el jugador. Este sistema permitió disponer de una base funcional para el desplazamiento de los enemigos, concentrando el desarrollo en la integración de las mecánicas propias del prototipo.

Cada enemigo se encuentra representado por un prefab independiente compuesto por los elementos necesarios para su funcionamiento, incluyendo el modelo tridimensional, los componentes de navegación, los scripts de inteligencia artificial y el sistema de vida. Estos prefabs fueron distribuidos dentro del escenario mediante puntos de aparición previamente definidos durante el diseño del nivel.

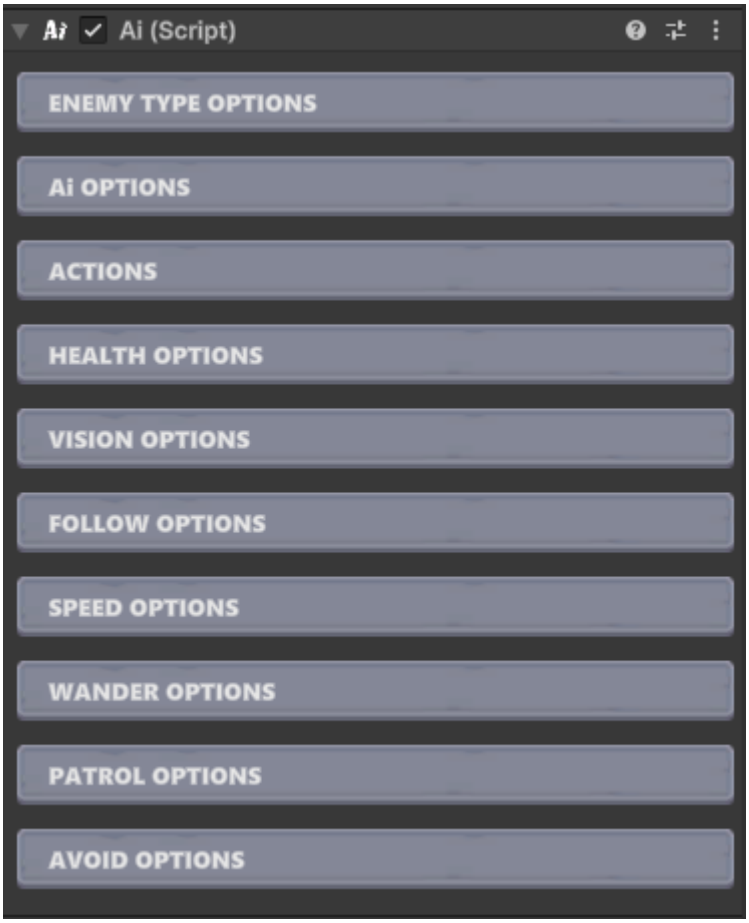
Ilustración 17

Prefab de enemigo melee

El comportamiento de los enemigos se basa en una serie de estados que determinan su respuesta frente a la presencia del jugador. Cuando el jugador ingresa dentro del rango de detección, el enemigo inicia la persecución utilizando el sistema de navegación del asset. Una vez alcanzada la distancia de ataque, ejecuta las acciones correspondientes hasta que el jugador abandona el área de detección o alguno de los dos personajes es eliminado.

Ilustración 18

Vista en inspector de funcionamiento de IA en el primer nivel



Con el propósito de integrar este sistema con las mecánicas desarrolladas para Inkfall, los enemigos fueron adaptados para responder correctamente al sistema de combate implementado. Esto permitió recibir daño proveniente de las diferentes armas del jugador,

reaccionar a las habilidades especiales y ejecutar la secuencia de eliminación correspondiente cuando su nivel de vida alcanza el valor mínimo establecido.

La utilización de prefabs permitió mantener una estructura uniforme para todos los enemigos presentes en el nivel, facilitando su distribución dentro del escenario y asegurando un comportamiento consistente durante la ejecución de la partida.

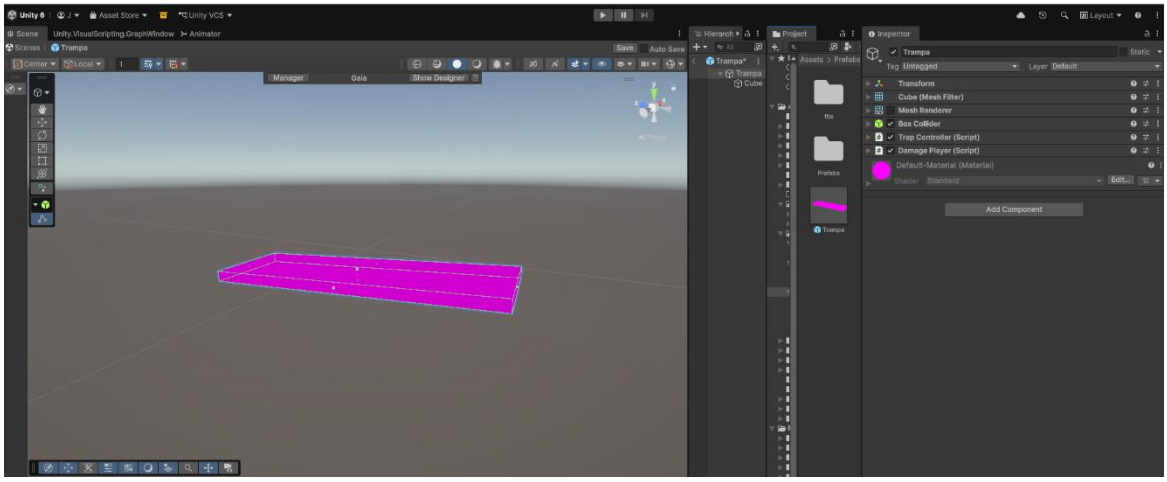
4.3.4. Sistema de trampas

Con el propósito de incrementar el nivel de desafío del escenario y fomentar la exploración cuidadosa del entorno, se implementó un sistema de trampas basado en pinchos distribuidos estratégicamente a lo largo del laberinto. Estas trampas constituyen obstáculos permanentes dentro del nivel y obligan al jugador a sincronizar sus movimientos con el comportamiento del escenario para evitar recibir daño.

Cada trampa fue desarrollada como un prefab independiente, permitiendo su reutilización en diferentes zonas del mapa sin necesidad de duplicar la lógica de programación. La estructura del prefab integra el modelo tridimensional, los colliders necesarios para la detección del jugador y los scripts encargados de controlar su comportamiento, facilitando la organización del proyecto y la distribución de las trampas durante el diseño del nivel.

Ilustración 19

Prefab de trampa del primer nivel



El funcionamiento del sistema se basa en un ciclo continuo de activación y desactivación controlado mediante temporizadores. Durante el estado de activación, los pinchos emergen gradualmente desde el suelo, permaneciendo visibles durante un intervalo de tiempo previamente definido. Posteriormente regresan a su posición inicial y permanecen desactivados hasta iniciar nuevamente el ciclo, generando un patrón repetitivo que el jugador debe identificar para desplazarse de forma segura.

La detección del jugador se realiza mediante colliders configurados como Trigger. Cuando el jugador permanece sobre la trampa mientras esta se encuentra activa, el sistema aplica daño de forma continua hasta que abandona el área de detección o la trampa vuelve a su estado inactivo. Esta lógica permite que el daño dependa tanto de la posición del jugador como del estado actual de la trampa, generando una interacción coherente con el resto de las mecánicas implementadas.

Ilustración 20

Vista en inspector del script Trap Controller

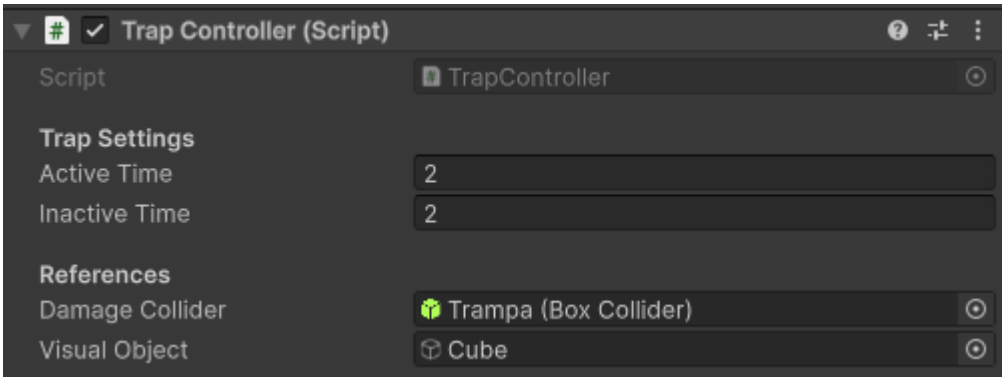
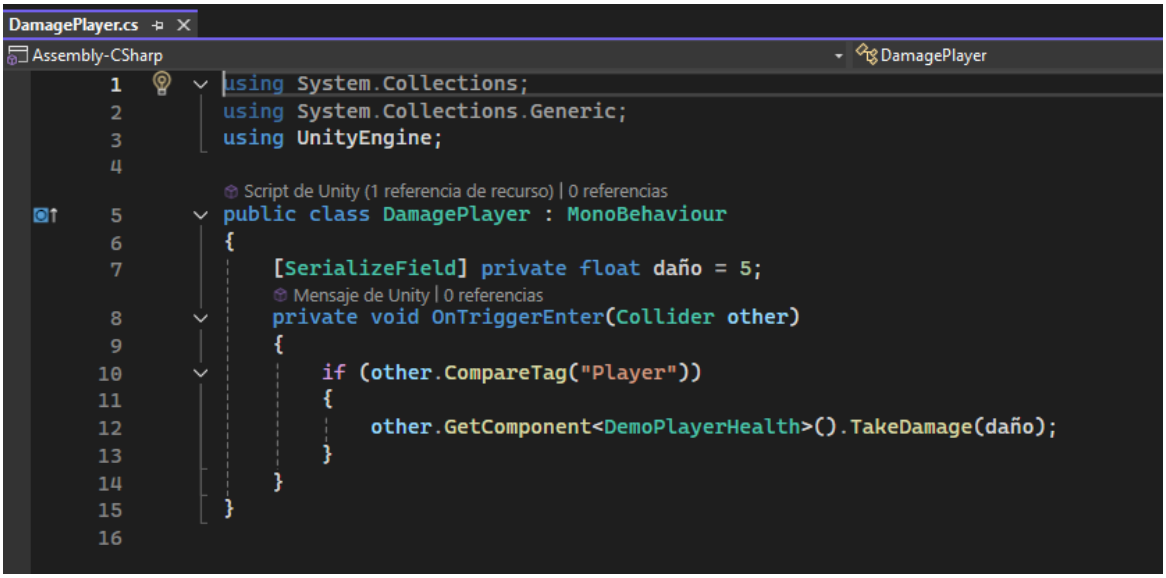


Ilustración 21

Método DamagePlayer del sistema de trampas del primer nivel



La utilización de prefabs permitió distribuir las trampas en diferentes sectores del laberinto manteniendo un comportamiento uniforme durante toda la partida. De esta manera, el sistema contribuye a aumentar la complejidad del escenario sin requerir configuraciones individuales para cada instancia colocada dentro del nivel.

Ilustración 22

Vista de trampa en escena en caja blanca

4.3.5. Desarrollo de mecánicas: sistema de dificultad

El sistema de dificultad implementado en Inkfall utiliza el número de muertes acumuladas por el jugador como variable de entrada para determinar los cambios que se aplican durante los siguientes intentos de la partida. El sistema registra cada muerte y, de acuerdo con el umbral alcanzado, determina el perfil de dificultad correspondiente a la progresión del juego. El sistema registra cada muerte y, de acuerdo con el umbral alcanzado, determina la configuración de dificultad correspondiente a la progresión del juego. Cada perfil contiene una lógica específica para aplicar los cambios asociados a un determinado estado de la progresión, permitiendo que las condiciones de la partida cambien progresivamente después de cada fracaso.

La adaptación se estructura mediante cuatro perfiles de dificultad implementados a partir de una clase base común. Estos perfiles permiten configurar progresivamente las condiciones del juego, incluyendo dos configuraciones diferentes de escenario mediante

instancias independientes de DifficultyProfile1. Los cambios implementados no se limitan a la modificación de parámetros numéricos, sino que reconfiguran las condiciones de juego mediante modificaciones en el entorno, la interfaz, las mecánicas de combate y el comportamiento de los enemigos. De esta manera, el sistema modifica progresivamente el desafío experimentado por el jugador a partir de los intentos fallidos, utilizando la muerte como condición de transición entre los diferentes estados de dificultad.

Para administrar este comportamiento se desarrolló un controlador central denominado `DifficultyManager`, implementado mediante el patrón Singleton. Esta elección permite mantener una única instancia del gestor durante la sesión de juego y conservar el estado asociado a la progresión de dificultad mientras se cargan nuevamente las escenas. El objeto que contiene el gestor utiliza `DontDestroyOnLoad`, evitando su destrucción durante los cambios de escena y permitiendo conservar el valor acumulado de `deathCount`. De esta manera, el número de muertes registrado puede utilizarse como condición para determinar el perfil de dificultad que debe aplicarse en los siguientes intentos.

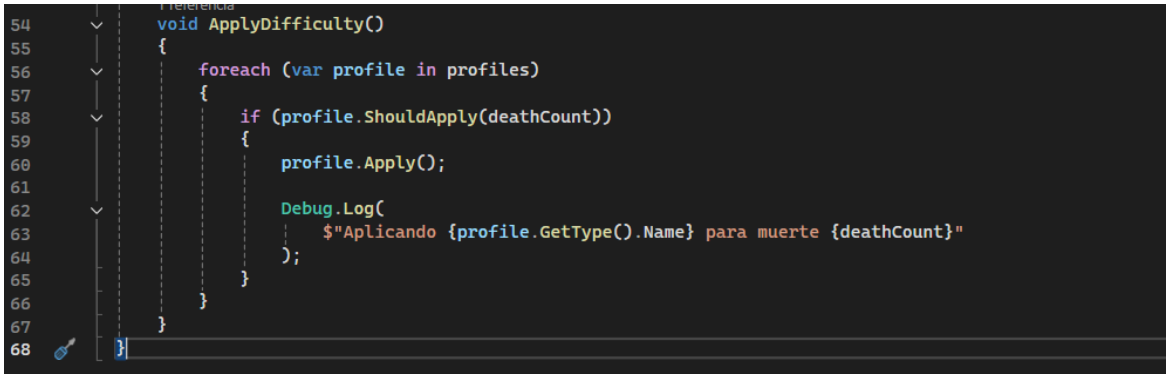
El ciclo de vida del gestor se encuentra asociado a la ejecución de la aplicación. Durante su inicialización, `DifficultyManager` comprueba si ya existe una instancia y, en caso de detectar una instancia previa diferente, destruye el objeto duplicado. Si no existe otra instancia, establece su referencia como instancia activa y configura su persistencia entre escenas. Además, el gestor se suscribe al evento `SceneManager.sceneLoaded`, lo que permite ejecutar la lógica de aplicación de dificultad cada vez que se carga una escena. Cuando el componente se deshabilita, la suscripción al evento es eliminada para evitar referencias persistentes innecesarias.

El contador `deathCount` se incrementa cuando se recibe el evento `DemoPlayerHealth.OnPlayerDeath`. Posteriormente, durante la carga de una escena, el

gestor vuelve a asociar el evento de muerte del jugador y ejecuta `ApplyDifficulty()`, método encargado de recorrer los perfiles disponibles y aplicar aquellos cuya condición de activación coincide con el número de muertes acumuladas. Esta estructura permite mantener la progresión de dificultad durante la sesión y asociar cada transformación con el intento correspondiente del jugador.

Ilustración 23

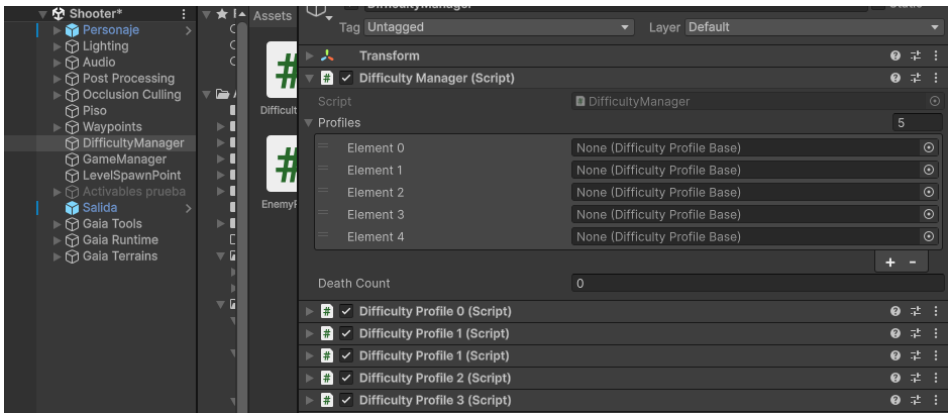
Método ApplyDifficulty del sistema de dificultad del primer nivel



La arquitectura del sistema se basa en perfiles de dificultad independientes. Cada perfil hereda de una clase base común e implementa únicamente la lógica correspondiente a un nivel específico de dificultad. Durante la carga de la escena, el gestor obtiene automáticamente los componentes que heredan de DifficultyProfileBase asociados al objeto que contiene el sistema de dificultad y ejecuta aquellos cuya condición de activación se cumple. Esta estructura permite incorporar diferentes configuraciones de dificultad mediante perfiles independientes, manteniendo separada la lógica de gestión de la progresión respecto de las modificaciones específicas de cada estado.

Ilustración 24

Vista en inspector de Difficulty Manager

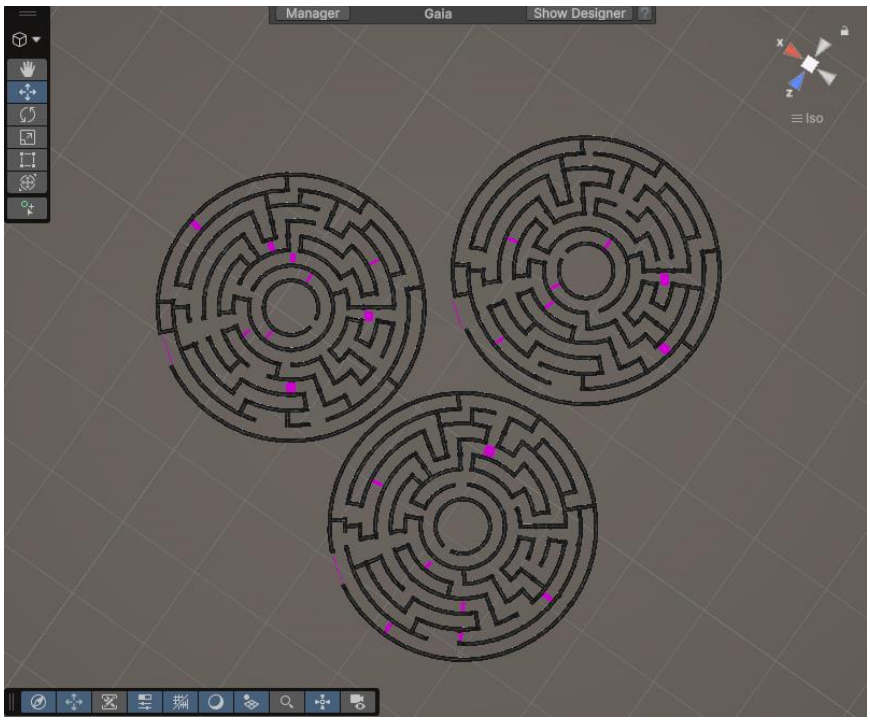


4.3.5.1. Primera muerte: modificación de la interfaz

Después de la primera muerte del jugador se activa la primera instancia de DifficultyProfile1, que introduce modificaciones tanto en la interfaz de usuario como en la configuración del escenario. En primer lugar, el método ActivateDelayedUI() deshabilita el componente HealthUI, utilizado para representar normalmente el estado de vida del jugador, y habilita HealthUIDelayed, modificando la forma en que se representa visualmente el daño recibido. De manera similar, el método ActivateFakeAmmo() deshabilita el componente TextAmmunitionCurrent del sistema de armas y habilita TextAmmunitionFake, sustituyendo la representación convencional de la munición por una versión modificada. Además, el perfil instancia el levelPrefab configurado en el Inspector, permitiendo cargar una primera variante del escenario. De esta manera, la primera transformación dinámica modifica la información visual proporcionada al jugador y las condiciones espaciales del nivel durante el siguiente intento.

Ilustración 25

Comparación de prefabs de laberintos



4.3.5.2. Segunda muerte: nuevos tipos de enemigos

Después de la segunda muerte se activa una segunda instancia de DifficultyProfile1, configurada con un levelPrefab diferente al utilizado durante el intento posterior a la primera muerte. Aunque se mantiene la estructura general del escenario tipo laberinto, esta nueva configuración incorpora nuevos tipos de enemigos mediante los sistemas de aparición definidos en el propio prefab del escenario. De esta manera, la transformación dinámica no se limita a modificar la distribución espacial del nivel, sino que amplía la composición de los adversarios que el jugador puede encontrar durante la partida.

El cambio se implementa reutilizando la lógica de programación de DifficultyProfile1, cuyo método SpawnLevel() instancia el prefab correspondiente en el objeto LevelSpawnPoint. Cada variante del escenario cuenta con sus propios sistemas de

aparición configurados para generar los enemigos definidos para esa etapa de la progresión. Como consecuencia, después de la segunda muerte el jugador se enfrenta a una combinación diferente de amenazas, lo que modifica las condiciones de combate y exige adaptar las estrategias utilizadas durante los intentos anteriores. Esta transformación permite incrementar la dificultad mediante la incorporación progresiva de nuevos comportamientos y tipos de enemigos, manteniendo la lógica general de gestión de la progresión separada de la configuración específica de cada escenario.

4.3.5.3. Tercera muerte: habilidades especiales aleatorias

Después de la tercera muerte se activa DifficultyProfile2, incorporando un sistema de activación aleatoria de habilidades especiales asociadas a las armas del jugador. Para ello, el perfil habilita el componente RandomSpecialActivator, encargado de obtener los objetos que implementan BaseWeaponSpecial y controlar la activación periódica de una de estas habilidades. El sistema utiliza un intervalo de diez segundos y, una vez transcurrido, selecciona aleatoriamente una de las habilidades especiales disponibles y ejecuta su método ForceSpecialAttack(). Posteriormente, el temporizador se reinicia y el proceso continúa durante la partida. Adicionalmente, el perfil habilita los elementos SpecialCooldownUI correspondientes al sistema de activación aleatoria, permitiendo representar visualmente el tiempo restante hasta la siguiente activación. Esta transformación introduce eventos impredecibles durante el combate, ya que las habilidades especiales pueden ejecutarse automáticamente y modificar temporalmente las condiciones de juego durante el intento del jugador.

4.3.5.4. Cuarta muerte: enemigos con mejoras aleatorias

Después de la cuarta muerte se activa DifficultyProfile3, que modifica las condiciones de combate mediante la carga de una configuración del escenario cuyos spawners se encuentran asociados a enemigos con el componente EnemyRandomBuff. De esta manera, el perfil no modifica directamente los enemigos que ya se encuentran en la escena, sino que carga un prefab de escenario configurado previamente con spawners que generan las variantes de enemigos preparadas para esta etapa de la progresión.

El componente EnemyRandomBuff supervisa el estado de vida de cada enemigo y detecta el primer daño recibido durante el enfrentamiento. Una vez identificado este evento, el sistema selecciona aleatoriamente uno de tres efectos: incremento de la velocidad de seguimiento mediante followSpeed, aumento de la vida del enemigo mediante Health o activación temporal de invulnerabilidad mediante _IsInvincible. El efecto seleccionado se aplica de forma individual al enemigo que recibió el primer daño y el sistema evita que se active nuevamente sobre el mismo enemigo. Esta transformación modifica el comportamiento de los adversarios durante el combate, ya que el jugador debe adaptarse a una mejora que se determina aleatoriamente después de iniciar el enfrentamiento con cada enemigo.

Ilustración 26

Prefab de enemigo mejorado en último nivel de dificultad

Finalmente, después de la quinta muerte se alcanza el límite establecido para la progresión de dificultad del primer nivel. En este punto, el sistema presenta una pantalla de Game Over y finaliza el intento actual, obligando al jugador a comenzar nuevamente desde la configuración inicial del nivel. De esta manera, la acumulación de muertes funciona como una condición de transición entre diferentes estados de dificultad hasta alcanzar un límite definido, momento en el cual el ciclo de progresión se reinicia con las condiciones base del juego.

Tabla 8

Sistema de dificultad en el primer nivel

Muerte/Intento	Perfil aplicado	Modificaciones Implementadas
Primera muerte	DifficultyProfile1	Se modifica la representación visual de la información del jugador mediante la activación de una barra de vida con actualización retardada y una representación modificada de la munición. Además, se instancia una configuración específica del escenario mediante un levelPrefab definido en el Inspector.
Segunda muerte	DifficultyProfile1	Se reutiliza el mismo perfil de programación, pero se configura un levelPrefab diferente. El nuevo escenario incorpora, mediante sus propios sistemas de aparición, nuevos tipos de enemigos que amplían las amenazas presentes durante la partida y requieren que el jugador adapte sus estrategias de combate.
Tercera muerte	DifficultyProfile2	Se habilita RandomSpecialActivator, que selecciona aleatoriamente una habilidad especial de las armas disponibles y ejecuta su activación automáticamente cada diez segundos. También se muestra una interfaz que permite visualizar el tiempo restante hasta la siguiente activación.

Muerte/Intento	Perfil aplicado	Modificaciones Implementadas
Cuarta muerte	DifficultyProfile3	Se instancia un escenario cuyos spawners generan enemigos que incorporan el componente EnemyRandomBuff. Después de recibir el primer daño, cada enemigo obtiene aleatoriamente un aumento de velocidad, un incremento de vida o un periodo temporal de invulnerabilidad, modificando sus condiciones de combate durante el enfrentamiento.

La separación de las transformaciones de dificultad mediante perfiles y configuraciones independientes permitió desarrollar un sistema en el que cada configuración puede incorporarse o modificarse de manera separada. La reutilización de DifficultyProfile1 mediante diferentes instancias permite cargar distintas configuraciones del escenario durante la progresión, mientras que DifficultyProfile2 incorpora modificaciones relacionadas con las habilidades especiales de las armas y DifficultyProfile3 permite introducir enemigos con comportamientos mejorados mediante la configuración de los spawners del escenario. Esta arquitectura mantiene separada la gestión de la progresión basada en muertes de las modificaciones específicas aplicadas en cada intento, facilitando la incorporación de nuevas transformaciones en futuras ampliaciones del sistema.

4.3.6. Arquitectura de clases y componentes del nivel shooter

Con el propósito de representar de manera simplificada el flujo de funcionamiento del sistema de dificultad dinámica, se presenta el pseudocódigo correspondiente al proceso de evaluación de las muertes acumuladas y aplicación de los perfiles de dificultad. Este procedimiento permite visualizar la secuencia lógica mediante la cual el sistema determina las transformaciones que deben aplicarse en cada intento del jugador.

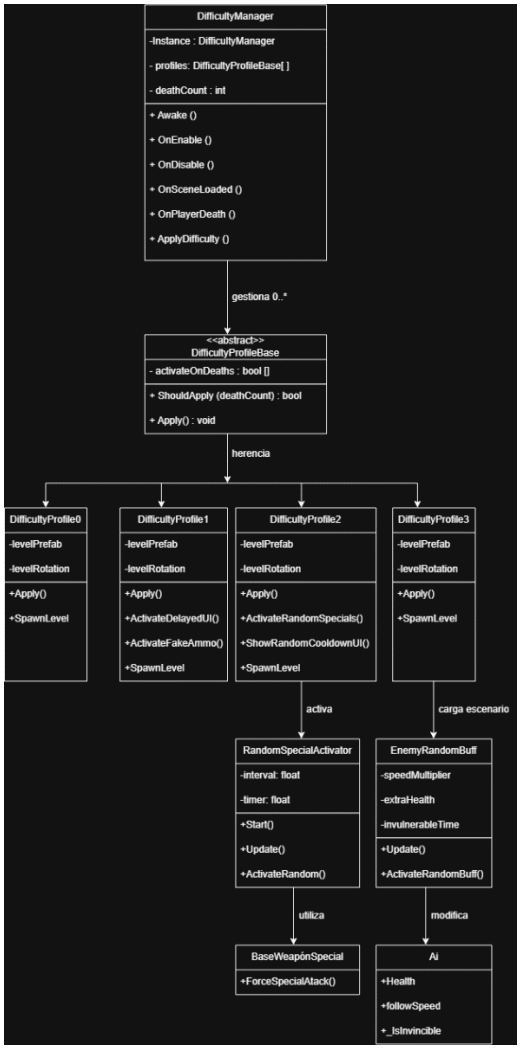
```
INICIO
  Inicializar DifficultyManager
  SI existe una instancia previa ENTONCES
    Destruir instancia duplicada
  SI NO
    Establecer instancia única
    Mantener instancia entre escenas
    Obtener perfiles de dificultad
  FIN SI
  Registrar eventos de carga de escena
  AL cargar una escena:
    Registrar evento de muerte del jugador
    Evaluar perfiles de dificultad
    PARA CADA perfil HACER
      SI corresponde al número de muertes ENTONCES
        Ejecutar Apply()
      FIN SI
    FIN PARA
  FIN AL
  AL producirse la muerte del jugador:
    Incrementar deathCount
  FIN AL
  SI deathCount = 0 ENTONCES
    Aplicar DifficultyProfile0
```

```
    SINO SI deathCount = 1 ENTONCES
        Aplicar primera instancia de DifficultyProfile1
    SINO SI deathCount = 2 ENTONCES
        Aplicar segunda instancia de DifficultyProfile1
    SINO SI deathCount = 3 ENTONCES
        Aplicar DifficultyProfile2
    SINO SI deathCount = 4 ENTONCES
        Aplicar DifficultyProfile3
    SINO SI deathCount = 5 ENTONCES
        Mostrar Game Over
        Reiniciar progresión
    FIN SI
FIN
```

El pseudocódigo representa el flujo general de funcionamiento del sistema de dificultad dinámica implementado en Inkfall. La variable deathCount actúa como condición de transición entre las diferentes configuraciones de dificultad. Después de cada muerte, el sistema incrementa el contador y, durante la carga del siguiente intento, evalúa las condiciones de activación configuradas en los perfiles. Cada perfil ejecuta de forma independiente las modificaciones correspondientes mediante su método Apply(). Una vez alcanzado el límite establecido de cinco muertes, se presenta la pantalla de Game Over y el jugador debe iniciar nuevamente la progresión desde la configuración base.

Ilustración 27

Diagrama de clases del sistema de dificultad del nivel 1

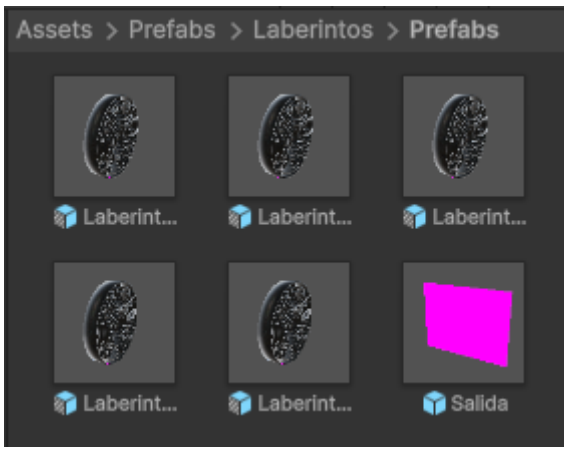


4.3.7. Diseño de nivel

Los escenarios fueron construidos mediante un conjunto de prefabs independientes, donde cada uno representa una configuración diferente del laberinto utilizada por el sistema de transformación dinámica basada en intentos fallidos. Esta organización permitió mantener separada la estructura física del escenario de la lógica encargada de gestionar su aparición durante la ejecución del juego.

Ilustración 28

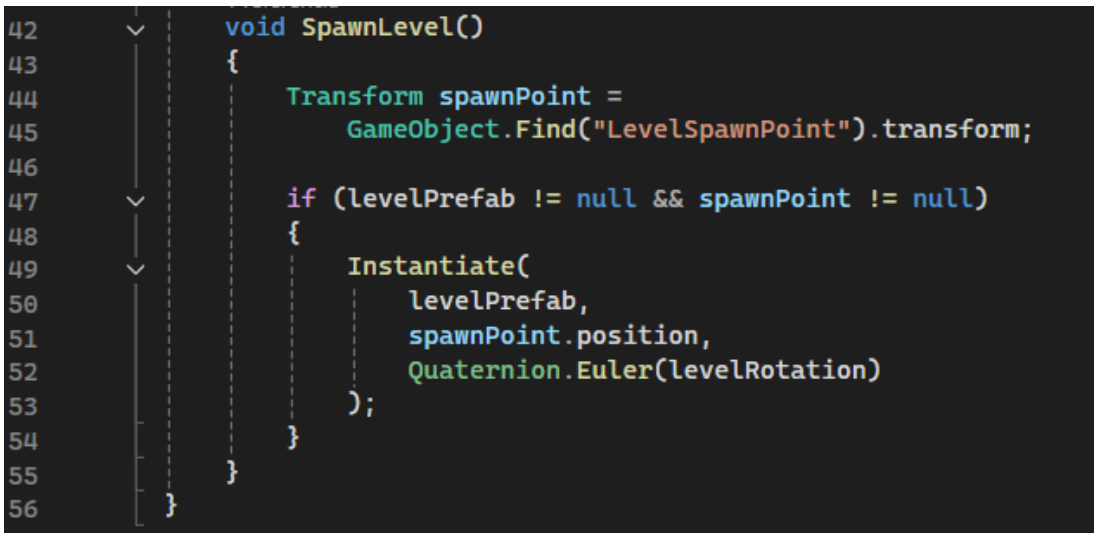
Vista en carpeta de los prefabs de laberintos en el primer nivel



Al iniciar una partida, o después de que el jugador pierde una vida, el sistema de dificultad determina qué configuración del laberinto debe utilizarse. Una vez seleccionado el nivel correspondiente, el prefab asociado es instanciado automáticamente en un punto de aparición previamente definido dentro de la escena. Este procedimiento permite reemplazar completamente el escenario sin necesidad de modificar manualmente los elementos presentes en el nivel.

Ilustración 29

Método SpawnLevel() del primer nivel

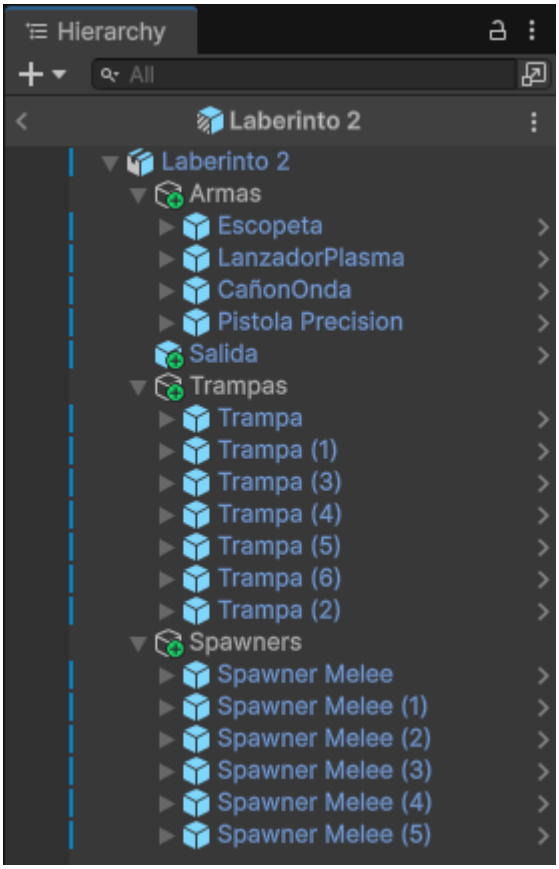


```
42      void SpawnLevel()
43      {
44          Transform spawnPoint =
45              GameObject.Find("LevelSpawnPoint").transform;
46
47          if (levelPrefab != null && spawnPoint != null)
48          {
49              Instantiate(
50                  levelPrefab,
51                  spawnPoint.position,
52                  Quaternion.Euler(levelRotation)
53              );
54          }
55      }
56  }
```

Cada prefab del laberinto contiene todos los objetos necesarios para el funcionamiento de la partida. Entre ellos se encuentran las trampas, las armas disponibles para el jugador, los puntos de aparición de enemigos y los diferentes elementos que intervienen en el desarrollo del nivel. Como resultado, cada configuración del escenario mantiene su propia distribución de obstáculos y recursos, permitiendo que el sistema de dificultad modifique la experiencia de juego mediante cambios en la organización espacial del entorno.

Ilustración 30

Vista en jerarquía de distribución del segundo laberinto



La utilización de prefabs independientes también facilitó el proceso de desarrollo en caja blanca, ya que permitió construir y ajustar cada versión del laberinto de manera aislada antes de integrarla al sistema general. Esto simplificó la incorporación de nuevos elementos y permitió realizar modificaciones sobre una configuración específica sin afectar el funcionamiento de las demás.

La distribución de enemigos, trampas y armas fue realizada manualmente durante el diseño del nivel, buscando mantener un equilibrio entre exploración, combate y riesgo. Cada punto de aparición fue colocado de acuerdo con la configuración particular del laberinto

correspondiente, permitiendo que la experiencia variara progresivamente conforme el sistema de dificultad introducía nuevas versiones del escenario.

Esta organización permitió desacoplar el diseño del escenario de la lógica de programación encargada de administrarlo, facilitando futuras modificaciones y proporcionando una estructura flexible para incorporar nuevos laberintos o variantes del nivel sin alterar el funcionamiento del resto de los sistemas implementados.

4.4. Desarrollo de sistemas principales: nivel de sigilo

El segundo nivel del prototipo corresponde a una experiencia de evasión en la que el jugador debe recorrer un entorno cerrado mientras administra una fuente de luz como recurso principal para sobrevivir. A diferencia del primer nivel, centrado en el combate, este escenario prioriza la exploración, la resolución de pequeños desafíos y la administración del riesgo mediante mecánicas que condicionan el avance del jugador.

La implementación del nivel se desarrolló utilizando una arquitectura modular dentro de Unity, donde cada mecánica fue construida como un sistema independiente. Entre los principales componentes se encuentran la mecánica del encendedor, el sistema de llaves y puertas, el rompecabezas del piano, las palancas, la inteligencia artificial del enemigo y el sistema de transformación dinámica basada en intentos fallidos. Cada uno de estos módulos se comunica con los demás mediante eventos, componentes y referencias controladas, permitiendo mantener un bajo acoplamiento entre los sistemas y facilitando futuras modificaciones.

El escenario fue construido en caja blanca utilizando geometría básica y prefabs reutilizables, permitiendo validar la lógica del juego sin depender de recursos artísticos definitivos. La distribución de los elementos interactivos y de los objetivos del nivel fue

diseñada para guiar progresivamente al jugador hacia la salida, obligándolo a interactuar con las diferentes mecánicas implementadas y generando una experiencia de juego completa desde el punto de vista funcional.

Ilustración 31

Vista general del segundo nivel en caja blanca

4.4.1. Sistema del encendedor

La mecánica principal del segundo nivel se basa en un encendedor portátil que el jugador debe mantener encendido durante su recorrido por el escenario. Este sistema fue desarrollado como un módulo independiente encargado de administrar el estado del encendedor, controlar su intensidad, gestionar el proceso de reencendido y establecer la interacción con la inteligencia artificial del enemigo.

El encendedor se encuentra implementado como un prefab integrado al GameObject del jugador, manteniendo una posición fija respecto a la cámara en primera persona. Esta estructura permite que el sistema acompañe permanentemente al jugador durante la partida,

facilitando la comunicación con el resto de los componentes del nivel sin depender de referencias externas.

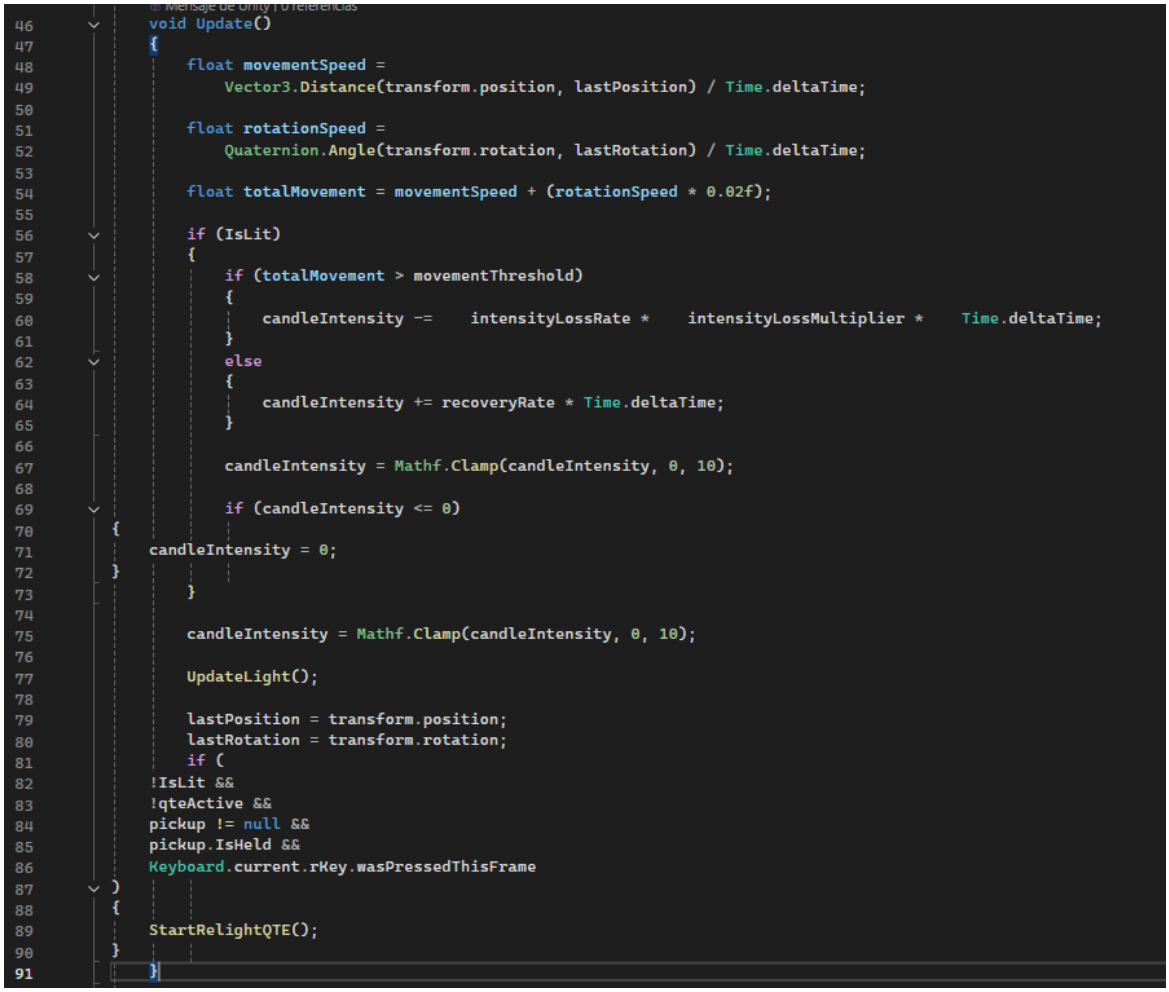
Ilustración 32

Vista en jerarquía del gameObject Player del segundo nivel

La lógica principal se encuentra implementada en la clase CandleVelocitySystem, responsable de controlar la intensidad de la llama, detectar el movimiento del jugador y administrar el estado general del encendedor. Este componente calcula continuamente el desplazamiento y la rotación del jugador para determinar la velocidad del movimiento. Cuando dicha velocidad supera un umbral previamente establecido, la intensidad de la llama disminuye progresivamente hasta extinguirse. En caso contrario, la intensidad se recupera de manera gradual hasta alcanzar nuevamente su valor máximo.

Ilustración 33

Método Update del sistema de apagado del encendedor

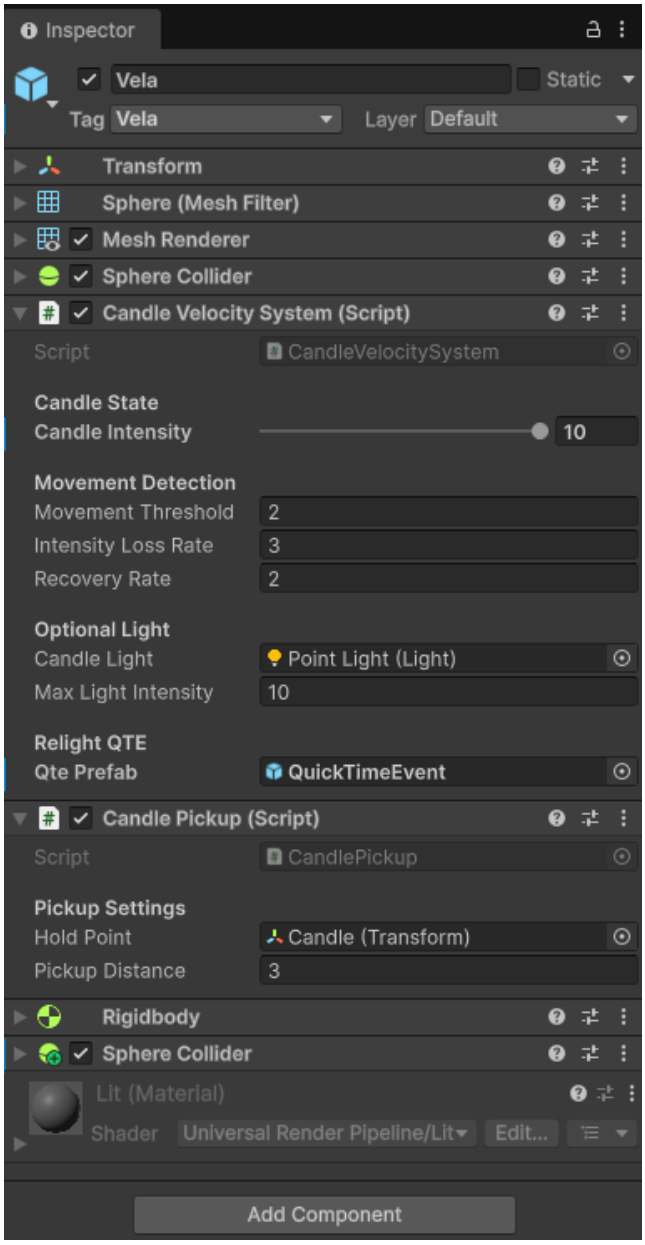


```
Mensaje de Unity | 0 referencias
void Update()
{
    float movementSpeed =
        Vector3.Distance(transform.position, lastPosition) / Time.deltaTime;

    float rotationSpeed =
        Quaternion.Angle(transform.rotation, lastRotation) / Time.deltaTime;

    float totalMovement = movementSpeed + (rotationSpeed * 0.02f);

    if (IsLit)
    {
        if (totalMovement > movementThreshold)
        {
            candleIntensity -=    intensityLossRate *    intensityLossMultiplier *    Time.deltaTime;
        }
        else
        {
            candleIntensity += recoveryRate * Time.deltaTime;
        }

        candleIntensity = Mathf.Clamp(candleIntensity, 0, 10);

        if (candleIntensity <= 0)
{
candleIntensity = 0;
}
    }

    candleIntensity = Mathf.Clamp(candleIntensity, 0, 10);

    UpdateLight();

    lastPosition = transform.position;
    lastRotation = transform.rotation;
    if (
!IsLit &&
!qteActive &&
pickup != null &&
pickup.IsHeld &&
Keyboard.current.rKey.wasPressedThisFrame
)
{
StartRelightQTE();
}
    }
```

Además de controlar el estado lógico del encendedor, el sistema actualiza dinámicamente la intensidad de la fuente de luz asociada mediante el componente Light de Unity. De esta forma, la iluminación visible para el jugador corresponde directamente al valor actual de intensidad almacenado por el sistema, proporcionando una representación visual coherente del estado de la mecánica.

Ilustración 34

Vista en inspector del GameObject Vela



Una vez que la intensidad alcanza el valor mínimo, el encendedor se considera apagado y deja de emitir iluminación. En este estado, el jugador puede iniciar el proceso de reencendido presionando la tecla correspondiente. Para ello, el sistema instancia automáticamente un prefab perteneciente al asset "Quick-Time-Event-Like-In-Dbd", el cual

presenta una secuencia interactiva inspirada en los eventos de habilidad utilizados en videojuegos de supervivencia. El jugador debe completar correctamente tres eventos consecutivos para restaurar completamente la intensidad del encendedor y reactivar su funcionamiento.

Ilustración 35

Método StartRelightQTE()

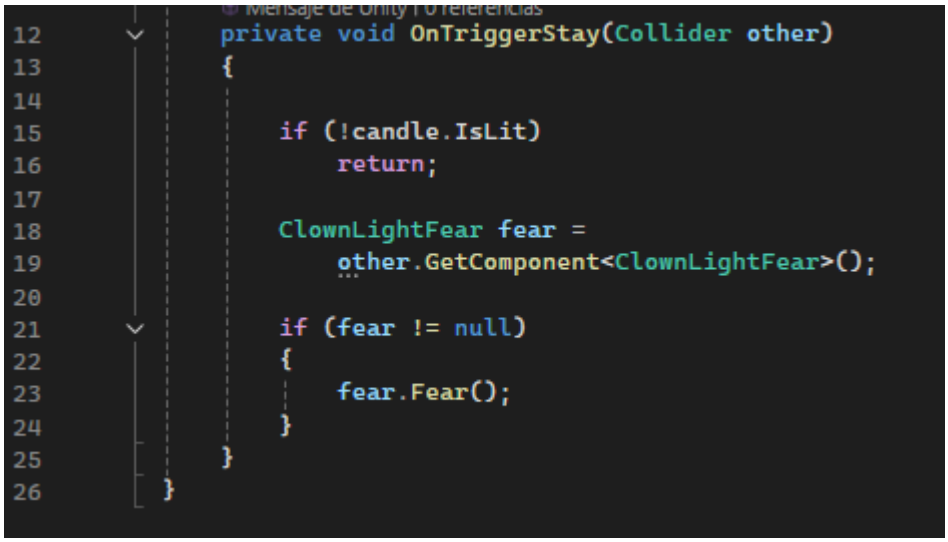
```
102     void StartRelightQTE()
103     {
104         qteActive = true;
105
106         Camera cam = Camera.main;
107         if (fpsController != null)
108         {
109             fpsController.isInteracting = true;
110         }
111
112         GameObject qte =
113             Instantiate(
114                 qtePrefab,
115                 cam.transform.position + cam.transform.forward * 0.05f,
116                 cam.transform.rotation
117             );
118
119         qte.transform.SetParent(cam.transform);
120
121         qte.transform.localPosition =
122             new Vector3(0f, 0f, 0.5f);
123
124         qte.transform.localRotation =
125             Quaternion.identity;
126
127         qte.transform.localScale =
128             Vector3.one *0.1f; // prueba 0.1, 0.05 o 0.02
129
130         QTE_Skillcheck_Behaviour skillcheck =
131         qte.GetComponentInChildren<QTE_Skillcheck_Behaviour>();
132
133         if (skillcheck != null)
134         {
135             skillcheck.OnCompleted += RelightCandle;
136         }
137     }
```

Como parte de la arquitectura del sistema, el encendedor incorpora un objeto hijo denominado FearZone, encargado de representar el área de influencia de la luz. Este objeto utiliza un collider configurado como Trigger y ejecuta la clase CandleFearZone, la cual verifica continuamente si el encendedor permanece encendido. Cuando un enemigo ingresa dentro del área de influencia y el encendedor continúa iluminando, el sistema invoca el

método correspondiente de la inteligencia artificial del payaso para activar su comportamiento de evasión. De esta manera, la luz deja de ser únicamente un elemento visual y pasa a convertirse en una mecánica que modifica directamente el comportamiento del enemigo.

Ilustración 36

Método OnTriggerStay() del GameObject FearZone



La implementación separa claramente la administración de la intensidad, la iluminación, la zona de influencia y el proceso de reencendido en componentes independientes. Esta organización facilita el mantenimiento del sistema y permite modificar cualquiera de estas funcionalidades sin afectar el funcionamiento general de la mecánica, manteniendo una arquitectura modular acorde con el resto del proyecto.

4.4.2. Sistema del piano

El rompecabezas del piano constituye uno de los mecanismos principales de progresión del segundo nivel. Su función es controlar el acceso a la siguiente zona del escenario mediante la resolución de una secuencia de notas musicales. Esta mecánica se

encuentra integrada dentro del flujo general del nivel, en el cual el jugador debe obtener una llave para acceder al sótano, activar el suministro eléctrico del piano mediante la caja de fusibles y, posteriormente, interpretar correctamente la melodía para desbloquear el acceso al sistema de alcantarillado.

La lógica del rompecabezas fue implementada mediante un controlador central denominado PianoManager y un conjunto de teclas independientes organizadas como objetos hijos del piano. Cada tecla incorpora el script PianoKey, encargado de gestionar la interacción con el jugador, reproducir la nota correspondiente y comunicar dicha acción al controlador principal. Esta arquitectura permite mantener desacoplada la interacción individual de cada tecla respecto al algoritmo encargado de validar la secuencia.

La interacción con las teclas utiliza el sistema de interacción implementado en el proyecto mediante la interfaz IInteractable. Antes de permitir la ejecución de una nota, el sistema verifica que el piano se encuentre energizado mediante la variable hasPower. Si el suministro eléctrico aún no ha sido activado, el jugador recibe un mensaje indicando que el instrumento no dispone de energía y no se procesa ninguna interacción.

Ilustración 37

Método Interact de las teclas del piano

```
15      public void Interact()
16      {
17          if (!PianoManager.Instance.hasPower)
18              return;
19
20          if (audioSource != null)
21              audioSource.Play();
22
23          PianoManager.Instance.PlayNote(note);
24      }
```

La alimentación eléctrica del piano depende de una caja de fusibles ubicada en el sótano del escenario. Esta mecánica utiliza el sistema de interruptores proporcionado por el asset Fps Horror Kit, sobre el cual se realizó una integración mediante el sistema de eventos de Unity. El interruptor correcto ejecuta el evento OnTurnOn, configurado desde el Inspector para invocar el método TurnPowerOn() del PianoManager. De esta manera, una vez activado el interruptor adecuado, el piano queda habilitado para comenzar la resolución del rompecabezas sin necesidad de modificar la lógica interna del asset.

Ilustración 38

Vista general de la caja de fusibles

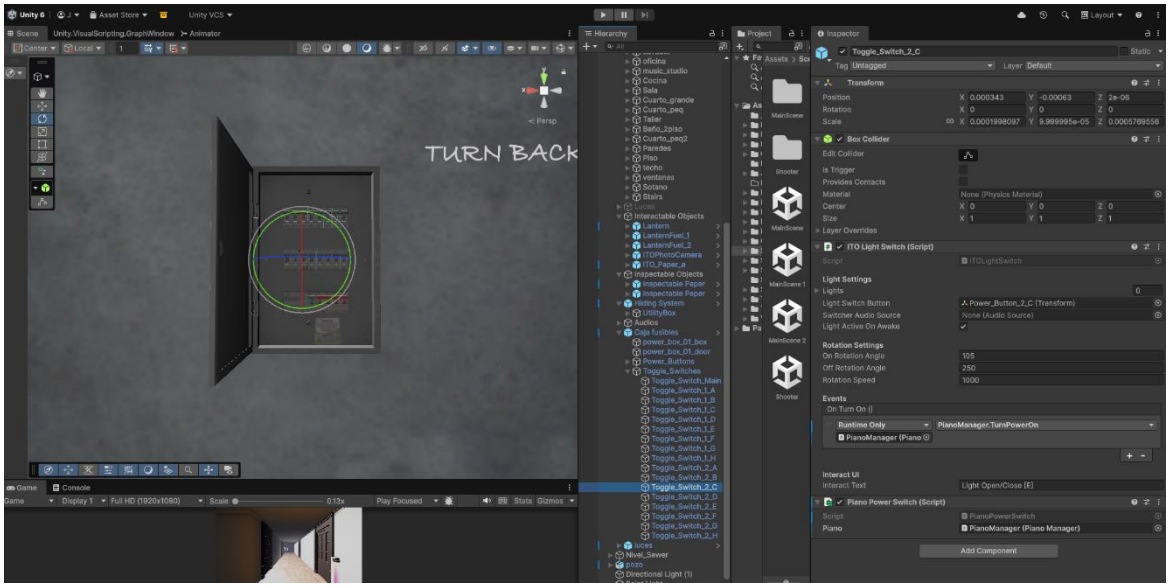
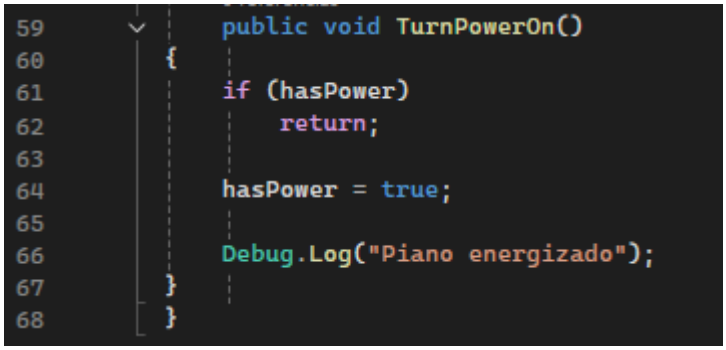


Ilustración 39

Método TurnPowerOn() de la caja de fusibles



```
59      public void TurnPowerOn()
60      {
61          if (hasPower)
62              return;
63
64          hasPower = true;
65
66          Debug.Log("Piano energizado");
67      }
68      }
```

Una vez energizado el instrumento, cada nota ejecutada por el jugador es enviada al método PlayNote() del PianoManager. Este controlador mantiene dos listas: una almacena la secuencia correcta definida para el rompecabezas y la otra registra las últimas notas interpretadas por el jugador. Cada nueva nota se incorpora a la lista de ejecución y, cuando el número de elementos supera el tamaño de la secuencia objetivo, la nota más antigua se elimina automáticamente. Este procedimiento mantiene una ventana deslizante de las últimas notas ejecutadas, permitiendo validar continuamente la melodía sin necesidad de reiniciar el progreso tras una nota incorrecta.

Ilustración 40

Vista general del piano

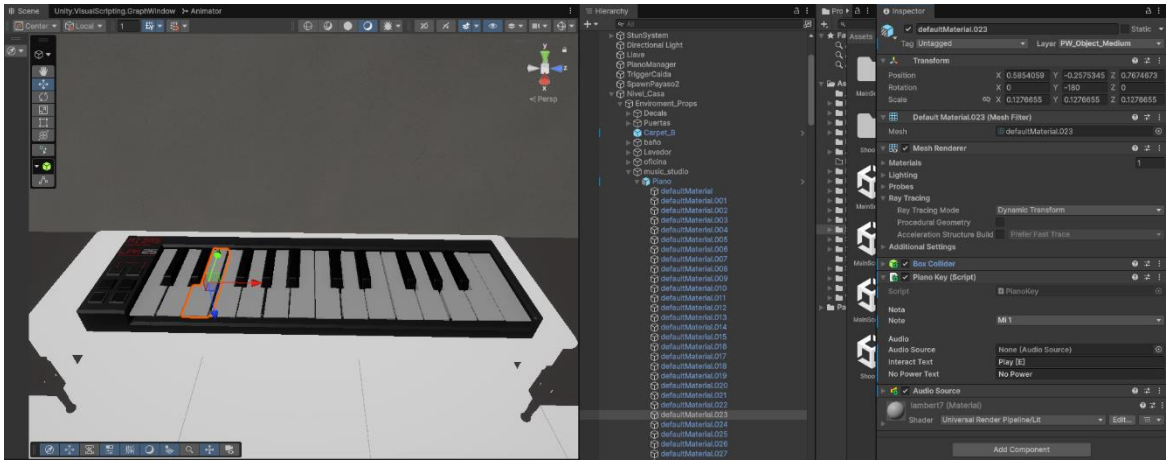
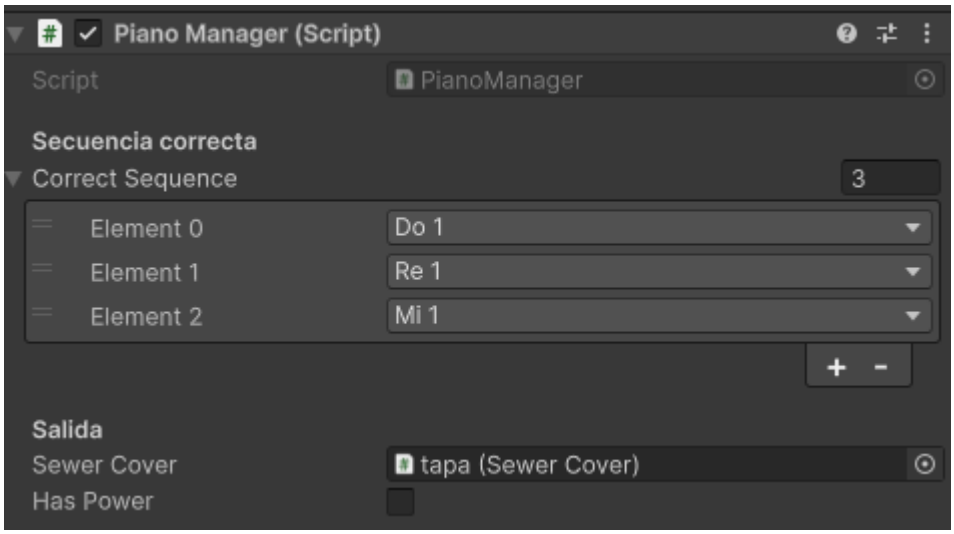


Ilustración 41

Vista en inspector del componente PianoManager

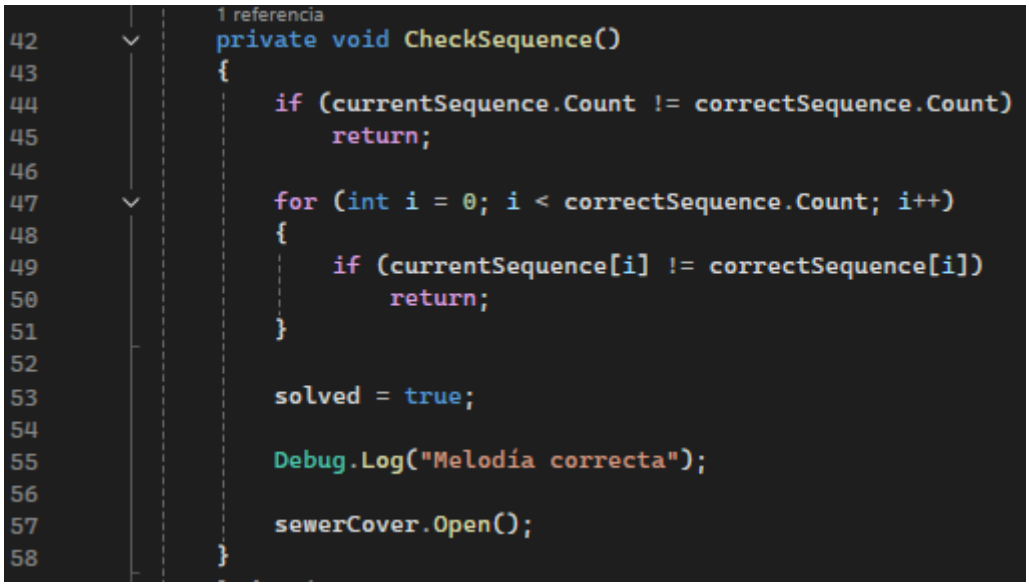


Cuando la secuencia almacenada coincide completamente con la melodía definida, el sistema marca el rompecabezas como resuelto e invoca el método Open() del objeto SewerCover, encargado de abrir la tapa de la alcantarilla que conduce al siguiente sector del nivel. A partir de este momento, el jugador puede acceder al sistema de alcantarillado, donde

continúa la progresión mediante la búsqueda de las válvulas necesarias para desbloquear la salida mientras evita al enemigo principal del escenario.

Ilustración 42

Método CheckSequence() del piano



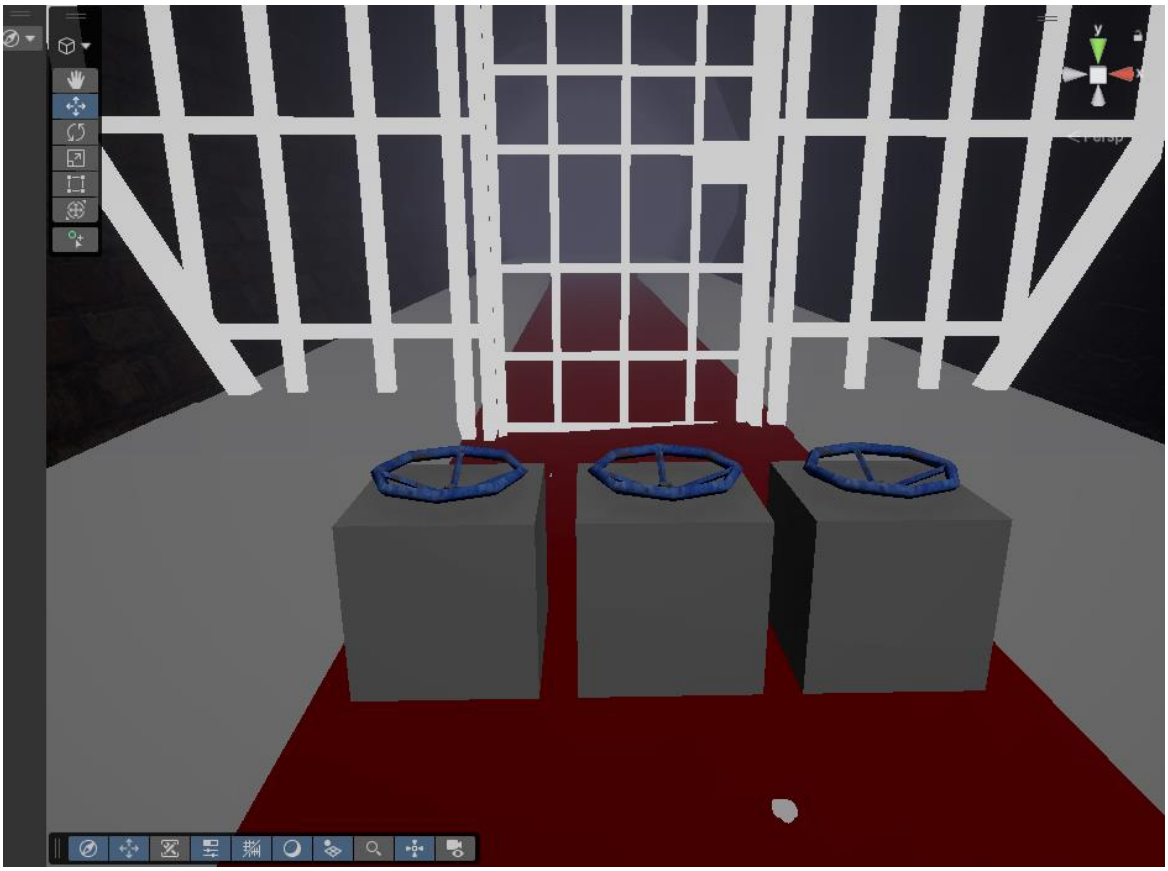
La implementación mantiene separados la interacción de las teclas, el suministro eléctrico, la validación de la secuencia y la activación de la alcantarilla, permitiendo modificar cualquiera de estos componentes de forma independiente. Esta organización facilita la incorporación de nuevas secuencias musicales o la reutilización del sistema en otros escenarios del proyecto sin alterar la arquitectura general del nivel.

4.4.3. Sistema de válvulas

Tras acceder al sistema de alcantarillado, el jugador debe localizar y activar tres válvulas distribuidas a lo largo del escenario para desbloquear la salida del nivel. Esta mecánica constituye el objetivo principal de la última sección del mapa y obliga al jugador a recorrer diferentes áreas mientras evita al enemigo que permanece activo durante toda la exploración.

Ilustración 43

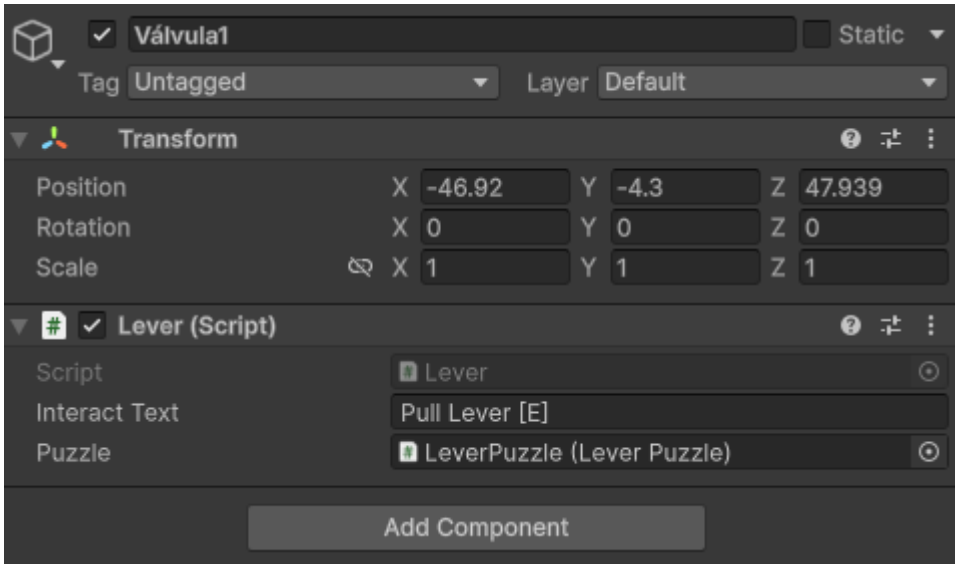
Vista en escena de las válvulas activables



El sistema fue implementado mediante tres p refabs idénticos de válvulas, cada uno equipado con el script Lever, responsable de gestionar la interacción individual con el jugador. La activación utiliza el mismo sistema de interacción empleado en el resto del proyecto mediante la interfaz IInteractable, permitiendo mantener un comportamiento uniforme entre todos los objetos interactivos del nivel.

Ilustración 44

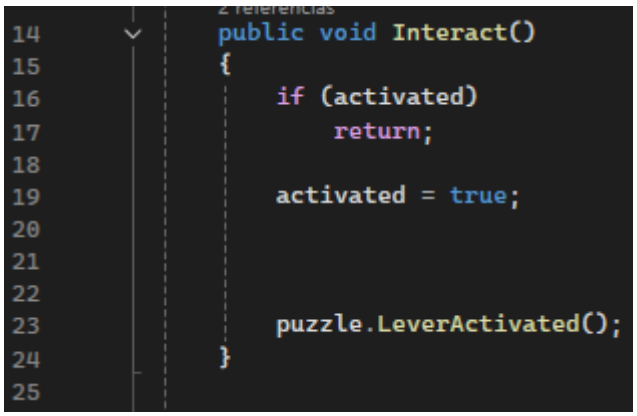
Vista en inspector del componente Lever de la válvula



Cuando el jugador interactúa con una válvula, el script verifica inicialmente si esta ya fue activada para impedir múltiples activaciones sobre el mismo objeto. Una vez validada la interacción, la válvula cambia su estado interno y notifica el evento al controlador central del rompecabezas mediante el método LeverActivated(). De esta manera, cada válvula únicamente es responsable de su propio comportamiento, mientras que la lógica de progresión permanece centralizada.

Ilustración 45

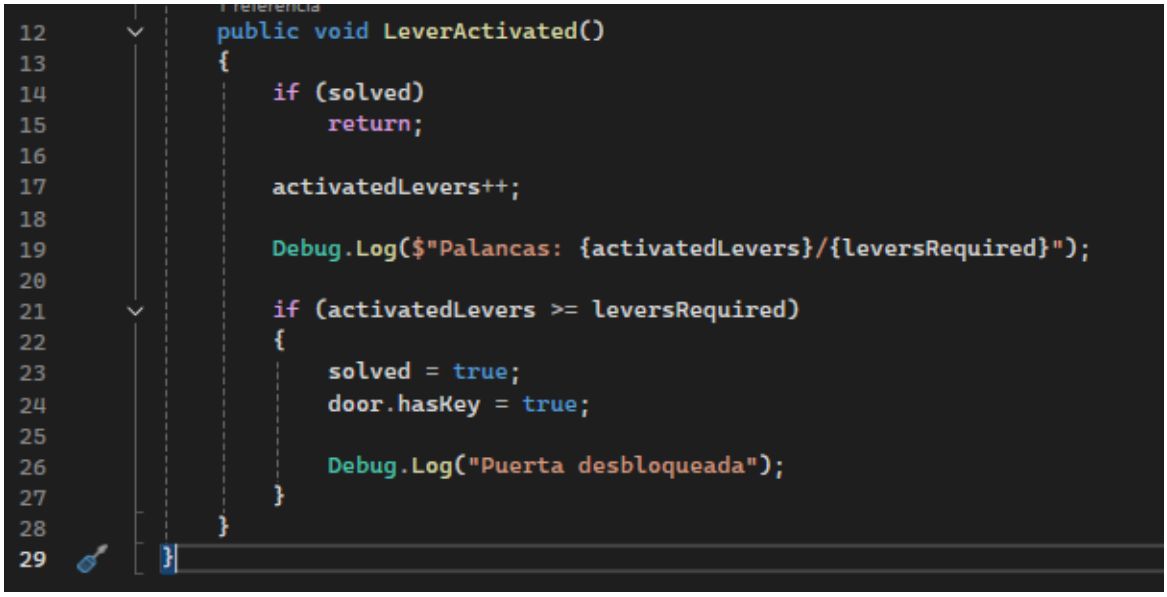
Método Interact() de la válvula



El control general del rompecabezas se encuentra implementado en el script LeverPuzzle, el cual mantiene un contador del número de válvulas activadas y define la cantidad necesaria para completar el objetivo. Cada vez que una válvula comunica su activación, el controlador incrementa el contador interno y verifica si se alcanzó el número requerido para finalizar el rompecabezas.

Ilustración 46

Método LeverActivated de la puerta de escape



```
1 referencia
public void LeverActivated()
{
    if (solved)
        return;

    activatedLevers++;

    Debug.Log($"Palancas: {activatedLevers}/{leversRequired}");

    if (activatedLevers >= leversRequired)
    {
        solved = true;
        door.hasKey = true;

        Debug.Log("Puerta desbloqueada");
    }
}
}
```

Cuando las tres válvulas han sido activadas, el sistema marca el rompecabezas como resuelto y habilita la apertura de la puerta final del escenario. Para ello, el controlador modifica la propiedad hasKey del componente DoorSystem, permitiendo que el jugador pueda abrir la puerta utilizando el mismo sistema de puertas empleado en el resto del nivel. Esta solución permitió reutilizar la lógica existente del asset sin necesidad de desarrollar un nuevo mecanismo de apertura.

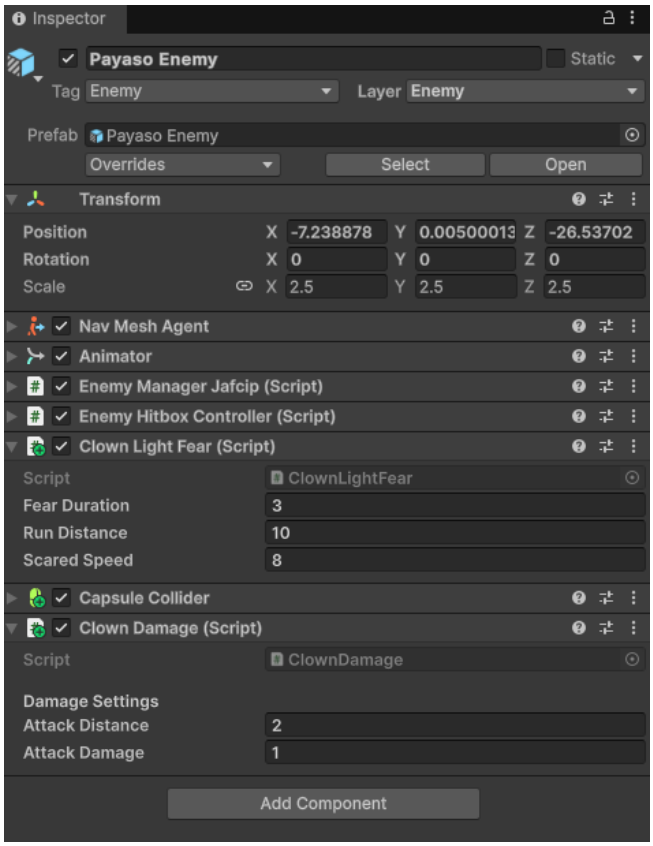
La arquitectura implementada mantiene separadas las responsabilidades entre la interacción individual de cada válvula y el control global del rompecabezas. Esta organización facilita modificar la cantidad de válvulas requeridas, reutilizar el sistema en otros escenarios o sustituir la puerta final por cualquier otro evento de progresión sin alterar el funcionamiento de los componentes individuales.

4.4.4. Sistema de inteligencia artificial del enemigo

El enemigo principal del segundo nivel utiliza el sistema de inteligencia artificial proporcionado por el asset Fps Horror Kit. Este sistema incorpora de forma nativa los comportamientos de patrulla, detección del jugador, persecución, búsqueda y reproducción de animaciones, permitiendo disponer de un enemigo completamente funcional desde las primeras etapas del desarrollo. Sobre esta base se implementaron nuevos componentes para adaptar el comportamiento del enemigo a las mecánicas específicas del proyecto.

Ilustración 47

Vista en inspector del GameObject Payaso Enemy

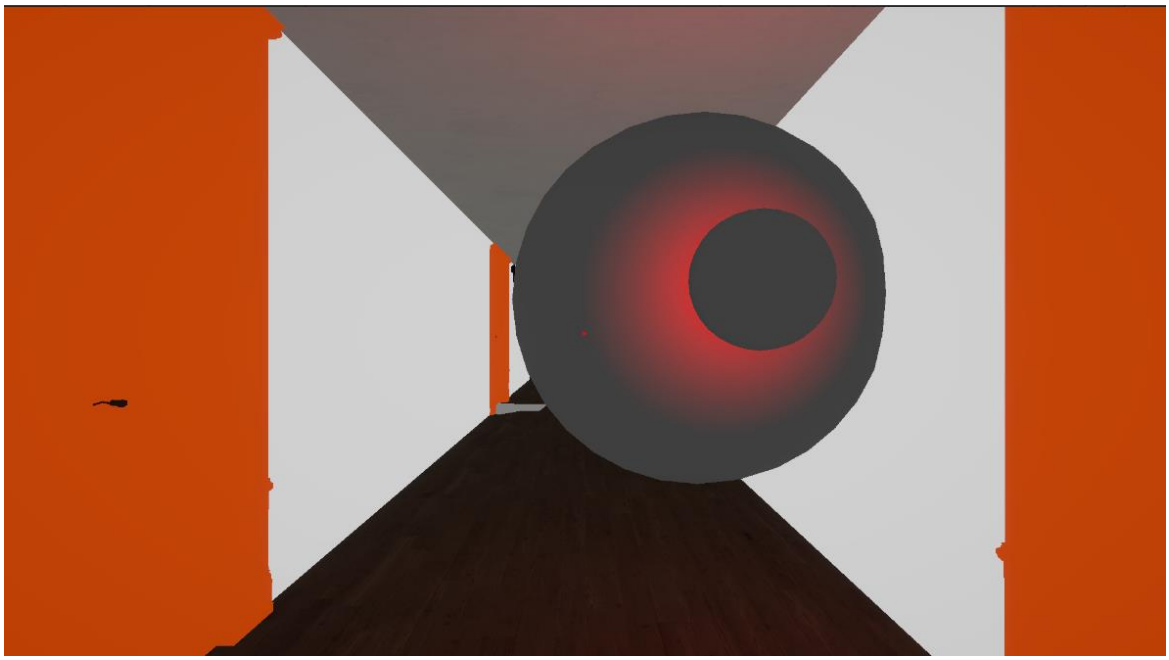


La primera modificación corresponde a la interacción entre el enemigo y el sistema del encendedor. Para ello se desarrolló el script ClownLightFear, encargado de alterar

temporalmente el comportamiento del enemigo cuando este ingresa en el área de iluminación generada por el jugador. Esta comunicación se realiza mediante el objeto FearZone, asociado al encendedor, el cual detecta mediante un Trigger cuándo el enemigo se encuentra dentro del radio de acción y ejecuta el método Fear().

Ilustración 48

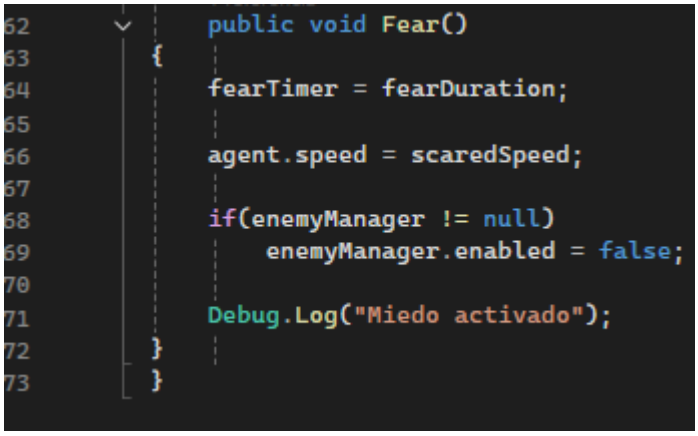
Vista en escena del GameObject FearZone



Al activarse el estado de miedo, el script deshabilita temporalmente el componente EnemyManagerJafcip, responsable de controlar la inteligencia artificial del asset, evitando que continúe ejecutando los comportamientos normales de persecución. Simultáneamente se incrementa la velocidad del NavMeshAgent y se calcula una nueva posición de destino en dirección opuesta al jugador, provocando que el enemigo se aleje durante un tiempo determinado. Una vez transcurrido dicho intervalo, el script restablece la velocidad original y reactiva nuevamente el controlador de inteligencia artificial, permitiendo que el enemigo continúe su comportamiento habitual.

Ilustración 49

Método Fear() del payaso

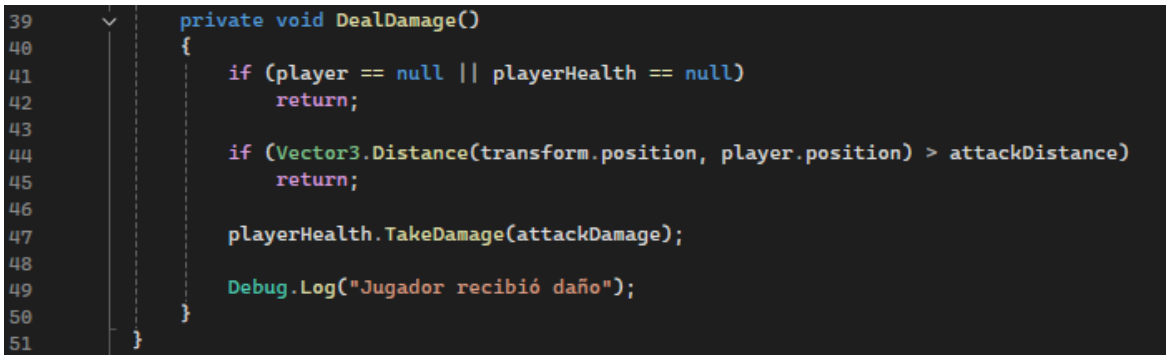


```
62      public void Fear()
63      {
64          fearTimer = fearDuration;
65
66          agent.speed = scaredSpeed;
67
68          if(enemyManager != null)
69              enemyManager.enabled = false;
70
71          Debug.Log("Miedo activado");
72      }
73      }
```

Además del comportamiento de huida, se implementó un sistema independiente para gestionar el daño al jugador. Aunque el asset controla la animación de ataque del enemigo, la aplicación del daño fue desarrollada mediante el script ClownDamage. Este componente se suscribe al evento OnAttackTriggered generado por EnemyManagerJafcip y, cada vez que se ejecuta una animación de ataque, verifica la distancia existente entre el enemigo y el jugador antes de invocar el método TakeDamage() del sistema de salud. Esta validación evita que el daño pueda aplicarse fuera del rango efectivo del ataque y mantiene sincronizada la lógica de combate con las animaciones del enemigo.

Ilustración 50

Método DealDamage() del payaso



```
39      private void DealDamage()
40      {
41          if (player == null || playerHealth == null)
42              return;
43
44          if (Vector3.Distance(transform.position, player.position) > attackDistance)
45              return;
46
47          playerHealth.TakeDamage(attackDamage);
48
49          Debug.Log("Jugador recibió daño");
50      }
51  }
```

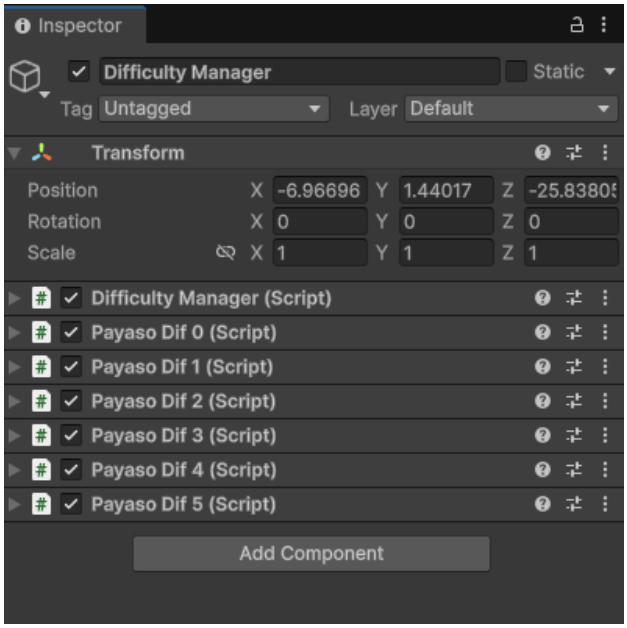
La separación entre la inteligencia artificial proporcionada por el asset y los componentes desarrollados para el proyecto permitió extender el comportamiento del enemigo sin modificar el código original del paquete. Esta arquitectura facilita futuras modificaciones sobre la mecánica del encendedor, el sistema de daño o la configuración del comportamiento del enemigo, manteniendo la compatibilidad con las actualizaciones del asset utilizado durante el desarrollo.

4.4.5. Desarrollo de mecánicas: sistema de dificultad

El segundo nivel reutiliza la misma arquitectura del sistema de dificultad implementada en el primer escenario. La gestión de la progresión continúa siendo responsabilidad del objeto DifficultyManager, el cual mantiene el registro de las muertes del jugador y aplica automáticamente los distintos perfiles de dificultad mediante componentes derivados de DifficultyProfileBase. Esta reutilización permitió mantener una estructura modular, evitando duplicar código y facilitando la incorporación de nuevos niveles de dificultad mediante componentes independientes.

Ilustración 51

Vista en inspector del GameObject DifficultyManger



A diferencia del primer nivel, donde las modificaciones se orientan principalmente al combate, el sistema de dificultad del segundo escenario incrementa progresivamente la presión psicológica y la complejidad de la exploración. Cada muerte introduce nuevas amenazas o modifica el comportamiento de sistemas ya existentes, obligando al jugador a adaptarse continuamente mientras intenta completar los objetivos del nivel.

El sistema de transformación dinámica basada en intentos fallidos se estructura mediante un administrador central denominado `DifficultyManager` y una clase abstracta base denominada `DifficultyProfileBase`. El `DifficultyManager` mantiene el contador de muertes del jugador mediante la variable `deathCount` y administra los diferentes perfiles de transformación registrados en el sistema. Al iniciar el objeto, el administrador obtiene automáticamente los componentes que heredan de `DifficultyProfileBase`, permitiendo incorporar los diferentes perfiles sin establecer manualmente cada uno de ellos en el código.

El contador de muertes se incrementa mediante la suscripción al evento `DemoPlayerHealth.OnPlayerDeath`. Cada vez que se produce la muerte del jugador, el método `OnPlayerDeath()` incrementa en una unidad el valor de `deathCount`. El administrador mantiene su instancia mediante el patrón Singleton y utiliza `DontDestroyOnLoad`, permitiendo conservar el objeto y el contador de muertes durante la recarga de escenas. Cuando una escena es cargada, el método `OnSceneLoaded()` vuelve a establecer la suscripción al evento de muerte del jugador y ejecuta `ApplyDifficulty()`, encargado de evaluar los perfiles disponibles.

Cada perfil hereda de `DifficultyProfileBase` y dispone de un arreglo booleano denominado `activateOnDeaths`, cuyos índices representan el número de muerte en el que puede activarse el perfil. El método `ShouldApply()` consulta el valor correspondiente al contador de muertes y devuelve `true` únicamente cuando la posición asociada se encuentra habilitada. Cuando la condición se cumple, `DifficultyManager` ejecuta el método `Apply()` del perfil correspondiente. De esta forma, la transformación del sistema se determina a partir del número de intentos fallidos registrados y permite aplicar modificaciones específicas sobre diferentes componentes del escenario y de las mecánicas del juego.

La estructura permite separar la lógica general de control de las muertes de las transformaciones específicas de cada perfil. Mientras `DifficultyManager` administra el conteo de muertes y la activación de los perfiles, cada clase derivada de `DifficultyProfileBase` contiene la lógica particular que debe ejecutarse cuando corresponde su activación. Esta organización permite incorporar diferentes transformaciones de manera independiente y mantener una estructura modular para la configuración de los cambios aplicados al juego.

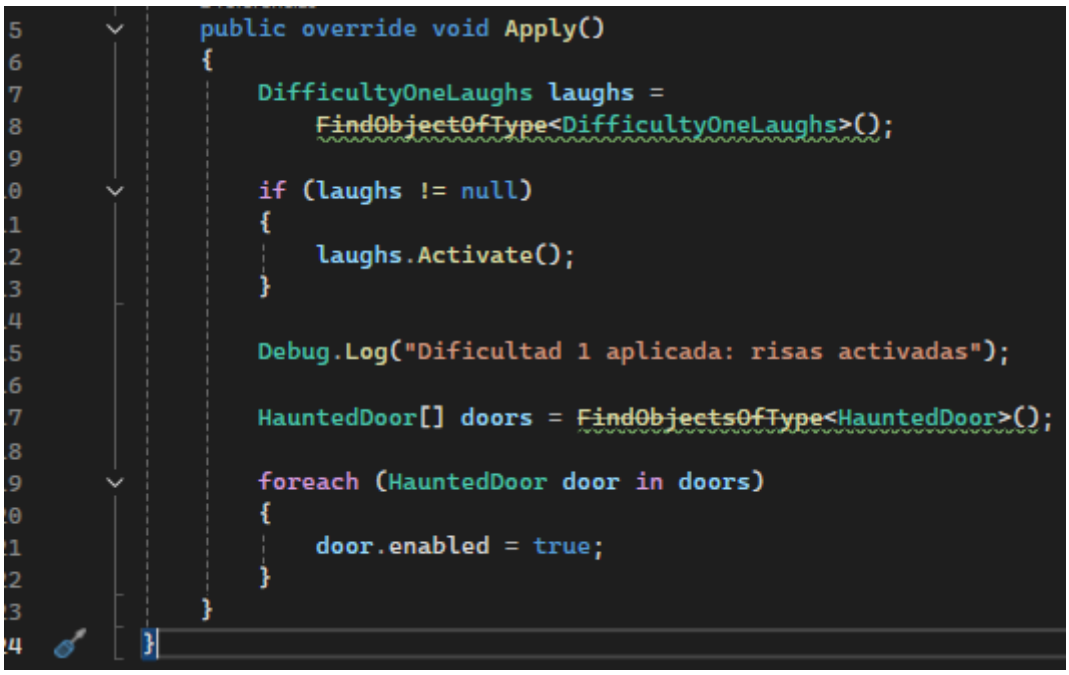
4.4.5.1. Primera muerte: activación de efectos ambientales

Tras la primera muerte del jugador se activa el perfil PayasoDif1, correspondiente al primer nivel de transformación dinámica del escenario. Este perfil modifica las condiciones ambientales mediante la activación del sistema de risas y el comportamiento de las puertas embrujadas. Para ello, el método Apply() localiza el componente DifficultyOneLaughs y ejecuta su método Activate(), habilitando los efectos de audio configurados para el escenario. Adicionalmente, se localizan todos los objetos que contienen el componente HauntedDoor y se habilitan sus componentes para permitir que las puertas comiencen a funcionar durante la exploración.

Esta transformación no modifica directamente los atributos del jugador ni los parámetros de detección del enemigo, sino que altera las condiciones del entorno y la percepción del jugador. La activación de las risas introduce una variación en la ambientación, mientras que las puertas embrujadas modifican dinámicamente el escenario durante la exploración. De esta manera, después del primer intento fallido, el jugador debe adaptarse a un entorno que presenta nuevos eventos y condiciones de navegación respecto al estado base representado por PayasoDif0.

Ilustración 52

Método Apply del perfil PayasoDif1



```
    public override void Apply()
    {
        DifficultyOneLaughs laughs =
            FindObjectOfType<DifficultyOneLaughs>();

        if (laughs != null)
        {
            laughs.Activate();
        }

        Debug.Log("Dificultad 1 aplicada: risas activadas");

        HauntedDoor[] doors = FindObjectsOfType<HauntedDoor>();

        foreach (HauntedDoor door in doors)
        {
            door.enabled = true;
        }
    }
}
```

4.4.5.2. Segunda muerte: activación de corrientes de aire

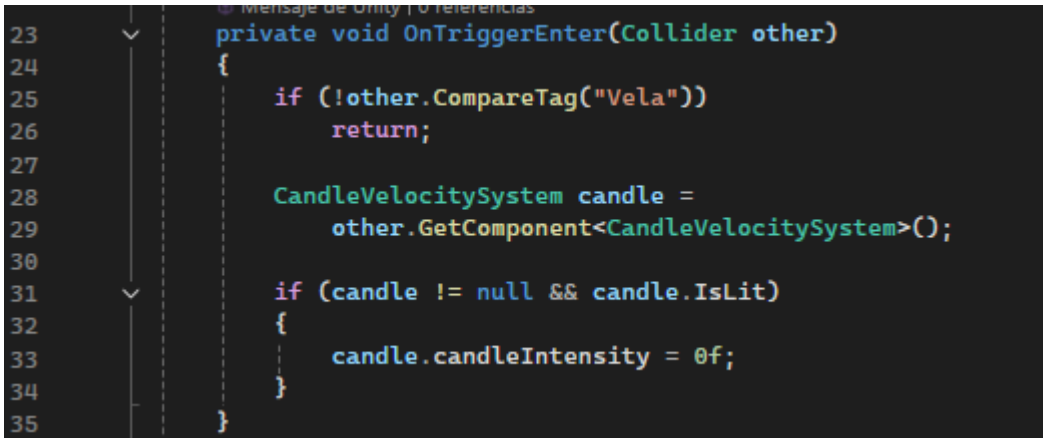
Tras la segunda muerte se activa el perfil PayasoDif2, que incorpora nuevas condiciones de riesgo relacionadas con la gestión de la fuente de luz del jugador. El método Apply() localiza el objeto WindSystem y activa los objetos hijos que forman parte del sistema, habilitando las corrientes de aire distribuidas en diferentes sectores del escenario. Estas corrientes se encuentran asociadas al comportamiento definido mediante el script WindZone, encargado de controlar periódicamente la activación y desactivación de las áreas de efecto mediante intervalos de tiempo variables.

Cuando una corriente de aire se encuentra activa y el jugador ingresa en su área de efecto con el encendedor encendido, el sistema comprueba el estado de la fuente de luz y modifica su intensidad. Adicionalmente, el perfil PayasoDif2 modifica el parámetro intensityLossMultiplier del componente CandleVelocitySystem, estableciéndolo en un valor

de 1.25, lo que incrementa la velocidad con la que disminuye la intensidad de la fuente de luz. En conjunto, estas modificaciones aumentan la presión sobre el jugador durante la exploración, ya que la pérdida de iluminación puede obligarlo a ejecutar nuevamente la mecánica de encendido mediante el evento de interacción correspondiente.

Ilustración 53

Método OntriggerEnter del sistema WindZone



4.4.5.3. Tercera muerte: incremento de la capacidad de detección

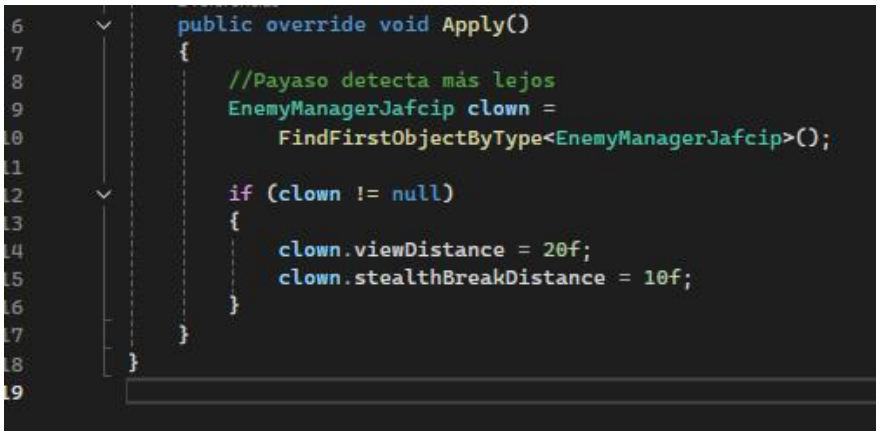
Tras la tercera muerte se activa el perfil PayasoDif3, el cual modifica directamente determinados parámetros del comportamiento del enemigo principal. Para realizar esta transformación, el método Apply() localiza el componente EnemyManagerJafcip asociado al payaso y modifica sus parámetros de percepción. Específicamente, el valor de viewDistance se establece en 20, mientras que stealthBreakDistance se establece en 10.

Estos cambios incrementan las condiciones de riesgo durante la exploración, debido a que el enemigo puede detectar al jugador desde una mayor distancia y abandonar las condiciones de sigilo ante una proximidad determinada. La transformación modifica, por tanto, el comportamiento de la inteligencia artificial sin alterar la estructura general de sus sistemas de persecución y ataque. Como consecuencia, el jugador dispone de menos

oportunidades para desplazarse de manera segura por el escenario y debe gestionar con mayor precisión sus movimientos y su exposición ante el enemigo.

Ilustración 54

Método Apply del perfil PayasoDif3



4.4.5.4. Cuarta muerte: activación de trampas de ralentización

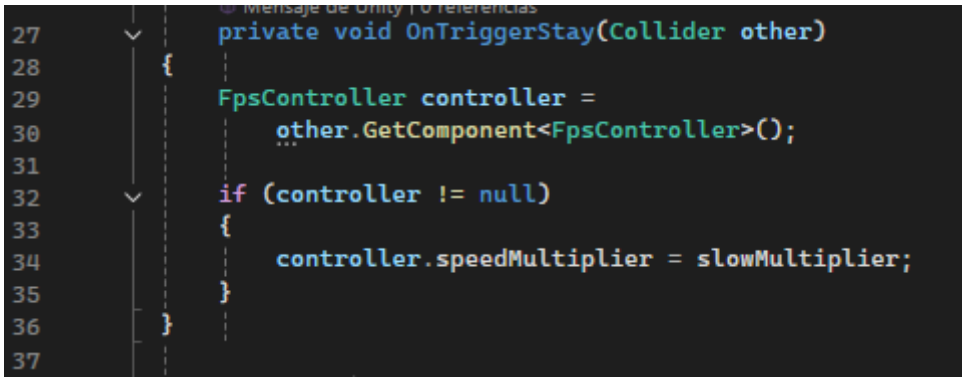
Tras la cuarta muerte se activa el perfil PayasoDif4, encargado de incorporar trampas de ralentización al escenario. El método Apply() localiza el objeto SlowSystem y activa los objetos hijos que forman parte de este sistema, habilitando las trampas distribuidas en diferentes zonas del nivel.

El comportamiento individual de estas trampas se gestiona mediante el script SlowTrap, que controla periódicamente la disponibilidad de cada zona de efecto mediante intervalos de activación y desactivación. Cuando una trampa se encuentra activa y el jugador permanece dentro de su área, el sistema obtiene el componente FpsController y modifica temporalmente su propiedad speedMultiplier, estableciendo el valor configurado para la ralentización. Al abandonar el área de la trampa, el multiplicador de velocidad se restablece a su valor normal. Esta transformación introduce una nueva condición de riesgo, debido a

que la reducción temporal de la velocidad de desplazamiento limita la capacidad del jugador para escapar o reposicionarse frente al enemigo.

Ilustración 55

Método OntriggerEnter() del sistema SlowTrap



4.4.5.5. Quinta muerte: activación de trampas de aturdimiento

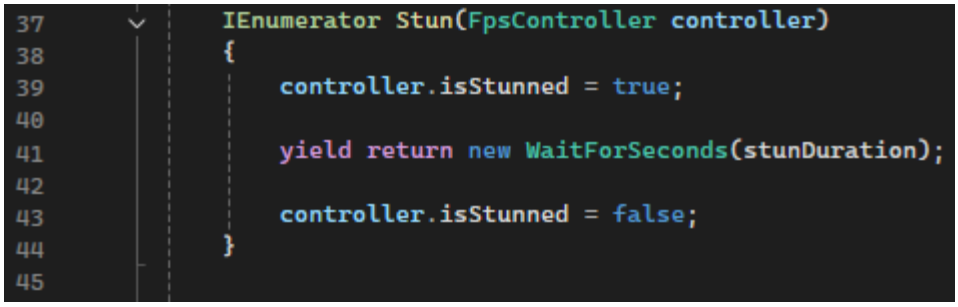
Tras la quinta muerte se activa el perfil PayasoDif5, correspondiente al último nivel de transformación dinámica implementado para el escenario. Este perfil localiza el objeto StunSystem y activa sus objetos hijos, habilitando las trampas de aturdimiento distribuidas en el nivel.

El comportamiento de estas trampas se implementa mediante el script StunTrap. Al entrar en contacto con una trampa activa, el sistema obtiene el componente FpsController del jugador y ejecuta una rutina que establece temporalmente la propiedad isStunned en true. Durante el tiempo definido por stunDuration, el jugador permanece en estado de aturdimiento y, una vez transcurrido este periodo, la propiedad se establece nuevamente en false. A diferencia de las trampas de ralentización, esta transformación limita temporalmente la capacidad de respuesta y movimiento del jugador, incrementando el riesgo de ser alcanzado por el enemigo durante una persecución.

Con la activación de PayasoDif5 se alcanza el último nivel de transformación dinámica definido para el escenario. Una vez que el jugador alcanza nuevamente esta condición de fallo y se completa el ciclo establecido de cinco transformaciones progresivas, el sistema muestra una pantalla de Game Over y requiere que el jugador reinicie la partida desde el estado inicial. De esta manera, la secuencia completa de intentos fallidos conduce progresivamente desde el estado base de PayasoDif0 hasta la condición de mayor dificultad representada por PayasoDif5, antes de finalizar el ciclo de juego.

Ilustración 56

Corutina Stun() de la trampa de aturdimiento



La organización mediante perfiles independientes permitió implementar cada modificación como un componente aislado, facilitando el mantenimiento del sistema y permitiendo incorporar nuevas condiciones de dificultad sin alterar la lógica principal del administrador. Esta arquitectura también favorece la reutilización del sistema en otros niveles del proyecto, modificando únicamente los perfiles asociados a cada escenario.

Tabla 9

Perfiles de dificultad del segundo nivel

Perfil de dificultad	Condición de activación	Modificaciones Implementadas
PayasoDif0	Estado base	No aplica ninguna transformación
PayasoDif1	Primera muerte	Activa las risas de fondo mediante DifficultyOneLaughs y habilita los componentes HauntedDoor presentes en la escena
PayasoDif2	Segunda muerte	Activa los objetos contenidos en WindSystem y establece intensityLossMultiplier del encendedor en 1.25
PayasoDif3	Tercera muerte	Establece viewDistance del payaso en 20 y stealthBreakDistance en 10
PayasoDif4	Cuarta muerte	Activa los objetos contenidos en SlowSystem
PayasoDif5	Quinta muerte	Activa los objetos contenidos en StunSystem

El sistema de transformación dinámica basada en intentos fallidos utiliza como variable de entrada la cantidad acumulada de muertes del jugador durante la progresión de la partida. Cada muerte incrementa el contador correspondiente y permite determinar el perfil de transformación que debe aplicarse. Los perfiles se ejecutan de manera progresiva, de modo que cada nuevo intento fallido habilita una transformación adicional sobre las condiciones previamente establecidas del escenario. De esta manera, las transformaciones no se basan en el incremento directo de valores estadísticos generales, sino en la activación o modificación de sistemas específicos del entorno, la inteligencia artificial y las mecánicas de interacción.

4.4.6. Arquitectura de clases y componentes del nivel de sigilo

Con el propósito de representar de manera simplificada el flujo de funcionamiento del sistema de dificultad dinámica, se presenta el pseudocódigo correspondiente al proceso

de evaluación de las muertes acumuladas y aplicación de los perfiles de dificultad. Este procedimiento permite visualizar la secuencia lógica mediante la cual el sistema determina las transformaciones que deben aplicarse en cada intento del jugador.

```
INICIO
  Inicializar DifficultyManager
  SI existe una instancia previa de DifficultyManager ENTONCES
    Destruir la instancia duplicada
  SI NO
    Establecer DifficultyManager como instancia única
    Mantener la instancia entre escenas
    Obtener perfiles de dificultad disponibles
  FIN SI
  Registrar eventos de carga de escena
  AL cargar una escena:
    Registrar evento de muerte del jugador
    Evaluar perfiles de dificultad disponibles
    PARA CADA perfil HACER
      SI el perfil está configurado para el número actual de muertes ENTONCES
        Ejecutar Apply() del perfil
      FIN SI
    FIN PARA
  FIN AL
  AL producirse la muerte del jugador:
    Incrementar deathCount
```

FIN AL

SEGÚN el número de muertes acumuladas:

CASO 0:

Aplicar PayasoDif0

Mantener la configuración base del escenario

CASO 1:

Aplicar PayasoDif1

Activar el sistema de risas

Habilitar las puertas embrujadas

CASO 2:

Aplicar PayasoDif2

Activar las corrientes de aire

Incrementar la velocidad de pérdida de intensidad del encendedor

CASO 3:

Aplicar PayasoDif3

Aumentar la distancia de visión del enemigo

Aumentar la distancia de detección de movimientos sospechosos

CASO 4:

Aplicar PayasoDif4

Activar las trampas de ralentización

CASO 5:

Aplicar PayasoDif5

Activar las trampas de aturdimiento

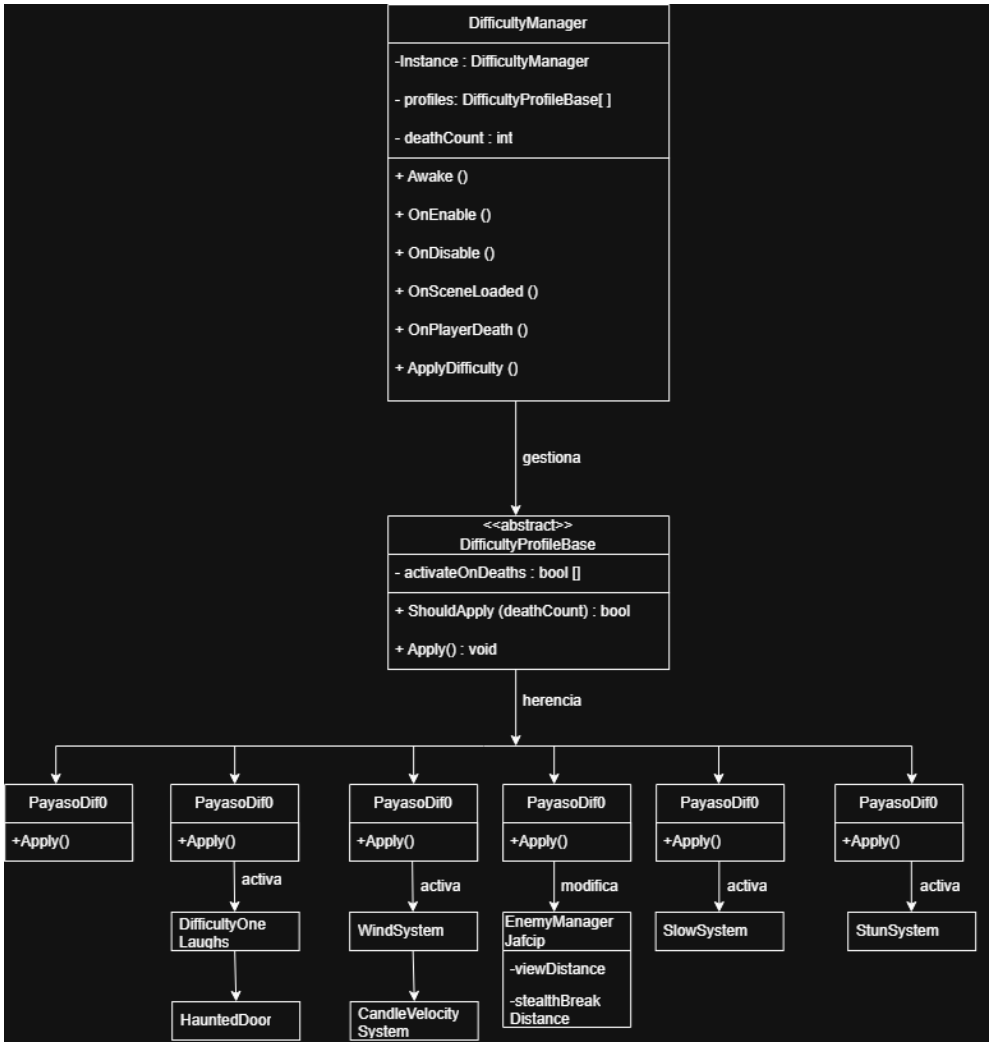
CASO MAYOR A 5:

```
            Mostrar pantalla de Game Over
            Finalizar el intento actual
            Reiniciar la progresión desde la configuración base
        FIN SEGÚN
    FIN
```

El pseudocódigo representa el flujo general de aplicación de las transformaciones dinámicas implementadas en el segundo nivel de Inkfall. El sistema utiliza el número de muertes acumuladas como condición para seleccionar y aplicar los diferentes perfiles de dificultad. Cada perfil modifica de manera independiente las condiciones del escenario, incorporando efectos ambientales, cambios en el comportamiento del enemigo y nuevas trampas que incrementan progresivamente el desafío. Una vez alcanzado el límite establecido para la progresión, el sistema finaliza el intento mediante la pantalla de Game Over y permite reiniciar el nivel desde su configuración base.

Ilustración 57

Diagrama de clase del sistema de dificultad del segundo nivel



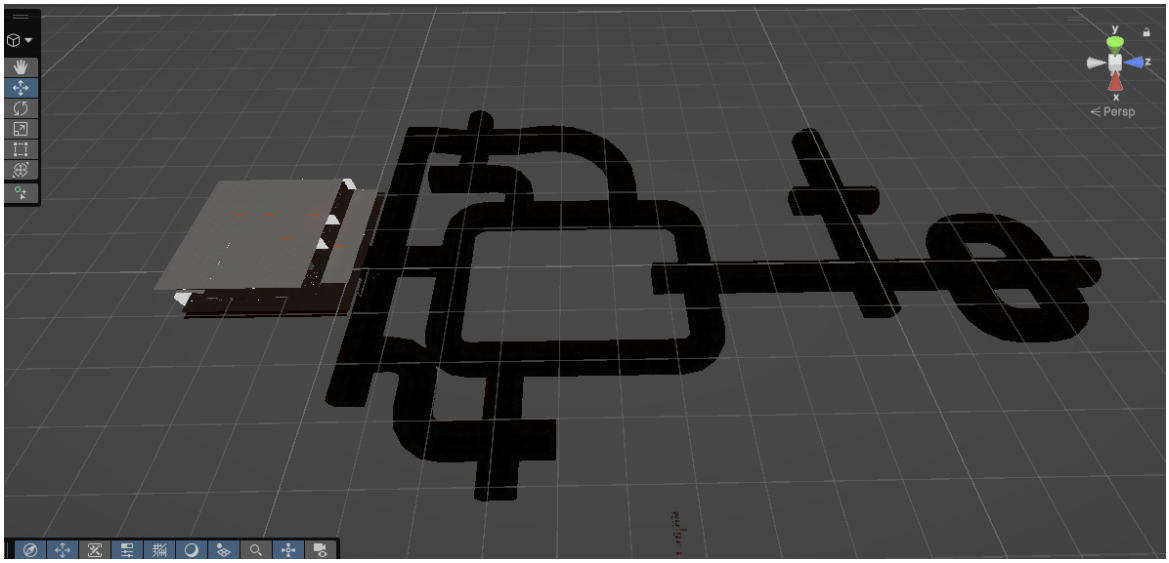
4.4.7. Diseño de nivel

Una vez implementados los sistemas que conforman el segundo nivel, se procedió a su integración dentro del escenario mediante el proceso de diseño de nivel. Esta etapa consistió en organizar espacialmente todos los elementos interactivos del mapa, establecer la progresión del jugador y distribuir las mecánicas de manera que la dificultad aumentara conforme avanzara la partida.

El escenario fue construido utilizando geometría en caja blanca, permitiendo validar el recorrido del jugador sin depender de modelos o recursos artísticos definitivos. La distribución de habitaciones, pasillos y zonas de exploración fue diseñada para guiar al jugador a través de una secuencia de objetivos, manteniendo siempre varias rutas de desplazamiento y espacios suficientes para la interacción con el enemigo principal.

Ilustración 58

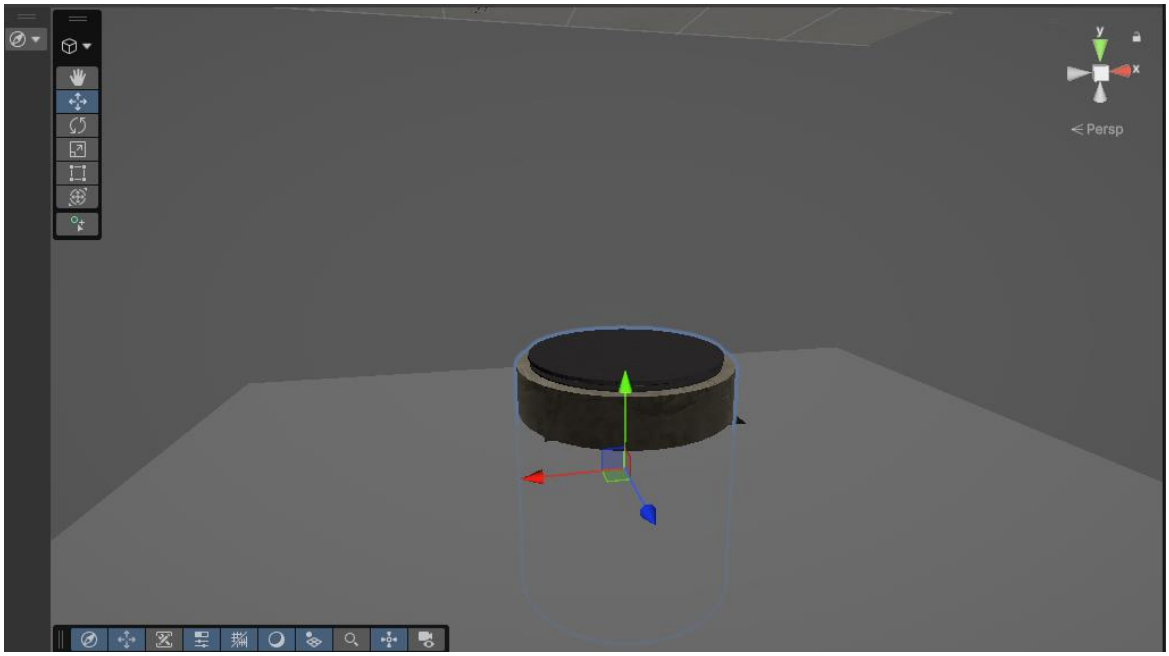
Vista aérea del diseño del segundo nivel



La progresión del nivel comienza en la planta superior del edificio, donde el jugador debe localizar una llave necesaria para acceder al sótano. Una vez obtenida, se habilita el acceso a la caja de fusibles, cuya activación energiza el piano y permite resolver el rompecabezas musical. Al completar correctamente la secuencia de notas, se abre la alcantarilla que conduce al sistema de alcantarillado, donde el jugador debe localizar las tres válvulas necesarias para desbloquear la puerta de salida del nivel.

Ilustración 59

Vista en escena del pozo de escape del segundo nivel



Durante el diseño del escenario se distribuyeron estratégicamente los distintos elementos interactivos, incluyendo llaves, puertas, caja de fusibles, piano, alcantarilla y válvulas. Esta distribución obliga al jugador a recorrer prácticamente la totalidad del mapa antes de completar los objetivos principales, incrementando el tiempo de exposición frente al enemigo y favoreciendo el funcionamiento del sistema de dificultad progresiva.

El enemigo principal también fue integrado durante esta etapa mediante la definición de su posición inicial. Asimismo, se ubicaron las distintas zonas de corrientes de aire y trampas que son habilitadas progresivamente por el sistema de dificultad, asegurando que cada modificación afecte sectores diferentes del escenario y genere nuevas condiciones de exploración conforme aumenta el número de muertes del jugador.

Ilustración 60

Vista en escena de ubicación de trampas del segundo nivel

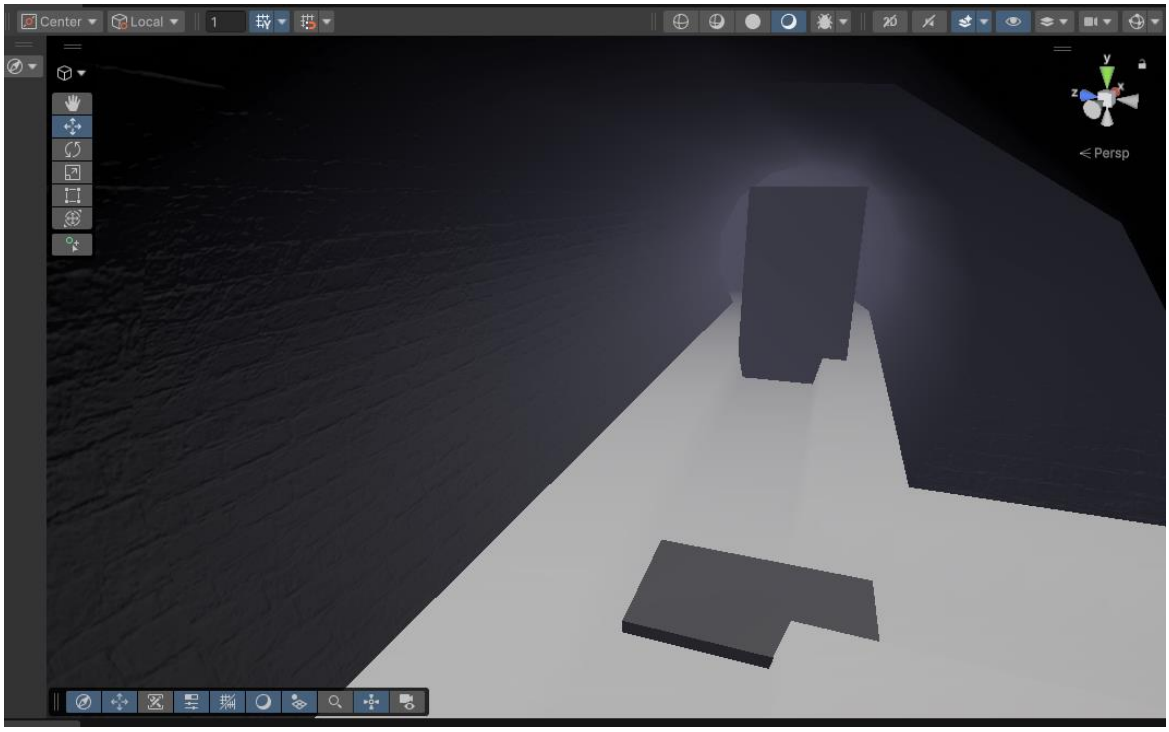
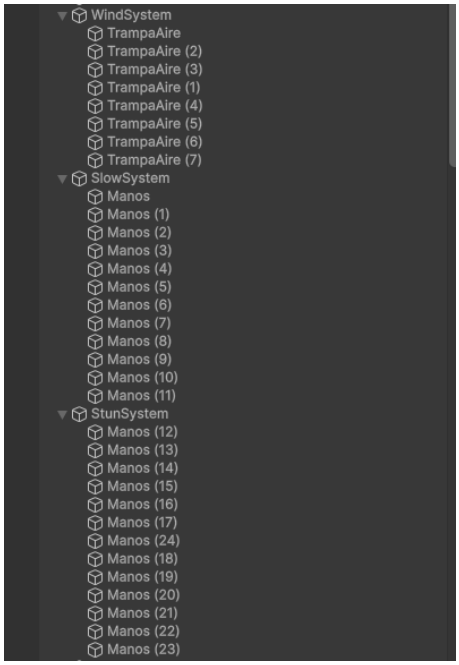


Ilustración 61

Vista en jerarquía de trampas del segundo nivel



Con el fin de facilitar futuras modificaciones del proyecto, los elementos principales del escenario fueron organizados como prefabs independientes y agrupados de acuerdo con su funcionalidad. Esta organización permite modificar fácilmente la distribución del nivel, reemplazar objetos temporales por recursos artísticos definitivos o incorporar nuevas mecánicas sin afectar el funcionamiento del resto de sistemas implementados.

4.5. Delimitación de la contribución técnica

El desarrollo de la base técnica de Inkfall se realizó mediante la integración de funcionalidades propias y recursos proporcionados por herramientas y assets de terceros. La utilización de estos recursos permitió disponer de sistemas base previamente desarrollados, sobre los cuales se implementaron, modificaron e integraron las mecánicas específicas requeridas por el prototipo. Por este motivo, es necesario diferenciar las funcionalidades

proporcionadas originalmente por los recursos externos de aquellas desarrolladas y configuradas específicamente para el presente trabajo.

Los recursos de terceros se utilizaron principalmente como base para funcionalidades generales del videojuego, como el controlador del jugador, sistemas de interacción, navegación e inteligencia artificial, movimiento y manejo de armas. La contribución propia se concentró en el diseño e implementación de las mecánicas específicas de Inkfall, la integración de los diferentes sistemas, la programación de eventos y condiciones particulares, los sistemas de rompecabezas, la progresión basada en intentos fallidos y la configuración de los escenarios funcionales en caja blanca. Asimismo, se desarrollaron scripts propios para extender o complementar las funcionalidades proporcionadas por los recursos utilizados, adaptándolas a los requerimientos del prototipo.

La Tabla 9 presenta la delimitación de la contribución técnica realizada en cada sistema implementado, diferenciando la funcionalidad proporcionada por los recursos de terceros, las modificaciones y configuraciones realizadas, los componentes desarrollados específicamente para el proyecto y las evidencias asociadas a su implementación.

Tabla 10

Delimitación técnica

Sistema	**Base utilizada**	**Funcionalidad original**	**Configuración propia**
Control del jugador	Low Poly Shooter Pack	Sistema base de control y movimiento del jugador	Detección de suelo, implementación de salto, configuración de cámara

Sistema	Base utilizada	Funcionalidad original	Configuración propia
Interacción	No aplica / FPS Horror Kit	Interacción con puertas y cajones	Recogida de armas, interacción con piano, caja de fusibles, llave, válvulas y encendedor
Combate	Low Poly Shooter Pack	Disparo y daño a enemigos	Habilidades especiales de cada arma, daño al jugador
Inteligencia artificial	Breadcrumb AI / FPS Horror Kit	Búsqueda y seguimiento del jugador	Habilidades especiales de enemigos, sistema de asustar al payaso con la luz
Encendedor	No aplica	No aplica	Encendido y apagado del encendedor, emisión de luz
Rompecabezas	No aplica	No aplica	Sistema de piano, caja de fusibles, alcantarilla, llave
Minijuego encendedor	Quick-Time-Event-like-in-DBD	Funcionamiento del sistema	Activación del sistema al apagarse el encendedor
Transformación dinámica	No aplica	No aplica	Implementación de todos los niveles de dificultad y scripts relacionados

Sistema	Base utilizada	Funcionalidad original	Configuración propia
Escenarios en caja blanca	Blender / Unity	Herramientas de modelado y construcción	Construcción de escenarios en caja blanca

La diferenciación presentada permite establecer el alcance de la contribución técnica realizada en el desarrollo de Inkfall. Los recursos y assets de terceros fueron utilizados como bases funcionales para acelerar la implementación de componentes generales del videojuego, mientras que las modificaciones, configuraciones e integraciones realizadas permitieron adaptar dichos componentes a los requerimientos específicos del prototipo. De manera complementaria, las mecánicas y sistemas particulares del juego fueron implementados mediante lógica propia, especialmente aquellos relacionados con los rompecabezas, la interacción entre sistemas y la transformación dinámica basada en intentos fallidos. Esta separación permite identificar con mayor claridad las funcionalidades preexistentes y las contribuciones desarrolladas específicamente para el presente trabajo.

4.6. Escalabilidad y mantenimiento

La base técnica desarrollada para Inkfall fue diseñada con el propósito de facilitar su mantenimiento y permitir la incorporación de nuevas funcionalidades durante futuras etapas del desarrollo. La organización del proyecto mediante componentes independientes permite que cada sistema pueda modificarse de manera aislada, reduciendo el impacto de los cambios sobre el resto de la aplicación. Esta característica facilita la corrección de errores, la optimización de mecánicas existentes y la actualización de funcionalidades sin afectar el comportamiento general del videojuego.

La arquitectura implementada también favorece la escalabilidad del proyecto. Nuevos niveles, enemigos, mecánicas, objetos interactivos o perfiles de transformación dinámica pueden incorporarse reutilizando la estructura existente y desarrollando únicamente los componentes específicos para cada nueva funcionalidad. De esta manera, la lógica principal del sistema permanece estable mientras se amplían las capacidades del videojuego.

Asimismo, la utilización de prefabs y clases base permite reutilizar configuraciones y comportamientos previamente implementados, disminuyendo la duplicación de código y simplificando la incorporación de nuevos elementos. En el caso del sistema de transformación dinámica basada en intentos fallidos, es posible añadir nuevos perfiles de dificultad implementando clases derivadas e incorporándolas al gestor correspondiente, sin modificar el funcionamiento del sistema encargado de administrar la progresión.

Finalmente, la arquitectura desarrollada proporciona una base preparada para integrar posteriormente recursos gráficos, animaciones, sonido, interfaces definitivas y contenido adicional del videojuego. Gracias a esta separación entre la lógica de programación y los elementos de presentación, el proyecto puede evolucionar de manera progresiva, facilitando tanto las labores de mantenimiento como la expansión del videojuego en futuras versiones.

4.7. Procedimiento para ampliar el sistema

Inkfall permite incorporar nuevos niveles de dificultad mediante la creación de perfiles independientes. Cada perfil contiene la lógica necesaria para aplicar modificaciones específicas al juego y es gestionado automáticamente por el componente DifficultyManager.

- Paso 1: Crear un nuevo perfil de dificultad

Para agregar un nuevo nivel de dificultad se debe crear un nuevo script que herede de la clase abstracta DifficultyProfileBase. Al heredar de DifficultyProfileBase, el nuevo perfil

tendrá acceso al sistema de activación mediante muertes y deberá implementar el método obligatorio Apply().

- Paso 2: Implementar la lógica del nuevo nivel

Cada perfil debe definir qué cambios realizará cuando sea activado, para esto se debe sobrescribir el método Apply(). Dentro de este método se puede añadir cualquier modificación relacionada con la dificultad, como modificar parámetros de enemigo, cambiar velocidad de movimiento, activar nuevos sistemas. El sistema no limita el tipo de modificación, ya que cada perfil posee independencia en su implementación.

- Paso 3: Añadir el nuevo perfil en el inspector

Seleccionar el GameObject que contiene DifficultyManager, presionar Add Component, buscar el nuevo script creado y añadirlo al objeto. DifficultyManager obtiene automáticamente todos los perfiles que estén agregados como componentes en el mismo GameObject

- Paso 4: Configurar cuando se activa el nivel

Cada perfil posee la variable activateOnDeaths. En el inspector, seleccionar después de que muerte se quiere activar el nuevo nivel de dificultad. En caso de ser necesario, se pueden agregar más muertes añadiendo nuevos elementos a este mismo componente

4.8. Conclusiones

El presente trabajo permitió desarrollar la base técnica del videojuego Inkfall mediante la programación de sistemas y mecánicas implementados en un entorno de caja blanca utilizando el motor Unity. La arquitectura desarrollada permitió integrar de manera organizada los diferentes sistemas que conforman el funcionamiento principal del prototipo, estableciendo una estructura modular que facilita la comunicación entre componentes y

proporciona una base sólida para la incorporación futura de recursos artísticos, sonoros y narrativos sin modificar la lógica central del videojuego.

Como parte del proceso de diseño, se elaboraron diagramas técnicos que describen la organización de los módulos, las relaciones entre los diferentes componentes y los flujos de interacción de las principales mecánicas implementadas. Esta documentación permitió definir previamente la estructura del sistema, facilitar la implementación de los distintos módulos y mantener una organización coherente durante el desarrollo del proyecto, contribuyendo además a mejorar la comprensión y el mantenimiento de la arquitectura propuesta.

La lógica de progresión basada en intentos fallidos fue implementada mediante un sistema modular de perfiles de dificultad, permitiendo controlar las condiciones de activación y transición entre los diferentes estados de dificultad de cada escenario. Esta arquitectura desacopló la gestión de la progresión de las modificaciones específicas aplicadas en cada nivel, facilitando la incorporación de nuevas transformaciones sin alterar el funcionamiento general del sistema.

Asimismo, se implementaron las mecánicas fundamentales del videojuego en un entorno de caja blanca, incluyendo el movimiento del jugador, los sistemas de interacción, el combate, el comportamiento de los enemigos y los diferentes sistemas de apoyo que intervienen durante la experiencia de juego. El desarrollo independiente de estos componentes permitió verificar su funcionamiento de manera aislada y posteriormente integrarlos dentro de una arquitectura común, favoreciendo la escalabilidad y el mantenimiento del proyecto.

La implementación de dos escenarios con dinámicas de juego diferentes permitió comprobar la adaptabilidad de la arquitectura desarrollada. El primer nivel integró mecánicas

de combate en un escenario tipo shooter, mientras que el segundo incorporó mecánicas de evasión, administración de recursos y transformación dinámica del entorno. A pesar de presentar objetivos y mecánicas distintas, ambos escenarios compartieron la misma base técnica, evidenciando la reutilización de componentes y la capacidad de la arquitectura para soportar diferentes tipos de experiencias de juego.

Finalmente, las pruebas funcionales y la evaluación realizada con los participantes permitieron verificar el funcionamiento de las mecánicas implementadas e identificar oportunidades de mejora dentro del prototipo. Los resultados del cuestionario evidenciaron una percepción predominantemente favorable en las funcionalidades evaluadas, destacando la mecánica de la vela con un 90 % de respuestas favorables y la inteligencia artificial de los enemigos con un 85 %. El sistema de combate y cambio de armas alcanzó un 60 % de respuestas favorables, mientras que el movimiento y control del personaje obtuvo un 55 %. Por otra parte, el sistema de cámara presentó el menor nivel de valoración favorable, con un 30 %, además de un 20 % de respuestas desfavorables, por lo que se identificó como uno de los principales aspectos susceptibles de mejora.

Estos resultados permitieron orientar el proceso de refinamiento hacia los sistemas que presentaron mayores oportunidades de mejora, particularmente el movimiento, el cambio de armas y el comportamiento de la cámara. Posteriormente, se realizaron nuevas pruebas funcionales para comprobar las correcciones implementadas y verificar la integración de los sistemas dentro del prototipo. En conjunto, la evidencia obtenida mediante las pruebas técnicas y la evaluación cuantitativa con los participantes permitió validar el funcionamiento de la base técnica desarrollada para Inkfall y sustentar el cumplimiento del objetivo específico relacionado con la evaluación del prototipo. De esta manera, se estableció una

base técnica modular y funcional sobre la cual puede continuar el desarrollo completo del videojuego.

4.9. Recomendaciones

Se recomienda mantener la arquitectura modular desarrollada durante las siguientes etapas del proyecto, incorporando nuevos sistemas y mecánicas mediante componentes desacoplados que preserven la independencia entre los diferentes módulos del videojuego. Esta estrategia facilitará el mantenimiento del código, reducirá el impacto de futuras modificaciones y permitirá ampliar las funcionalidades sin comprometer la estabilidad del sistema.

Se recomienda continuar la integración progresiva de recursos gráficos, animaciones, efectos visuales, sonido y narrativa sobre la base técnica desarrollada, procurando conservar la separación entre la lógica de programación y los elementos de presentación. De esta manera, será posible realizar ajustes tanto en el apartado técnico como artístico sin generar dependencias innecesarias entre ambos procesos.

Como trabajo futuro, se recomienda ampliar el sistema de transformación dinámica mediante nuevos perfiles de dificultad y reglas de adaptación que consideren variables adicionales al número de intentos fallidos, como el desempeño del jugador, el tiempo de resolución de los objetivos o la utilización de recursos durante la partida. Esto permitirá construir experiencias de juego más adaptativas y personalizadas.

Finalmente, se recomienda continuar realizando pruebas con usuarios pertenecientes al público objetivo durante las siguientes fases de desarrollo del videojuego. La evaluación continua de las mecánicas implementadas permitirá identificar nuevas oportunidades de

mejora y validar la incorporación de futuras funcionalidades antes de la integración definitiva de los recursos visuales y narrativos del proyecto.

Referencias

[1] K. Salen and E. Zimmerman, Rules of Play: Game Design Fundamentals. Cambridge, MA, USA: MIT Press, 2004.

[2] T. Fullerton, Game Design Workshop: A Playcentric Approach to Creating Innovative Games, 3rd ed. Boca Raton, FL, USA: CRC Press, 2014.

[3] G. K. Sepúlveda, F. Besoain, and N. A. Barriga, "Exploring dynamic difficulty adjustment in videogames," in 2019 IEEE CHILEAN Conference on Electrical, Electronics Engineering, Information and Communication Technologies (CHILECON), 2019, pp. 1–6.

[4] J. Schell, The Art of Game Design: A Book of Lenses, 2nd ed. Boca Raton, FL, USA: CRC Press, 2015.

[5] J. Gregory, Game Engine Architecture, 3rd ed. Boca Raton, FL, USA: CRC Press, 2018.

[6] M. Zohaib, "Dynamic Difficulty Adjustment (DDA) in Computer Games: A Review," Advances in Human-Computer Interaction, vol. 2018, pp. 1–12, 2018, doi: 10.1155/2018/5681652.

[7] D. Ang and A. Mitchell, "Comparing Effects of Dynamic Difficulty Adjustment Systems on Video Game Experience," in Proceedings of the Annual Symposium on Computer-Human Interaction in Play (CHI PLAY '17), Amsterdam, The Netherlands, 2017, pp. 317–327, doi: 10.1145/3116595.3116623.

[8] M. Borg, V. Garousi, A. Mahmoud, T. Olsson, and O. Stålberg, "Video Game Development in a Rush: A Survey of the Global Game Jam Participants," IEEE Transactions on Games, vol. 12, no. 3, pp. 246–259, 2020, doi: 10.1109/TG.2019.2910248.

[9] B. B. Marklund, H. Engström, M. Hellkvist, and P. Backlund, "What Empirically Based Research Tells Us About Game Development," The Computer Games Journal, vol. 8, pp. 179–198, 2019, doi: 10.1007/s40869-019-00085-1.

[10] A. Khalifa, M. C. Green, G. Barros, and J. Togelius, "Intentional Computational Level Design," arXiv:1904.08972, 2019.

[11] S. Stahlke and P. Mirza-Babaei, “Making the Thing,” in The Game Designer's Playbook: An Introduction to Game Interaction Design. Oxford, U.K.: Oxford University Press, 2022, pp. 280–305, doi: 10.1093/oso/9780198845911.003.0010.

[12] I. Millington and J. Funge, Artificial Intelligence for Games, 3rd ed. Boca Raton, FL, USA: CRC Press, 2019.

[13] S. Rogers, Level Up! The Guide to Great Video Game Design, 2nd ed. Chichester, U.K.: Wiley, 2014.

[14] P. D. Paraschos and D. E. Koulouriotis, “Game difficulty adaptation and experience personalization: A literature review,” International Journal of Human–Computer Interaction, vol. 39, no. 1, pp. 1–22, 2023, doi: 10.1080/10447318.2021.2020008.

[15] A. J. A. Seyderhelm and K. Blackmore, “Systematic Review of Dynamic Difficulty Adaption for Serious Games: The Importance of Diverse Approaches,” SSRN Electronic Journal, 2022, doi: 10.2139/ssrn.3982971.

[16] R. S. Pressman y B. R. Maxim, Software Engineering: A Practitioner's Approach, New York, USA: McGraw-Hill, 2020.

[17] Unity Technologies, *Unity Manual*. Unity Technologies, 2025. [Online]. Available: https://docs.unity3d.com/Manual/. Accessed: Jul. 21, 2026.

[18] Blender Foundation, Blender Manual. Blender Foundation, 2025. [Online]. Available: https://docs.blender.org/manual/en/latest/. Accessed: Jul. 21, 2026.

[19] I. Sommerville, *Software Engineering*, 10th ed. Boston, MA, USA: Pearson, 2016.

[20] I. Mistrik, R. Bahsoon, P. Eeles, R. Roshandel, and M. Stal, Eds., Relating System Quality and Software Architecture. Waltham, MA, USA: Morgan Kaufmann, 2014, doi: 10.1016/C2013-0-00646-9.

[21] K. Werbach and D. Hunter, *For the Win: How Game Thinking Can Revolutionize Your Business*. Philadelphia, PA, USA: Wharton Digital Press, 2012.

[22] R. Hunicke, M. LeBlanc, and R. Zubek, “MDA: A Formal Approach to Game Design and Game Research,” in Proceedings of the AAAI Workshop on Challenges in Game AI, 2004.

[23] M. J. P. Wolf, *The Medium of the Video Game*. Austin, TX, USA: University of Texas Press, 2001.

[24] K. Isbister, *How Games Move Us: Emotion by Design*. Cambridge, MA, USA: MIT Press, 2016.

[25] J. Nielsen, Usability Engineering. San Francisco, CA, USA: Morgan Kaufmann, 1994.

[26] S. Rabin, Introduction to Game Development, 2nd ed. Boston, MA, USA: Charles River Media, 2010.

[27] C. Bateman and R. Boon, 21st Century Game Design. Hingham, MA, USA: Charles River Media, 2005.

[28] M. Csikszentmihalyi, Flow: The Psychology of Optimal Experience. New York, NY, USA: Harper & Row, 1990.

[29] G. Calleja, In-Game: From Immersion to Incorporation. Cambridge, MA, USA: MIT Press, 2011.

[30] J. H. Murray, Hamlet on the Holodeck: The Future of Narrative in Cyberspace. New York, NY, USA: Free Press, 1997.

[31] D. Church, “Formal abstract design tools,” Gamasutra, 1999.

[32] E. Adams and A. Rollings, Fundamentals of Game Design, 2nd ed. Upper Saddle River, NJ, USA: Prentice Hall, 2007.

[33] H. Jenkins, “Game Design as Narrative Architecture,” in First Person: New Media as Story, Performance, and Game, N. Wardrip-Fruin and P. Harrigan, Eds. Cambridge, MA, USA: MIT Press, 2004.

[34] R. Hunicke, “The Case for Dynamic Difficulty Adjustment in Games,” in Proceedings of the 2005 ACM SIGCHI International Conference on Advances in Computer Entertainment Technology (ACE '05), Valencia, Spain, 2005, pp. 429–433, doi: 10.1145/1178477.1178573.

[35] B. Buxton, Sketching User Experiences: Getting the Design Right and the Right Design. San Francisco, CA, USA: Morgan Kaufmann, 2007.

[36] M. Richards and N. Ford, Fundamentals of Software Architecture: An Engineering Approach. Sebastopol, CA, USA: O'Reilly Media, 2020.

[37] G. Booch, J. Rumbaugh, and I. Jacobson, The Unified Modeling Language User Guide, 2nd ed. Boston, MA, USA: Addison-Wesley, 2005.

[38] R. Nystrom, Game Programming Patterns, vol. 68. Carrollton, GA, USA: Genever Benning, 2014

[39] L. Bass, P. Clements, and R. Kazman, Software Architecture in Practice, 3rd ed. Boston, MA, USA: Addison-Wesley, 2012.