Escuela
Politécnica
Superior

King of the Magnets:

Juego de Realidad Virtual en Unreal Engine

Grado en Ingeniería Multimedia

Trabajo Fin de Grado

Universitat d'Alacant
Universidad de Alicante

Autor:
Ennio Fernando Velásquez Berlingeri

Tutor/es:
Carlos J. Villagrá Arnedo

Julio 2023

Resumen

En este trabajo se documenta el desarrollo de un videojuego para realidad virtual. No solo un prototipo o una prueba de concepto, sino un videojuego que se pueda disponer en el mercado, con todos los elementos que hacen de este un producto completo y coherente. Pasando por todas las etapas de desarrollo, desde la concepción de la idea, hasta el resultado final y pulido que cualquier persona se puede descargar y jugar.

Se ha obtenido como resultado una experiencia jugable de comienzo a fin, con sus respectivos menús, interfaces, salas, progresión, arte, sonido, y acabado general que hacen de este un producto que se puede publicar en el mercado. El juego saca el máximo provecho de las capacidades de la realidad virtual, utilizando mecánicas, técnicas de inmersión, y diseño estético que promueven la inmersión utilizando la visión, el audio, y los movimientos. A este producto se le denomina demo debido a que con su pulido y acabado actual, tiene campo para ampliarse con más niveles y contenido, así extendiendo el tiempo de juego necesario para explorar más en profundidad las mecánicas. Sin embargo, el producto obtenido es una experiencia completa con todas las características de un producto terminado, y se pretende publicar en su estado actual como una demo.

Los aspectos de los que más se hablan en este documento incluyen mecánicas, progresión de juego, diseño de niveles, estética visual, diseño de sonido, arte, narrativa, y el aspecto más fundamental, la programación. Todos estos son los elementos que un juego completo necesita tener. En este proyecto se hace especial énfasis en el aspecto técnico a la hora de programar mecánicas, solucionar problemas, y utilizar adecuadamente las tecnologías en cuestión. Haciendo especial mención del desarrollo de inteligencia artificial, mecánicas de juego, optimización, arte técnico, y uso de diversas herramientas, sumado a una variedad de técnicas para lograr el resultado deseado en cada instante.

Se explica no solo cómo se ha desarrollado un producto, sino también la visión y los métodos utilizados para trazar esta ruta de proyecto y avanzar en cada momento del desarrollo. Adicionalmente, se explora el ambiente actual de la realidad virtual, y el lugar que este tipo de proyectos tienen en el mercado.

Se explican las distintas tecnologías de última generación utilizadas para hacer realidad este proyecto, como dispositivos de realidad virtual y motores de videojuegos con sus respectivas herramientas, virtudes, y restricciones.

Motivación, justificación y objetivo general

Llegando al final de mi carrera, muchos compañeros míos tenían ideas claras de lo que querían hacer para su TFG, sea investigaciones en temas que siempre han querido profundizar, o incluso contar una historia a través de una animación o juego.

Cuando me tocó escoger a mí, sin saber que quería hacer en concreto, tomé una decisión. Quiero que mi TFG sea un reto. Un reto que pusiera a prueba mis habilidades técnicas, mis habilidades creativas, y ultimadamente, mis habilidades como ingeniero.

Sin dudarlo, decidí que escogería dos tecnologías de las que no tengo ni idea, y con las que nunca haya trabajado, y haría un proyecto en base a ello. Estas tecnologías fueron la realidad virtual, y *Unreal Engine*.

Mi objetivo en este proyecto es, no solo desarrollar el producto, sino aprender desde cero *Unreal Engine*, y aprender desde cero como se producen aplicaciones de realidad virtual. Por lo que mi meta es, más allá de desarrollar un producto final, es que al final de este trabajo, pueda considerarme alguien capaz de defenderse con estas tecnologías, poniendo a prueba mi versatilidad como desarrollador en el área multimedia.

Además, estas no son tecnologías que haya escogido de forma totalmente arbitraria. *Unreal Engine* es un motor que está entre los más usados en la industria de videojuegos AAA, y la realidad virtual es una tecnología nueva e innovadora que está en etapas muy tempranas de su madurez como industria, y se proyecta que tendrá mucho crecimiento como mercado en los próximos años. Por lo que además del reto que me supone aprender estas tecnologías, me dan una serie de conocimientos y habilidades que considero que me serán de inmenso valor en mi futuro profesional.

Finalmente, mi objetivo secundario es aprender a trabajar en el desarrollo de un producto que se pueda publicar y vender, y obtener toda esta valiosa experiencia que hace falta para entrar a la industria. No obstante, este objetivo se encuentra en segundo plano ante mi objetivo de aprender a utilizar las tecnologías previamente mencionadas.

Agradecimientos

Quiero agradecerle a mi familia. A mi madre Esther y mi padre Ennio José por siempre haberme apoyado, y haber confiado en mis decisiones, mis capacidades, y lo más importante, haber creído siempre en mí.

También quiero agradecerle a mi pareja Andrea, que ha estado a mi lado desde antes de comenzar en Multimedia. Gracias por darme esa fuerza que tanto he necesitado para afrontar el futuro y todos sus retos.

Haber llegado aquí habría sido imposible sin el soporte de mis amigos de Multimedia, ellos son mis compañeros en este viaje y siempre nos hemos ayudado y nos seguiremos ayudando. El talento y la habilidad no sirven de nada si no tienes amigos en los que confiar y apoyarte. Danny, Caballero, Adri, Iván, el otro Iván, entre muchísimos más, ellos saben quiénes son.

Finalmente, quiero agradecer a todos los profesores de Multimedia que me han enseñado tanto y han sido un ejemplo a seguir para mí en mi futuro profesional, mencionando especialmente a mi tutor Carlos Villagrá, a Alicia Garrido, y al talentosísimo Fran Gallego.

Dedicatoria

Este trabajo va dedicado a mi mamá, que no le gustan los videojuegos. Aun así, siempre me ha apoyado.

Citas

¡Adelante con los faroles!

Ennio Velásquez (Abuelo)

El sabor de la vida es el sabor de la lucha.

Fernando Berlingeri (Nonno)

Índice de contenidos

Índice de figuras

1. Introducción

La realidad virtual (VR) es una de las tecnologías recientes con promesa de tener un enorme impacto en nuestro futuro. Es un mercado con un vivaz crecimiento que no para de llamar la atención de grandes compañías e inversores. Esta tecnología consta en la utilización de dispositivos como pantallas, cascos, y visores para sumergir a los usuarios en una experiencia virtual que busca simular los sentidos como la visión y la audición.

La realidad virtual tiene un sinfín de aplicaciones, de las cuales quizá la más importante, y que más huella ha dejado a día de hoy, es el entretenimiento y los videojuegos.

En este trabajo se va a documentar el desarrollo de un videojuego para realidad virtual. No solo un prototipo o una prueba de concepto, sino un videojuego que se pueda disponer en el mercado, con todos los elementos que hacen este un producto completo y coherente. Pasando por todas las etapas de desarrollo, desde la concepción de la idea, hasta el resultado final y pulido de forma que cualquier persona se puede descargar y jugar. Se explicará no solo cómo se ha desarrollado un producto, sino también la visión y los métodos utilizados para trazar esta ruta de proyecto y avanzar en cada momento del desarrollo.

Los aspectos de los que más se van a hablar incluyen mecánicas, progresión de juego, diseño de niveles, estética visual, diseño de sonido, arte, narrativa, y el aspecto más fundamental, la programación. Todos estos son los elementos que un juego completo necesita tener. En este proyecto se hará especial énfasis en el aspecto técnico a la hora de programar mecánicas, solucionar problemas, y utilizar adecuadamente las tecnologías en cuestión. Haciendo especial mención del desarrollo de inteligencia artificial, mecánicas de juego, optimización, arte técnico, y uso de diversas herramientas, sumado a una variedad de técnicas para lograr el resultado deseado en cada instante.

Adicionalmente, se va a explorar el ambiente actual de la realidad virtual, y el lugar que este tipo de proyectos tienen en el mercado. Se explicarán las distintas tecnologías de última generación utilizadas para hacer realidad este proyecto, como dispositivos de realidad virtual y motores de videojuegos con sus respectivas herramientas, virtudes, y restricciones.

Como nota adicional, al este ser un documento con mucha terminología inglesa, debido al ámbito informático del proyecto, se adjunta un glosario al final del mismo con diversas definiciones utilizadas.

Esta memoria se ha organizado de forma de que a continuación, en el apartado 2, se presenta el marco teórico. En el apartado 3 se hará una sección de objetivos, donde se explicarán los objetivos generales del proyecto. En el apartado 4 se explicará la metodología a emplear durante el desarrollo. En el apartado 5 se hará una sección de análisis en el que se explorarán las herramientas que pueden hacer este proyecto posible, además de inspeccionar datos del mercado para tener una idea de la viabilidad del producto. El apartado 6, para efectos prácticos el cuerpo del trabajo, explicará en detalle todo el proceso de desarrollo del juego de comienzo a fin. Finalmente, el apartado 7 analizará las conclusiones del proyecto y el resultado obtenido.

2. Marco teórico

A continuación, se va a informar sobre las tecnologías que juegan un papel en este proyecto, y se va a explorar el mercado actual que servirá como antecedentes y proporcionará referencias de la viabilidad del producto a realizar.

2.1. Realidad virtual

Se le conoce a la realidad virtual como el uso de un ordenador para sumergir a un usuario en un entorno sensorial que simula la realidad. Esto se logra a través de dispositivos interactivos que envían y reciben información como gafas, cascos, guantes, pantallas, o trajes.

El termino realidad virtual se acuñó en 1987 por Jaron Lanier, un ingeniero pionero en la tecnología. También se le conocía a la realidad virtual como 'realidad artificial'. Desde incluso muchos años antes hubo muchos prototipos e innovaciones con la intención de crear un entorno virtual inmersivo e interactivo, como el uso de pantallas panorámicas, gafas para observar figuras tridimensionales, e interfaces de interacción como lápices y guantes.

La realidad virtual, hoy en día, se entiende como el uso de unas gafas con pantallas muestran una imagen del entorno simulado, y con el uso de sensores de movimiento, animan el entorno a tiempo real para dar la ilusión de que el usuario se encuentra ahí. Con el uso de guantes o controles, el usuario incluso puede tener cierta sensación de tacto, e interactuar con objetos del entorno.

La realidad virtual tiene aplicaciones en el área del ocio, la educación, el arte, y el trabajo. Y es una tecnología que sigue en desarrollo y crecimiento.

Dispositivos de realidad virtual contemporáneos incluyen gafas con su propio sistema operativo que permiten ejecutar sus aplicaciones de forma independiente e inalámbrica, como es el caso del *Meta Quest* 2, gafas que sirven como soporte para utilizar un teléfono móvil como dispositivo de realidad virtual (utilizando su giroscopio para detectar los movimientos de la cabeza del usuario), hasta incluso dispositivos que sirven como periféricos para un ordenador, como es el caso del *Valve Index*. Este último tipo de dispositivo se conoce como PCVR.

Entre los dispositivos de realidad virtual más populares en el mercado actual se encuentran el *Meta Quest* 2, con su sistema operativo propio y capacidad de jugar inalámbricamente sin depender de ningún otro equipo, además de que tiene la capacidad de conectarse a un ordenador y usarse como PCVR. El *Valve Index*, que ofrece grandes capacidades técnicas y gráficas, sin embargo, es exclusivo

de PCVR, por lo que necesita un ordenador en el cual jugarse. Finalmente se mencionará el *Google Cardboard*, que sirve como una introducción barata al público al mundo de la realidad virtual, ya que es simplemente un soporte de cartón que sostiene el móvil como si fueran unas gafas de realidad virtual convencionales. El móvil es el que ejecuta las aplicaciones de realidad virtual dando una simulación bastante conseguida sin uso de una tecnología demasiado sofisticada (Lowood, 2023).

Figura 1. Gafas de realidad virtual Meta Quest 2
(Fuente Meta: https://www.meta.com/es/quest/products/quest-2)

Figura 2. Google Cardboard
(Fuente Google: https://arvr.google.com/cardboard)

2.2. Realidad Virtual para PC (PCVR)

PCVR son las iniciales de PC (*Personal Computer*) y VR (*Virtual Reality*), que viene siendo nada más que aplicaciones de Realidad Virtual que son ejecutadas en ordenadores convencionales a través de periféricos como gafas, cascos, y mandos. A pesar de que la interfaz de la simulación se hace a través de las gafas, la aplicación es procesada por el ordenador, permitiendo de esta manera, la ejecución de programas mucho más exigentes a nivel gráfico y computacional.

Usualmente las desarrolladoras de realidad virtual publican 2 versiones de su juego: Una versión optimizada y ligera para que pueda ser ejecutada en el sistema operativo del *Meta Quest* 2, y una versión con mejoras gráficas y funcionales considerable que es publicada para PC para ser jugada como PCVR.

Usualmente los productos desarrollados para la realidad virtual en PC tienen como objetivo experiencias más grandes e inmersivas, sacando provecho de la inmensa capacidad de procesamiento que ofrece un ordenador de sobremesa. A diferencia de los juegos de VR móviles, los juegos de PCVR, por lo general, buscan explorar nuevos territorios de lo que es posible en la realidad virtual, ofreciendo experiencias más completas y con muchas menos restricciones técnicas.

2.3. Productos referentes

A continuación, se mostrarán 3 ejemplos de aplicaciones de PCVR que servirán como ejemplo y referencia para el tipo de aplicación de realidad virtual que se desarrolla en este trabajo. A pesar de que hay muchas funcionalidades y especialidades de las distintas aplicaciones de realidad virtual, para efectos de este trabajo nos enfocaremos en aplicaciones dedicadas al ocio y entretenimiento digital. Cabe destacar que todos estos juegos que se van a exponer tienen versiones tanto para el *Meta Quest* 2 como para PCVR. Siendo las versiones de PCVR mucho menos restringidas a nivel técnico y más fieles a la visión original del proyecto. Debido a esto, hablaremos únicamente de las versiones de PCVR, ya que exponen de forma más limpia y completa el producto final. Además, cabe destacar, que el proyecto que se pretende desarrollar en este trabajo tiene como objetivo ser exclusivamente de PCVR, por lo que las referencias más acertadas van a provenir de productos de semejantes necesidades.

2.3.1 Bonelab

Bonelab es un juego de acción y combate, enfocado en un sistema de físicas profundo donde se puede interactuar con el mundo del juego y se pueden usar una variedad de armas, desde palos, espadas y hachas, hasta pistolas, y rifles. Idealmente se juega de pie haciendo movimientos con el cuerpo completo (Stress Level Zero, 2022). De este juego se ha obtenido inspiración por la variedad de enemigos, y la manera de utilizar las distintas armas que se encuentran por el juego mientras el jugador avanza en la mazmorra.

Figura 3. Screenshot de Bonelab
(Fuente Steam: https://store.steampowered.com/app/1592190/BONELAB)

2.3.2 I Expect You To Die

I Expect You To Die es un juego de puzles en el que te pones en el rol de un espía con poderes psíquicos. En este juego interactuarás con objetos, obtendrás pistas, y superarás las misiones, ya sea desactivar una bomba, encontrar unos planos secretos, o manejar un submarino. En cada nivel el jugador es estacionario, por lo que idealmente se juega sentado. El personaje al tener poderes psíquicos, el jugador puede mover cosas a distancia y atraer objetos a través del aire, lo que permite interactuar con todo el entorno a pesar de que el jugador no se mueva (Schell Games, 2016). De este juego se ha obtenido inspiración por su ambiente cómico con los diálogos de los personajes, y las maneras interesantes de interactuar con el entorno para resolver puzles, como encendiendo cosas en fuego, abriendo palancas, entre muchos otros.

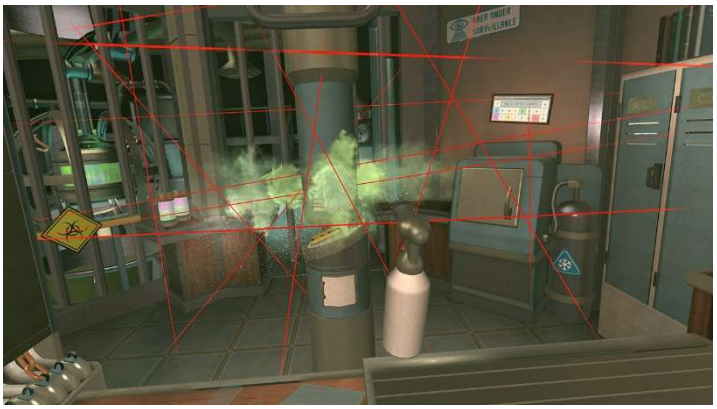

Figura 4. Screenshot de I Expect You To Die
(Fuente Steam: https://store.steampowered.com/app/587430/I_Expect_You_To_Die)

2.3.3 *Superhot* VR

Superhot VR es un juego de acción en el que superarás niveles venciendo una serie de enemigos. La mecánica principal del juego es que, si no te mueves, el tiempo se ralentiza, y mientras más te mueves, el tiempo se acerca más a su velocidad normal, por lo que acabas esquivando balas, lanzando objetos, y disparando a cámara lenta a las distintas oleadas de enemigos. En este juego puedes usar cualquier cosa como arma, desde una maceta que puedes agarrar y lanzar hasta un rifle automático en cada mano (SUPERHOT Team, 2016). Este juego ha inspirado este proyecto por sus mecánicas de combate divertidas y satisfactorias, cuando lanzas objetos y vences enemigos.

Figura 5. Screenshot de Superhot VR
(Fuente Steam: https://store.steampowered.com/app/617830/SUPERHOT_VR)

3. Objetivos

A continuación, se exponen los objetivos de este proyecto. En cada apartado, los objetivos están ordenados por relevancia, comenzando por los más importantes, y finalizando con los más secundarios y específicos.

Desarrollo de un producto jugable

El objetivo principal de este proyecto, que prima sobre cualquier objetivo, y con máxima prioridad es desarrollar un videojuego de realidad virtual para PC, también conocido como PCVR. El propósito es, que más que un prototipo, el producto desarrollado sea una experiencia completa, incluyendo su progresión, apartado visual, jugabilidad y diseño de sonido.

Para el final de este proyecto es indispensable que haya un *build* jugable con su debido tutorial y progresión de niveles en forma de juego, evitando que el producto final incluya niveles de prototipos y partes incompletas. El juego realizado debe poder exponerse ante el público como algo jugable que pueden experimentar, más que un producto incompleto o en desarrollo.

El cumplimiento de este objetivo implica una ramificación en una serie de objetivos individuales más pequeños que proceden a explicarse a continuación.

Aprendizaje sobre *Unreal Engine* y realidad virtual

Es prioridad de este trabajo familiarizarse y obtener experiencia en ambas tecnologías; *Unreal Engine* y realidad virtual. Para el final del proyecto se espera haber conseguido un nivel de conocimiento y habilidad suficientemente competente para poder acceder en la industria utilizando estas tecnologías.

Prototipado y pulido de mecánicas

En este trabajo se hace mucho enfoque en prototipar mecánicas propias en *Unreal Engine* y su pulido para que con estas mecánicas haya mucho potencial de hacer niveles divertidos, variados y originales.

Profundizado en técnicas de *technical art*

Se espera trabajar y aplicar diversas técnicas que eleven el apartado gráfico del juego haciendo uso de herramientas y “trucos” de *technical art* para lograr un juego estéticamente agradable y original. Estas técnicas incluyen efectos de postprocesado, partículas, animación, y técnicas de optimización gráfica.

Publicación de un producto comercial

Se pretende que al final del desarrollo el producto obtenido tenga un nivel de calidad y acabado mínimo para poder ser publicado, sea gratis o pago, en alguna plataforma de distribución de videojuegos.

4. Metodología

Se ha decidido llevarlo a cabo con una metodología en 3 fases como se representa en el diagrama que se encuentra a continuación.

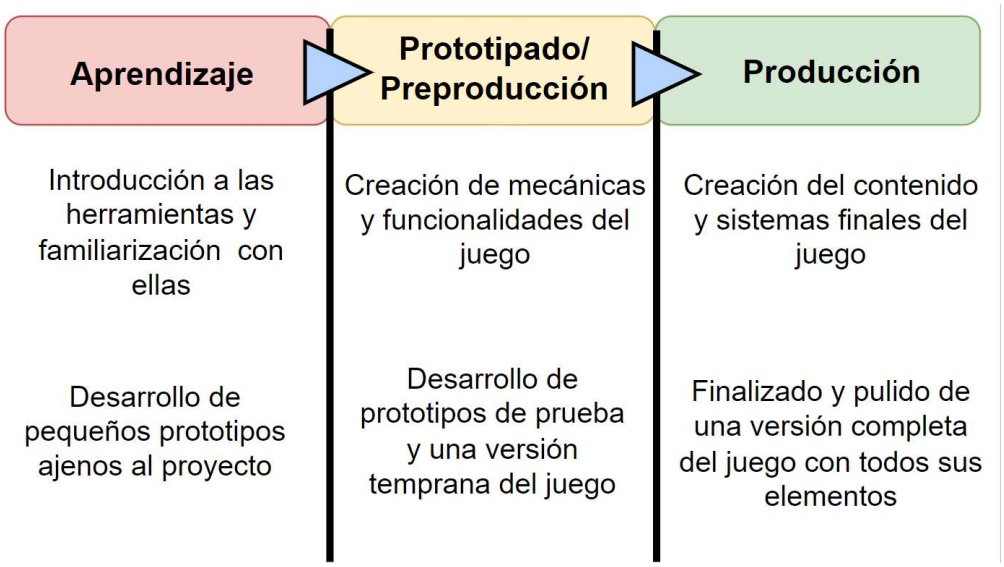

Figura 6. Estructura de la metodología.
(Fuente: Elaboración propia)

La primera fase será una fase de aprendizaje y preparación, en la cual se estudiará sobre *Unreal Engine* y sobre realidad virtual. El objetivo de esta fase es aprender desde cero a utilizar *Unreal Engine*, y eventualmente obtener suficiente soltura en dicha herramienta como para avanzar a la segunda fase.

La segunda fase se trata de una fase de prototipado y diseño. Diseñar un juego y luego desarrollarlo es una receta para el desastre, basado en experiencias personales, por lo que lo primero que se va a hacer en esta segunda fase es prototipar a partir de una idea. Con dicha idea de juego en mente, se prototiparán las mecánicas y se pondrán a prueba los conceptos que funcionan y los que no, de esta manera dando una imagen mucho más concisa y realista del producto a desarrollar. Estos prototipos serían prototipos rápidos que no sean demasiado elaborados de desarrollar, también conocido como prototipos *'quick and dirty' (rápido y sucio)*, cuya única intención es probar un concepto de diseño de forma rápida, ignorando buenas prácticas de programación, dado que el código de este prototipo será desechado. En el final de esta fase ya se deberían haber prototipado las mecánicas y sistemas que estarán presentes en el juego.

La tercera fase es el desarrollo del producto, partiendo de las decisiones de diseño y prototipos provenientes de la fase anterior, esta fase se trata de pulir las mecánicas y sistemas previamente

establecidos y crear todo el contenido del juego. Principalmente se comenzaría por pulir las mecánicas, sistemas de enemigos o puzles o cualquier tipo de interacciones del juego. Con esto hecho, se crea todo el contenido del juego, lo cual incluye niveles, enemigos, retos y progresión. Finalmente, se crea el arte 3D, efectos especiales, sonido, e interfaz. Todo esto se produce al final, para no perder de vista la importancia de las mecánicas y niveles en las fases más tempranas de la producción.

5. Análisis

En este apartado se analizarán distintos factores que juegan un rol en posicionar las características tecnológicas y comerciales de este trabajo y el producto que se desarrolla con él. Todo lo explicado a continuación dará una visión más completa del rol de este proyecto en el ambiente actual de los juegos de realidad virtual, y de que tecnologías y herramientas sacará provecho.

5.1. Herramientas

A continuación, se van a describir las herramientas que serán utilizadas para desarrollar el juego, y cuáles son sus características más importantes, tanto su papel en la industria, como sus elementos más relevantes para efectos de este juego.

5.1.1 *Unreal Engine* 5

Unreal Engine es un motor de videojuegos creado por *Epic Games*. Posee numerosas herramientas y capacidades gráficas para la creación y diseño de contenidos multimedia, como videojuegos o aplicaciones a tiempo real. Además, posee compatibilidad multiplataforma, permitiendo desarrollar aplicaciones para ordenadores, consolas, móviles, hasta equipos de realidad virtual.

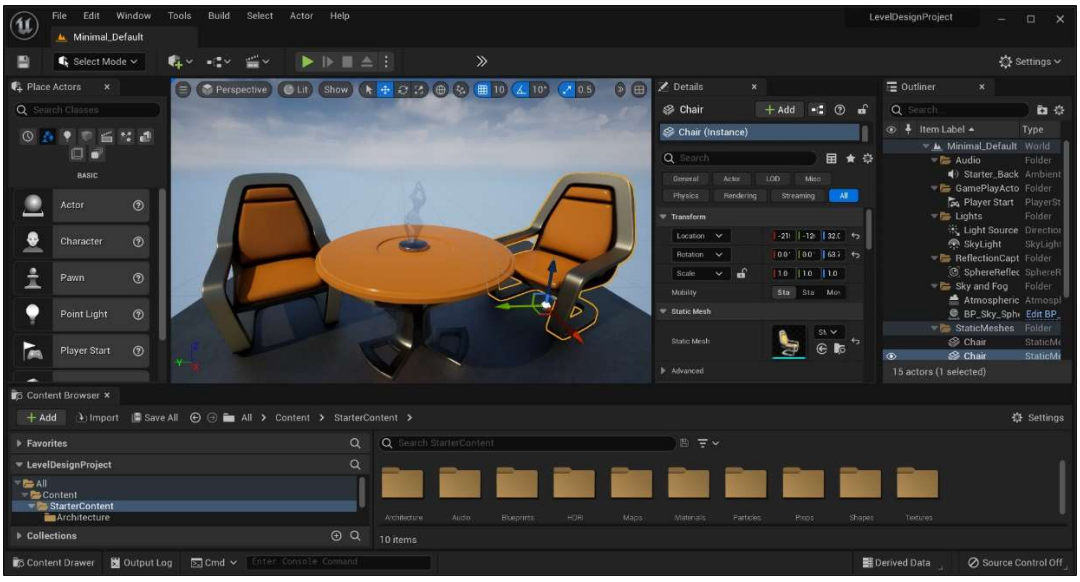

Figura 7. Interfaz de Unreal Engine.
(Fuente: Documentación de Unreal Engine: https://docs.unrealengine.com/5.1/en-US/level-editor-in-unreal-engine/)

Una de las características más importantes de *Unreal Engine* es la existencia de *Blueprints*, un sistema de programación visual basado en gráficos de nodos. Con *Blueprints* se puede programar prácticamente lo mismo que con la programación convencional en C++ que *Unreal Engine* incluye.

La última versión de *Unreal Engine*, *Unreal Engine* 5, destaca por sus grandes capacidades gráficas a tiempo real, generando gráficos de alta calidad y fácil integración en proyectos. Las herramientas más destacables de esta versión del motor son Nanite y Lumen. Nanite es una tecnología de renderizado con la capacidad de manipular millones de triángulos a la vez, permitiendo manejar mallas 3D para escenarios y personajes de una forma mucho más eficiente y detallada. Lumen es una tecnología de iluminación dinámica de alta calidad que permite generar gráficos realistas de forma eficiente a tiempo real (Epic Games, 2022) .

Figura 8. Demostración gráfica de Unreal Engine 5.
(Fuente GPUOpen: https://gpuopen.com/ue5-announce/)

En cuanto a su posición en el mercado, es uno de los motores de juegos más importantes en la industria, solo a la par de Unity. *Unreal Engine* ha sido adoptado por numerosos estudios, tanto de la industria de videojuegos como la industria de la animación y la arquitectura. Entre las empresas de videojuegos que utilizan esta herramienta se encuentran Electronic *Arts*, *Activision Blizzard*, *CD Projekt RED*, y *Riot*. Este motor se ha usado en productos de renombre mundial como *Fortnite*, *Gears Of War*, *Batman: Arkham Knight, Borderlands*, entre muchos otros.

Las alternativas existentes en el mercado son Unity y Godot. Estos tienen sus ventajas y desventajas. Sin embargo, *Unreal Engine* ha sido el motor que prima sobre estos dos debido a su potente capacidad gráfica, y su robusto repertorio de herramientas avanzadas para manejar realidad virtual, ambas cosas que Unity y Godot carecen.

5.1.2 *Meta Quest* 2 (*Oculus Quest* 2)

El *Meta Quest* 2 es un dispositivo de realidad virtual desarrollado por la empresa *Meta*, anteriormente conocida como *Facebook*. Este dispositivo cuenta con una pantalla LCD de 1832 x

1920 pixeles por ojo y un campo de visión de 90 grados. Posee dos mandos de movimiento. Este equipo posee una precisa tecnología de seguimiento para la cabeza y para los mandos.

Figura 9. Meta Quest 2.
(Fuente GPUOpen: https://www.meta.com/es/quest/products/quest-2/)

El *Meta Quest* 2 permite a los usuarios disfrutar experiencias de realidad virtual de alta calidad de forma inalámbrica y autónoma, sin necesidad de un ordenador o una consola, sin embargo, este equipo se puede conectar a un ordenador para ejecutar aplicaciones de PCVR, es decir, aplicaciones de realidad virtual que se ejecutan sobre la plataforma de PC. Esta versatilidad hace al *Meta Quest* 2 un producto líder en la industria de la realidad virtual, recibiendo criticas positivas por sus cualidades y facilidad de uso (Meta, 2023).

Este dispositivo ha vendido más de 14 millones de unidades, y la compañía *Meta*, ha realizado una importante inversión tanto en la tecnología como el producto, esto puede ser indicio del plan y la proyección a largo plazo que puede tener este producto en la industria (Delgado, 2022).

En este caso, la alternativa al *Meta Quest* 2 habría sido el *Valve Index*, que en muchos aspectos tiene mejores capacidades que este. Sin embargo, la flexibilidad que ofrece el *Meta Quest* 2, sumado a su enorme público en el mercado, hacen que este sea el ganador de la competición, y sea el indicado para este proyecto.

5.2. Estudio de mercado

En este apartado se va a realizar un estudio breve de mercado sobre la situación actual de la industria de juegos de RV, y la posición de productos similares a este proyecto en este mercado.

En 2020 el mercado de la realidad virtual en videojuegos tenía un tamaño de 6.26 mil millones de USD, y se estima que para 2028 crezca a 53.44 mil millones USD (Fortune Business Insights, 2021).

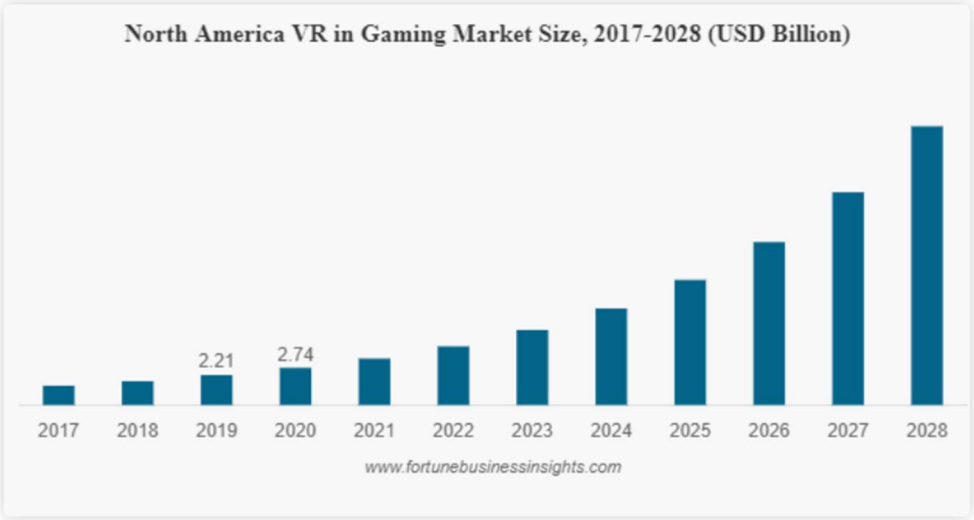

Figura 10. Diagrama del alza del mercado de los videojuegos de VR
(Fuente Fortune Business Insights: https://www.fortunebusinessinsights.com/industry-reports/virtual-reality-gaming-market-100271)

La demanda de juegos de realidad virtual está en alza principalmente por la adopción del público del hardware de realidad virtual, por productos como *Meta Quest* 2, *Valve Index*, *Playstation* VR, y productos futuros como *Meta Quest* 3, e incluso *Apple Vision Pro*, los cuales tienen como objetivo captar las grandes masas y llevar la realidad virtual al mercado ***mainstream***.

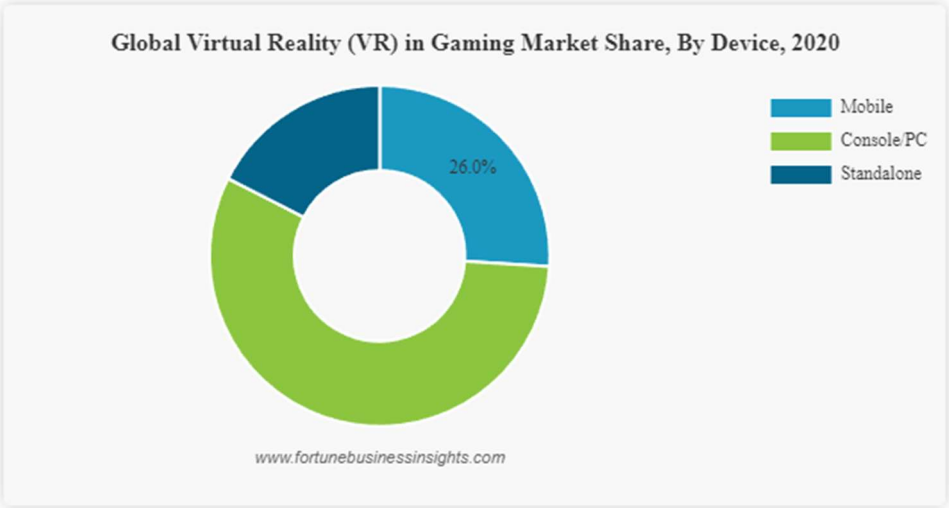

Figura 11. División del tamaño de mercado de los juegos VR por plataforma
(Fuente Fortune Business Insights: https://www.fortunebusinessinsights.com/industry-reports/virtual-reality-gaming-market-100271)

Como se observa en la gráfica, la mayor parte de este mercado se sitúa en los juegos de consola y PC. Sin embargo, no se debe subestimar la realidad virtual para móviles con un 26% del mercado. A pesar de esto, el estudio referenciado remarca que consola y PC seguirán siendo los líderes de esta industria (Fortune Business Insights, 2021).

En resumidas cuentas, debido al crecimiento en ventas de hardware para realidad virtual, el mercado de videojuegos para estas plataformas tiene una proyección importante.

Dado que este es un mercado nuevo, y en crecimiento, la competencia no es tan grande como en otras plataformas como juegos móviles o juegos de PC y consolas.

Como referencia, en *Steam* había 50.361 videojuegos en 2020, de los cuales solo un 7.68% eran de realidad virtual. De la demográfica que los juega, más de la mitad (52.28%) utilizaban cualquier versión de los *Oculus* de Facebook, incluido el *Meta Quest* 2. Esto nos muestra que la industria de videojuegos de realidad virtual es un mercado muy inmaduro aún, y hay mucho espacio para innovación y productos nuevos (Dean, 2023).

Se concluye que el mercado de videojuegos de realidad virtual es una industria en crecimiento, con demanda en aumento y un número relativamente bajo de juegos a disposición de los jugadores en comparación a otras plataformas, por lo que, un producto de las características que se proponen tiene un valor que hace que valga la pena desarrollarlo.

6. Desarrollo

En este apartado se va a documentar el proceso de desarrollo del proyecto. Se pretende documentar el progreso de forma paralela al desarrollo, por lo que se explicarán las distintas fases y situaciones que se han ido presentando a lo largo de la producción en el orden en el que han surgido. Primero se hablará de toda la preparación y aprendizaje necesarios para familiarizarse con las tecnologías con las que se va a trabajar. Luego, se explicará en detalle la etapa de preproducción y prototipado previos a la creación del juego definitivo. Finalmente se hablará de la fase de desarrollo, en la que se explicará la creación concreta del contenido que definirá al producto final, teniendo todo el prototipado previo como fundamento.

6.1. Preparación y Aprendizaje

En esta primera etapa de desarrollo el pilar principal ha sido el aprendizaje de *Unreal Engine* 5. Por lo que las primeras semanas del proyecto han consistido en seguir pequeños cursos y tutoriales sobre la herramienta, desarrollando pequeños prototipos jugables, para así conseguir una familiarización con las características y funcionalidades que ofrece *Unreal Engine* 5. A continuación se explicará que es lo que se ha aprendido de *Unreal Engine* en esta etapa inicial.

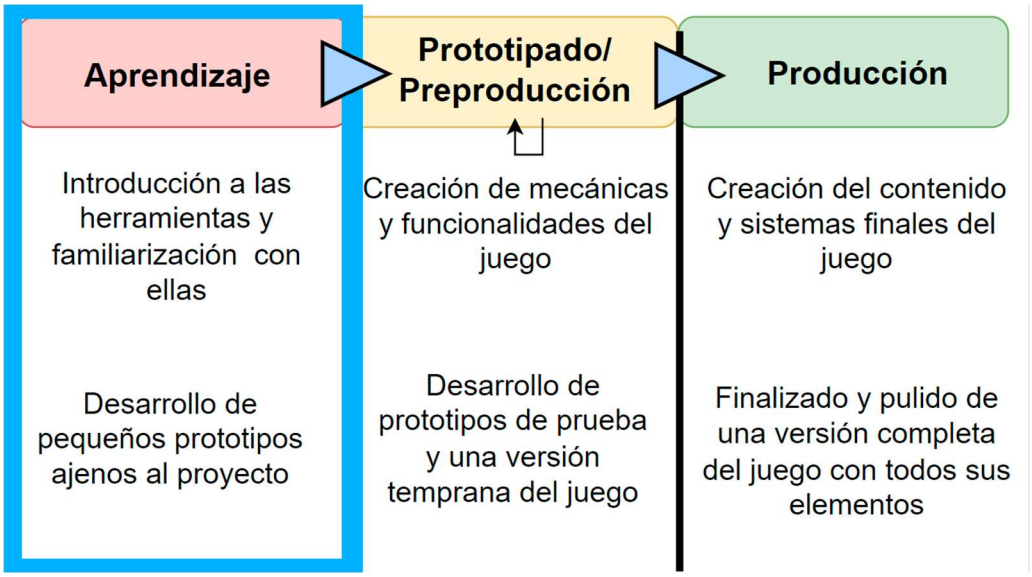

Figura 12. Fase de aprendizaje en la metodología
(Fuente: Elaboración propia)

Cabe destacar que a medida que se ha avanzado a las próximas etapas se han aprendido muchas otras funcionalidades, y se ha profundizado mucho más en las que se mencionan a continuación,

sin embargo, todo lo explicado en este segmento del trabajo se limita a lo aprendido antes del avance a la fase de prototipado, y todas las nuevas funcionalidades aprendidas y utilizadas serán explicadas en donde hayan tenido lugar en la línea de tiempo del desarrollo.

Por su parte, se aclara que las funcionalidades explicadas a continuación están explicadas desde el punto de vista que les di a la hora de aprenderlas y los usos que yo mismo les di durante esta etapa de aprendizaje.

Interfaz y estructura

Lo primero aprendido fue la interfaz de *Unreal*, principalmente como crear y cargar plantillas de juego. También se ha estudiado funciones básicas del motor como utilización del *viewport,* la jerarquía de la escena, el explorador de contenidos, y la ventana de detalles de un objeto (conocido como Actor en *Unreal*). Esta interfaz de por si permite crear, eliminar, y manipular elementos en la escena del juego. Con el uso de una plantilla de juego predeterminada, lo primero que se ha hecho es jugar con el movimiento de la cámara para explorar la escena, seleccionar objetos de la escena, editar sus valores en la ventana de detalles, y añadir elementos a la escena. También se han hecho jerarquías de actores asignando algunos actores como hijos de otros y entendiendo cómo funciona dicho sistema.

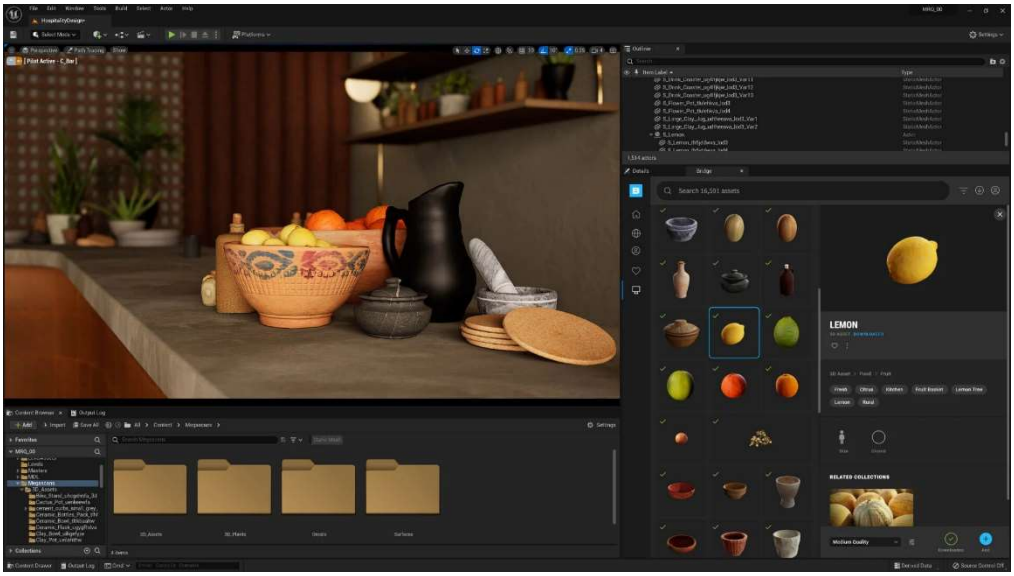

Figura 13. Ejemplo de interfaz de Unreal Engine 5
(Fuente Blog de Unreal Engine: https://www.unrealengine.com/en-US/blog/unreal-engine-5-is-now-available)

Actores

Los actores son los objetos dentro de una escena de *Unreal*. Dependiendo del tipo de actor que sean, estos poseen propiedades físicas, posición en el mundo, y componentes que les añaden funcionalidad. Con el uso de los actores y componentes predefinidos en *Unreal*, hay muchas funcionalidades que se pueden utilizar en un proyecto. También hay un sistema de herencia en el que se pueden crear actores genéricos y crear subclases a partir de estos actores para hacer variaciones de un tipo de actor y que todos mantengan propiedades de la clase de la que heredaron su funcionalidad.

Con esto por ejemplo se puede hacer una pelota, que tenga su posición en el espacio, malla esférica y propiedades físicas. Y a partir de esta pelota se pueden crear subclases que sean variaciones de la original. Una pelota de tenis que sea pequeña y tenga un rebote más pronunciado, y una bola de boliche que sea pesada, densa, y rebote muy poco.

Componentes

Como se ha mencionado previamente, los componentes son funcionalidades que puedes asociar a un actor. Distintos componentes darán distintas propiedades y capacidades a cada actor. Estos incluyen un componente de movimiento, un componente de malla que les permite dibujarse en la escena a través de un modelo 3D, un componente de cámara para mostrar donde estará la cámara de juego para dicho actor cuando sea usado como jugador, entre muchísimas más. También se pueden crear componentes customizados, que añaden la funcionalidad deseada por el desarrollador y estas se crean a través de *Blueprints* o C++, ambos serán explicados a continuación.

Blueprints

Blueprints quizá sea la funcionalidad más característica de todo *Unreal Engine*. En principio *Blueprints* es un sistema de programación visual basada en nodos, lo cual permite estructurar una lógica de juego sin tener que programar con código. *Blueprints* permite el manejo de variables y almacenamiento de datos.

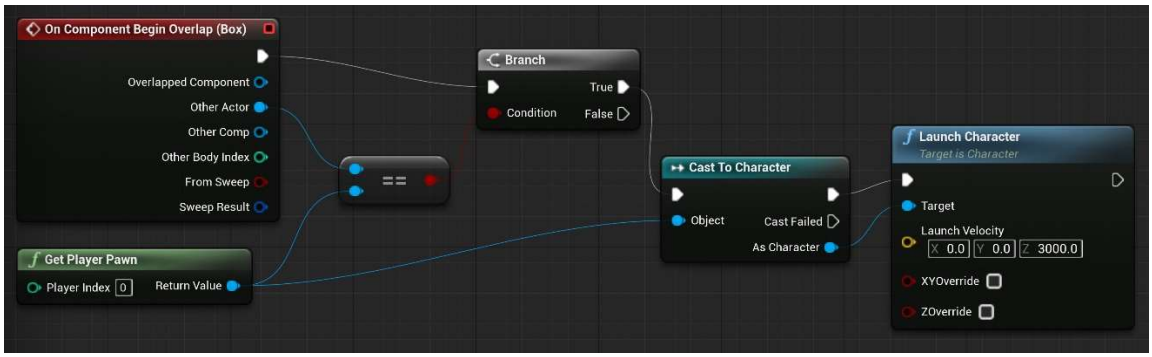

Figura 14. Ejemplo de código de Blueprints.

(Fuente Documentación de Unreal Engine: https://docs.unrealengine.com/4.27/en-US/ProgrammingAndScripting/Blueprints/QuickStart/)

La característica más llamativa durante el aprendizaje de *Blueprints* es la capacidad de crear código específico para una escena, un componente, y un actor, lo cual facilitaría la creación de eventos específicos en ciertas partes del juego e incrementa la versatilidad de desarrollo de forma considerable sin tener que alterar estructuras de objetos y herencia, ni teniendo que utilizar técnicas de programación avanzadas para hacer un sistema que funcione de esta manera con código convencional. En definitiva, *Blueprints* tiene muchas funcionalidades que hacen ciertas acciones mucho más convenientes que hacerlas por código.

Se ha utilizado *Blueprints* para imprimir mensajes en pantalla, captar *inputs* provenientes de periféricos, manejar movimientos de cámara sencillos, creación de proyectiles, y contadores de munición. También se han creado actores y componentes en *Blueprints*.

Físicas

En paralelo al aprendizaje de *Blueprints*, se han ajustado a discreción valores de las propiedades físicas de los actores, entre estos la gravedad, fricción, masa, e incluso tamaño. También se ha probado importar mallas 3D al motor, y observar las colisiones por defecto que poseen. Se ha aprendido a cambiar propiedades de las mallas de colisión mediante al editor de mallas.

Una vez entendidos estos conceptos, se ha probado funciones de *Blueprints* que permiten asignar velocidades e impulsos a los actores para hacer todo tipo de efectos, como saltos, lanzamiento de proyectiles, y rotación de personajes.

C++ para Unreal

A pesar de que la característica más remarcable de *Unreal Engine* es *Blueprints*, este también tiene soporte de C++. Se pueden crear clases, actores, e incluso componentes en C++ usando programación orientada a objetos bajo la estructura de *Unreal*. El uso de C++ en *Unreal* permite el

uso de las herramientas más sofisticadas de C++ que *Unreal* no necesariamente posee, como las librerías de estructura de datos, manejo más manual de la herencia con la capacidad de crear funciones virtuales, sobrecargas, y constructores específicos para actores y componentes.

Principalmente dos usos para C++ fueron los que más me llamaron la atención en este contexto. Primero, el hecho de que *Blueprints* tiene muy poca variedad de contenedores de datos, como colas o listas enlazadas. Todas estas estructuras podrían implementarse en sus propias clases de *Blueprints*, pero considero mucho más conveniente C++ en esta aplicación ya que C++ de por si trae muchas de estas estructuras de datos, y en el caso que haya que implementar dicha estructura por cuenta propia, en C++ igualmente es mucho más sencillo debido al manejo manual de la memoria. Segundo, me sorprendió alegremente la capacidad de crear clases en C++ y poder crear subclases en *Blueprints* y viceversa, dado que esto permite sacarles el máximo provecho a ambas herramientas de programación y abre el abanico de posibilidades de una forma substancial.

Sin embargo, la única desventaja que experimenté utilizando C++ en este contexto es el hecho de que los tiempos de compilación son elevados, el editor de *Unreal* muchas veces no detecta las clases creadas en C++ y toca cerrar y abrir varias veces para que funcione. Incluidas veces en las que *Unreal* borraba el progreso que hayas hecho en C++ y había que compilar otra vez, reiniciar todo, y asegurarse que todas las referencias dentro de *Unreal* estén en su lugar. Sin mencionar las veces que *Unreal Engine* '*crashea*' por razones fuera del alcance del programador, y con solamente reiniciar el editor, y a veces el ordenador, se arreglan estos problemas. Todo esto es un contratiempo que no ocurre con *Blueprints*. En *Blueprints* se compila y guarda y todo funciona a la perfección.

Con C++ se crearon componentes con funcionalidades específicas, se crearon clases base de las cuales luego se heredaba la funcionalidad a clases de *Blueprints*, e incluso crear subclases de las mismas clases de *Blueprints* para añadir ciertas funcionalidades en C++ más específicas a cada variación. En este caso se creó una clase Tanque que tenía de funciones abstractas de las cuales se creó un jugador y una torreta. Ambos con las mismas funciones base, el jugador podía moverse y disparar usando inputs, y la torreta tenía una inteligencia artificial básica que apuntaba al jugador si estaba en su campo de visión y disparaba acordemente.

Eventos

Unreal tiene un sistema de eventos en el que hay eventos predefinidos que disparan llamadas a funciones que se pueden definir por el usuario, y un sistema para poder crear eventos propios. Entre los eventos predefinidos se han usado eventos de colisión, eventos de muerte cuando se

acaba la vida de un componente de vida, eventos que reinician el nivel y eventos que disparan la pantalla de muerte del jugador.

Lumen

Lumen es la tecnología de iluminación más reciente de *Unreal* que da un acabado pulido y realista a tiempo real. Se ha construido una escena y ajustado los materiales y texturas para que se vean bien con la iluminación de Lumen. Se ha creado luz solar y la iluminación de una escena de una mazmorra, con efectos de postprocesado como niebla, oclusión ambiental y ajuste de exposición de la luz.

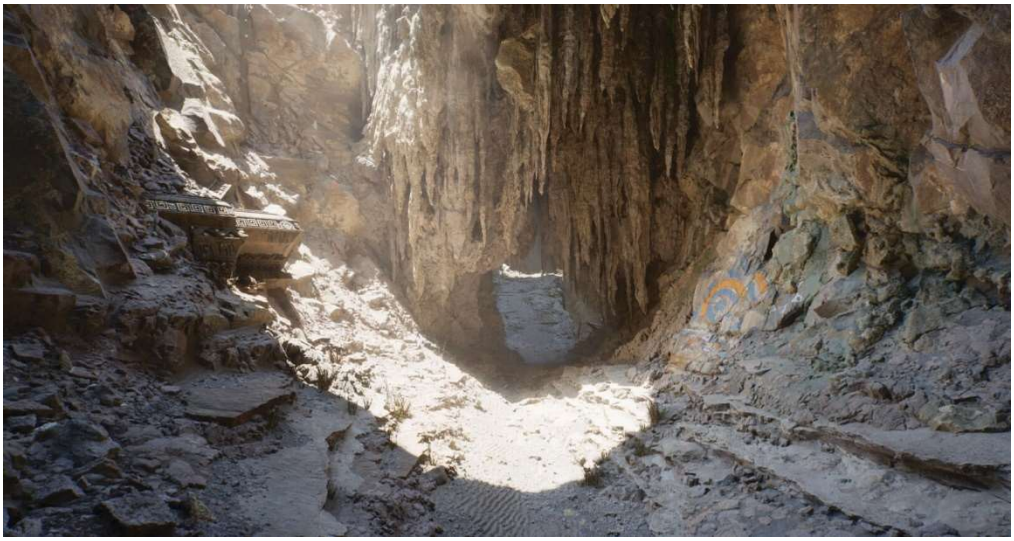

Figura 15. Ejemplo de Lumen
(Fuente Blog de Unreal Engine: https://docs.unrealengine.com/5.1/es-ES/lumen-global-illumination-and-reflections-in-unreal-engine/)

Modos de Juego

En *Unreal* hay una estructura de código llamado Modo de Juego, estos te permiten manejar condiciones de victoria y relaciones entre actores, por ejemplo, manejar la muerte de los actores, sumar puntos, y restar puntos. Este código tiene que estar separado de los actores, pues si hay varios actores, algunos mueren, y otros aparecen. Debe haber una estructura que de forma unificada maneje el estado de la partida y las relaciones entre entidades.

Controladores de personajes

El controlador de personaje sirve como una interfaz entre la persona jugando, y el actor que controla en el juego. A los actores controlables por el jugador se les llama peón. En el caso en el que el peón maneje el *input* del jugador, si el peón muere, por ejemplo, el jugador no tendría manera de introducir *input* al juego, por lo que el controlador de personajes maneja todos los inputs que el jugador pretende hacer, y da las instrucciones al peón o peones de las acciones que deberían

tomar. Este sistema también permite hacer que haya múltiples peones controlables de forma simultánea o alternante.

Conclusión de esta fase

Como toda tecnología, la habilidad que uno adquiere depende de lo que se profundice en la misma, sea a nivel de teoría o práctica. *Unreal Engine* es una herramienta tan grande y con tantas funcionalidades que el tiempo dedicado a este trabajo podría haberse invertido en su totalidad al simple hecho de aprender y profundizar sus detalles técnicos y funcionales. Sin embargo, esta no es la idea, y hay un proyecto que realizar. En definitiva, avanzar a la segunda fase de este trabajo ha sido un salto de fe el cual dependía totalmente de un factor altamente subjetivo: que tan confiado me sintiera de lo que he aprendido hasta ahora para poder proceder a la segunda fase del proyecto. Una vez aprendido lo que se ha explicado previamente, se ha decidido avanzar a la fase de prototipado.

6.2. Preproducción y prototipado

Una vez aprendidos los fundamentos de *Unreal Engine*, se avanza a la segunda fase, prototipado. En esta fase se desarrollan prototipos de mecánicas y sistemas por dos razones fundamentales: probar qué ideas de mecánicas de juego funcionan en el juego de verdad, y hacerse una idea concisa de cómo se implementan de forma limpia estos sistemas.

Si se iteran los prototipos y pulen las mecánicas, más adelante el desarrollo del producto final es mucho más robusto tanto a nivel de diseño como a nivel de estructura e implementación de los sistemas y mecánicas.

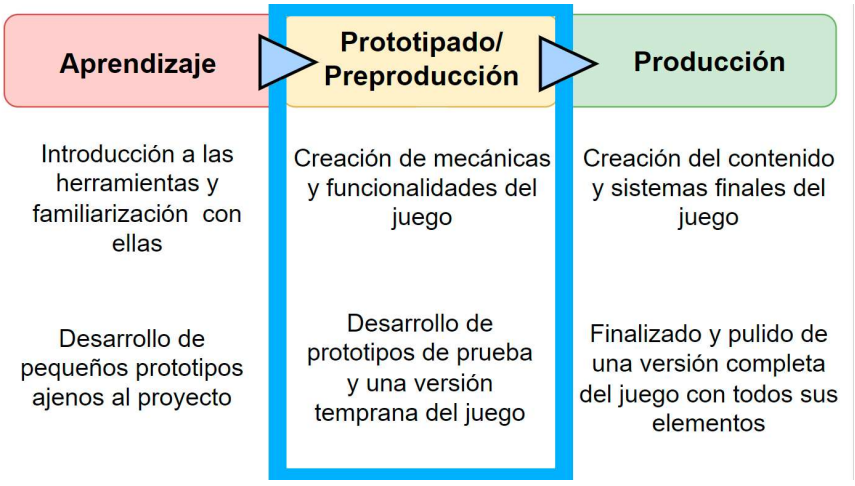

Figura 16. Fase de prototipado en la metodología
(Fuente: Elaboración propia)

Se pretende que al final de este apartado se haya aprendido, probado, y concretado lo suficiente sobre las mecánicas de juego para tener una imagen más realista y concreta de la experiencia, con un *game design* definido. Además, todos los sistemas y mecánicas que estarán presentes en el juego final ya deberán estar implementadas para el final de esta fase.

Se procede a documentar el avance realizado durante esta etapa.

Idea inicial

Antes de desarrollar cualquier prototipo, lo primero es tener ideas de mecánicas para implementar.

La mayor ventaja, y también limitación de los juegos de realidad virtual para productos como el *Meta Quest* 2 es que solo se pueden diseñar mecánicas utilizando el casco y los mandos, que corresponden a la cabeza y las manos. Esto supone una limitación debido a que, por ejemplo, si se le da demasiada libertad de movilidad al jugador, como permitiéndole moverse con el joystick, saltar o hacer cualquier elemento de movilidad brusca, se pueden generar fuertes efectos de mareo y malestar en muchos jugadores. Debido a esto, hay que tener mucha precaución y cuidado con las mecánicas de movilidad que se vayan a implementar. Que el juego sea estático y prácticamente todas las acciones que se hagan dentro del juego sean una réplica directa de los movimientos que realice el jugador en su espacio de juego es altamente preferible.

Con todo esto en mente, la primera idea de juego que se tuvo fue de un juego en el que la mecánica principal se base en el lanzamiento de cosas. El jugador agarrará objetos con sus manos y tendrá que apuntar y lanzar, esto sería atómicamente la mecánica principal del juego sobre la que todo lo demás será construido. Todos los otros elementos de jugabilidad serán adiciones secundarias que elevarán la mecánica y le darán juego a la misma.

Ahora, un juego de solo lanzar cosas sin mas no es para nada divertido, por lo que se pretende darle un giro divertido y único a esta mecánica. Una idea es poder atraer objetos de vuelta a la mano del jugador, permitiendo que lanzando algo, este objeto se le devuelva al jugador como un búmeran, pero la idea es que se puede lanzar lo que sea como un búmeran, latas, espadas, rocas, o cualquier otro elemento. Esta mecánica tendría varios usos, desde vencer enemigos o cortar objetos a distancia lanzando tu espada y que se devuelva, como para resolver puzles, por ejemplo, lanzando un palo al fuego para encenderlo y que cuando vuelva sirva como antorcha. Sin embargo, esta mecánica es la que va a estar más sujeta a cambio, incluso puede ser desechada totalmente si se prueba y los resultados no son los deseados. Este concepto es el que más se beneficiará de los prototipos, ya que estos nos dirán si la idea funciona, es divertida, y que cosas cambiarle y ajustarle para sacarle el máximo provecho antes de crear el documento de diseño y producir el juego.

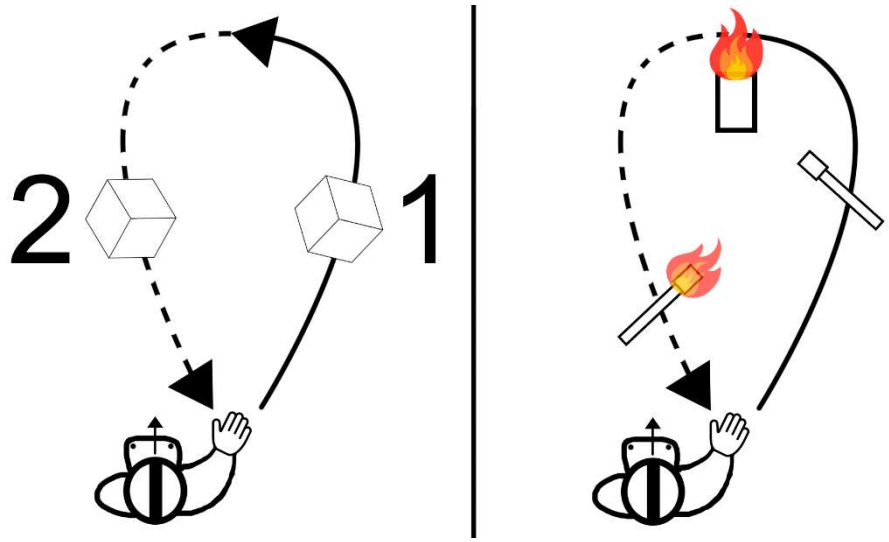

Figura 17. Diagrama de concepto de mecánicas
(Fuente: Elaboración propia)

Plantilla VR de Unreal Engine

Aquí se comienza a prototipar. Afortunadamente, *Unreal Engine* nos ahorra mucho trabajo técnico, ya que incluye una plantilla de realidad virtual que correctamente detecta el *Meta Quest* 2 y los mandos. Incluye unas mecánicas simples de movimiento por teletransportación, capacidad de agarrar objetos con las manos, y unas pistolas de juguete que puedes usar para disparar pelotas.

No obstante, tras probar la plantilla, inmediatamente se hace evidente que estas mecánicas son muy carentes y están ahí de forma demostrativa. Esto es una plantilla después de todo. Por lo que hay que ponerse manos a la obra, y prototipar nuestra mecánica principal, el lanzamiento.

Mecánicas de Lanzamiento: Punto de partida

El realismo de este lanzamiento no es tan importante, siendo la prioridad principal el *game feel,* es decir, que se sienta bien utilizar la mecánica, y sea gratificante y entretenido para el jugador utilizar dicha mecánica. Una vez esta mecánica este completa, se va a proceder a prototipar la mecánica del búmeran, usando el lanzamiento como base.

Antes del prototipado de esta mecánica, por redundante u obvio que parezca, se va a dar una breve definición de lo que un lanzamiento en el juego debe lograr, para poder entender mejor los obstáculos y problemas que se han encontrado.

La mecánica de lanzamiento debe lograr una sencilla tarea, que el jugador agarre un objeto, y al hacer el movimiento de lanzamiento y soltar el objeto, este objeto será disparado por el aire con una velocidad lineal y una velocidad angular. La velocidad lineal nos dice hacia dónde y con que velocidad está moviéndose por el aire, y la velocidad angular como rota y a que dirección. En resumidas cuentas, para nuestro juego, tenemos que hacer que en base al movimiento que haga el

jugador, la velocidad lineal y velocidad angular de lanzamiento del objeto sean lo más adecuadas posibles, y permitan lanzar objetos de forma cómoda, precisa y creíble.

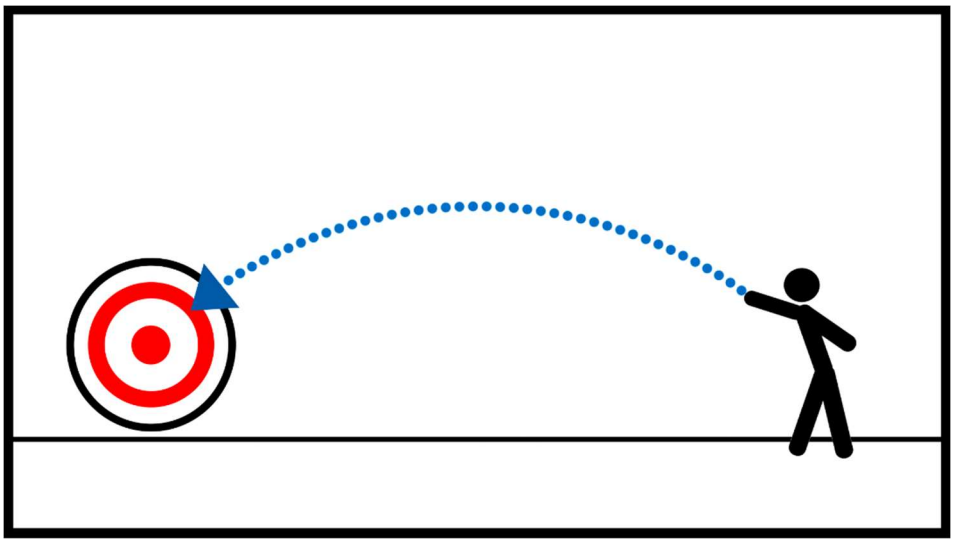

Figura 18. Representación de lanzamiento ideal
(Fuente: Elaboración propia)

Dicho esto, hablemos de cómo funciona esto en la plantilla de VR de *Unreal*. En la plantilla de *Unreal* es posible lanzar objetos, dado que ya hay una mecánica de agarrar y soltar objetos, sin embargo, esta mecánica se nota que funciona de forma inadecuada, siendo imposible apuntar y lanzar con una fuerza deseada por el jugador. A veces al lanzar el objeto simplemente cae al suelo sin velocidad, otras veces sale volando rápidamente hacia abajo o hacia atrás de forma impredecible, otras veces se lanza hacia adelante, pero la velocidad es muy baja y la precisión es muy deficiente. Es una mecánica sumamente incómoda y frustrante, como se aprecia en el diagrama a continuación. Se procederán a mostrar diagramas ilustrativos que representan en grandes rasgos cómo funcionaba el lanzamiento en cada momento.

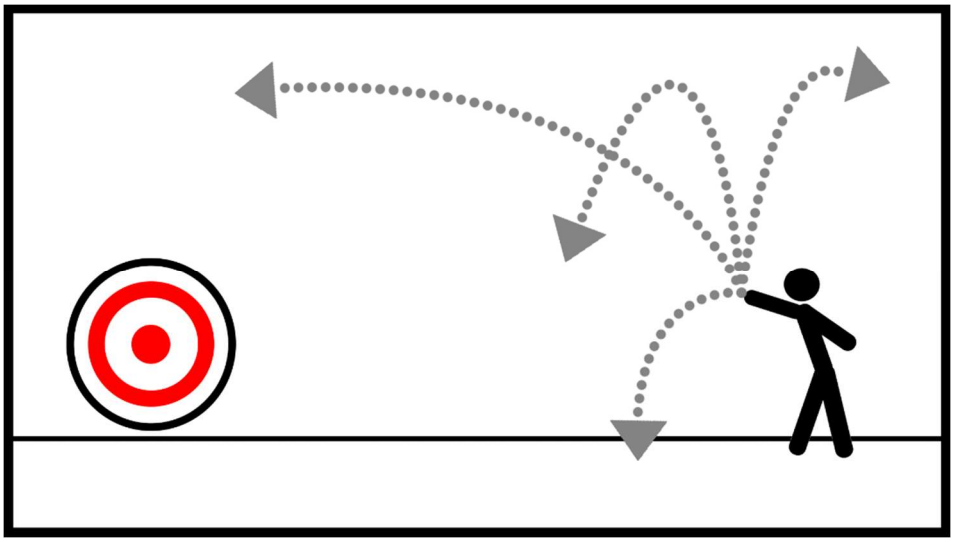

Figura 19. Representación de lanzamiento de la plantilla
(Fuente: Elaboración propia)

Este indebido funcionamiento se debe a por cómo está implementado el sistema de agarrar y soltar objetos en la plantilla, tal como se puede apreciar en el código de *Blueprints* del juego. Lo que hace el juego cuando agarras algo es quitarle sus propiedades físicas y colisiones para enlazarlo a la mano del jugador y este lo pueda mover libremente, y cuando lo suelta, se desenlaza de la mano del jugador y se le devuelven sus propiedades físicas, asignándole la velocidad lineal y angular que el objeto tenía en el último *frame* antes de soltarlo. Intuitivamente uno pensaría que no hay problema con esto, ya que el lanzamiento en la vida real funciona así, el objeto se lanza con la velocidad que tenía antes de soltarlo, pero lamentablemente, en el juego esto no se refleja de esa manera por dos razones principales.

La primera razón es que el último instante de un lanzamiento ignora todo el movimiento previo al lanzamiento, que en la vida real es la que le da su velocidad real al objeto que se lanza. La segunda razón es el pequeño margen de error humano al lanzar, si en el último instante moviste la mano ligeramente para atrás en forma de látigo, el objeto volará a toda velocidad en esa dirección, o si el movimiento arqueado del lanzamiento acaba con una inclinación hacia abajo, el objeto irá directamente hacia abajo. Procederemos a intentar corregir estos problemas.

Mecánicas de Lanzamiento: Velocidad lineal

Se ha comenzado con intentar asignar una velocidad lineal adecuada, que corresponda a todo el movimiento previo al lanzamiento y nos ahorre problemas de movimientos a direcciones erráticas del lanzamiento. Antes de continuar, se eliminó toda velocidad e inercia previa al lanzamiento para

que todas las velocidades que tenga este al ser lanzado se las asignemos nosotros manualmente desde *Blueprints*.

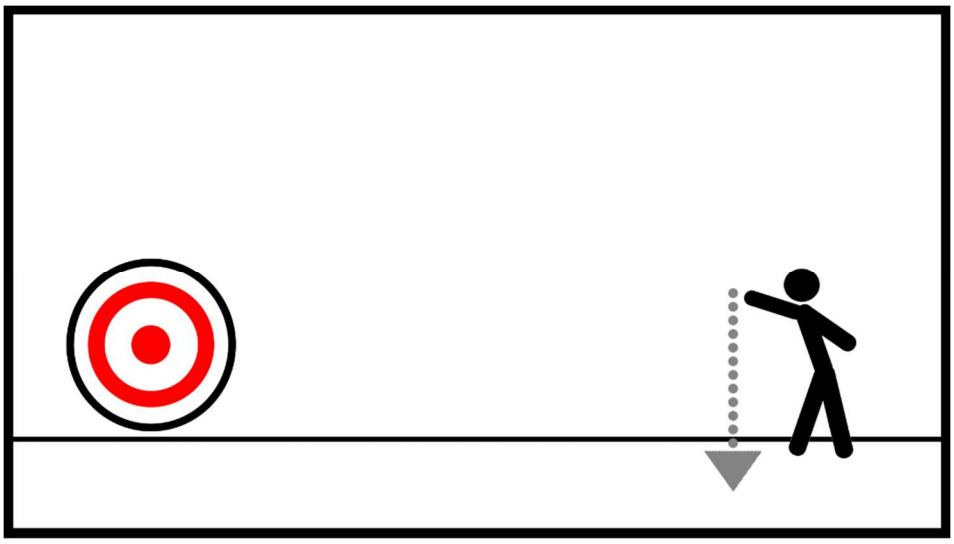

Figura 20. Representación de lanzamiento sin velocidad
(Fuente: Elaboración propia)

Como método se intentaron un par de conceptos. En vez de utilizar la velocidad del último *frame*, se utilizaría la velocidad de un numero definido de *frames* antes del último. Otro concepto fue almacenar un vector que vaya a ser la velocidad de lanzamiento y en cada *frame* hacer una mezcla, o *blend* entre este vector y el vector de la velocidad lineal del objeto, para que el vector de velocidad de lanzamiento fuera un acumulado de muchas velocidades anteriores. Ninguno de estos fue el definitivo, pues tras ensayo y error, y algo de investigación, se decidió utilizar el método que el *game developer* Karl Lewis explica en un post de su blog de desarrollo de su juego *Snowball Showdown*.

Este método consiste en hacer una lista de las 5 últimas velocidades lineales del objeto, y que la velocidad de lanzamiento sea la velocidad promedio entre estas, corrigiendo errores humanos y tomando en cuenta el movimiento previo al lanzamiento (Lewis, 2019).

Se ha implementado este sistema con un par de cambios, de forma que, en vez de utilizar la velocidad del mando, se utiliza la velocidad del objeto, para que se replique de forma realista ese pequeño efecto palanca que hay entre el centro de la palma y el centro del objeto que se está lanzando. Se ha notado una mejoría inmediata, haciendo que los lanzamientos pasen de ser 'totalmente erráticos' a ser simplemente imprecisos. Aún quedan muchas cosas por hacer, pero definitivamente este es un buen comienzo.

El siguiente elemento que hace falta en un lanzamiento es la palanca de la muñeca que se hace al rotar la mano justo al final del movimiento de lanzamiento. Aunque sutil, esta palanca juega un rol esencial a la hora de mejorar la precisión y fuerza de un lanzamiento, y sin ella, el lanzamiento se sigue sintiendo difícil de controlar.

Mecánicas de Lanzamiento: Palanca de la muñeca y velocidad angular

Para suerte nuestra, Karl Lewis también habló de como implementó este efecto palanca. Calculando el producto vectorial entre el vector entre el centro de la mano y el centro de masa del objeto, y el vector de la velocidad angular de la mano, se obtiene un vector perpendicular que nos da esa velocidad añadida por el efecto palanca. Se suma esta velocidad a la velocidad lineal y notamos los resultados, ya hay efecto palanca. Además, asignamos al objeto la velocidad angular calculada para que este gire con el lanzamiento (Lewis, 2019).

Con estas dos técnicas implementadas, los lanzamientos tenían unos comportamientos similares a los que se retratan a continuación.

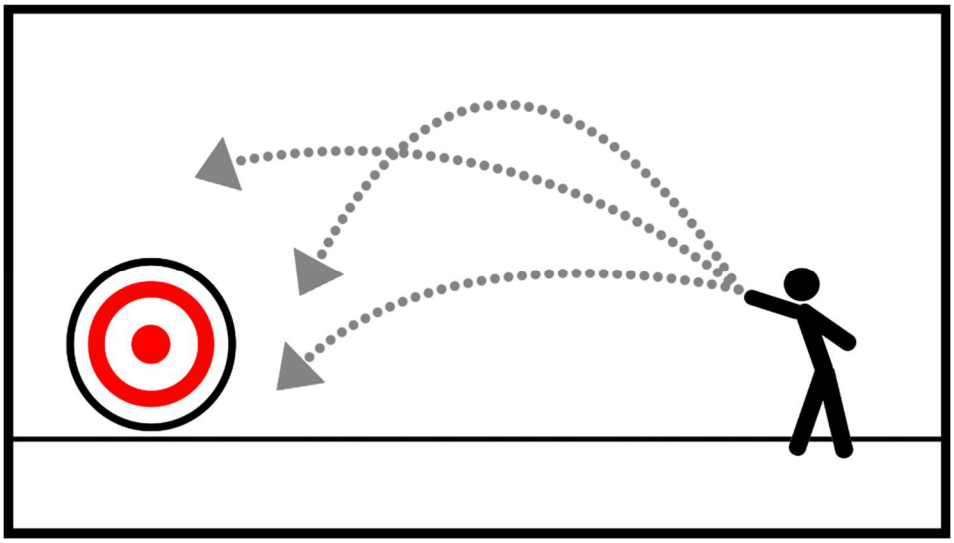

Figura 21. Representación de lanzamiento con velocidad lineal arreglada
(Fuente: Elaboración propia)

Ahora es mucho más sencillo y controlable lanzar cosas. Sin embargo, probando este lanzamiento se notaba un problema crucial. Se usaba también la velocidad angular del último *frame*, dando problemas similares a los de usar el último *frame* de la velocidad lineal. Se ha resuelto creando otra lista de vectores, pero esta vez para calcular el promedio de la velocidad angular del objeto. Ahora funcionaba mucho mejor el lanzamiento de objetos y ya tenía una sensación más utilizable.

Mecánicas de Lanzamiento: Punto de lanzamiento y objetos alargados

Probando las mejoradas mecánicas de lanzamiento, se notaba un problema específico a la hora de lanzar objetos pequeños (alrededor de 20cm) en comparación a lanzar objetos alargados (más de 20cm). Cuando se lanzan objetos alargados, estos rotan y calculan su palanca basada en la distancia entre el centro de masa del objeto y la mano del jugador, lo cual tiene sentido y gracia a la hora de lanzar estos objetos alargados semejantes a espadas. Sin embargo, con objetos pequeños o no alargados había un problema fundamental, usar el vector de distancia entre la mano y el objeto hacía que la precisión de la palanca cambiara drásticamente a si estas agarrando un objeto un poco más hacia la izquierda, un poco más hacia la derecha, y todo esto dependía de por donde agarraras el objeto.

Cabe destacar, que en la vida real cuando agarras algo, lo balanceas con los dedos y lo posicionas en de un modo natural, esto sumado a la sensación de peso ofrece una noción muy buena sobre el centro de masa del objeto y como tendrías que lanzarlo para que tenga la trayectoria que deseas. Sin embargo, en el juego esto no es así, cuando agarras el objeto, este se queda en la posición relativa a la mano del instante en el que se agarró, haciendo que la línea entre el centro de la mano y el centro de masa estén considerablemente más desalineados en el mundo virtual. Con los objetos alargados este problema no es relevante, pues, al ser alargado, dependiendo de cómo agarres el objeto, hay más margen de error en como lo puedes agarrar sin que esté muy desalineado, y es más fácil tener noción de como lanzar las cosas.

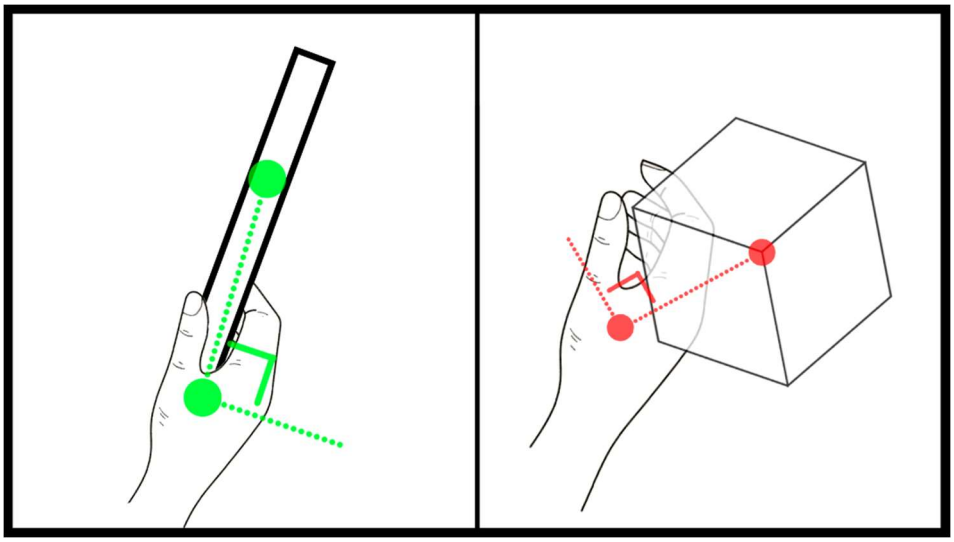

Figura 22. Representación de distintos alineamientos entre la mano y el objeto agarrado (Fuente: Elaboración propia)

Dado que en la vida real uno centra en su mano el objeto a lanzar, y en el juego no queda bien hacer esto, se decidió hacer un pequeño truco para darle la ventaja al jugador.

Creamos algo que llamamos 'Punto de lanzamiento'. El punto de lanzamiento denominamos a un punto imaginario encima de la palma de la mano sobre el dedo medio, que proyectado hacia la palma da una línea recta, y es el que rota de forma más perpendicular cuando uno gira la muñeca. La línea entre este punto y el centro de la mano es la línea que recorre un objeto mientras es lanzado por la mano, si hacemos que los objetos pequeños calculen su palanca en base a este punto de lanzamiento, lanzar objetos pequeños se sentirá y tendrá una precisión exactamente igual los agarres por donde los agarres, haciendo que el jugador no tenga que detenerse a pensar por donde está agarrando este objeto pequeño antes de lanzarlo. Este punto de lanzamiento sería usado solo si la distancia entre el centro del objeto y la mano es menor a un umbral, permitiendo lanzar objetos alargados de la misma manera que antes.

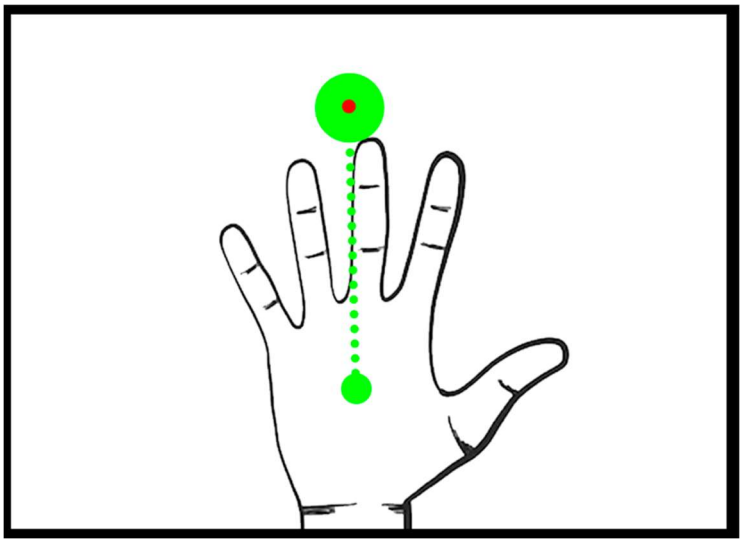

Figura 23. Representación de la localización del 'punto de lanzamiento'
(Fuente: Elaboración propia)

Una vez implementado este cambio, lanzar objetos pequeños se ha vuelto mucho más preciso y cómodo. Además, se ajustó la rotación de los objetos alargados cuando son lanzados para dar un efecto más realista.

Mecánicas de Lanzamiento: Pulido y últimos toques

Finalmente, aunque el lanzamiento funcionaba casi perfecto, tenía un par de ligeros problemas de precisión. Todo esto se puliría probando las mecánicas y ajustando números de variables que calculan el lanzamiento a base de ensayo y error.

Un pequeño cambio que tuvo un efecto inmenso fue mover el punto de lanzamiento. En vez de que esté justo encima del dedo medio de la mano, se movió medio centímetro hacia la dirección de la palma, haciendo que este punto se asemeje más el último punto de contacto entre la mano y el objeto cuando se lanza algo tomando en cuenta el grosor del dedo. Esto hizo que la precisión incrementara de forma importante y fuera mucho más sencillo apuntar.

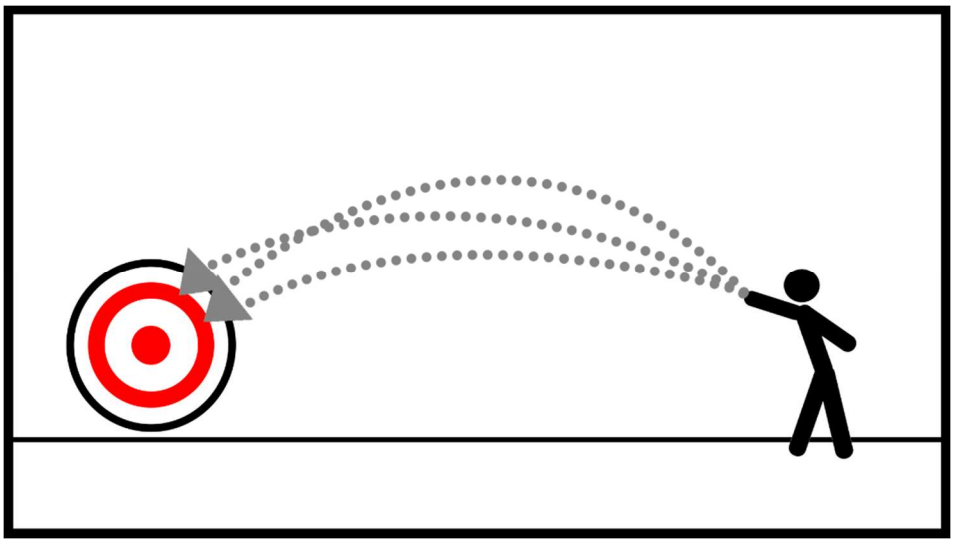

Figura 24. Representación del lanzamiento con efecto palanca ajustado
(Fuente: Elaboración propia)

Como se aprecia en el diagrama anterior, los lanzamientos ahora son mucho más intuitivos, y el margen de error corresponde más a el error humano que naturalmente puede haber en cualquier lanzamiento.

También se ajustó el tamaño de las listas de velocidades lineales y angulares, para que el lanzamiento se sintiera bien en distintas maneras de lanzar un objeto (utilizando todo el brazo, usando solo la muñeca, haciendo movimientos cortos y rápidos, o lanzamientos con mucho movimiento).

Asistencia de puntería: Elección de objetivo

A pesar de que el lanzamiento ya tenga un *game feel* adecuado, para que jugar el juego sea verdaderamente cómodo y satisfactorio, se procede a implementar un sistema de asistencia de puntería. Este sistema tendrá la responsabilidad de ajustar la velocidad inicial de lanzamiento para que el jugador tenga más facilidad a la hora de atinarle a objetivos a distancia con el objeto que esté lanzando.

Para comenzar, se ha editado el mapa donde se ha estado testeando hasta ahora, que era el mapa por defecto de la plantilla de VR de *Unreal Engine*. Se ha hecho mucho más amplio, y se han

colocado diversos modelos *placeholder* alrededor del mapa que servirán como objetivos, con los cuales probaremos la asistencia de puntería.

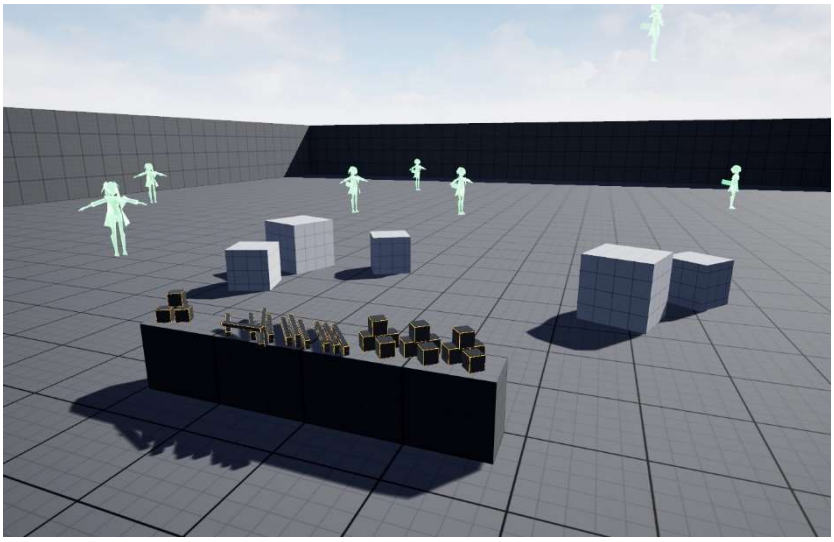

Figura 25. Screenshot del escenario de prueba
(Fuente: Elaboración propia)

Se ha utilizado una función de *Blueprints* llamada '*Predict Projectile Path'*, que, en base a la velocidad y posición inicial de un proyectil, te devuelve una serie de puntos que representan la trayectoria que llevaría dicho proyectil hasta colisionar. Se ha hecho que en instante que se lance un objeto, se llame a esta función y se pueda visualizar el camino que recorrerá el proyectil.

Lo primero es lo más simple, se hace que, si se predice que el proyectil colisionará con el objetivo, no se va a hacer ninguna corrección de trayectoria. No hay que corregir un lanzamiento que no lo necesita.

Antes de definir como corregir un lanzamiento, la primera cuestión que hay que resolver es la siguiente: ¿Cómo se decide cual es el objetivo al que se debe asistir el lanzamiento? Una vez sepamos cual es el objetivo al cual el jugador estaba apuntando, ya podemos corregir el lanzamiento. Dado que un lanzamiento es una curva, y queremos que no sea difícil apuntar a objetos tanto lejanos como cercanos, se ha decidido utilizar como punto de partida dos radios para los objetivos.

El primer radio, que llamaremos radio de colisión, definiría la esfera por la cual, si el lanzamiento pasa por ella, se considera que le ha dado al objetivo. Objetos más grandes tendrán radios mayores, permitiendo que la corrección de lanzamientos no vaya siempre al mismo punto.

El segundo radio, que llamaremos radio de detección, sería una esfera con un radio mucho mayor, que sería la guía para escoger el objetivo. Si el lanzamiento pasa por esta esfera más grande, se tomaría su respectivo objetivo como el objetivo de lanzamiento al cual el jugador tenía la intención de apuntar.

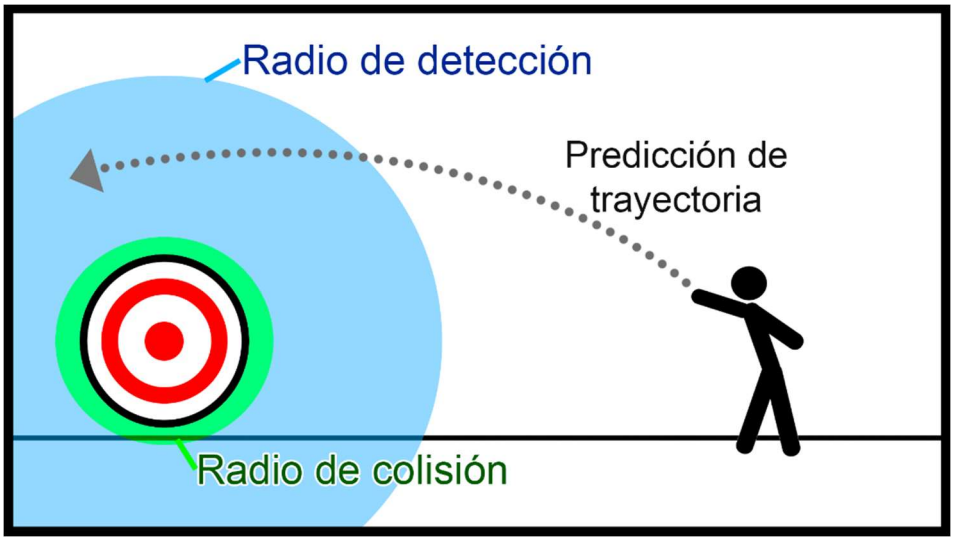

Figura 26. Diagrama de radios para la asistencia de puntería
(Fuente: Elaboración propia)

Para tener *feedback* de cuál era el objetivo que está escogiendo el juego, se ha hecho que cuando este sea seleccionado, este cambie de color por un par de segundos. Cabe destacar que aún no se implementa ninguna corrección de trayectoria, solo se observa que objetos cambian de color al lanzar un proyectil. El objetivo que cambie de color es el que el algoritmo está escogiendo como objetivo para la corrección.

Tras probar este sistema, se notó un problema, ya que cuando un objeto está muy lejos, es más difícil apuntarle a su radio de detección, y hacer que el juego sepa que ese es tu objetivo. Se ha solucionado de forma sencilla, en cada instante del juego, el radio de detección de un objetivo cambia, se hace más grande si el jugador está lejos y se hace más pequeño si el jugador está cerca. Esto hace que, desde la primera persona del jugador, todos los radios de detección tengan una proporción similar, haciendo que apuntar a un objeto lejano sea igual de fácil que a un objeto cercano, además, esto hace imposible que entres dentro de un radio de detección, ya que eso haría que todos los lanzamientos sean dirigidos a un solo objetivos.

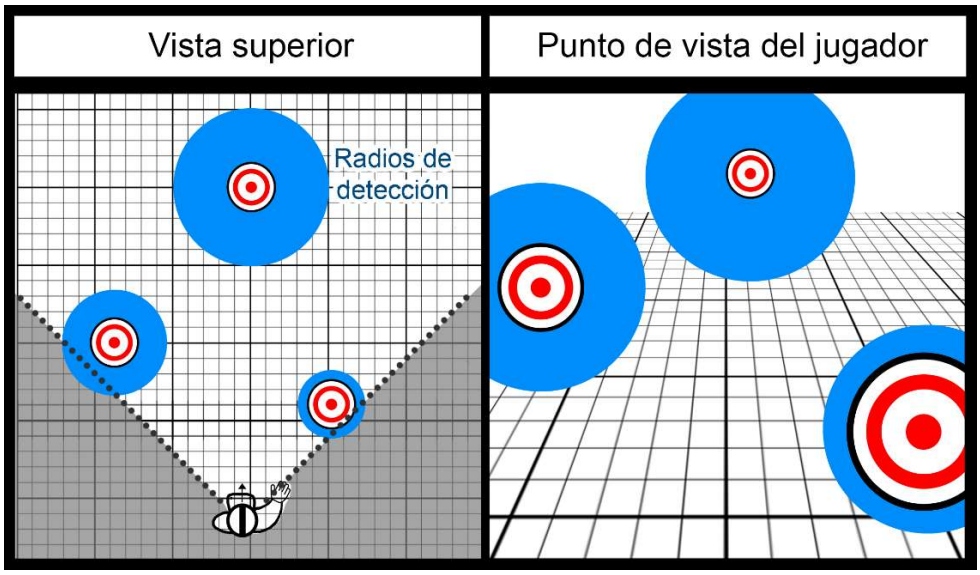

Figura 27. Representación de las proporciones relativas de los radios de detección (Fuente: Elaboración propia)

En el diagrama anterior se puede apreciar que los radios de detección más lejanos al jugador son más grandes. Esto se hace para que desde el punto de vista del jugador, todos tengan el mismo tamaño relativo, por lo que apuntar a algo lejano va a tener la misma dificultad que apuntar a algo cercano.

Sin embargo, salió a flote otro problema. Muchas veces la trayectoria de un proyectil pasaba por más de un solo radio de detección. ¿Cuál de los dos objetivos debería utilizarse? La solución ha sido sencilla. Se escoge el objetivo el cual la trayectoria del proyectil pase más cerca de su centro.

Con esto finalizado, tras probar y ajustar valores, se ha conseguido que se sienta acertada la predicción del juego al indicar el objetivo al que se está apuntando.

Asistencia de puntería: Corrección de trayectoria

Ya sabemos cuál es el objetivo al que estamos apuntando, ahora solo hay que desviar el lanzamiento inicial a una trayectoria nueva que asista este lanzamiento. Esto se logra con una función de *Blueprints* llamada '*Suggest Projectile Velocity*', que, recibiendo un punto inicial, un punto final, y una velocidad inicial, te calcula la velocidad de lanzamiento que debe dársele a un proyectil para que llegue a su destino.

Para lograr que se sienta genuina esta corrección y no sea demasiado obvio, se han tomado dos decisiones fundamentales. Primero, se va a utilizar siempre la velocidad inicial de lanzamiento que el jugador ya le haya estado proporcionando al proyectil antes de la corrección. Y segundo, el punto final de la trayectoria de corrección va a ser la posición de corte de la trayectoria inicial sobre el

radio de detección, proyectado al radio de colisión. Esto quiere decir, que, si la trayectoria original se cruza con el radio de detección por la parte superior de este, el punto al cual se hará la corrección de lanzamiento será la parte superior del radio de colisión. De la misma manera, si se cruzó el radio de detección cerca del centro, se lanzará el objeto hacia el centro del radio de colisión.

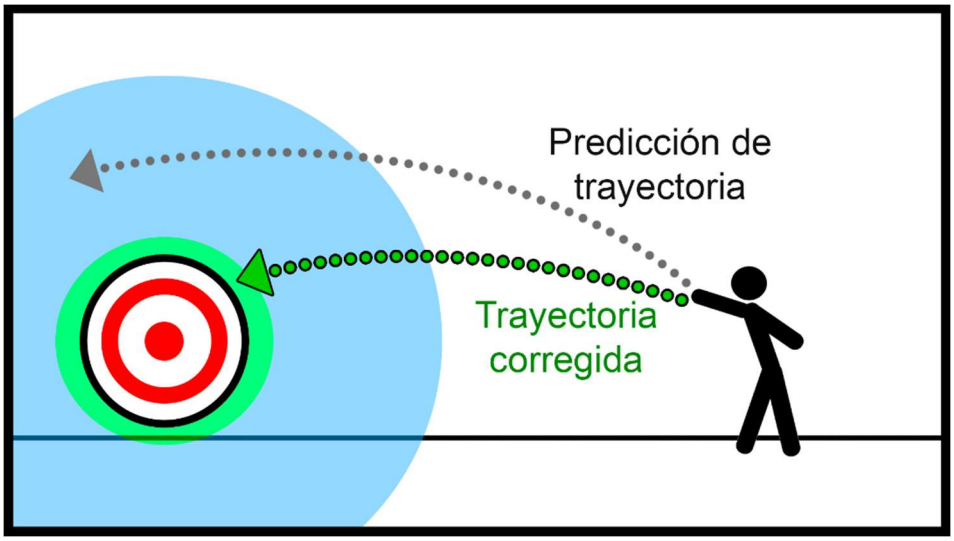

Figura 28. Diagrama de corrección de trayectoria
(Fuente: Elaboración propia)

Ya tenemos el punto final al cual vamos a apuntar, ahora simplemente utilizamos la función '*Suggest Projectile Velocity*' para saber que velocidad inicial darle al proyectil una vez es lanzado, y aplicársela. Cabe destacar también que se ha implementado esto de forma que, si la velocidad inicial de lanzamiento no es suficiente para atinarle al objetivo, no se hace ninguna asistencia.

Con esto implementado, los lanzamientos asistidos se sienten auténticos y satisfactorios de utilizar. Tras probarlo, es fácil acertarle a el objetivo que se desee sin que se sienta muy evidente la asistencia que el juego hace por ti.

Efecto imán

Con el lanzamiento funcionando de forma satisfactoria y como se pretende, ahora sigue programar la mecánica principal de nuestro videojuego, el verdadero corazón del proyecto.

La intención del 'Efecto imán' que se le quiere dar a las mecánicas principales, es que el jugador al agarrar un objeto con la mano, este se 'enlace' a dicha mano, haciendo que cuando lo lance, el jugador pueda devolver el objeto a su mano a voluntad, semejante al martillo de Thor en las películas de Marvel.

Para lograr esto, primero se ha preparado el sistema de 'enlazado', en el que, al agarrar un objeto, la mano y el objeto guardan referencias del uno al otro, y pueden comunicarse. Al agarrar otro objeto las referencias se sustituyen. El jugador también tiene la posibilidad de 'desenlazar' un objeto que no esté agarrando dándole al botón A de la mano correspondiente. Aunque se lance el objeto, mientras la mano no haya 'desenlazado' por ninguna de las razones previamente mencionadas, estos seguirán enlazados, llevando al siguiente paso.

Al lanzar el objeto, se chequea si el jugador vuelve a presionar el botón de agarre. Si no hay ningún objeto en su rango de agarre, y hay un objeto enlazado a su mano, se le envía un mensaje al objeto para que se devuelva a la mano, siempre que el jugador siga presionando el botón de agarrado. Esto se aprecia en el siguiente *Blueprint*, en el que si al intentar agarrar, no encuentra nada que pueda agarrar, atrae el '*Bound Component*', o componente enlazado, en el caso que exista.

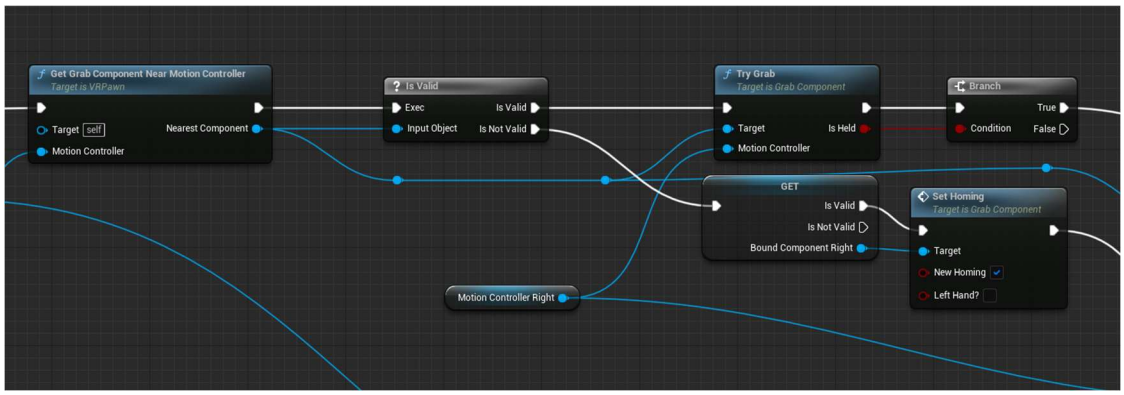

Figura 29. Blueprints de decisión entre agarrar o atraer
(Fuente: Elaboración propia)

Para las físicas del objeto volviendo, se ha hecho que el objeto en cuestión esté en estado '*isHoming*' que se traduce a 'está volviendo'. Cuando el objeto está en este estado se desactiva su gravedad, para que las fuerzas que se le apliquen pasen a ser las que nosotros definamos en su comportamiento.

Primero se ha intentado asignar una fuerza en el objeto que vaya hacia la mano. Este método se sentía natural y daba la sensación de imán que se quería obtener. Sin embargo, esto daba un problema fundamental, ya que, al asignar esta fuerza, el objeto mantenía su velocidad que tenía antes de que se aplicara esta fuerza, haciendo que en la mayoría de los casos en vez de que el objeto volara hacia la mano como un búmeran, hiciera un efecto de orbita, como se muestra en la gráfica a continuación.

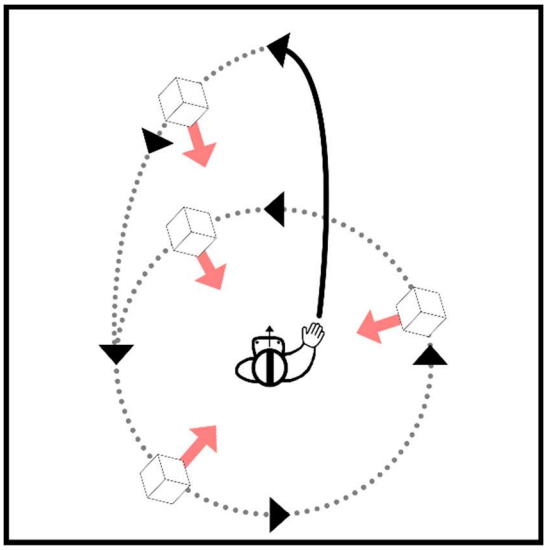

Figura 30. Representación del efecto órbita
(Fuente: Elaboración propia)

Esto ocurre porque la fuerza centrífuga de la alta velocidad del objeto terminaba balanceándose con la fuerza centrípeta del efecto imán, que busca acercar el objeto al jugador. En la gráfica representada, las flechas negras indican la dirección del objeto al ser atraído, y las flechas rojas la fuerza de atracción hacia el jugador.

Una manera de solucionar esto es hacer que la fuerza de atracción a la mano tenga mayor magnitud. No obstante, esto haría que los objetos obtuvieran velocidades muy elevadas, dando problemas con las físicas y jugabilidad, además de no sentirse bien. Y, por si fuera poco, el problema aún ocurría en algunos casos.

A continuación, se intentó asignar una velocidad que fuera una interpolación entre la velocidad actual del objeto en vuelo y la velocidad deseada en dirección hacia la mano. Esto funcionaba mejor a la hora de atraer el objeto a la mano, pero al ser un método poco natural que ignoraba mucho las físicas, se sentía rígido y artificial, sobre todo en el desvío inicial del objeto.

La solución final que se encontró fue una mezcla de ambos métodos. Primero, cuando el jugador empieza a atraer el objeto, se asigna una fuerza hacia la mano, dando ese efecto natural y magnético que se pretendía. Pero para evitar la órbita, y hacer que el objeto vaya recto hacia la mano, una vez el objeto ya ha empezado a devolverse, se calcula que su velocidad esté bajo cierto rango de diferencia a la velocidad deseada, cuando esté en este rango, se deja de aplicar la fuerza y se interpola la velocidad para ajustarla hacia donde se pretende. En otras palabras, si el objeto volador ya está volando a una dirección lo suficiente similar a la deseada, se hace este pequeño ajuste para evitar hacer la órbita, dando una sensación mucho más natural y acertada.

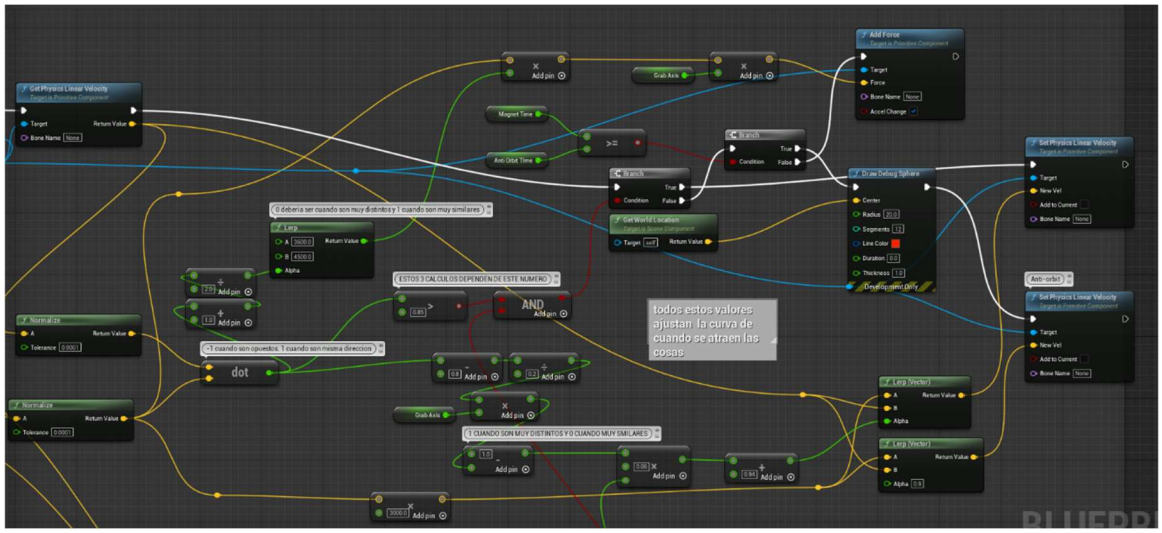

Figura 31. Blueprints de atracción
(Fuente: Elaboración propia)

Una vez implementado, se puede decir que el problema estaba arreglado, pero en algunos casos específicos en el que el objeto fuera en direcciones muy distintas a la deseada, o el jugador bruscamente evitara que el objeto llegara a su mano, este nunca llegaba al punto de corregir la órbita, y acababa orbitando nuevamente. Para solucionar esto se ha usado una variable que contabiliza el tiempo que el objeto tiene intentando llegar a la mano. Si sucede el caso que el objeto tenga más de cierto tiempo en el aire sin lograr llegar a su destino, automáticamente se cambiará al método de atracción con interpolación, con unos ajustes de valores mucho más agresivos. Esto elimina finalmente el efecto órbita.

Atrapar el 'búmeran'

Ahora sigue hacer que el objeto vuelva a ser agarrado una vez vuelve a la mano.

Si se hacía que se chequeara una distancia, es decir, si el objeto está a X unidades de la mano, se agarra de vuelta, era la respuesta intuitiva. Pero, las altas velocidades que alcanzaba este objeto, hacían que este chequeo fuese impreciso y fallara.

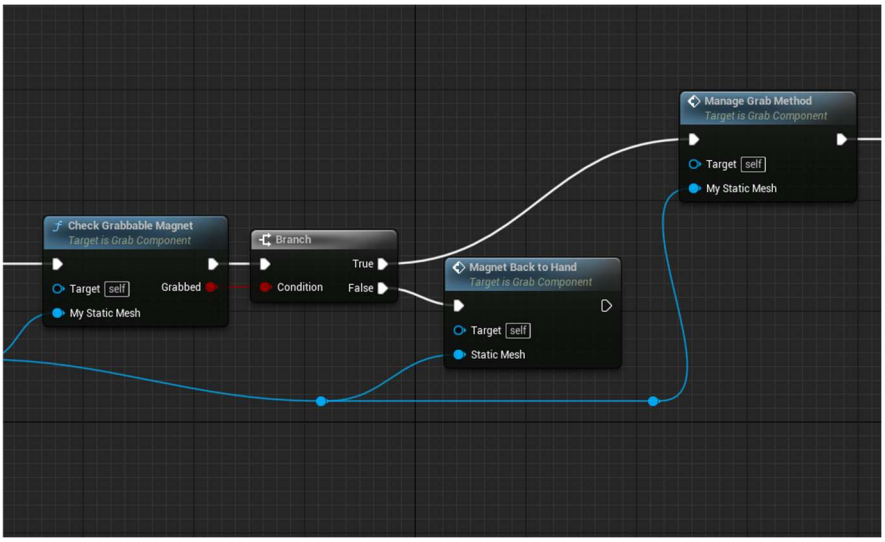

Figura 32. Blueprint de elección entre atraer o atrapar
(Fuente: Elaboración propia)

La solución ha sido simple. Cuando el objeto va más rápido, el rango de chequeo es mayor, para corregir la precisión por la velocidad.

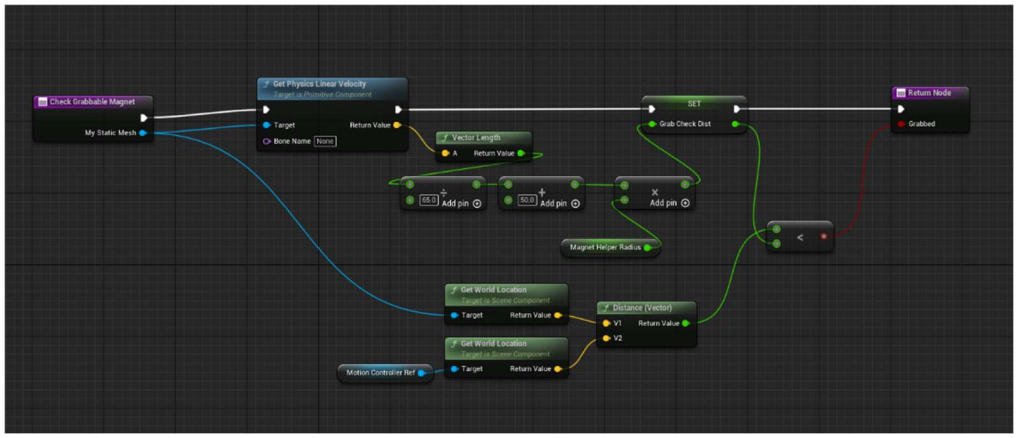

Figura 33. Blueprint de cálculo de elección entre atraer o atrapar
(Fuente: Elaboración propia)

Lanzamiento con curva

Para dar un toque de diversión al juego, se ha intentado prototipar un sistema que dependiendo de cómo lances el objeto, este volará en una curva en el aire hasta el objetivo. Si se lanza el objeto con una rotación vertical, el objeto volará de forma normal. Sin embargo, si se lanza con una rotación en horizontal, este hará un efecto de curva en el aire. Dependiendo de que tanta rotación haya, y de la inclinación que esta tenga, la curva será afectada.

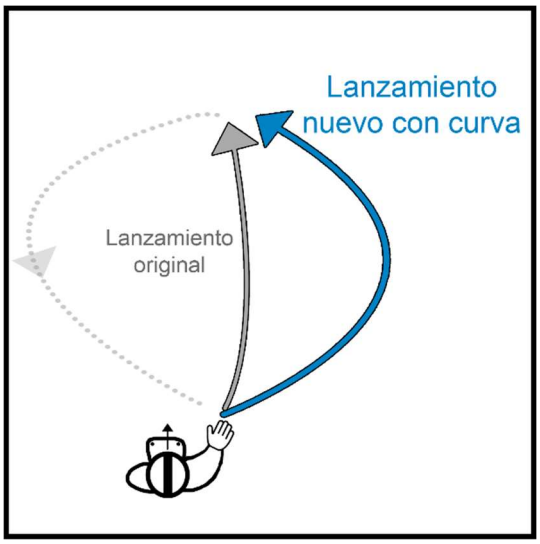

Figura 34. Representación de lanzamiento con curva
(Fuente: Elaboración propia)

La implementación comenzó con calcular "que tan horizontal" es un lanzamiento. Saber esto es tan sencillo como obtener la velocidad de rotación en el eje vertical (En este caso Z) del objeto. Si es 0, es que no está girando horizontalmente, y si es positiva o negativa, está girando horizontalmente a una dirección u otra. Simplemente, a partir de este valor, se desfasa la velocidad inicial de lanzamiento un poco hacia la izquierda o hacia la derecha respectivamente, para que cuando el objeto en vuelo corrija su trayectoria, logremos el efecto de curva.

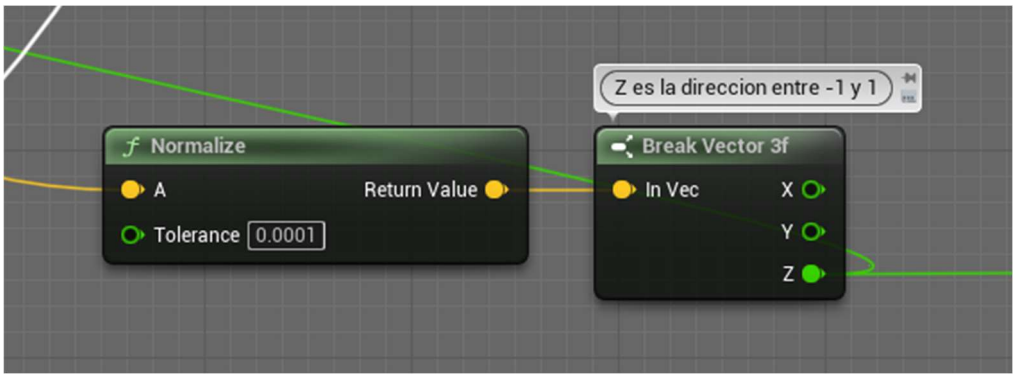

Figura 35. Blueprint de obtención de horizontalidad del lanzamiento
(Fuente: Elaboración propia)

Después se calcula de la siguiente manera el desfase inicial para el lanzamiento.

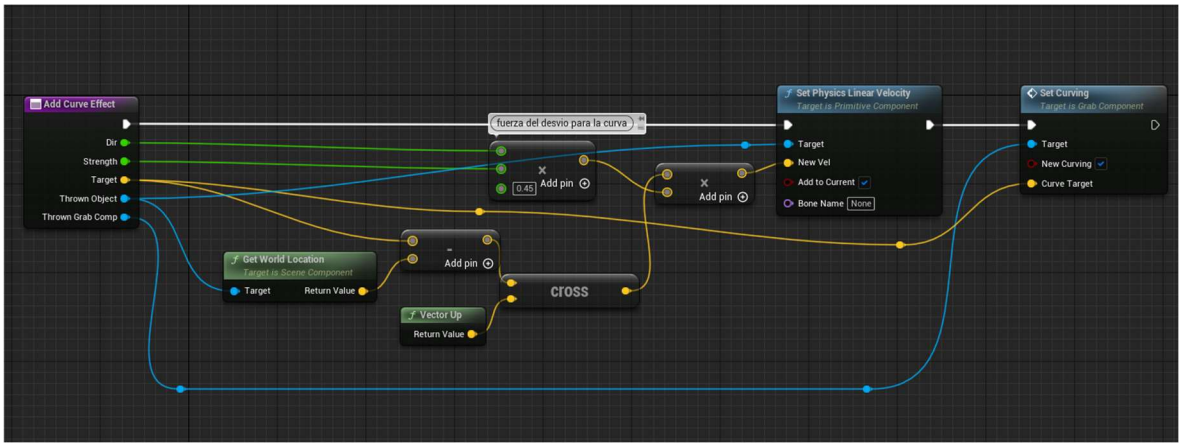

Figura 36. Blueprint de cálculo de desfase inicial de lanzamiento
(Fuente: Elaboración propia)

Una vez se hace este desfase, se le indica al objeto que haga efecto de curva, indicándosele el punto de objetivo hacia el que iba a volar originalmente.

Para corregir la trayectoria, se hace el siguiente cálculo de vectores.

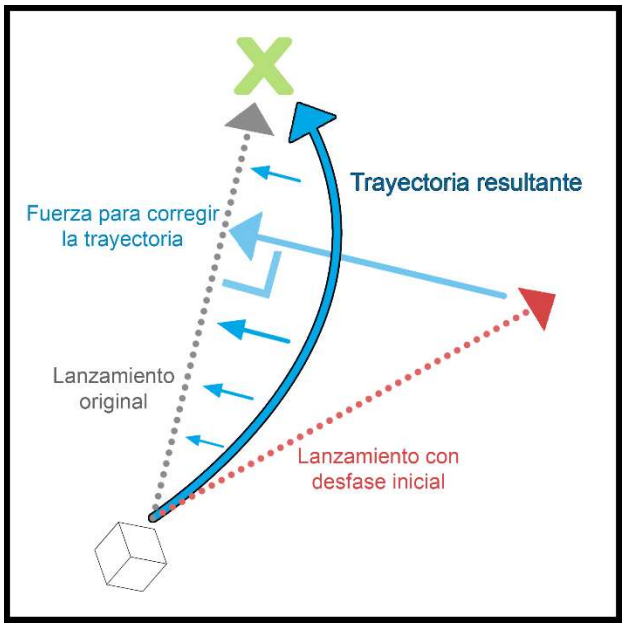

Figura 37. Diagrama del cálculo de corrección de trayectoria
(Fuente: Elaboración propia)

Se calcula la proyección de la velocidad actual sobre el vector distancia entre ambos objetos, luego a este vector se le resta la velocidad actual. Esto resulta en un vector perpendicular a la distancia entre los objetos que corresponde a el desfase horizontal que tiene esta velocidad actual. Este vector es la corrección que queremos aplicarle al efecto que vuela, para que al final del recorrido ya se haya corregido dicho desfase inicial, resultando en el efecto curva.

Se usa este vector para añadirle una fuerza al objeto volador, corrigiendo la trayectoria de forma natural. A continuación, se muestra la implementación.

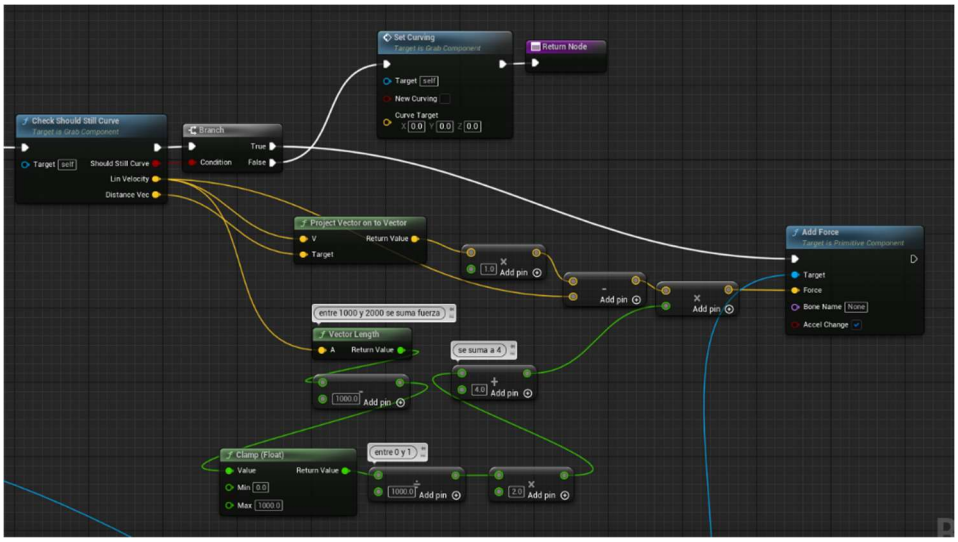

Figura 38. Implementación de la corrección de trayectoria
(Fuente: Elaboración propia)

Movimiento del jugador

Dado que ya el lanzamiento funciona de forma adecuada, se le dan los toques finales al movimiento del jugador.

Hasta ahora se ha usado la teletransportación por defecto que viene con la plantilla de VR. Pero ya que este es un juego de acción, se requiere que las mecánicas tengan una movilidad adecuada.

La idea es asignarle al peón del jugador que herede de la clase *Character*, que le añade 2 elementos fundamentales: Una colisión de capsula, que se va utiliza para las colisiones con el entorno, y el componente de movimiento, que le añade todas las funcionalidades para poder moverse de forma correcta con una enorme variedad de configuraciones. Sin embargo, esto supone un problema inmediatamente.

Los sistemas de realidad virtual utilizan un sistema de coordenadas propio para poder mover al jugador dentro de este ambiente virtual manteniendo un punto de referencia estático, lo que permite al jugador moverse y que su representación dentro de las aplicaciones se mueva acordemente. El problema que genera este sistema para esta implementación en particular, significa que para que el jugador pueda moverse correctamente dentro de *Unreal*, se debe mantener un punto de referencia estático, este punto de referencia siendo la posición del Actor del jugador en *Unreal*, lo que significa que si posicionamos una capsula en el componente raíz del Actor, el jugador con moverse en la vida real hará que la capsula, con sus colisiones, se quede en el punto

de referencia del área de juego, mientras que el jugador se ha movido a otro lugar. Esto en esencia rompe el juego, permitiendo al jugador atravesar paredes, y que las colisiones con el mundo con las que el jugador recibe daño estén separadas del jugador. La solución intuitiva es mover la capsula a la posición relativa del jugador en el área de juego, sin embargo, esto no se puede en *Unreal* dado que la capsula es el componente raíz, y no se puede mover de forma relativa. La capsula siempre se encontrará en el punto de referencia del área de juego, no en el jugador.

Utilizando el tutorial de personajes en VR del *youtuber* Just2Devs como referencia, se ha implementado la técnica para solucionar este problema. Esta técnica consiste en el funcionamiento que se explicará a continuación (Just2Devs, 2021).

En cada *frame* del juego se mide el desfase horizontal entre el punto de referencia del área de juego, después se mueve la capsula a este desfase, posicionando la capsula donde está el jugador. Sin embargo, como el jugador es hijo de la capsula, el jugador también se mueve. Lo que se hace es mover al jugador a que tenga una posición relativa de cero, posicionándolo en la cápsula, que ya está en la posición que debería estar (Just2Devs, 2021).

Se repite este proceso por cada *frame*, y da la ilusión que la capsula sigue al jugador, cuando lo que sucede es que el jugador se está moviendo hacia la capsula en cada *frame*, y a la capsula se le añade el desfase que ha tenido el jugador en relación a la capsula en el *frame* anterior.

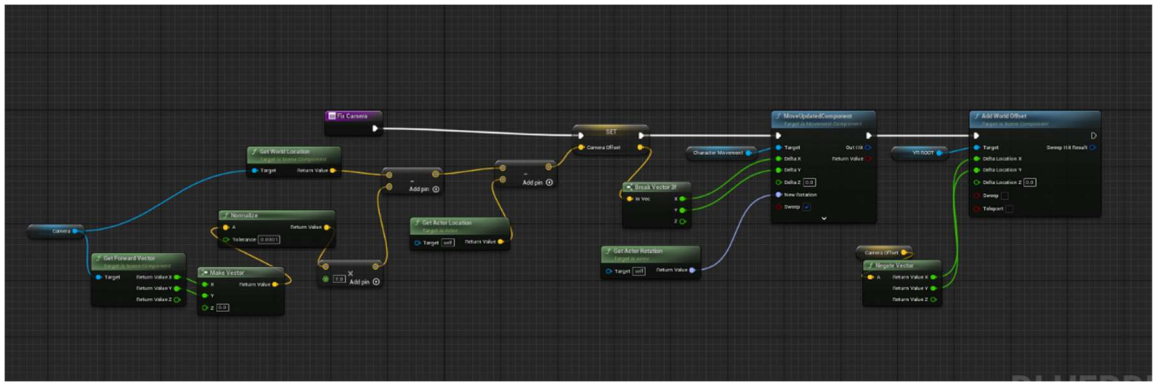

Figura 39. Implementación de movimiento de capsula del jugador
(Fuente: Elaboración propia)

Para que este movimiento de una mejor sensación, en vez de posicionar la capsula exactamente en el jugador, se posiciona un poco atrás de hacia donde el jugador este viendo, para que la capsula con la que el jugador choca con el mundo esté más cerca del cuerpo del jugador real, en vez de estar centrada en la cara.

Adicionalmente, el desfase vertical del jugador en cada *frame* se utiliza para editar la altura de la capsula, haciendo que si el jugador se agacha, la capsula se haga más pequeña y el jugador pueda entrar en áreas más bajas.

Enemigos terrestres – bases

Se procede a la implementación de los enemigos. Primero se hará una base de un enemigo humanoide terrestre, y después la de un enemigo volador. Primero se implementará el enemigo terrestre, que tendrá 2 variaciones, uno que ataca cuerpo a cuerpo y otro que ataca a distancia. El arte conceptual es el que se ve a continuación.

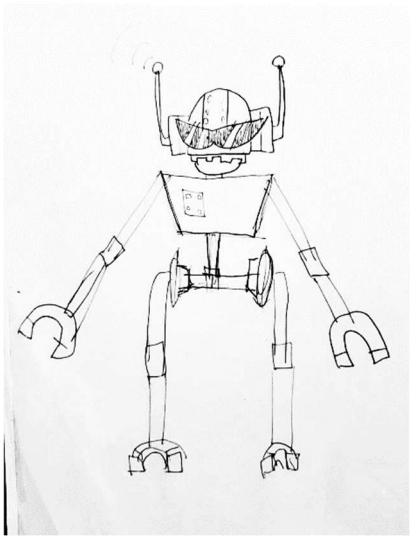

Figura 40. Arte conceptual del enemigo terrestre
(Fuente: Elaboración propia)

Primero se ha creado un peón con *Movement Controller* de *Unreal*, que, para efectos prácticos, es un personaje terrestre que se mueve a través de una capsula de colisión. Se le ha añadido un *Target Component* de los que hemos creado anteriormente para que el enemigo sea considerado un objetivo para cuando lances objetos, y se le aplique la asistencia de puntería. Se le ha creado un controlador de inteligencia artificial, componente que sirve como cerebro del enemigo y contendrá toda la lógica de la inteligencia artificial, un *blackboard*, componente que almacena información sobre la inteligencia artificial, y un *behavior tree* que definirá su comportamiento y las decisiones que tomará. Se crea cada uno de estos componentes en su propia clase, para que cada variación de enemigo tenga componentes que hereden de estas clases y cambien su funcionalidad a una más específica.

Este enemigo tendrá un comportamiento básico. Se le acercará al jugador, y cuando este a un rango adecuado, atacará. Pero por ahora solo tenemos el esqueleto de esta estructura, y falta implementación concreta.

Enemigos terrestres – Percepción y *Pathfinding*

Se le ha aplicado a este enemigo base un sistema de percepción de *Unreal*, el cual permite definirle al enemigo un campo de visión para percibir su entorno. A continuación, se muestra la implementación de lo que pasa una vez se percibe al jugador mediante este sistema automático.

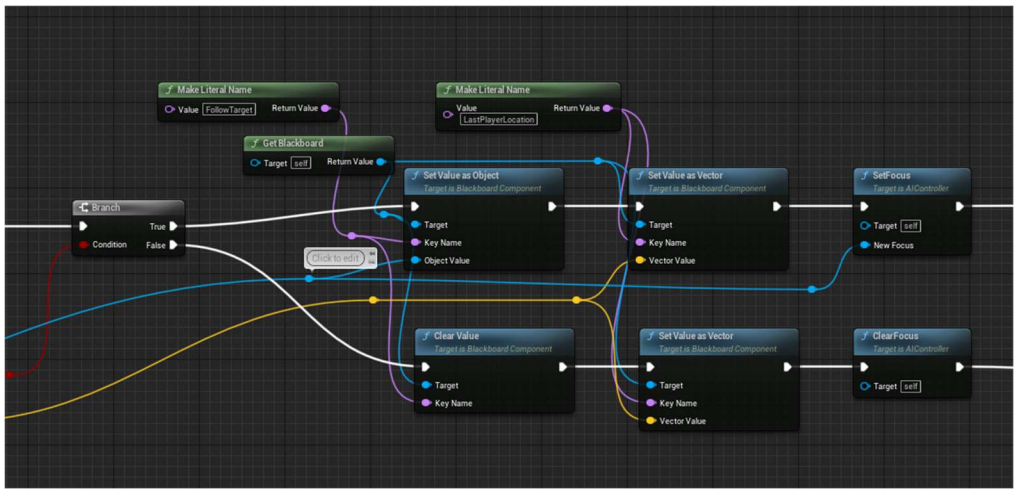

Figura 41. Blueprint de percepción del jugador
(Fuente: Elaboración propia)

Cuando se percibe al jugador, se actualizan los valores del *blackboard* de forma acorde.

Se ha hecho que cuando detecte visualmente al jugador, lo persiga utilizando la función *MoveTo* del controlador de inteligencia artificial, que utiliza *NavMesh* calculados automáticamente en el mapa para seguir al jugador mediante búsqueda de caminos (*Pathfinding*) y esquive obstáculos en el camino.

Si el enemigo deja de percibir al jugador, este se quedará quieto.

Enemigos terrestres - Inteligencia

Con estas bases hechas, se empieza con la implementación más específica para los 2 tipos de enemigos terrestres que se implementarán. El enemigo que te ataca cuerpo a cuerpo, y un enemigo que te ataque a distancia, lanzándote proyectiles de energía.

Primero se crean hijos del controlador de IA, un controlador para cada enemigo. Este controlador de IA posee una función llamada *Attack*, la cual será sobrecargada por cada versión de este componente, haciendo que cada uno tenga su forma de atacar.

Todos los enemigos terrestres utilizarán el mismo *behavior tree*, ya que seguirán la misma lógica de ataque. Solo habrá 2 cosas que cambian: Los enemigos cuerpo a cuerpo se acercarán al jugador mucho, mientras que los que atacan a distancia tendrán un radio de aceptación más alto para

decidir atacar al jugador, y el enemigo cuerpo a cuerpo atacará activando una colisión a forma de *hitbox*, mientras que el enemigo a distancia instanciará un proyectil hacia el jugador.

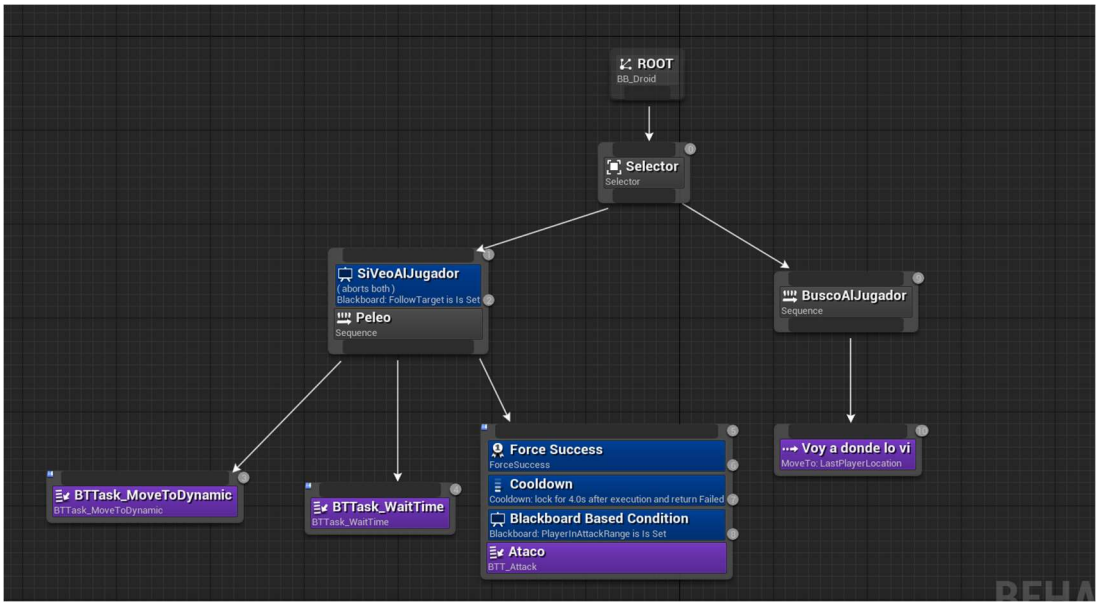

Figura 42. Behavior tree de los enemigos terrestres
(Fuente: Elaboración propia)

Este es el *behavior tree* que compartirán estos enemigos. Se aprecia que, si ven al jugador, se acercan y proceden a atacar. En cambio, si no ven al jugador, se mueven a la última posición en la que lo vieron y no hacen nada más. De aquí solamente es relevante la implementación propia de los nodos *BTTask_MoveToDynamic* y *BTTask_Attack*.

BTTask_MoveToDynamic hace algo muy similar a un *MoveTo* Actor, con la diferencia que la implementación creada permite cambiar el radio de aceptación de forma dinámica a partir de una variable en su controlador, esto se ha hecho para tanto el enemigo que se acerca mucho al jugador y el que se mantiene lejos puedan usar el mismo *behavior tree*, centralizando la lógica, haciéndola más escalable y fácil de editar.

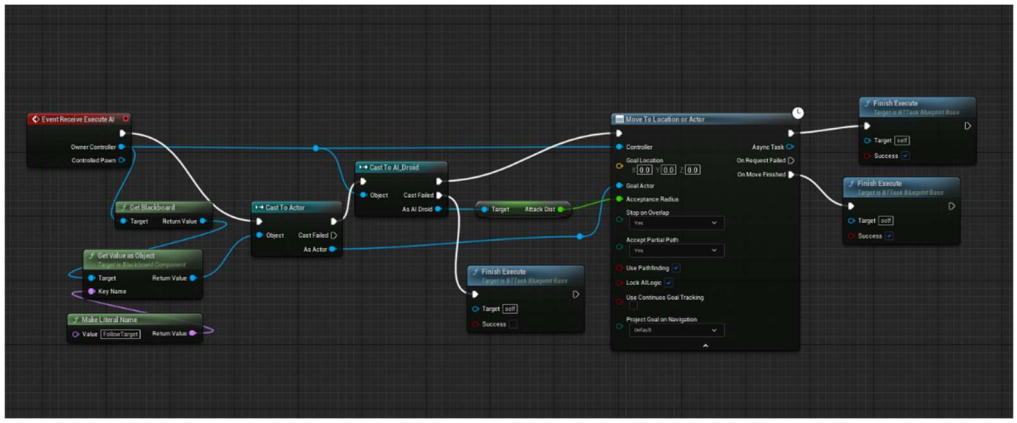

Figura 43. Blueprint del movimiento dinámico
(Fuente: Elaboración propia)

El nodo *BTTask_Attack* hace una cosa muy sencilla. Similarmente a el nodo anterior, este nodo simplemente llama a la función *Attack* del controlador de IA. Por lo que ambos controladores que hemos hecho sobrecargarán esta función, permitiéndoles tener distintas formas de atacar, aun compartiendo la misma lógica.

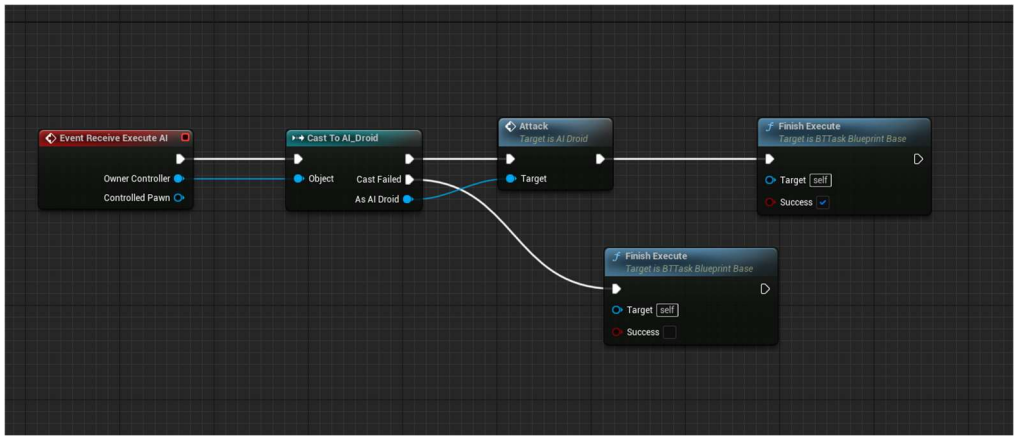

Figura 44. Blueprint del nodo de ataque
(Fuente: Elaboración propia)

Finalmente, en el controlador de cada enemigo se le asigna un radio de aceptación distinto, y sobrecargamos la función *Attack*.

Con esto implementado, hemos logrado exitosamente tener 2 tipos de enemigos terrestres que comparten lógica y se comportan de la manera deseada.

El ataque del personaje melee llama a un evento para poder hacer uso de la función *Delay*, la cual permite esperar un tiempo definido antes de continuar la ejecución. La implementación es la siguiente.

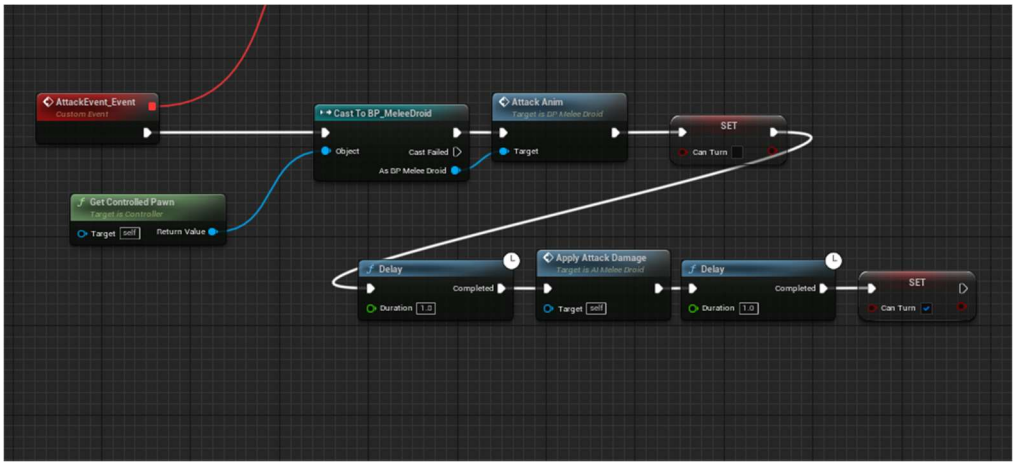

Figura 45. Blueprint de función de ataque cuerpo a cuerpo
(Fuente: Elaboración propia)

El ataque del personaje a distancia simplemente instancia un proyectil con su posición y orientación necesaria.

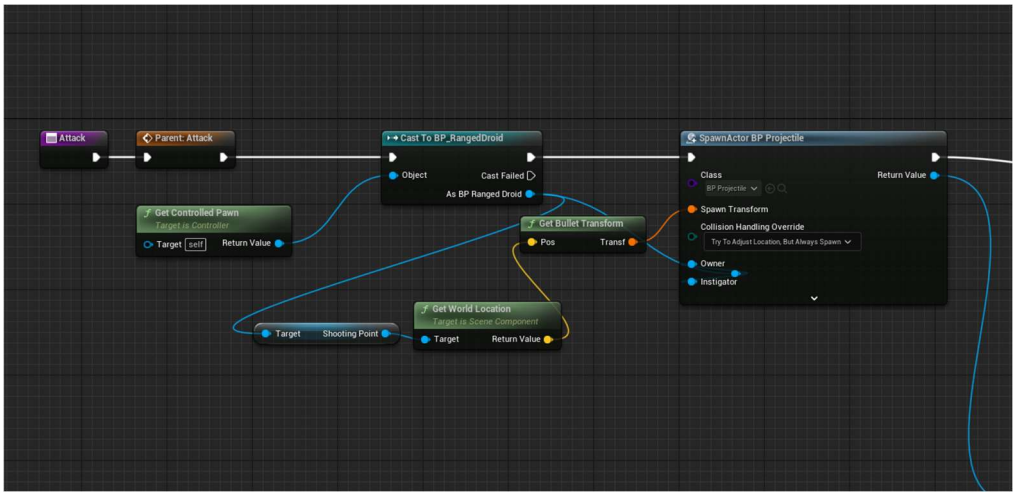

Figura 46. Blueprint de función de ataque a distancia
(Fuente: Elaboración propia)

Enemigo volador - Bases

Con la inteligencia artificial de ambos enemigos terrestres terminada, se procede a hacer un último tipo de enemigo, el enemigo volador. La idea es que vuele como un dron y tenga un comportamiento más de patrulla. Mientras que los enemigos terrestres solo esperan al jugador y lo buscan, el enemigo volador patrullará de forma automática por el mapa y atacará al jugador cuando lo vea, y cuando lo deje de ver, seguirá patrullando (A este enemigo se le refiere dentro del código del juego como '*Bird*' debido a que es un pájaro mecánico).

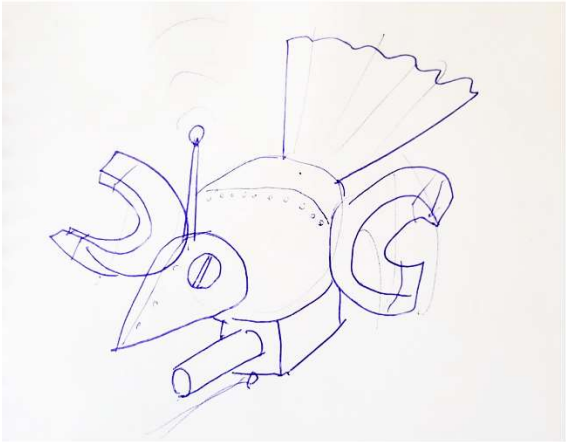

Figura 47. Arte conceptual de enemigo volador
(Fuente: Elaboración propia)

Para esto creamos un controlador de IA, y un *behavior tree* totalmente nuevos, ya que no compartirán lógica con los enemigos terrestres.

En vez de usar un *Movement Component*, se utilizará un *Floating Pawn Movement Component*, que permite definir movimientos para actores que vuelan o flotan, a diferencia del *Movement Component* convencional que solo permite mover actores terrestres.

Lo primero que se implementa es que estos enemigos puedan volar, así que se hace una implementación sencilla. Cuando el enemigo se crea, este escogerá aleatoriamente una altura dentro de cierto rango, esta altura será la altura de vuelo que tendrá de forma habitual. Se escoge de forma aleatoria para que todos los enemigos vuelen a alturas distintas, y no se junten todos en una misma altura. Se define una variable llamada *goingUp* en el controlador de IA, este puede ser verdadero o falso. Cada *frame* se revisará la altura actual del actor, y en base a la altura que tiene en ese momento si el actor debe subir o bajar. En base a si sube o baja, se añade movimiento hacia arriba o hacia abajo. Para darle una sensación más estilizada a estos enemigos, se hará que tengan un margen de distancia entre el punto en el que deciden subir y deciden bajar, para que en vez de que se queden estáticos a una altura, suban y bajen en un movimiento ondular dando una impresión más cercana a una levitación. La implementación es la siguiente.

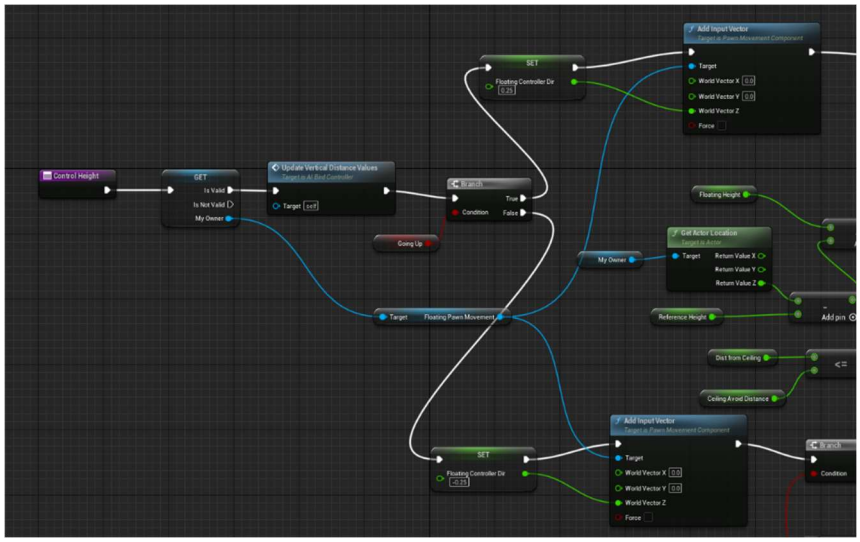

Figura 48. Blueprint de control de altura del enemigo volador
(Fuente: Elaboración propia)

Adicionalmente se ha hecho un *sphere cast* hacia arriba del enemigo en todo momento que chequee colisiones por encima de él. Esto se usa para saber si hay un techo y a que distancia está. Esto permite que, si el enemigo quiere volar a una altura mayor a la del techo, este se quede levitando debajo del techo con un margen definido. Así el enemigo no atraviesa, ni se choca contra los techos.

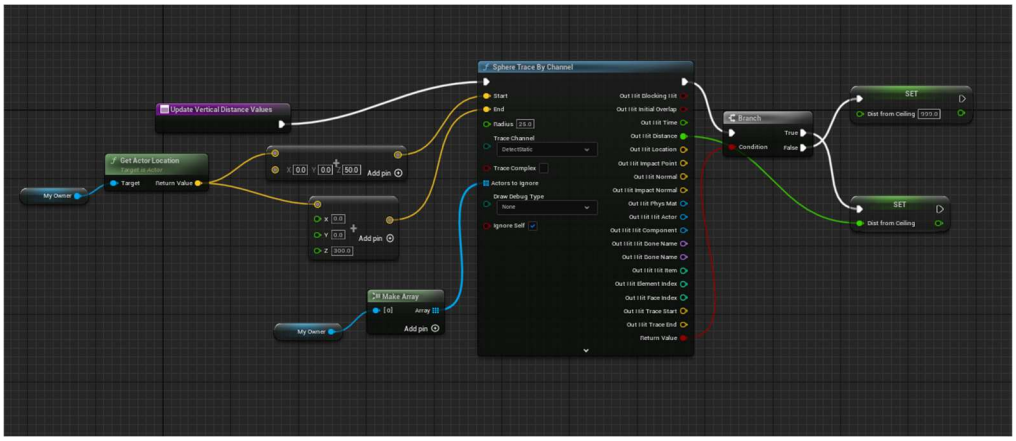

Figura 49. Blueprint de control para no chocar con el techo
(Fuente: Elaboración propia)

Con esto, ya tenemos las bases del enemigo volador hechas para poder comenzar a modelar su inteligencia artificial.

Enemigo volador - Inteligencia

Para esta inteligencia artificial se ha ideado el siguiente *behavior tree*.

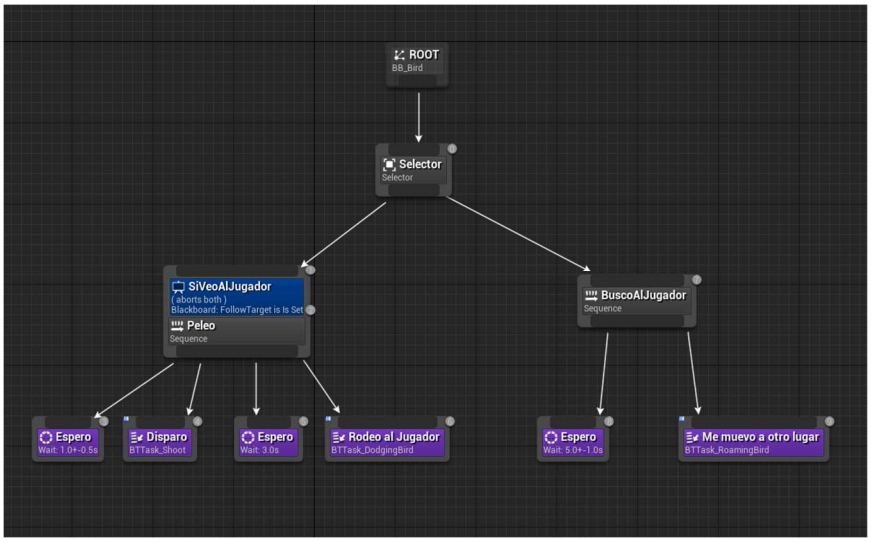

Figura 50. Behavior tree del enemigo volador
(Fuente: Elaboración propia)

Como se aprecia, si ve al jugador, el enemigo va a disparar, esperar y moverse alrededor del jugador. Si no, va a patrullar, esperando y después de unos segundos moviéndose a otro lugar.

En su ataque este dispara un proyectil idéntico al enemigo terrestre que ataca a distancia.

De aquí nos interesa la implementación de *BTTask_DodgingBird* y *BTTask_RoamingBird*.

El primer nodo que se desarrolló fue *BTTask_RoamingBird*. Este nodo hace una tarea simple, buscar hacía que dirección puede patrullar, y moverse ahí. A pesar de la simplicidad en concepto de esta tarea, su implementación tiene cierto nivel de complejidad.

Primero para buscar la dirección hacia donde se va a mover, escoge un ángulo aleatorio entre 0 y 360 grados en intervalos de 24 grados. A esta dirección se le hace un *sphere cast* para asegurarse que no choque con nada en su camino. Si se encuentra un obstáculo, le suma 24 grados al ángulo que buscó, y chequea de nuevo. Así hasta un máximo de 4 chequeos. Si los 4 chequeos fallan, se mueve hacia la última dirección buscada, pero la distancia justa para que no choque con el obstáculo.

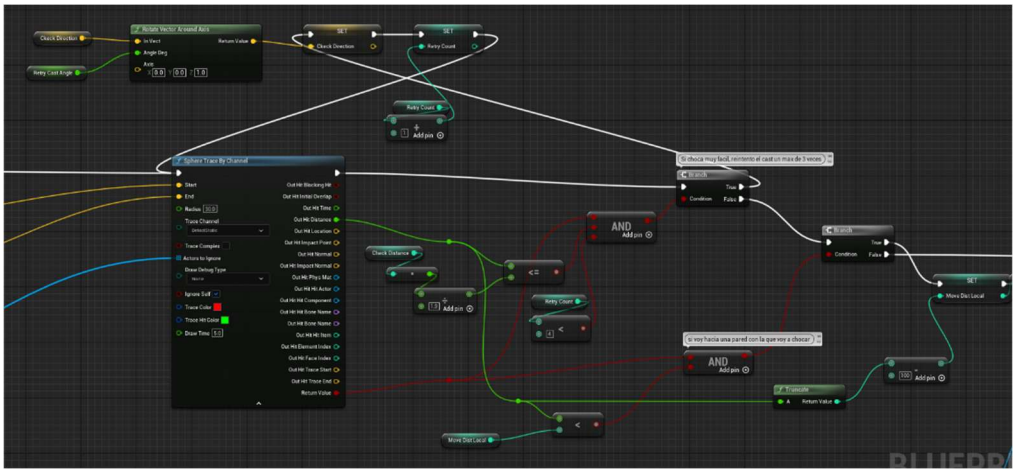

Figura 51. Blueprint del sistema de navegación
(Fuente: Elaboración propia)

Este sistema de búsqueda de direcciones nos garantiza que no choque contra paredes ni se acerque demasiado a ellas, tiene tendencia a alejarse de paredes, y en el peor de los casos, moverse paralelamente a la pared. Esto nos sirve para que cuando este enemigo este patrullando cubra mucha área y tenga más posibilidades de ver al jugador con su sistema de percepción.

A continuación, se desarrolló el nodo *BTTask_DodgingBird*. Este nodo se encarga de que cuando el pájaro ve al jugador, se intenta posicionar en un rango de distancia del jugador para luego atacar. Para que no se quede quieto, entre cada ataque se mueve a otra dirección rodeando al jugador.

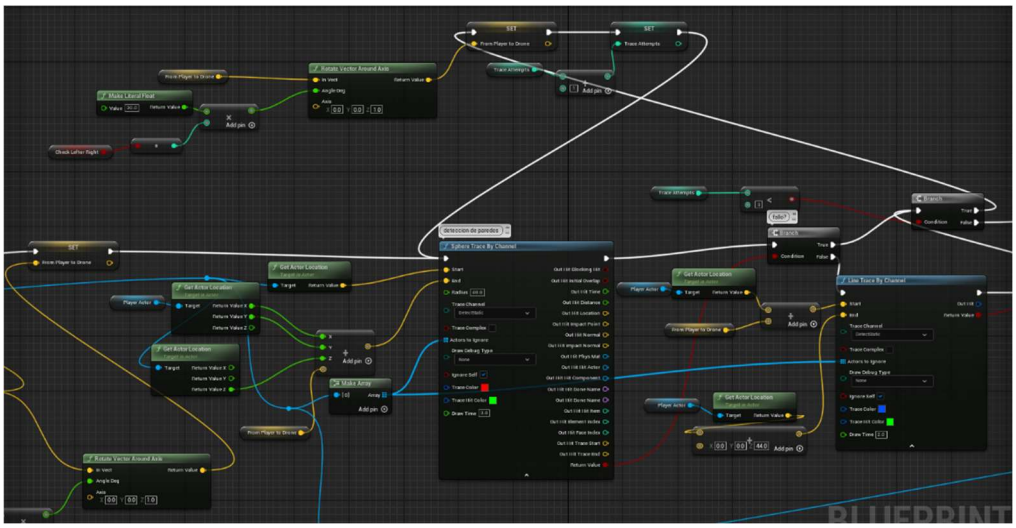

Figura 52. Blueprint del sistema de rodear al jugador
(Fuente: Elaboración propia)

Una vez se llama este nodo, se escoge aleatoriamente una dirección, si va a moverse a su izquierda o a la derecha. Ahora, basado en esto escoge un ángulo entre 35 y 90. Esto representa hacia dónde va a intentar moverse el pájaro de forma radial, tomando al jugador como centro, generando un

rango de posibles posiciones a las que intentará moverse. En el siguiente diagrama se muestra los ángulos relativos al jugador a los cuales este enemigo podría intentar moverse.

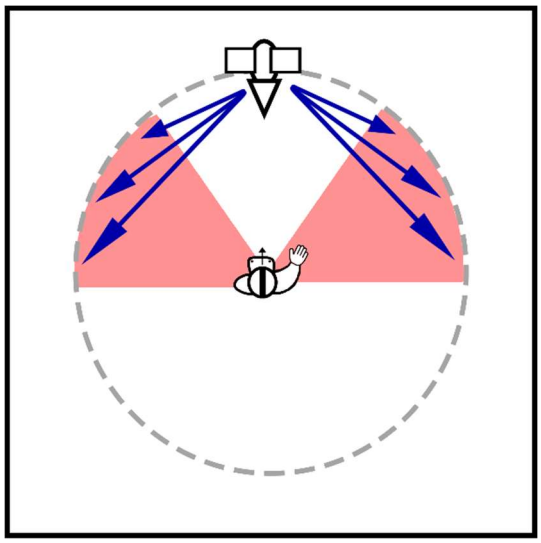

Figura 53. Diagrama de movimiento del enemigo volador frente al jugador
(Fuente: Elaboración propia)

A diferencia del nodo anterior, no solo se van a chequear choques contra paredes. También se va chequear que desde la posición a la que se va a intentar mover no se bloquee la posición con el jugador. Ya que al enemigo le interesa atacar al jugador, no va a moverse detrás de un muro por accidente, por ejemplo.

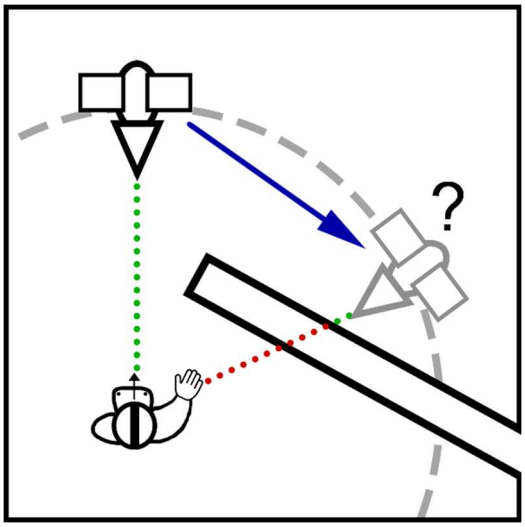

Figura 54. Diagrama de movimiento incorrecto del enemigo volador
(Fuente: Elaboración propia)

Si el chequeo falla por cualquier razón, similarmente al nodo anterior, va a intentar un máximo de 4 veces, sumando 30 grados cada vez, y se mueve hasta donde puede usando el último chequeo si todas fallan.

Finalmente, con estos nodos implementados, el enemigo tiene un movimiento convincente y divertido.

Modelos y *rigs*

Para que el juego contenga cierto nivel de identidad, se ha decidido hacer que los enemigos sean modelos propios.

Solo hay que hacer 2 modelos, el del enemigo terrestre, que será el mismo modelo para el cuerpo a cuerpo y el de distancia, y el enemigo volador.

Se ha elegido hacer una estética robótica para ahorrar esfuerzos en temas del *skinning* de los personajes para que se muevan de forma adecuada con el esqueleto.

Primero se han modelado ambos enemigos a partir de los artes conceptuales anteriormente mostrados en las figuras 40 y 47.

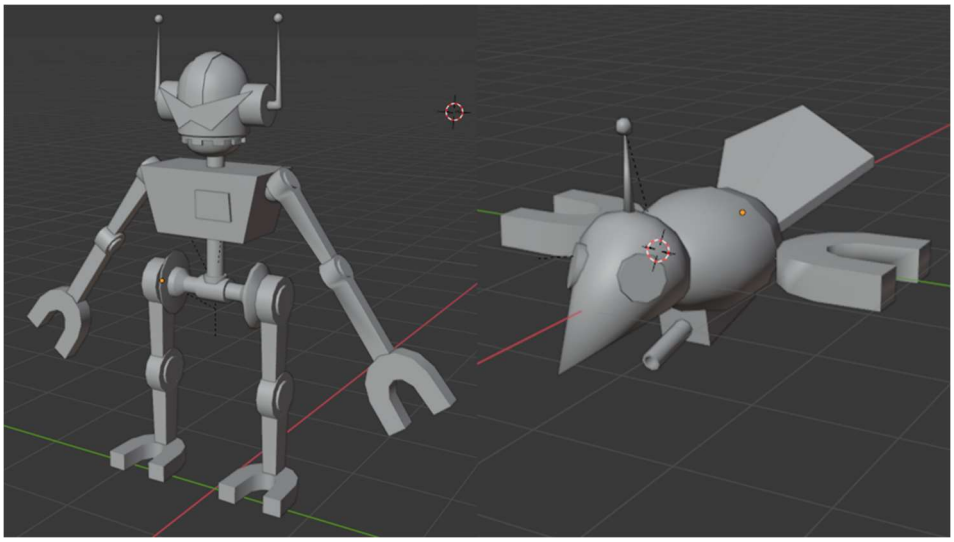

Figura 55. Modelos de los enemigos
(Fuente: Elaboración propia)

Estos modelos se han hecho en *Blender* desde cero a partir de figuras primitivas y funciones de modelado básicas.

Se ha procedido a crear un esqueleto para cada enemigo. Es importante en el esqueleto que cada hueso corresponda a una sola pieza móvil del robot, ya que se moverán estas piezas estáticas en relación de unas a otras.

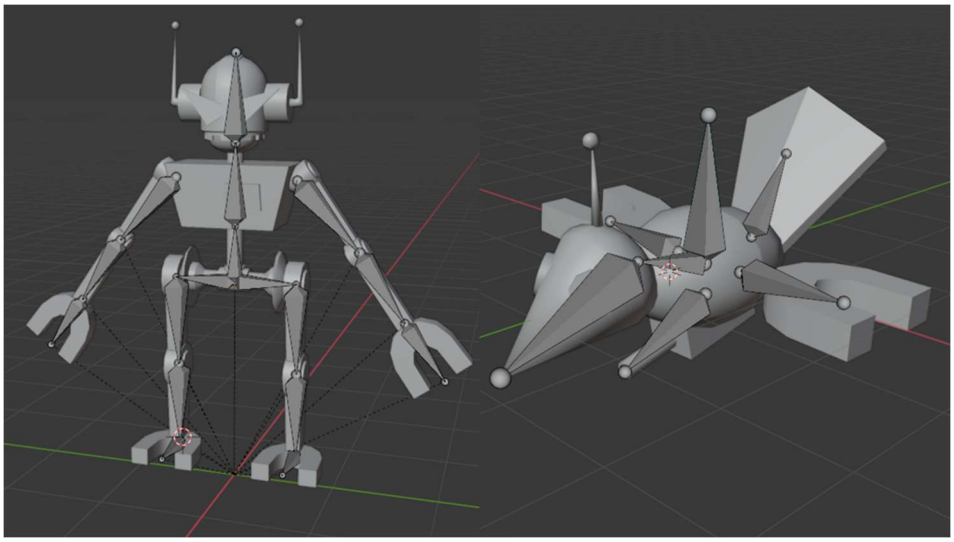

Figura 56. Modelos de los enemigos con sus esqueletos
(Fuente: Elaboración propia)

Finalmente, para terminar el *rigging* en *Blender*, solo queda emparentar cada pieza del robot a su hueso correspondiente, esto logra que el cuerpo del robot se mueva como se pretende.

Estos modelos se exportan como FBX para importarse a *Unreal*.

Animación física y salud de enemigos

Se pretende que estos modelos tengan animaciones físicas, es decir, que, en vez de tener animaciones convencionales hechas a mano, sus movimientos sean reacciones a las físicas del juego. En este caso solo serán 2. Que el enemigo reaccione a cuando el jugador le pegue y su cuerpo retroceda, y que cuando el enemigo muera, su cuerpo caiga y se deje llevar por las físicas, esto último se le conoce como *ragdoll*.

Para esto hace falta hacer un *rigging* de un material físico para cada personaje. El *rigging* del material físico consiste en asignar a cada hueso del *rig* hecho en *Blender* un cuerpo físico con colisiones definidas.

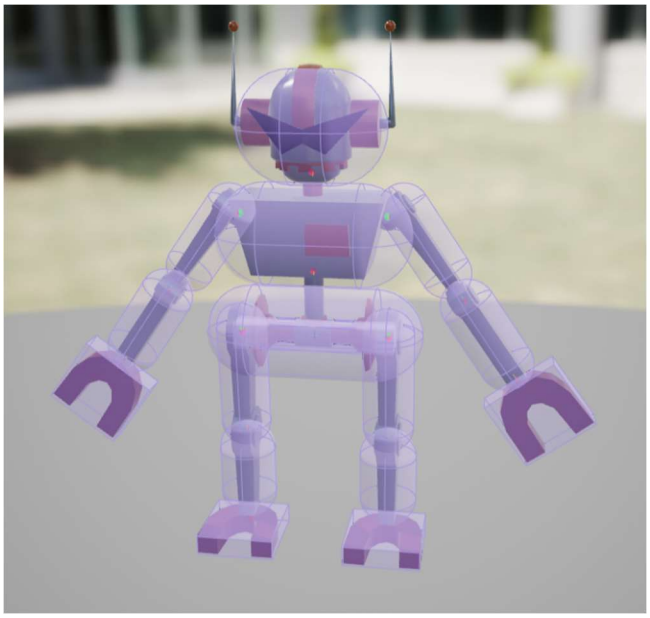

Figura 57. Material físico del enemigo terrestre
(Fuente: Elaboración propia)

Una vez tenemos todos los cuerpos físicos asignados, definimos sus relaciones entre sí, conocido como *constraints*. Estos *constraints* indican al *rig* que cuerpos físicos están juntos, y sus límites de rotación, para que, por ejemplo, una rodilla solo se flexione 90 grados hacia atrás y no se pueda extender hacia adelante. Todo esto contribuye a que el cuerpo físico se mueva de forma más convincente.

Con todos estos elementos terminados tenemos los *rigs* preparados para animaciones físicas.

Al simular las físicas, el personaje ya hace *ragdoll* de forma adecuada, ahora solo queda hacer las animaciones de reacciones a los ataques al jugador.

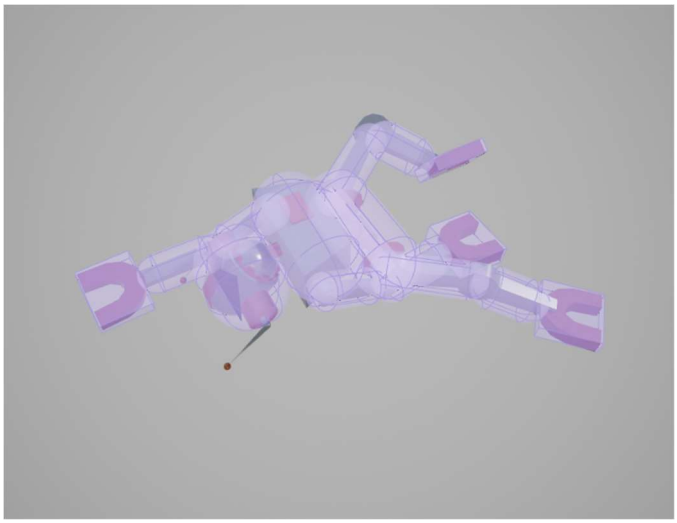

Figura 58. Ejemplo de ragdoll
(Fuente: Elaboración propia)

Hacer las reacciones a los golpes fue más problemático, por lo que se recurrió al tutorial del *youtuber PrismaticaDev* para lograr el resultado deseado (PrismaticaDev, Physical Animation: The Ultimate Starter Guide [UE4/UE5], 2022).

Para poder generar reacciones a los ataques, se hace una simulación parcial, quiere decir, en vez de hacer que el cuerpo físico simule sus físicas por completo, se hace que solo ciertas partes del cuerpo simulen físicas. Se utiliza una función de *Unreal* que permite mezclar, o hacer *blend*, entre la simulación física y las animaciones convencionales de los *rigs*, que permite que reaccionen a eventos de su entorno con simulaciones físicas parciales, y de forma discreta poco a poco quitarles peso a las animaciones físicas frente a las convencionales y detener la simulación. Esto nos permitirá que el enemigo mientras se mueva, reaccione a un golpe sin perder el balance por completo, y luego suavemente recomponerse y continuar con sus animaciones (PrismaticaDev, Physical Animation: The Ultimate Starter Guide [UE4/UE5], 2022).

Ahora, para poder hacer esta reacción a ataques, solo queda que el jugador pueda hacer daño al enemigo, y una vez sepamos en que parte del cuerpo le ha dado, ahí asignamos un impulso para que el enemigo reaccione.

Primero chequeamos que cuando un objeto con velocidad choque con un enemigo, buscamos el hueso más cercano al punto de impacto, y luego buscamos la velocidad relativa del objeto que hizo daño en ese punto. Utilizamos esta velocidad como velocidad del impulso, y aplicamos este impulso al hueso correspondiente.

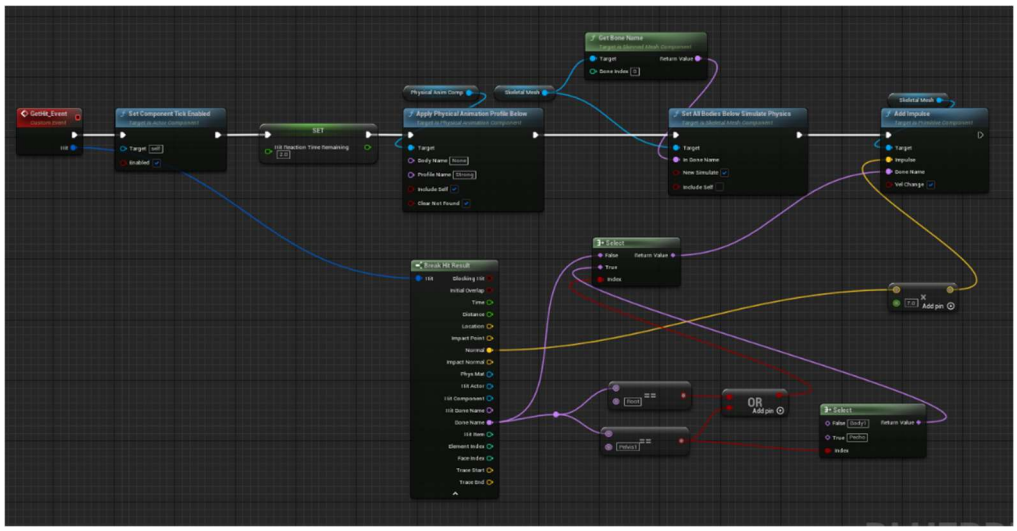

Figura 59. Blueprint de impulso al hueso correspondiente en impacto
(Fuente: Elaboración propia)

Adicionalmente, como ya estamos llevando cuenta de los objetos que hacen daño a los enemigos, hacemos un sistema de vida simple en el que se añaden a los enemigos una variable de números enteros como puntos de vida, y el golpe que recibe le resta puntos de vida. Finalmente, cuando el enemigo se queda sin puntos de vida, se desactiva su inteligencia artificial y se simula un *ragdoll* completo, para demostrar que el enemigo ha sido vencido. Después de unos pocos segundos el enemigo se elimina.

Tipos de armas

Para darle más profundidad al juego, se han conceptualizado 2 armas principales. Una espada de acero que haga un daño elevado, y una espada de madera que haga menos daño, pero puede encenderse en fuego, lo cual tendrá unos usos que se explicarán en la sección siguiente a esta.

Se ha añadido una variable, siendo un numero entero, al componente de objeto para agarrar, esta variable indica el daño que hace este objeto cuando choca con un enemigo. Este valor es por defecto 1. Lo que quiere decir que cualquier objeto que se pueda agarrar puede hacer daño, no solo las espadas. Respectivamente los daños son los siguientes: Todos los objetos por defecto hacen 1 de daño, la espada de madera hace 2 de daño, la espada de acero hace 3 de daño. Los puntos de vida de los enemigos están en constante variación debido a balanceos del juego, pero en un principio los enemigos terrestres tienen 6 puntos de vida, y los voladores tienen 3.

Ahora para encontrar los modelos que se van a usar, se ha descargado un paquete de *assets* gratis de la tienda de *Unreal* llamado *Infinity Blade: Weapons*. De aquí se han encontrado modelos de armas que corresponden con el concepto inicial. Adicionalmente se ha añadido un arma basada en uno de los *assets* del paquete, siendo una espada de globo. Esta espada de globo se le ha asignado valores de fricción linear altos, lo que quiere decir que cae lentamente y al lanzarla pierde su velocidad rápidamente, haciendo que se comporte como una espada de globo de vedad. La intención de esta arma es añadir variedad a los objetos que el jugador encuentre y darle un tinte un poco más cómico al juego.

Figura 60. Armas del juego
(Fuente: Elaboración propia)

Finalmente, se ha añadido una espada gigante que hace 10 de daño, inspirada en el arma principal del protagonista del famoso manga *Berserk*. Se han ajustado algunos valores de las físicas de lanzamiento para que no se rompan con el uso de esta espada. Sin embargo, no es certera la inclusión de esta arma en el juego final.

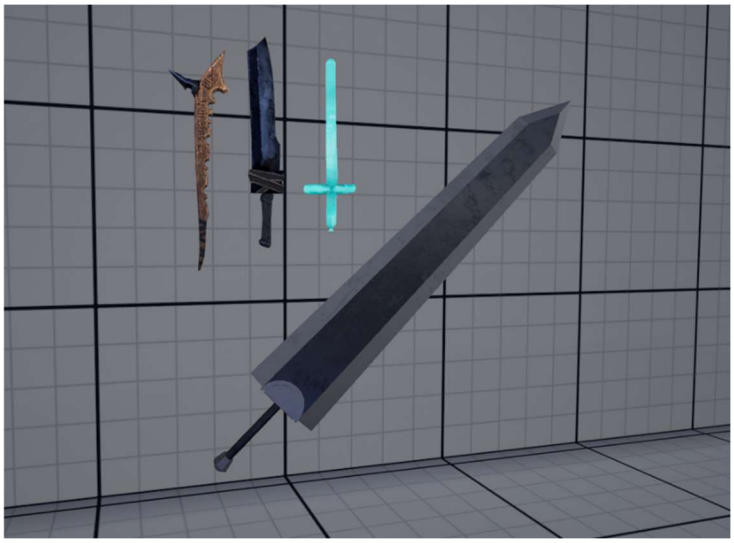

Figura 61. Armas del juego, y la Dragonslayer
(Fuente: Elaboración propia)

Con esto concluido, se han incluido en el juego de forma correcta todas las armas que se pretendían.

Fuego y torres de fuego

Como se mencionó en el apartado anterior, la espada de madera va a tener una mecánica de fuego, sumado a un objeto en el mapa que sea una torre que se pueda encender en fuego, que servirá como proveedor de fuego dentro del juego, y también se podrá usar para puzles, por ejemplo, teniendo que encender todas las torres que están apagadas en un área.

Figura 62. Demostración de la mecánica de fuego
(Fuente: Elaboración propia)

Se ha utilizado un sistema de partículas de un *asset* pack gratis de la tienda de *Unreal* llamado *FX Variety pack*. Se ha editado el efecto de fuego para que se adecue más a el uso que se le dará en este juego.

Simplemente con detectar colisiones entre objetos que puedan interactuar con el fuego, y chequear si están encendidos, hacer este sistema de interacciones es relativamente sencillo. Además, se ha hecho que la espada globo reviente cuando toque fuego.

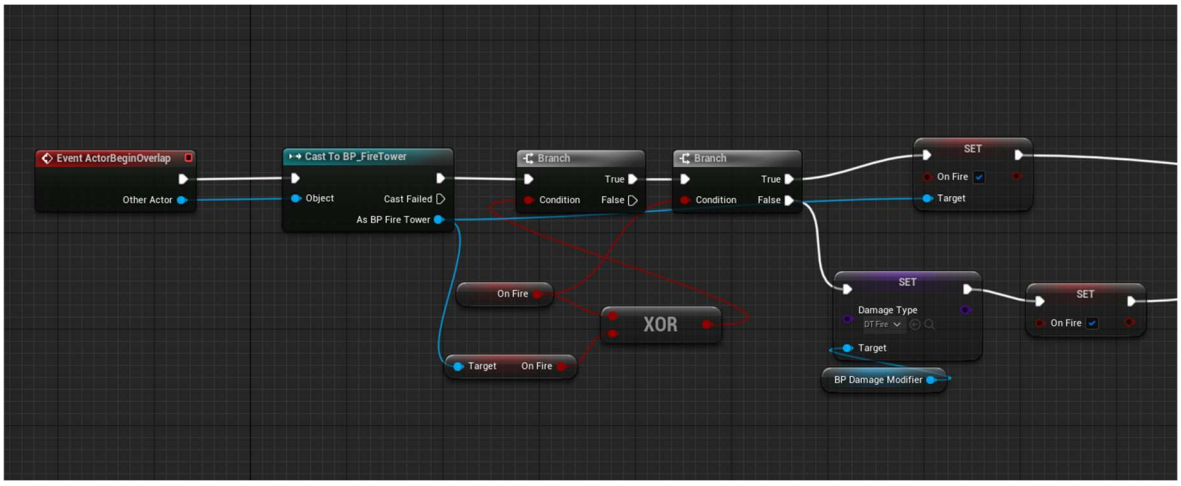

Figura 63. Implementación del sistema de fuego
(Fuente: Elaboración propia)

Palancas, botones, y puertas

Para poder hacer niveles más interesantes, faltan unos últimos pocos elementos.

Se ha hecho una palanca que se puede agarrar y activar con la mano. Esta palanca tiene unas luces que indican rojo si está bloqueada, y no se va a poder tirar para abajo del todo, o verde si esta desbloqueada y se puede activar.

Figura 64. Palanca del juego
(Fuente: Elaboración propia)

Esta palanca se va a usar para abrir puertas, y activar eventos dentro del juego. El sistema de bloqueo de estas palancas sirve para que se el jugador no pueda avanzar hasta que haga alguna acción o serie de acciones que desbloqueen la palanca según como dicte el diseño de cada nivel.

Estas palancas también se han usado para crear un dispensador de objetos, con la idea de si el jugador pierde un arma o un objeto importante, pueda “materializar” otro, destruyendo el objeto creado anteriormente. Este funcionamiento ha sido inspirado en los dispensadores de cubos de Portal 2.

Figura 65. Dispensador del juego
(Fuente: Elaboración propia)

Se ha creado un objeto botón, que tiene como función estar en un estado desactivado, y una vez es golpeado por un objeto físico que el jugador haya lanzado, este se activará. Cuando el botón está

desactivado tiene un color rojo brillante, y cuando se activa, este color se cambia a verde brillante. Este elemento se utilizará para diseñar distintos retos a medida que el jugador progrese.

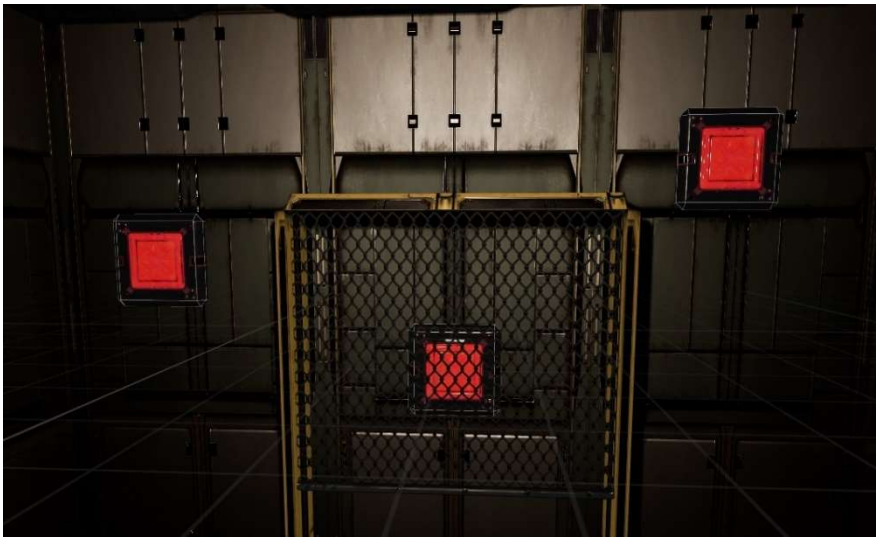

Figura 66. Botones del juego
(Fuente: Elaboración propia)

Finalmente se ha desarrollado un sistema simple de puertas para controlar la progresión de niveles.

En principio cada habitación o conjunto de habitaciones contendrá un reto, sea de puzles, reto con las mecánicas, o vencer enemigos. Para separar estas unidades en "niveles" se ha creado un pequeño túnel con dos puertas. Este sistema de puertas separará niveles, permitiendo pasar, pero no volver.

Figura 67. Puerta del juego
(Fuente: Elaboración propia)

Este sistema de puertas funciona de la siguiente manera. Hay dos puertas a cada punta del pasillo, una que se llamaremos la puerta de afuera y otra que llamaremos la puerta de adentro. La puerta de afuera tendrá una palanca que la activa desde fuera del pasillo. Este es el lado por el que el usuario entrará. Por defecto la palanca está bloqueada, por lo que en cada nivel se programará que condiciones deberán cumplirse para que esta palanca se desbloquee y el jugador pueda avanzar. El jugador no puede utilizar la palanca para cerrar la puerta. Una vez el jugador abre la puerta y entre, una colisión dentro del pasillo cerrará de nuevo la puerta de afuera, evitando que el jugador pueda volver. Una vez dentro del pasillo, se encontrará del otro lado la puerta interior y su palanca respectiva. Esta palanca siempre estará desbloqueada, por lo que el jugador puede usar esta área como un área segura entre niveles. Adicionalmente, se pueden meter objetos y armas dentro de esta área para ponerlos a disposición del jugador antes de entrar al siguiente reto. Una vez abra la puerta interior, se podrá pasar a la siguiente habitación.

Este elemento se ha creado emparentando actores dentro de otros. Se tenía un actor palanca con su funcionalidad, y se ha creado un actor puerta que tan solo puede abrirse y cerrarse a través de eventos y llamadas. Se ha creado el actor *DoorSystem*, o sistema de puertas, que tiene el pasillo en cuestión, las dos palancas y las dos puertas como hijos. A través de eventos, se ha hecho que este *DoorSystem* funcione como un sistema que comunica actores dentro de este y los utiliza como componentes. Esto permite crear comportamientos más complejos a partir de actores creados previamente con funcionalidades definidas.

A continuación, se muestra la estructura del actor *DoorSystem*, y como se utilizan eventos para comunicar entre si a los actores hijo.

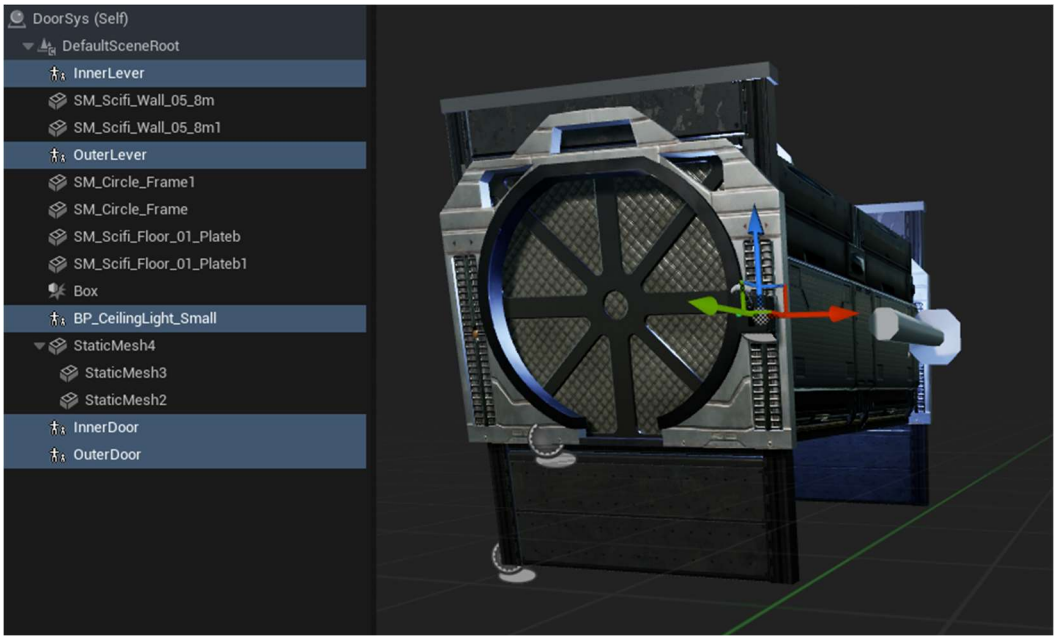

Figura 68. Demostración de actores compuestos
(Fuente: Elaboración propia)

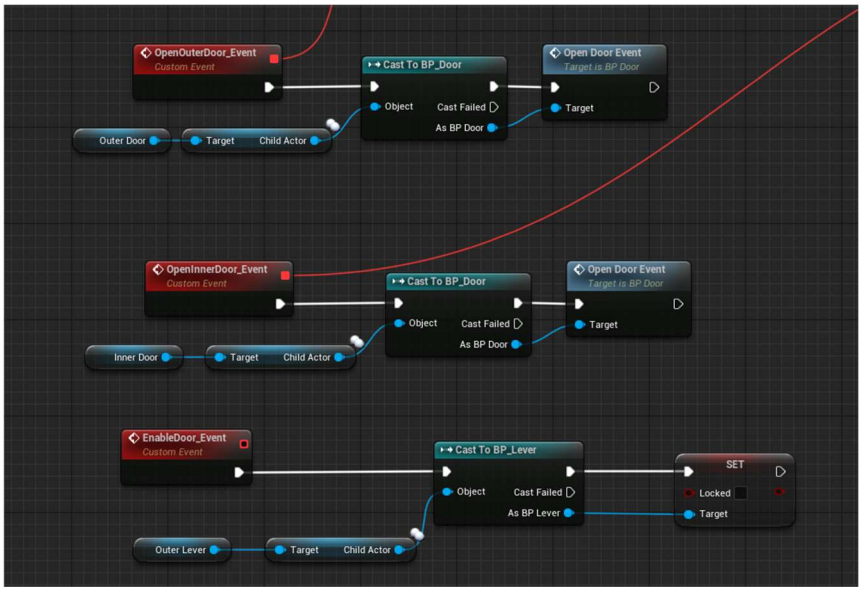

Figura 69. Ejemplo de comunicación entre actores en un actor compuesto
(Fuente: Elaboración propia)

Conclusión de esta fase

En esta etapa se han creado las mecánicas del juego, sobre las cuales revolverá el diseño de niveles y la experiencia de juego final, además de todos los otros sistemas de *gameplay* que son los bloques de construcción para cada reto en esta experiencia virtual.

Tras mucho ensayo y error, esta etapa ha sido fructífera y se ha desarrollado de la manera esperada; creando distintas mecánicas, añadiendo y quitando ideas, probando conceptos, y aprendiendo detalles más profundos sobre las herramientas de desarrollo.

Al prototipar este juego se han conceptualizado distintas técnicas para lograr los resultados deseados, encontrando detalles a favor y en contra de las mismas. Ha habido problemas y se ha tenido que idear soluciones a los mismos, tanto a nivel de diseño como a nivel de programación.

Ultimadamente, tras acabar la fase de prototipado, ya hemos concretado las mecánicas de juego. Los elementos que el jugador se encontrará, como armas, puertas, botones y palancas ya están programados, y funcionando debidamente. Finalmente, los enemigos con su inteligencia artificial y sistemas de interacción, incluyendo vida, y animaciones físicas, también tienen un rol concretado y finalizado en lo que a la experiencia de *gameplay* concierne.

Por todas las razones previamente mencionadas, se considera terminada la fase de prototipado, y se da inicio a la fase de producción, en la cual solamente utilizaremos los elementos ya creados para construir niveles y crear la experiencia final.

6.3. Producción

Esta fase de producción se enfocará en utilizar todos los elementos previamente creados para construir una experiencia final. Esto incluirá creación de niveles, implementación de efectos visuales, pulido visual, optimización, sonido, y progresión del juego. El objetivo de esta fase es pasar de elementos de *gameplay* separados, a una experiencia completa de comienzo a fin, que pueda ser jugada y distribuida como producto.

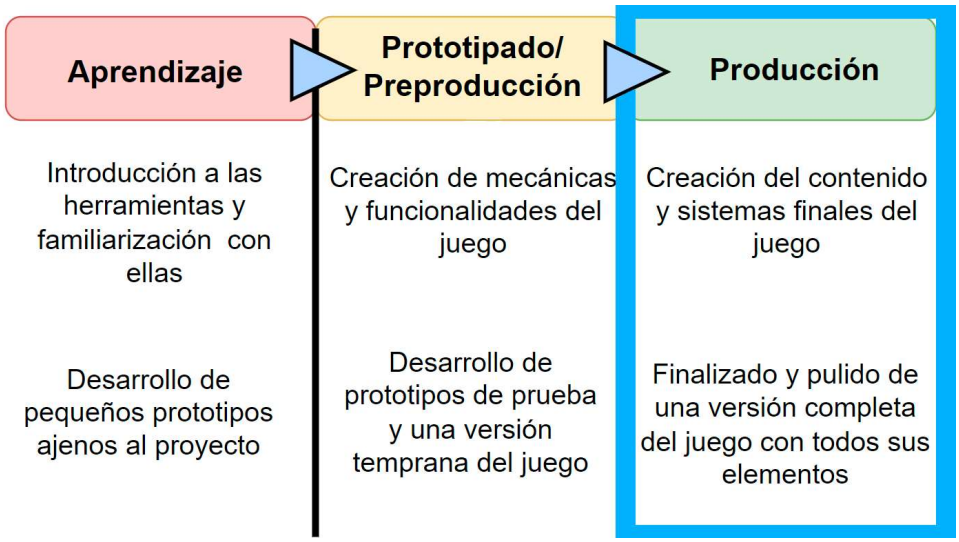

Figura 70. Fase de producción de la metodología
(Fuente: Elaboración propia)

Diseño de niveles

Para construir esta experiencia, es pertinente un buen diseño, el cual dictarán los retos que se le pondrán al jugador desde el momento que empiece a jugar.

El diseño tiene tres objetivos principales: Lograr una curva de aprendizaje, dar la sensación de progresión, y finalmente, hacer que el jugador se sienta inmerso en la experiencia. Todos estos son elementos de diseño bien conocidos que se explican en libros como '*Level Up! The Guide to Great Video Game Design',* de Scott Rogers (Rogers, 2010).

El primer objetivo es la curva de aprendizaje. Todas las experiencias de los videojuegos se pueden resumir en una esencia principal: El aprendizaje. Este aprendizaje incluye conocer las mecánicas, enemigos, y habilidades que como jugador uno necesita para poder avanzar en dicho juego. Sin este aprendizaje, no hay avance. Por eso se utiliza el concepto de curva de aprendizaje, en el cual se asume que cuando el jugador comienza a jugar no sabe nada del juego, y las mecánicas tienen que irse presentando al jugador de forma progresiva. Garantizando que en todo momento este enfrentando un reto, pero que a la vez este tenga las herramientas para enfrentarse a los mismos. La diversión de una experiencia linear como esta radica en su curva de aprendizaje, y en el balanceo de la misma que el diseñador paute. Debido a todo esto, lo primero que se ha diseñado es una lista de niveles, y que mecánica o reto deberá aprender a dominar el jugador en cada una de estas áreas. Esto nos dará una sólida guía para la progresión que debe tener el juego.

1. El jugador debe aprender a moverse en el mundo virtual.
2. El jugador debe aprender a agarrar y lanzar objetos.
3. El jugador debe aprender a golpear enemigos utilizando armas de su entorno.
4. El jugador debe aprender la mecánica de fuego
5. El jugador debe aprender la mecánica de lanzamiento y atracción.

Estos 5 puntos se han traducido de forma directa a las primero 5 áreas del juego, para poder completar el siguiente reto, el jugador debe haber dominado la mecánica que aprendió previamente. Esto garantiza que el jugador siempre se esté divirtiendo, debido a que apenas domine una mecánica, se enfrentará al reto de aprender una mecánica nueva, mientras domina y pone a prueba todo lo que ha aprendido hasta ese momento. Lo que nos lleva a nuestro segundo objetivo, la sensación de progresión.

Una vez se ha completado el aprendizaje y el jugador ya conoce las bases del juego y las mecánicas, lo único que aportará diversión es la sensación de progresión. Los siguientes niveles simplemente son distintas combinaciones de mecánicas que aumentan la complejidad del área y ponen a prueba al jugador a dominar de forma más efectiva su conocimiento y habilidades sobre las mismas. Esto

hace que el jugador tenga que profundizar en las mecánicas del juego. Con retos de progresiva dificultad bajo las mismas reglas ya establecidas, se logra un aprendizaje distinto al anterior. El primer aprendizaje es el aprendizaje de una mecánica nueva, pero el aprendizaje que se busca a la hora de crear progresión, es el de enseñar distintas maneras en las que las mecánicas pueden usarse y hacer que el jugador domine estas al máximo. Una buena progresión, adicionalmente, culmina con un reto final que pone a prueba todo lo que el jugador ha aprendido hasta ese momento.

Finalmente, el tercer punto: la inmersión. Este es un punto más artístico y abstracto. La inmersión no tiene tanta importancia en otro tipo de juegos, pero como se está desarrollando un juego de realidad virtual, la inmersión es necesaria para mantener entretenido al jugador y poder sacarle el máximo provecho a estas mecánicas que utilizan la interacción entre el jugador y este mundo virtual. En este caso, la inmersión va a consistir en hacer niveles que se sientan creíbles bajo el contexto de una mazmorra con robots.

Utilizando todos estos principios como referencia, se ha bocetado, y finalmente concretado el siguiente diseño de niveles:

El juego comienza en un ascensor que va descendiendo. El jugador no puede salir mientras sigue bajando, en este momento a través de un altavoz se escucha al antagonista del juego, el rey de los imanes. En este diálogo este personaje te explicará que estas en su mazmorra con el fin de vencerlo, y que tendrás que superar sus retos. Después el ascensor se detiene de golpe, salen chispas, y el ascensor cae. La finalidad de esto es generar la impresión y miedo de caer en el ascensor. Finalmente, el ascensor cae, el jugador esta ileso, y se ha llegado a la primera habitación del juego.

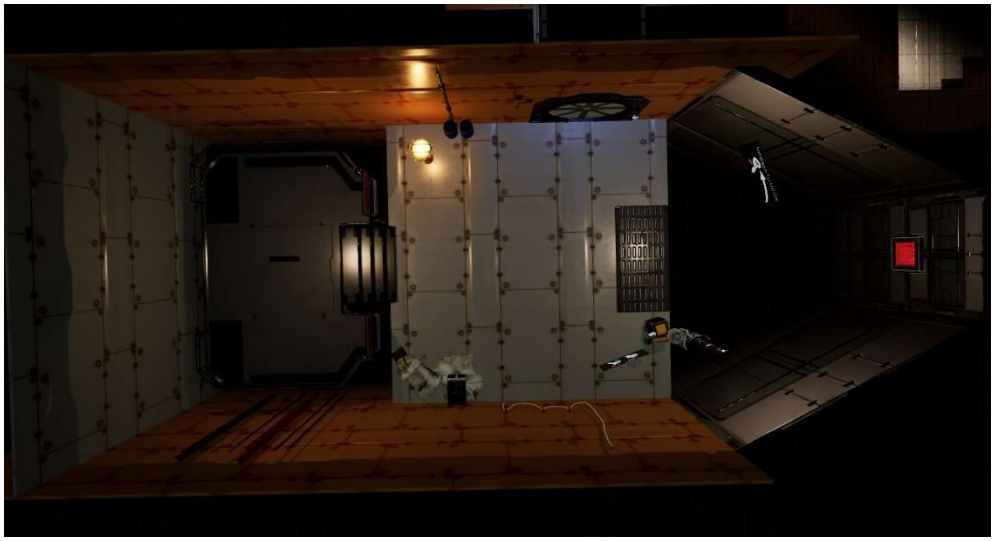

Figura 71. Primera habitación
(Fuente: Elaboración propia)

En esta primera habitación el jugador aprende a moverse y a explorar su entorno. Mas adelante se encuentra una puerta cerrada, un dispensador de espadas de madera, y un botón fuera de alcance para el jugador. Para poder avanzar el jugador debe dispensar una espada, y lanzarla al botón, desbloqueando la puerta. La finalidad de esto es enseñar al jugador como agarrar y lanzar cosas, además de familiarizarlo con los botones y dispensadores.

Figura 72. Segunda habitación
(Fuente: Elaboración propia)

Al avanzar a esta segunda habitación el jugador se encuentra una espada de metal, y dos enemigos. El enemigo que ataca cuerpo a cuerpo y el enemigo que ataca a distancia. El fin de esta habitación es familiarizar al jugador con el combate. Para avanzar el jugador debe agacharse y pasar por un agujero en la pared más adelante. A pesar de que vencer a los enemigos es opcional, esto es algo complicado ya que están cubriendo la salida.

Figura 73. Tercera habitación
(Fuente: Elaboración propia)

Después se llega a la tercera habitación. Aquí habrá un enemigo de ataque cuerpo a cuerpo, y un puzle que resolver. Hay un dispensador de espadas de madera, dos torres de fuego, y 3 botones fuera de alcance. La peculiaridad es que uno de los botones está cubierto por una cerca. Una de las torres de fuego esta encendida y la otra apagada, la solución es encender en fuego la espada de madera y propagar el fuego a la segunda torre. Esto hará que la cerca baje, desbloqueando el acceso al último botón, permitiendo al jugador avanzar. La finalidad de esta habitación es enseñar al jugador cómo funciona la mecánica de fuego y el tipo de puzles que podrá encontrarse más adelante.

Figura 74. Cuarta habitación
(Fuente: Elaboración propia)

Tras avanzar, se llega a la cuarta habitación. Aquí el jugador se encuentra un agujero enorme delante de él. Hay cuatro torres de fuego: dos en el agujero y dos a sus lados en la plataforma, sumado a un dispensador de espadas de madera. De las torres de fuego solamente una de las que están en el agujero esta encendida. El jugador sabe que tiene que encender las 4 torres porque es algo similar a lo que tuvo que hacer en la habitación anterior. Sin embargo, la torre encendida sobre el agujero está muy lejos, y si lanza la espada, esta cae al vacío. Tanto el rey de los imanes con su dialogo como algún tutorial que aparezca en pantalla le enseñarán al jugador como atraer de vuelta objetos que ha lanzado, esto le servirá para lanzar la espada a la torre encendida, atraer la espada en llamas de vuelta, y encender las torres restantes. Después de esto se eleva un puente que permite al jugador avanzar a través del vacío.

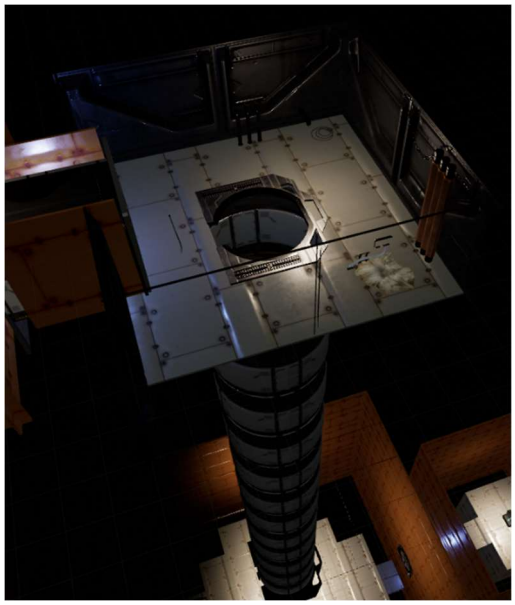

Figura 75. Quinta habitación
(Fuente: Elaboración propia)

Se llega a la habitación cinco. Aquí solamente hay un agujero muy profundo delante del jugador. De manera chistosa, el rey de los imanes reta al jugador a probar su valentía y lanzarse por el agujero, y que, si se lanza, podrá avanzar y robarle su arma suprema. En esta habitación el jugador solo tiene que saltar al vacío. La finalidad de esta habitación es cambiar un poco el ritmo del juego, y hacer que el jugador tenga que saltar al vacío, experiencia que asusta y es emocionante para muchos jugadores de realidad virtual.

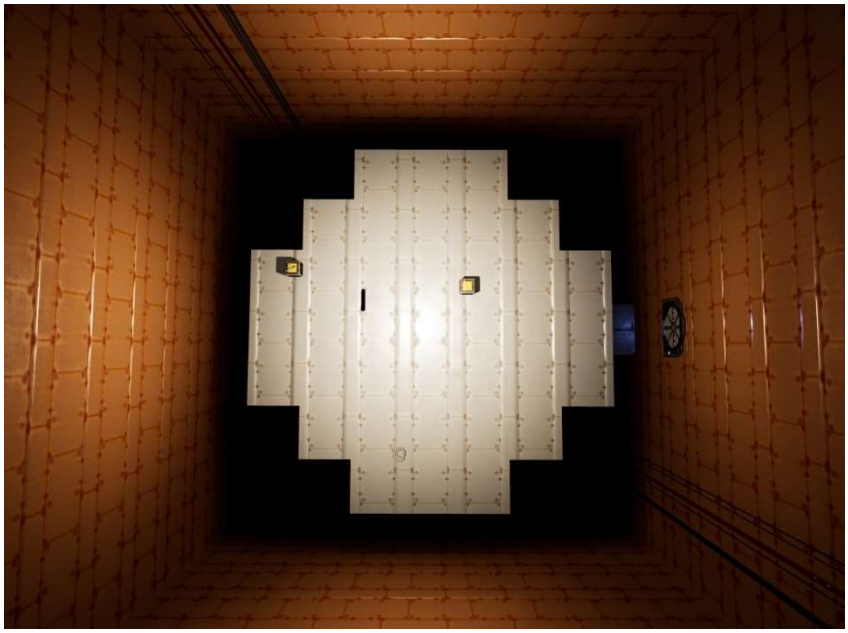

Figura 76. Sexta habitación
(Fuente: Elaboración propia)

Tras caer por el agujero, el jugador cae en medio de una amplia plataforma. Tiene una espada de madera a su alcance, y una espada de metal algo más lejos. Aparecerán cuatro enemigos voladores que el jugador tendrá que vencer para avanzar.

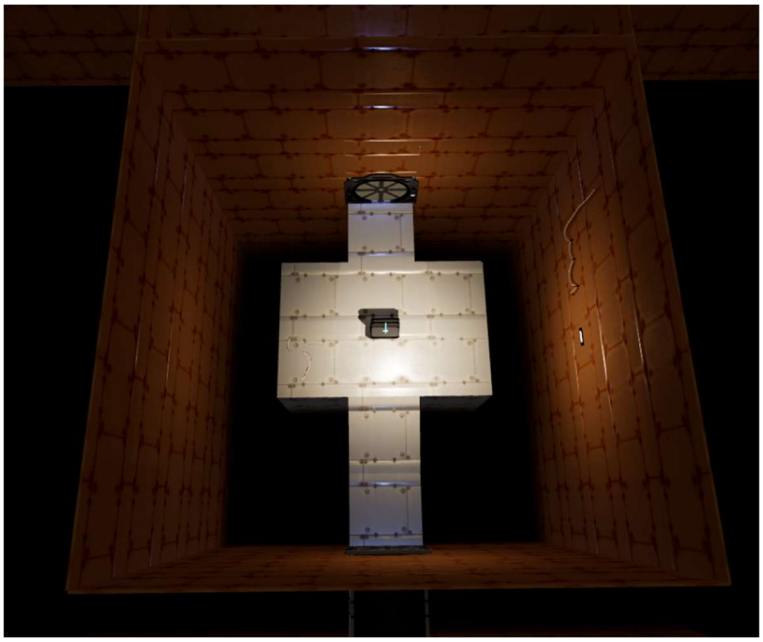

Figura 77. Séptima habitación
(Fuente: Elaboración propia)

Tras vencer a los enemigos, el jugador llega a esta pequeña habitación con una espada de globo en un pedestal. El rey de los imanes explica que esa es su arma suprema. Esto es un momento con intenciones de resultar cómico para el jugador. Tras unos segundos caen dos robots cuerpo a cuerpo desde arriba. El jugador debe vencerlos para avanzar. Si el jugador usa la espada de globo, se dará cuenta que no se puede lanzar muy lejos, al ser de globo, y que necesita asestar muchos golpes a los enemigos para vencerlos. Para efectos del *gameplay*, esta arma no tiene mucha utilidad más allá del chiste.

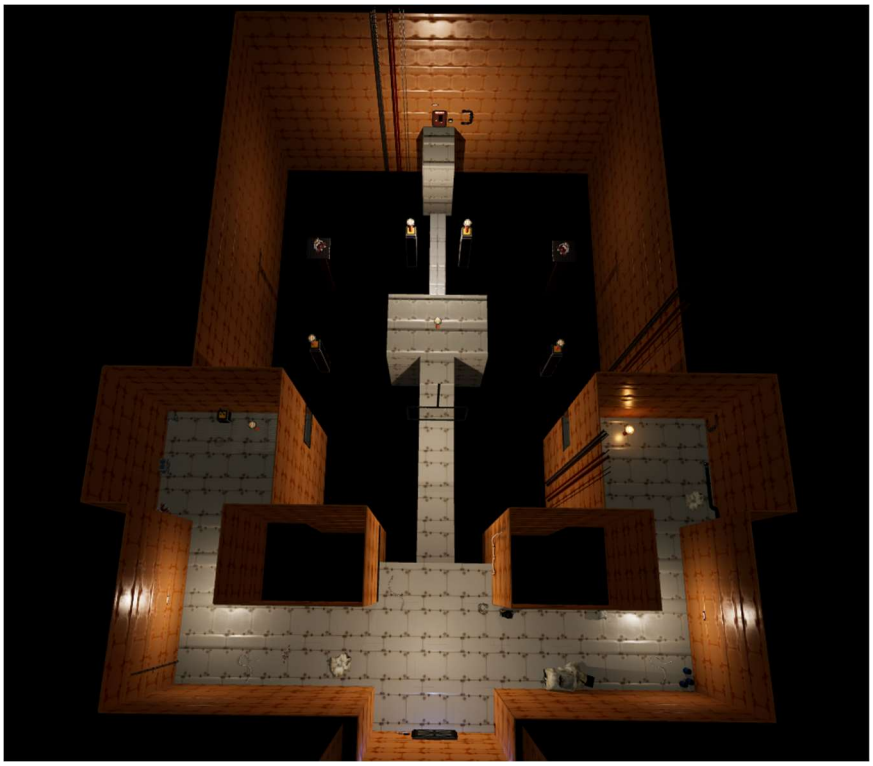

Figura 78. Octava habitación
(Fuente: Elaboración propia)

Finalmente se llega a la última habitación que se pretende desarrollar para este demo del juego. Esta es la más grande de todas y con mayor nivel de dificultad. Hay un pasillo a cada lado del jugador, y una plataforma hacia adelante. Como el paso está bloqueado a la plataforma del centro, el jugador debe avanzar por ambos pasillos y encender las torres de fuego al final de cada uno. Hay todo tipo de enemigos escondidos en estas zonas. Al encender ambas torres se abre el acceso a la plataforma de en medio. Apenas llegue, tendrá dos enemigos de ataque a distancia en plataformas atacándole, y aparecerán 4 enemigos voladores. Hay 5 torres de fuego apagadas, si el jugador logra encender todas, se levantará un puente que le permitirá avanzar. Vencer a los enemigos es opcional. Una vez el jugador cruce el puente, y active la palanca, se habrá terminado la demo.

Assets modulares

Para traer a la vida estos niveles se han utilizado paquetes de *assets* gratis de la tienda de *Unreal Engine*. Estos incluyen muros, pisos, techos, cajas, y todo tipo de decoraciones. La temática en la que se ha enfocado la búsqueda de estos paquetes de *assets* ha sido ciencia ficción, sobre todo elementos que puedan dar la sensación de “mazmorra mecánica con robots” al jugador.

Los *asset packs* utilizados son: *BlinkAndDashVFX, FXVarietyPack, InfinityBlade Weapons and Effects, Modular SciFi, Polar Sci-Fi Facility*, y *Spaceship Interior Environment*.

Iluminación y optimizaciones

Una vez armados las bases de los primeros niveles, se han introducido luces y efectos de postprocesado a los mismos. Se han escogido luces cálidas, además de una luz ambiental con una intensidad decente, para garantizar la correcta iluminación de todo el mundo.

Una vez armados estos niveles, ha surgido un problema de inmensa importancia. El rendimiento del juego había disminuido. El juego estaba rozando un rendimiento de 20FPS (fotogramas por segundo) en algunas áreas. Como referencia, la realidad virtual debería correr como mínimo a 60FPS para que sea viable utilizarla. Esto ha sido un obstáculo que detuvo totalmente el progreso en la creación de niveles.

Se ha utilizado la herramienta *Unreal Insights* del propio *Unreal Engine*. Esta herramienta permite monitorizar el coste temporal de cada rutina que se ejecuta en el juego. Tras usar esta herramienta, se ha notado que la gran diferencia entre rendimientos de una zona que corría a 40FPS y una que corría a 20FPS era en la sección de renderizado, específicamente, el cálculo de sombras. A continuación, se muestra una captura *Unreal Insights*, y como su interfaz representa gráficamente el coste temporal de cada proceso del juego, y como estos afectan el rendimiento.

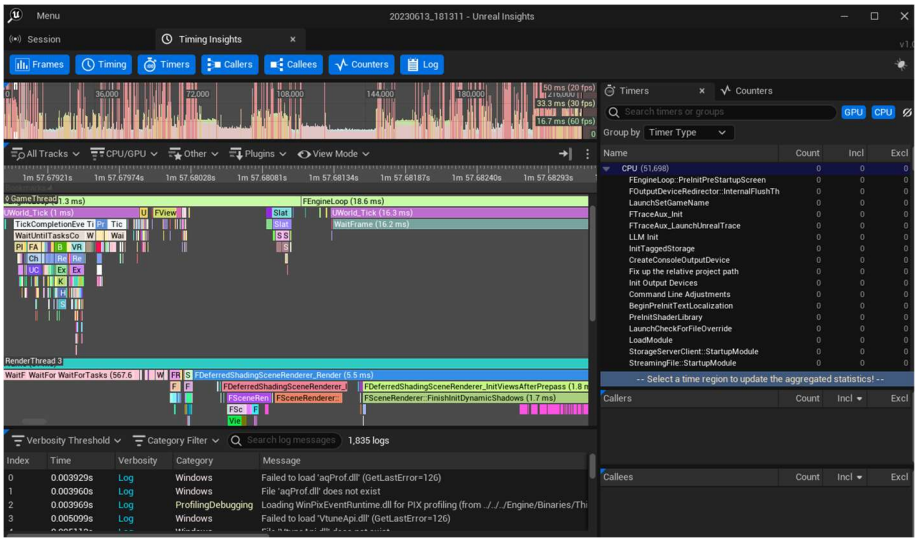

Figura 79. Screenshot de Unreal Insights
(Fuente: Elaboración propia)

De esta forma, se han removido todas las sombras dinámicas del juego, estas perteneciendo a el jugador, enemigos, y objetos interactivos.

El rendimiento había mejorado considerablemente, sin embargo, aún el juego podía optimizarse más. Se ha usado una herramienta llamada *Oculus Performance Window*, integrada en *Unreal*, la cual analiza la escena de juego y notifica de posibles problemas de rendimiento. Se ha confirmado

de esta manera que aún se podía mejorar el rendimiento de las luces, pues las luces se estaban calculando de forma dinámica y a tiempo real, lo que lleva a la siguiente optimización.

Para corregir este problema de las luces, que hacían cálculos muy costosos cada iteración del *loop* de juego, se han reemplazado las luces dinámicas por luces estacionarias. La principal característica de las luces estacionarias es que estas precalculan su iluminación y sus sombras, renderizándose las mismas a un *lightmap*, o mapa de luz, siendo una capa de texturas sobre los objetos que tienen la iluminación calculada de antemano. De esta manera, nada de esto se tiene que calcular de forma redundante decenas de veces por segundo.

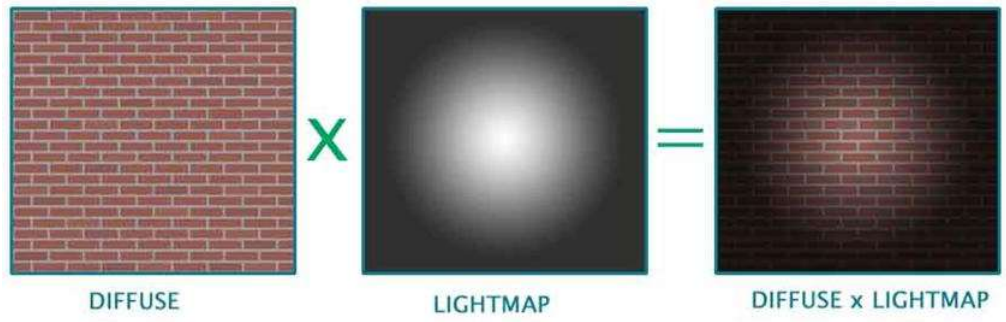

Figura 80. Ejemplo de lightmaps
(Fuente Keshav Channa:
https://www.flipcode.com/archives/Light_Mapping_Theory_and_Implementation.shtml)

El uso de luces estacionarias, *lightmaps*, y la desactivación de sombras dinámicas finalmente llevó el rendimiento del juego a su punto deseado.

Una vez resuelto este problema, se ha terminado de construir e iluminar los primeros niveles y se ha avanzado a darles lógica a los mismos, como se explicará en la siguiente sección.

Level Blueprints

Como se ha explicado en el apartado 6.1 de aprendizaje, *Blueprints* es el lenguaje de programación visual que utiliza *Unreal Engine*. Los *Level Blueprints* son una funcionalidad de *Unreal* que permite crear *Blueprints*, es decir, lógica de juego, para cada nivel en específico. Esto también se le conoce como *level scripting*. En los *Level Blueprints* se conectan mediante eventos los sistemas previamente creados, para crear una lógica específica a cada zona del juego. Como se aprecia en el siguiente *screenshot*.

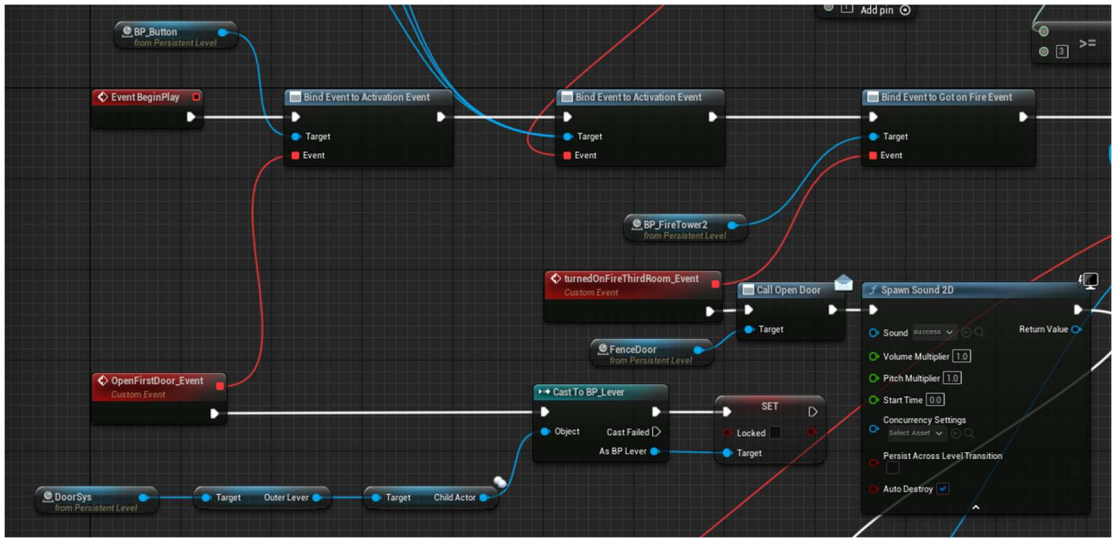

Figura 81. Ejemplo de comunicación del Level Blueprint a través de eventos
(Fuente: Elaboración propia)

Este *Level Blueprint* recoge referencias a los objetos que se han introducido al nivel, y el diseñador asigna eventos y funcionalidades extra a estos. Por ejemplo, cuando se activa un botón, este dispara el evento *OnButtonActivated*, y eso es todo. Para las puertas se ha creado un evento llamado *OpenDoorEvent*, el cual cuando es llamado, la puerta pasa de estar bloqueada a estar desbloqueada. En el *Level Blueprint* es tan sencillo como asignar un evento personalizado al *OnButtonActivated*, y establecer que cuando ocurra ese evento, se llame el *OpenDoorEvent* de la puerta en cuestión. Esto resulta en que la funcionalidad del botón y de la puerta están enlazados, y cuando se active el botón, la puerta se desbloqueará.

Por simple que sean estas funcionalidades, la existencia del *Level Blueprints* abre un inmenso abanico de opciones para poder crear todo tipo de funcionalidades y lógica especificas a cada nivel sin necesidad de cambiar el funcionamiento de los elementos individuales del juego.

Esto se utilizará para definir la lógica para completar cada nivel, sea activando botones, encendiendo antorchas, o venciendo enemigos, además de para disparar diálogos y elementos de tutorial en zonas específicas del juego.

Vida y Checkpoints

Debido a que ya se han programado los patrones de ataque de los enemigos de antemano, siendo estos los proyectiles y los ataques cuerpo a cuerpo, ya había un sistema funcional para detectar en qué momento el jugador debía recibir daño, por lo que solamente quedaba añadir dicha funcionalidad, para poder lograr un ciclo de juego en el que este pueda morir, y reaparecer en un *checkpoint*.

Se ha hecho uso de un elemento que existe en *Unreal Engine* llamado *Game Instance*, o instancia de juego. Este es un objeto que persiste desde el momento de inicio del juego hasta el fin de la ejecución del mismo. Este objeto permite almacenar información y funcionalidades. Debido a su naturaleza persistente a lo largo del juego, tiene mucha utilidad para almacenar información como *checkpoints*, vida, y configuraciones.

Se ha implementado un sistema de vida, en el que el jugador tiene ciertos puntos de vida, y un número máximo de puntos de vida. Cuando este recibe daño, la instancia de juego se encarga de restar vida al jugador, y cuando la vida se acabe, llamar a la función de muerte, que se encargará de recargar el nivel que este cargado en dicho momento. Adicionalmente, se llama a la función de muerte cuando el jugador cae por un agujero fuera del área de juego.

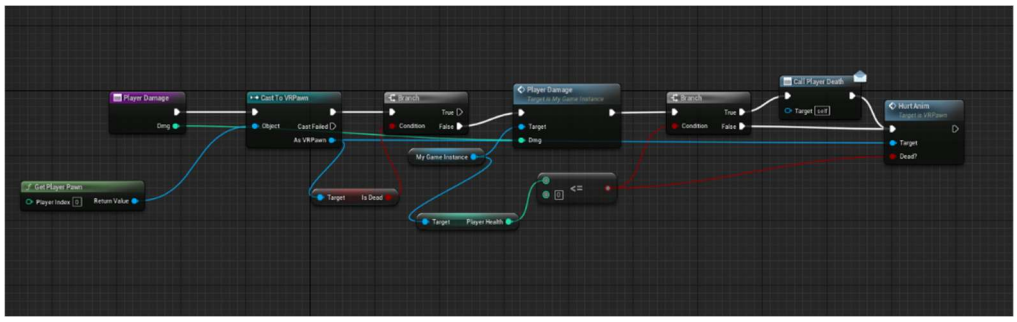

Figura 82. Blueprint de sistema de vida
(Fuente: Elaboración propia)

También se ha creado un sistema de *checkpoints*, o punto de guardado. Para lograr esto se ha añadido funcionalidad a las puertas del juego. Se ha hecho que cuando el jugador entre por una puerta, se almacene en la instancia de juego la posición del jugador en dicho momento. Esta posición quedará almacenada hasta que el jugador pase por otra puerta y sobrescriba esta variable. Al almacenar esto en la instancia del juego, independientemente de si el jugador muere y se carga de nuevo el nivel, esta información no se pierde. Entonces una vez el nivel se carga de nuevo, el jugador al ser creado chequea en la instancia de juego si hay algún *checkpoint* que haya sido almacenado, y si es el caso, se mueve el jugador a esta posición.

La siguiente captura muestra que es lo que se hace cuando el jugador pasa por una puerta. Primero se cierra la puerta detrás de él. Luego se actualiza el *checkpoint*, y se le restablece la vida al jugador.

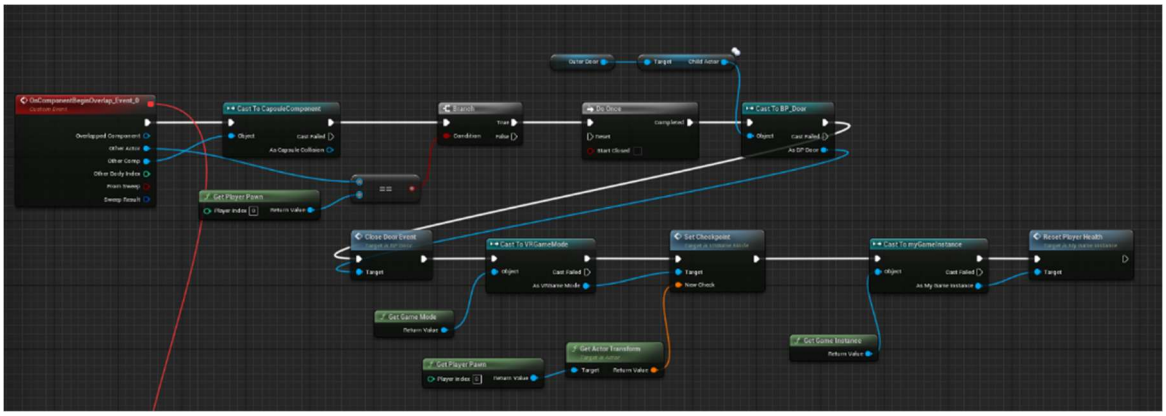

Figura 83. Implementación del checkpoint en las puertas
(Fuente: Elaboración propia)

El resultado final de este sistema es que el jugador puede recibir daño de parte de los enemigos, y cuando muere, este reaparecerá en el lugar correspondiente a la última puerta que cruzó, en estas pequeñas áreas seguras que se mencionaron cuando se habló del sistema de puertas, que se encargan de separar secciones del nivel.

Manos del jugador

Hasta este momento las manos del jugador eran representaciones gráficas de los mandos. Se ha procedido a implementar representaciones gráficas de manos al jugador, con el fin de mejorar la experiencia de juego.

Se ha utilizado como base los modelos de manos para VR creados por el artista Quinn Kuslich. Estos modelos son ofrecidos de forma gratuita.

Figura 84. Modelos de manos de Quinn Kuslich
(Fuente Quinn Kuslich: [https://www.patreon.com/QuinnKuslich/posts?filters[tag]=Animation](https://www.patreon.com/QuinnKuslich/posts?filters[tag]=Animation))

Se ha editado en el programa *Blender* el modelo de estas manos, para adaptarlas mejor a la narrativa y tono del juego. Se tuvo la intención de crear unas manos con unos “guantes con poderes

magnéticos", que tuvieran detalles que cambiaran de color dependiendo de ciertos factores dentro del juego. A continuación, se muestra el resultado final de dicho modelo, y se explicará cómo se ha conseguido.

Figura 85. Modelo final de la mano, hecho a partir del modelo de Quinn Kuslich
(Fuente: Elaboración propia)

Primero se ha importado el modelo de la mano a *Blender*, y se ha editado su modelo. Se ha reemplazado el dedo robótico por un dedo normal y se ha añadido media esfera en la parte trasera de la mano.

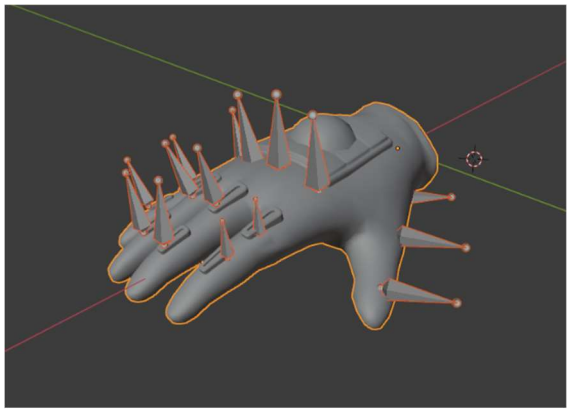

Figura 86. Huesos de la mano
(Fuente: Elaboración propia)

A continuación, se editaron los huesos de la mano para que coincidan con el nuevo poligonaje. Se ha tenido que hacer un *skinning* desde cero a la mano, debido a que los cambios en poligonaje habían sido muy grandes. Esto significa que se ha tenido que definir cómo se comporta el modelo en reacción a los huesos definidos. Después de este tedioso progreso, se ha logrado el resultado deseado.

Se ha importado a *Unreal* los modelos de dichas manos y se les han asignado las animaciones ofrecidas por el artista, que sorprendentemente, funcionaron de forma correcta a pesar de todos los cambios que habían sufrido la malla y los huesos del modelo. Adicionalmente, se asignaron materiales de cuero para una mejor estética.

Se ha usado el árbol de animaciones de *Unreal* para poder definir qué pose de manos poner en cada momento que respondieran a los botones y áreas del mando que el jugador toque. Este árbol de animaciones funciona como una máquina de estados que permite configurar los distintos estados y las transiciones entre ellos.

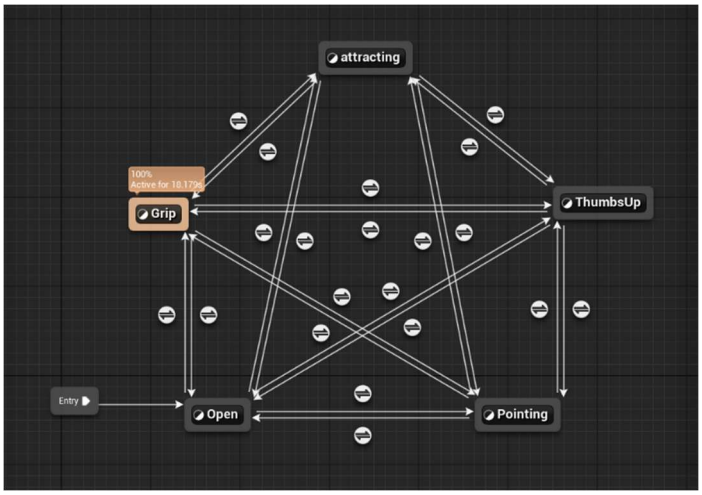

Figura 87. Árbol de animación de las manos
(Fuente: Elaboración propia)

Finalmente se ha implementado el sistema de cambio de color a los detalles oscuros del guante. Estos servirán para informar al jugador. Cuando estén estos detalles en negro, significa que el jugador no puede atraer ningún objeto con esa mano usando sus poderes magnéticos, cuando estos detalles estén iluminados significa que el jugador tiene un objeto enlazado a esa mano y puede atraerlo, y finalmente cuando los detalles se iluminen de forma muy aclarada, esto señala que el jugador está atrayendo el objeto. Cuando el jugador este sujetando un objeto, la mano hará animación de estar cerrada. Adicionalmente se ha usado una animación a medias, entre la animación de la mano cerrada y la mano abierta, dando una impresión de mano semiabierta cuando el jugador esté atrayendo algo. Y para concluir, la mano derecha se ilumina de color azul, y la mano izquierda de color rojo.

Animaciones de los enemigos

A este punto los enemigos tienen animaciones físicas que reaccionan a los ataques del jugador, y caen cuando son vencidos. Debido a que las animaciones físicas eran la parte complicada, se han

hecho mucho más temprano durante el desarrollo, y se ha decidido crear las animaciones convencionales en este punto.

Se ha utilizado *Control Rig*, una herramienta que te permite crear controladores para un modelo dentro de *Unreal*, y crear animaciones dentro del mismo editor, sin tener que importarlas de otros programas. *Control Rig* posee muchas otras funciones, pero se mencionan solo las que se han utilizado en este trabajo.

Se ha creado un *Control Rig* para el modelo de enemigo terrestre y uno para el enemigo volador.

Para el enemigo volador ha sido fácil, pues solo se ha tenido que crear una animación de aleteo, las animaciones físicas y movimiento del enemigo se encargan del resto.

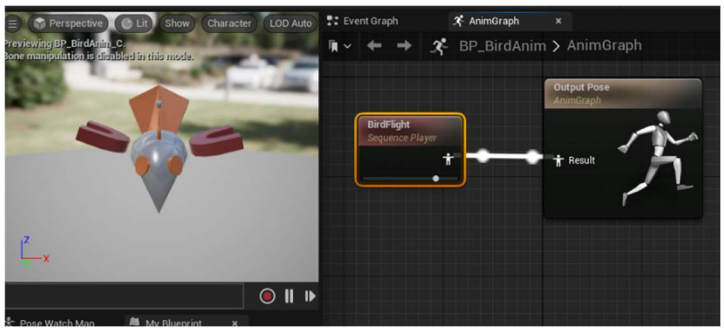

Figura 88. Animation Blueprint del enemigo volador
(Fuente: Elaboración propia)

Para el enemigo terrestre se ha creado una animación para cuando esta quieto, una para cuando camina, una para cuando golpea, y una en la que levanta los brazos cuando ve al jugador.

Este ha utilizado un árbol de animaciones más complejo, en el que se combinan dinámicamente distintas animaciones dependiendo de la situación.

Cuando el enemigo no ve al jugador, este debe tener los brazos abajo, y cuando ve al jugador, debe levantarlos, por lo que se ha usado un nodo llamado *Layered Blend Per Bone*, que permite mezclar animaciones distintas separándolas por partes del cuerpo. Se ha utilizado esto para que cuando no ve al jugador tenga los brazos de la posición quieta, y cuando lo encuentra, levanta los brazos. Esto solamente afecta los brazos.

Para la locomoción del enemigo se ha usado una animación de caminar, y la velocidad de esta animación se rige por qué tan rápido se mueve el enemigo en cada momento, dando una ilusión convincente del movimiento. Cuando el enemigo vaya a golpear este ejecutara la animación de golpe en su totalidad.

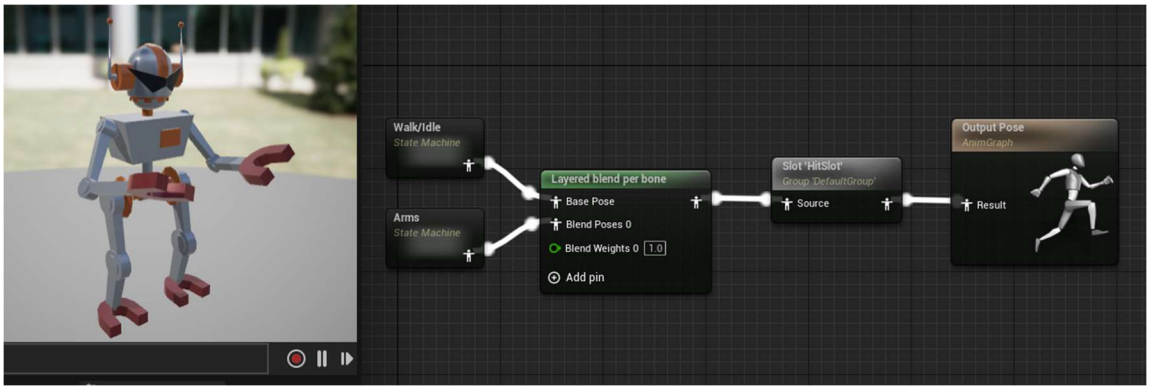

Figura 89. Animation Blueprint del enemigo cuerpo a cuerpo terrestre
(Fuente: Elaboración propia)

Como nota adicional, los enemigos que atacan a distancia siempre tendrán los brazos levantados, debido a que están sujetando un imán.

Estas animaciones funcionan de forma adecuada de forma simultánea a las animaciones físicas, es más, resaltan de forma positiva las animaciones físicas.

VFX y partículas

Se han creado una serie de efectos visuales y partículas para mejorar la experiencia de juego. Estos incluyen efectos de fuego, efectos especiales en las manos y los objetos cuando están siendo atraídos por el jugador, efectos de daño, chispas y explosiones a los enemigos, efectos visuales cuando se activa un botón y efectos de laser cuando se crean objetos en los dispensadores.

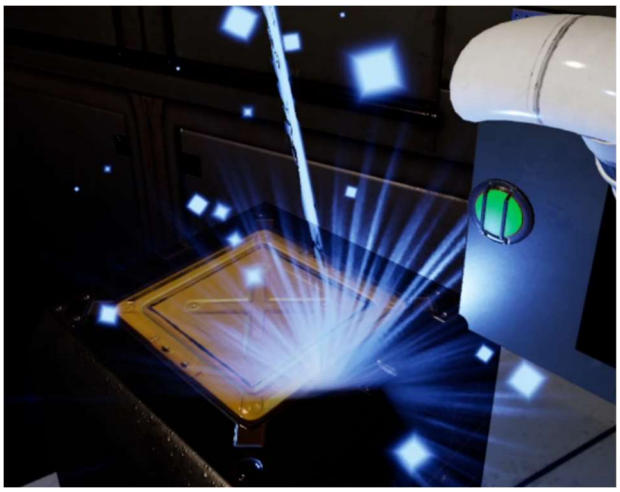

Figura 90. Efectos visuales del dispensador materializando un arma
(Fuente: Elaboración propia)

Todos estos efectos previamente mencionados han sido creados a partir de paquetes de partículas gratis de la tienda de *Unreal*, en los que se han editado dichos efectos para darles el aspecto deseado dentro del juego. Los paquetes utilizados son *BlinkAndDashVFX, FXVarietyPack*, e *InfinityBladeEffects*.

A continuación, se explicarán los efectos creados de forma manual desde cero.

Se ha utilizado un widget bidimensional sobre la cámara del jugador para poder posicionar elementos 2D en la vista del jugador. Se ha creado un efecto de difuminado a negro para transiciones entre escenas, en el que se posiciona una imagen totalmente negra en esta pantalla del jugador, y se anima para que se difumine, afectando su opacidad a lo largo del tiempo. Similarmente se ha utilizado un efecto similar, pero con una pantalla roja por debajo de la negra, que dará un pequeño difuminado rojo cuando el jugador recibe daño. Cuando el jugador muere, se difumina la pantalla a rojo con una opacidad a medias para que pueda observar su entorno durante unos pocos segundos, luego se difumina todo a negro y se reinicia el nivel.

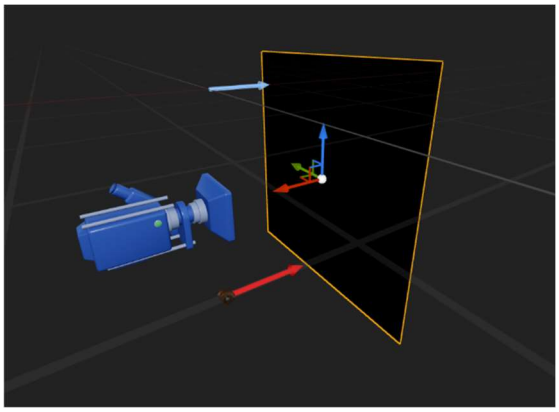

Figura 91. Widget 2D que se encarga de oscurecer y cubrir la cámara
(Fuente: Elaboración propia)

Finalmente se ha creado un emisor de partículas utilizando la configuración de renderizado de en forma de lazo, que permite dibujar efectos de partículas siguiendo un camino continuo. Este efecto se ha pintado de blanco y se le ha dado un corto tiempo de vida, adicionalmente se ha hecho que la línea dibujada en su trayecto se haga más delgada con el tiempo. El resultado es una línea de acción color blanca que demarca el camino que han seguido los objetos arrojados. Esto permite dar una visibilidad más clara de la posición y trayectoria de los objetos arrojados, además de ser visualmente gratificante.

Figura 92. Emisor de partículas de la línea de movimiento
(Fuente: Elaboración propia)

Figura 93. Demostración del efecto de línea de movimiento
(Fuente: Elaboración propia)

Sonido

En un juego de realidad virtual la inmersión es fundamental, por lo que un buen diseño de sonido es esencial en este tipo de juegos.

Se han usado exclusivamente sonidos de la página '*freesound.org*' la cual ofrece una gran biblioteca de sonidos bajo la licencia de *Creative Commons*, la cual permite su uso, distribución y edición. Muchos de los sonidos utilizados en el juego han sido editados a través el uso de Audacity para ajustarlos a las necesidades del juego en todo momento.

En grandes rasgos, se han añadido sonidos ambientales para el juego, incluyendo sonido de pasos al jugador, sonidos de los objetos cuando chocan y caen, efectos robóticos a los enemigos, sonidos

que indican cuando se abre una puerta o resuelve un puzle, sonidos de laser para los dispensadores, sonidos mágicos con toques tecnológicos cuando el usuario utiliza sus guantes magnéticos, sonidos a las palancas y puertas, efectos de fuego a las torres de fuego y la espada de madera, y efectos de sonido de viento.

Todo esto se ha logrado utilizando la función *Spawn Sound* de *Unreal*, que permite crear sonidos tanto en posiciones como hacer que sigan al componente de algún actor. Esta función da la libertad de escoger cuando se elimina el sonido y como se escuchará en el juego.

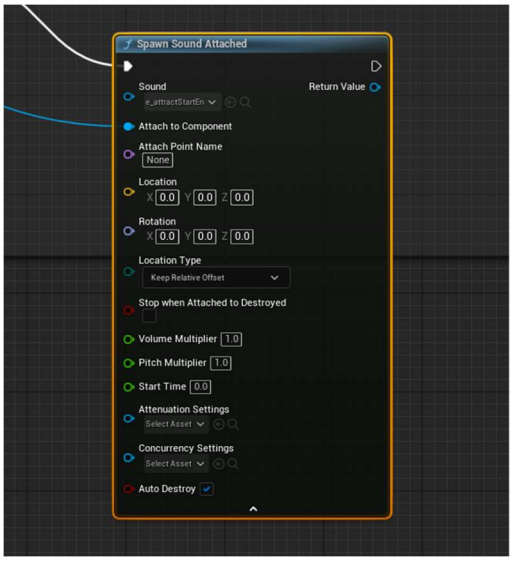

Figura 94. Ejemplo de Spawn Sound
(Fuente: Elaboración propia)

Cabe destacar el efecto de viento que se ha hecho para las armas y el jugador, en principio, estas son un sonido de viento que suena a mayor volumen y con un tono más alto mientras más rápido se mueven estos mismos objetos. Para las armas esto da el efecto de cortar el viento cuando lanzas algo o cuando lo mueves con fuerza. En el jugador, esto da el efecto de que cuando el jugador está cayendo, se escucha el viento. Esto se logró con un plugin de Unreal llamado *MetaSound*, el cual funciona con nodos y permite diseñar de forma precisa el sonido a partir de simples y parámetros. En el caso del sonido de viento, simplemente se envía como parámetro la velocidad del objeto, y se hace que a partir de cierta velocidad emita sonido a partir de un clip de sonido de viento, y mientras más velocidad tenga se aumenta la altura del sonido, además de su volumen. Adicionalmente, también se le añade ruido rosa para que tenga un efecto más natural con el cambio de volumen y tono. Esta técnica ha sido inspirada por una técnica similar utilizada por el *youtuber PrismaticaDev* en su tutorial de *MetaSounds* (PrismaticaDev, Intro to MetaSounds in Unreal Engine! [New UE5 Series], 2022).

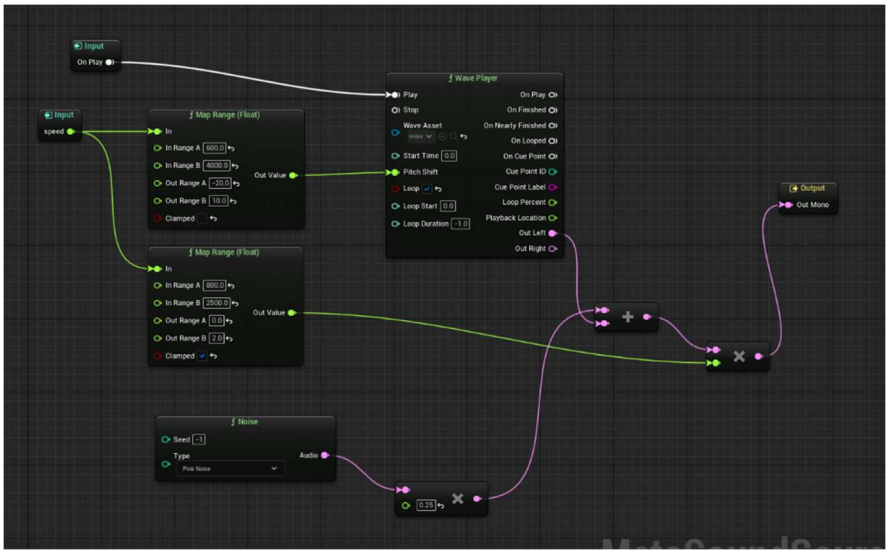

Figura 95. Metasound del sonido de viento
(Fuente: Elaboración propia)

Diálogos

Acercándose al fin de la producción del juego, ya se tienen niveles, un *loop* de juego, sonido, y toda clase de pulidos. A partir de ahora solamente quedan dar los toques finales para que esta sea una experiencia completa y jugable de comienzo a fin.

Hablando de la historia del juego por un momento, se resume en lo siguiente: El jugador es un retador que se adentra en la mazmorra del rey de los imanes, y tiene el objetivo de superar sus retos y llegar al fin de la mazmorra para vencerlo, y poder ser nombrado el próximo rey de los imanes. Para darle a todo esto un toque más cómico, el personaje del rey de los imanes estará hablándole al jugador a través de altavoces durante el juego. Los diálogos del rey de los imanes le explicarán al jugador la historia, dará pistas sobre las mecánicas y como avanzar, y finalmente, dará un elemento cómico de entretenimiento a lo largo del juego, dado que este personaje sigue un estereotipo de villano caricaturesco al que todo le sale mal.

Se ha escrito un guion con todos los diálogos de este personaje, y se han grabado. A pesar de que la actuación de voz de un programador no es lo óptimo, para efectos del proyecto, va a ser suficiente. A estas voces se le han dado efectos de radio para resaltar el hecho que se escucha a través de altavoces.

En *Unreal* se ha creado un gestor de diálogos, que garantizará una serie de cosas esenciales en el juego. Primero, que los diálogos solo puedan sonar uno a la vez, es decir, si el jugador se adelanta y dispara un diálogo del personaje mientras este está diciendo otra línea, en vez de solaparse los diálogos, debería detenerse el primero y comenzar el nuevo. Segundo, que las voces estén situadas cerca del jugador en el espacio tridimensional para asegurar que siempre se podrán escuchar, y

finalmente, que haya un efecto de sonido antes de que el personaje comience a hablar y otro cuando este termine de hablar, esto indicará al jugador cuando comienza y termina un diálogo.

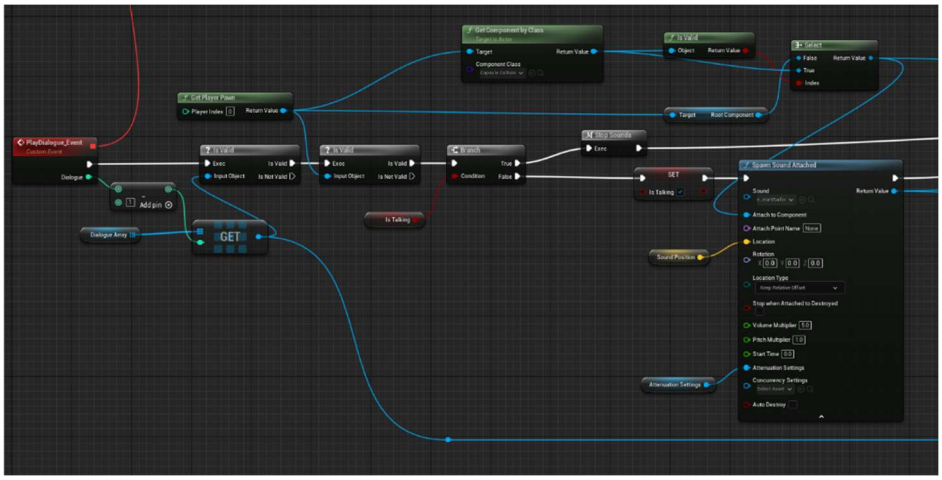

Figura 96. Blueprint del gestor de diálogos
(Fuente: Elaboración propia)

Simplemente se ha creado un evento que disparará los diálogos, y en el *Level Blueprint* se llama a este evento cuando haga falta.

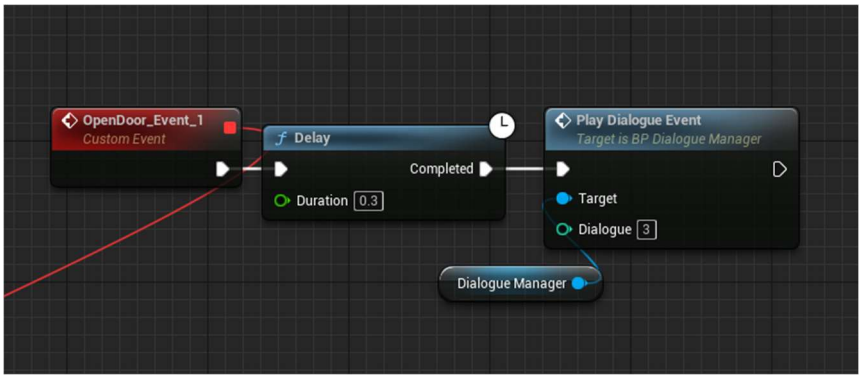

Figura 97. Ejemplo de llamada al gestor de diálogos
(Fuente: Elaboración propia)

Escena del ascensor

Para darle un comienzo más natural al juego, en vez de que el jugador aparezca de la nada al comienzo del juego, se ha puesto al jugador en un ascensor. En este ascensor estará bajando el jugador cuando el rey de los imanes comienza a hablar y se presenta. En este diálogo se le da a entender la historia, por qué está ahí, y que es lo que tiene que hacer. Una vez el personaje termina de hablar el ascensor se detiene con un sonido duro, como si algo se hubiera roto. Luego salen un par de chispas eléctricas en el suelo, y el ascensor empieza a caer. Esto tiene la intención de dar un susto al jugador, sacando provecho de la realidad virtual para elevar situaciones terroríficas como

estar en un ascensor que está cayendo. Finalmente, este cae sobre una plataforma con un fuerte golpe. Y entre la cortina de humo resultante se revela la primera habitación del juego.

Esto se ha logrado con el uso de timelines, funciones de *Unreal* que te permiten hacer animaciones simples, como la del ascensor cayendo, y con eventos asociados como el evento de cuando salen las chispas o empieza a caer el ascensor. Se han usado sistemas de partículas como los mencionados anteriormente para hacer los efectos visuales, y se han usado efectos de sonido con la función *Spawn Sound*.

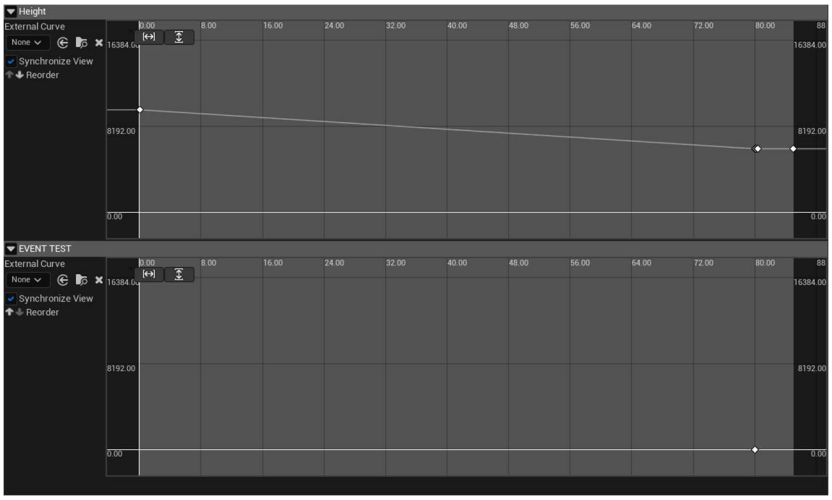

Figura 98. Ejemplo de timeline que anima al ascensor
(Fuente: Elaboración propia)

Menús, configuraciones, y tutoriales

Finalmente tenemos el juego terminado, ahora solo queda envolverlo, para poder iniciarlo y jugarlo de comienzo a fin, como es el objetivo.

Se ha creado un nuevo nivel con una habitación vacía y alguna decoración. Se usará esta como menú de inicio. Aquí se ha hecho una variación del jugador en la que este no se puede mover, ni hacer ninguna acción relativa al *gameplay*. Se han creado elementos 2D de menú posicionados en el mundo 3D, y se ha utilizado el *Widget Interaction* que ya venía implementado con la plantilla de VR para que el jugador pueda interactuar con este menú. El jugador simplemente con apuntar con las manos, puede interactuar y presionar botones del menú. Se ha creado un título del juego en 3D en *Blender* y se ha introducido en el juego. El menú resultante es el siguiente:

Figura 99. Menú de inicio
(Fuente: Elaboración propia)

Se ve el título del juego, los créditos en la parte inferior, y a la derecha el panel con todo lo interactivo. Un botón de iniciar juego, que cargará el juego y hará que el jugador aparezca en el ascensor del comienzo. Una caja que, si marcas, cambiará la rotación de la cámara, para que cuando el jugador voltee su cámara usando los mandos, se haga de forma suavizada con movimiento realista, o se haga inmediatamente rotando la cámara 45 grados a la dirección escogida. Esta última configuración es importante debido a que girar la cámara de forma suavizada puede resultar mareante para muchos usuarios, por lo que la opción de giro inmediato es una buena alternativa a tener. Nos encontramos también un *slider* de volumen, que permite cambiar el volumen del sonido del juego. Finalmente está el botón para salir del juego.

Para hacer la habitación de fin de juego, se ha duplicado esta habitación de menú principal, y cambiado los paneles por uno que pone “Fin de la demo. Gracias por jugar.” Seguido de un botón para ir al menú principal, y uno para salir del juego.

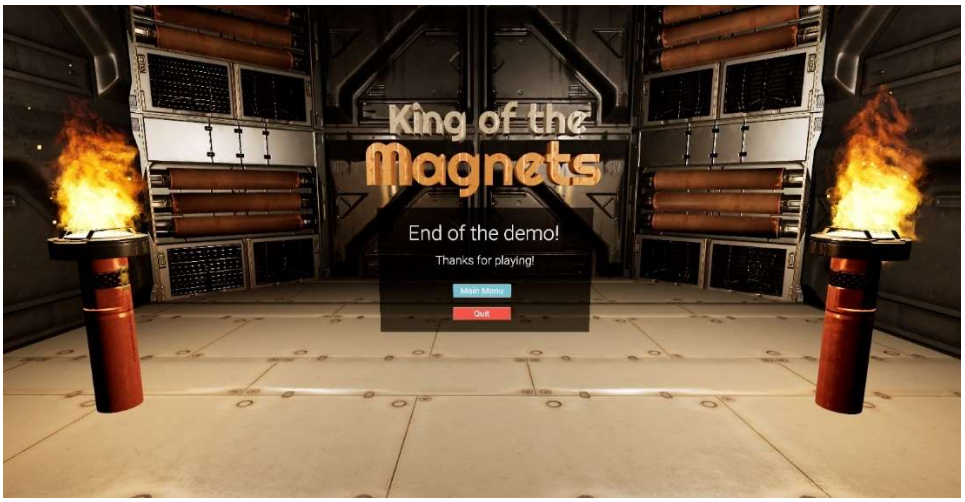

Figura 100. Menú de final de juego
(Fuente: Elaboración propia)

Esta habitación será cargada cuando el jugador complete todos los niveles que hay actualmente en el juego.

Finalmente se ha utilizado el sistema de paneles flotantes para crear paneles de tutorial en las primeras zonas del juego, que enseñarán al jugador los controles para poder moverse, agarrar, lanzar, y atraer objetos.

Figura 101. Demostración de los paneles de tutorial en el juego
(Fuente: Elaboración propia)

Conclusión de la producción

Se ha logrado el objetivo de hacer una experiencia lineal jugable de comienzo a fin. En esta etapa se ha producido todos los elementos que hacen que el producto pase de ser un simple prototipo, a un producto real. Sin embargo, por falta de tiempo, el juego ha quedado corto en número de habitaciones y tiempo de juego, por lo que se ha decidido dejar hasta donde se ha podido, siendo un demo de los primeros niveles del juego. Esto sirve mucho como un corte vertical que muestra

de manera pulida y terminada las primeras áreas de este juego, a falta simplemente continuar y crear más contenido.

Por todo esto, y todo lo explicado en el apartado de producción, esta etapa se considera un éxito, cumpliendo con los objetivos y las expectativas que se han tenido para que este juego cumpla.

7. Conclusiones

7.1. Resultado del proyecto

Tras todas las fases de desarrollo y pulido, se puede concluir que el proyecto ha sido un éxito rotundo. Se ha obtenido un producto jugable que cumple con las expectativas iniciales del proyecto y con el estándar de calidad proyectado desde un principio. A continuación, se resumirá las características del producto obtenido.

Se ha desarrollado un videojuego de realidad virtual en *Unreal Engine*. El título del juego es *King of the Magnets*. Es un juego de acción y puzles que utiliza la original mecánica de atraer objetos como si fueran un bumerang después de haber sido lanzados. El juego se puede jugar de comienzo a fin, y ofrecer una experiencia entretenida e inmersiva, sacándole máximo provecho a las ventajas de la realidad virtual para evocar emoción al jugador.

El juego comienza con un menú de inicio. Este menú, y todos los otros dentro del juego, tienen un acabado visual pulido y agradable. Se ha tenido sumo cuidado en que toda parte de la experiencia tenga un acabado profesional.

Una vez comienza la experiencia, el juego tiene un paso adecuado a la hora de contar la historia, enseñar las mecánicas, y presentar distintos elementos del juego en forma de tutoriales.

Cada nivel del juego tiene un propósito que se siente natural conforme se avanza en el juego, sumado a el arte de entornos que se adecua a la historia y contribuye a la sensación de inmersión y aventura.

El juego saca el máximo provecho de las ventajas de la realidad virtual, utilizando elementos visuales y auditivos para generar una sensación de inmersión. Se utilizan efectos visuales y partículas que demuestran la acción que está ocurriendo en el juego. Se utiliza el audio para dar una mayor inmersión e intensificar cada momento del juego, sea el sonido ambiental de mazmorra mecánica mientras se explora, o los sonidos intimidantes de los enemigos y los golpes durante escenas de acción. Todo esto contribuye a una sensación general de inmersión en el mundo del juego.

La experiencia de juego y diseño de niveles contribuye a generar en el jugador distintas emociones a la hora de jugar. Esto incluye la emoción y adrenalina a la hora de combatir con enemigos, esquivando proyectiles, dando espadazos, sumado a la satisfacción de lanzar una espada y darle a

un enemigo, cayendo éste entre chispas y explosiones para que la espada vuelva a tu mano y la atrapes en el aire sin dificultad.

También se saca provecho de la inmersión en el juego para generar sentimiento de intimidación en el jugador: Encontrarse en un ascensor que está cayendo, luego explorar lugares peligrosos y desconocidos, encontrarse con enemigos más altos que el jugador, y en ocasiones sentir sensaciones extremas como caer de una gran altura, o encontrarse encerrado en una habitación claustrofóbica con robots hostiles.

Una vez se termina el juego, el jugador aparece en un menú indicando que ha terminado la demo, pues el juego en un futuro podría tener más contenido profundizando en las mecánicas y enemigos existentes actualmente. Sin embargo, la experiencia tiene suficiente complejidad y pulido para poder considerarse una experiencia jugable en todo su derecho.

El uso de *Unreal Engine* 5 ofrece al juego un acabado visual muy profesional, con sistemas de iluminación y efectos visuales que hacen el juego una experiencia agradable.

En conclusión, se ha conseguido desarrollar un juego de realidad virtual de forma satisfactoria. Este juego contiene todo lo esperado para una experiencia de acción en realidad virtual, y tiene un acabado visual y mecánico digno de ser un producto que se pueda disponer en el mercado.

Figura 102. Captura de escena de acción
(Fuente: Elaboración propia)

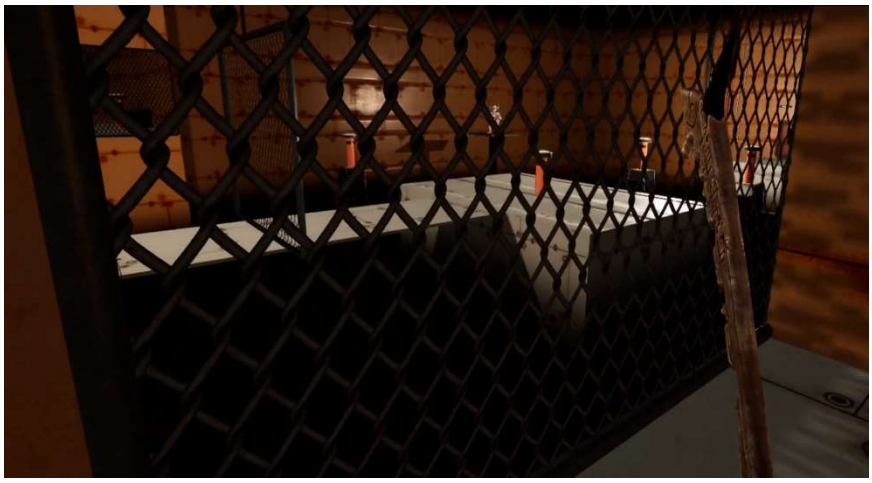

Figura 103. Captura de escena de exploración
(Fuente: Elaboración propia)

Figura 104. Captura de escena de puzles
(Fuente: Elaboración propia)

7.2. Revisión de objetivos

A continuación, se hará una revisión de todos los objetivos declarados en este documento, y se proporcionará una breve conclusión del resultado de cada uno en el producto final, y, en definitiva, en qué medida se ha cumplido cada objetivo. Se destaca que estos objetivos están ordenados por importancia, significando que el primer objetivo mencionado es el más principal e importante, y el último mencionado será más complementario y secundario.

Desarrollo de un producto jugable

Como objetivo principal de este proyecto, se considera que este apartado se ha superado con creces. Como se ha mencionado en el resultado del producto, el juego se puede jugar de comienzo

a fin, con una progresión natural entre escenas, jugabilidad e historia, sumado a que posee mecánicas pulidas, y elementos audiovisuales acabados. Todas estas son características de un juego que se puede encontrar en el mercado, y está un paso más allá de considerarse un mero prototipo, por lo que el desarrollo de un producto jugable, el objetivo principal de este proyecto, se considera un logro.

Aprendizaje sobre *Unreal Engine* y realidad virtual

El segundo objetivo de este proyecto ha sido el aprendizaje de la herramienta de *Unreal Engine* 5, además de dominar el funcionamiento de la tecnología de la realidad virtual en sus distintas aplicaciones.

El aprendizaje de *Unreal Engine* se ha logrado, debido a que se ha tenido que desarrollar este producto con todas sus etapas de producción enteramente en *Unreal Engine*, desde el prototipado hasta el producto final. Naturalmente, el proceso de haber logrado todo esto habiendo empezado de un punto de partida sin conocimiento alguno de cómo funciona esta herramienta, concluye un éxito en este objetivo, con un enorme aprendizaje no solo de la herramienta, sino también de las distintas fases de producción, pasando por todos los aspectos relevantes del desarrollo de un producto de estas características. El conocimiento general obtenido de todas las funciones que ofrece *Unreal Engine* cumple satisfactoriamente lo deseado de este proyecto.

En cuanto a la realidad virtual, para el desarrollo de las mecánicas, se ha profundizado en el funcionamiento de estos dispositivos, tanto a nivel técnico como a nivel de diseño. En el aspecto tecnológico se ha tenido que lidiar con la manera en la que el motor de juego recibe la información del dispositivo de realidad virtual. Con sus ventajas e inconvenientes se ha tenido que diseñar sistemas, como el sistema de movimiento y el de agarre/lanzamiento. Estos sistemas han necesitado una comprensión de la información que proporcionan los dispositivos de realidad virtual, como la posición y rotación de cada componente, incluyendo la frecuencia con la que se recibían datos. A nivel de diseño se ha tenido que comprender la experiencia de usuario en entornos virtuales, y aprender las mejores prácticas para crear un juego divertido e inmersivo, que no maree ni intimide al jugador más de lo necesario, resultando en una experiencia cómoda y correcta.

Por todas las razones previamente mencionadas, se considera un éxito el aprendizaje de *Unreal Engine* y realidad virtual. El producto resultante sirve como prueba del dominio conseguido en este lapso de tiempo.

Prototipado y pulido de mecánicas

Siendo un objetivo que es consecuencia del objetivo principal, se considera exitoso debido a que, para la realización del juego terminado, se ha pasado por la etapa de prototipado y pulido de mecánicas. Esta etapa ha sido el fundamento sobre el cual el producto final se soporta. Por ende, este objetivo ha sido logrado, superando las expectativas iniciales.

Profundizado en técnicas de *technical art*

Durante el desarrollo, todo el aspecto de *technical art* ha quedado en un segundo plano, comparándolo con el desarrollo de mecánicas. Sin embargo, aún se han obtenido una serie de logros dignos de mencionar. Se han creado animaciones físicas en los enemigos que funcionan de manera semi procedural y responden de manera natural al daño que reciben. También se ha trabajado en el desarrollo de materiales que generen la ilusión de niebla. Se ha trabajado de forma intensiva en la iluminación, la optimización de la misma, y la utilización correcta de la misma para dar los mejores resultados posibles con un ahorro en coste computacional considerable. Finalmente se ha hecho mucho hincapié en los sistemas de partículas y efectos visuales, los cuales ofrecen ese acabado visual final y pulido. A pesar de que no se han explorado los *shaders* o técnicas que ahonden en el motor gráfico, los logros obtenidos son un éxito que juegan un rol considerable en hacer que el juego sea semejante producto.

Publicación de un producto comercial

Finalmente, como último objetivo, la publicación comercial del producto. A pesar de que el producto cumple con los estándares de calidad para poder ser vendido en plataformas de distribución, el juego carece de contenido y tiempo de juego para poder ser considerado algo más que una demostración. Sin embargo, esta demo posee cualidades que, por su cuenta, son merecedoras de ser publicadas, aunque sea de forma gratuita. Por esta razón, a falta de más contenido, esta demostración sirve como anticipación de lo que se puede esperar de una versión completa y terminada de este juego.

7.3. Retrospectiva sobre la metodología

La metodología de 3 fases: aprendizaje, prototipado y producción ha funcionado de maravilla. Cada fase ha cumplido de forma satisfactoria lo que se esperaba obtener de cada una, y ayudaba mucho a enfocar el trabajo en los aspectos importantes de cada fase, evitando distracciones o trabajo redundante.

Cabe destacar que el único imprevisto de esta metodología recae en el hecho de que la etapa de aprendizaje ha tomado mucho más tiempo del esperado. El proyecto ha sido comenzado en octubre, y ha sido terminado en julio, un total de 9 meses. La proyección inicial que se tenía era 2 meses de aprendizaje, 4 meses de prototipado, y 3 meses de producción. Sin embargo, la fase de aprendizaje tomó mucho más tiempo de lo esperado, siendo alrededor de 4 meses, dejando menos tiempo para las dos etapas subsiguientes. Esto se debe a que se infraestimó la complejidad de *Unreal Engine* y el tiempo necesario para dominarlo. Sin embargo, esto no afectó al proyecto de forma negativa, dado que el desarrollo ha sido mucho más fluido y cómodo con la experiencia que habían tomado esos 4 meses de aprendizaje. Una vez completada la fase de aprendizaje, se ha llevado el prototipado y la producción de forma fructífera.

7.4. Trabajo futuro

Como se ha mencionado anteriormente, a pesar de que el resultado obtenido ha sido un éxito, el juego necesita más contenido para que tenga más tiempo de juego, se puedan explorar las mecánicas a profundidad, y se pueda completar la historia. Por esta razón el juego por ahora se denomina simplemente como una demo, y el trabajo futuro constaría en crear más niveles, una pelea de jefe final contra el rey de los imanes, y concluir la historia. De esta manera, el juego será un producto comercial completo y terminado, que se podrá vender en el mercado.

Glosario

A

Asset: Recursos que se usan en el videojuego, ya sean modelos 3D, animaciones, texturas, etc.

Asset pack: Paquete de assets.

B

Behavior tree: Arbol de comportamiento.

Blackboard: Pizarra.

Blend: Mezcla.

Build: Resultado ejecutable creado a partir del código compilado del proyecto.

Ch

Character: Personaje.

Checkpoint: Punto de guardado o punto de control en un videojuego.

C

Constraint: Restricción.

F

Feedback: Señales en el juego que informan al jugador.

Frame: Frame; Fotograma.

G

Game design: Diseño de juegos.

Game feel: Sensación del juego.

Gameplay: Jugabilidad.

H

Hitbox: Caja de colisión.

I

Input: Entrada de información de parte del usuario.

J

Joystick: Palanca de control.

L

Level scripting: Secuencia de comandos para niveles

Lightmap: Mapa de iluminación.

Loop: Ciclo.

P

Pathfinding: Busqueda de caminos.

Placeholder: Provisional.

R

Ragdoll: Simulación de físicas de un personaje en el que imita un personaje muerto sin movimiento y es afectado por las físicas del entorno.

Rig: Modelo 3D con capacidad de mover articulaciones.

Rigging: Proceso de preparación de un rig.

S

Screenshot: Captura de pantalla.

Shaders: Sombreador. Programa informático que realiza cálculos gráficos.

Skinning: Definir el comportamiento de una malla 3D en respuesta al movimiento de sus respectivos huesos.

Slider: Deslizador.

T

Technical art: Arte técnico.

V

Viewport: Ventana de la interfaz.

Referencias

Dean, B. (27 de Marzo de 2023). *Steam Usage and Catalog Stats for 2023*. Obtenido de Backlinko: https://backlinko.com/steam-users#steam-games-by-genre

Delgado, M. (1 de 7 de 2022). Meta Quest 2 se convierte en el headset VR más exitoso con 14,8 millones de unidades vendidas. *Vandal*, págs. https://vandal.elespanol.com/noticia/1350754902/meta-quest-2-se-convierte-en-el-headset-vr-mas-exitoso-con-148-millones-de-unidades-vendidas/ . Obtenido de https://vandal.elespanol.com/noticia/1350754902/meta-quest-2-se-convierte-en-el-headset-vr-mas-exitoso-con-148-millones-de-unidades-vendidas/

Epic Games. (Abril de 2022). *Unreal Engine*. Obtenido de https://www.unrealengine.com/es-ES

Fortune Business Insights. (Septiembre de 2021). *Virtual reality gaming market 100271*. Obtenido de Fortune Business Insights: https://www.fortunebusinessinsights.com/industry-reports/virtual-reality-gaming-market-100271

Just2Devs. (28 de Septiembre de 2021). *VR Character in Unreal Engine 4*. Obtenido de Youtube: https://youtu.be/pX-j4HQNAiY

Lewis, K. (26 de Julio de 2019). *How to Improve Throwing Physics In VR (feat. Snowball Showdown)*. Obtenido de Karl Lewis Design: https://karllewisdesign.com/how-to-improve-throwing-physics-in-vr/

Lowood, H. E. (16 de Junio de 2023). *virtual reality*. Obtenido de Britannica: https://www.britannica.com/technology/virtual-reality

Meta. (2023). *Quest 2*. Obtenido de Meta: https://www.meta.com/es/quest/products/quest-2/

PrismaticaDev. (26 de Noviembre de 2022). *Intro to MetaSounds in Unreal Engine! [New UE5 Series]*. Obtenido de Youtube: https://youtu.be/92RIT7tObDg

PrismaticaDev. (19 de Febrero de 2022). *Physical Animation: The Ultimate Starter Guide [UE4/UE5]*. Obtenido de Youtube: https://youtu.be/46NfgXlnCzM

Rogers, S. (2010). *Level Up! The Guide to Great Video Game Design.* Wiley.

Schell Games. (13 de Diciembre de 2016). *I Expect You To Die*. Obtenido de Steam: https://store.steampowered.com/app/587430/I_Expect_You_To_Die

Stress Level Zero. (29 de Septiembre de 2022). *BONELAB*. Obtenido de Steam: https://store.steampowered.com/app/1592190/BONELAB

SUPERHOT Team. (25 de Febrero de 2016). *SUPERHOT VR*. Obtenido de Steam: https://store.steampowered.com/app/617830/SUPERHOT_VR