I.E.S. Clara del Rey
Comunidad de Madrid

UNIÓN EUROPEA
Fondo Social Europeo
El FSE invierte en tu futuro

Blinded

by

CIMA
PRODUCTIONS

Ciclo Formativo de Grado Superior
Desarrollo de Aplicaciones Multiplataforma

Autores
Francisco de Asís Ciudad Arranz, Mario López Moraga, Mateusz Yatsyk Szumilas

Tutor
Luis Manuel Moreno Lara

Coordinadores
José Luis López Álvarez, Luz Susana Canelo Oliva

I.E.S. Clara del Rey

ÍNDICE DE CONTENIDOS

No se encuentran entradas de índice.

INDICE DE ILUSTRACIONES

INTRODUCCIÓN

Este proyecto parte de la idea de buscar nuevos retos para los integrantes del equipo, tanto a nivel particular como en grupo.

Buscábamos algo que nos permitiera crecer individualmente, desarrollando nuevas destrezas y habilidades, pero que también implicara coordinación, para potenciar nuestra capacidad de trabajar en equipo.

Además, queríamos integrarlo en el ámbito del módulo de Empresa e Iniciativa Emprendedora, desarrollando los estudios y plan de empresa necesarios para un posible lanzamiento del producto a nivel comercial.

Tras barajar distintas posibilidades nos decantamos por el desarrollo de un videojuego. Este desarrollo implicaba aprender un nuevo lenguaje de programación (C#) y múltiples herramientas asociadas, como Unity o Blender.

Para coordinar el trabajo en equipo utilizamos git (en concreto Github), la herramienta más popular para la gestión de código fuente y coordinación de trabajo de desarrollo, algo que sin duda necesitaremos en nuestro futuro trabajo como desarrolladores.

El producto desarrollado es un videojuego multijugador en el que dos equipos asimétricos, con distinto número de jugadores y habilidades por equipo, compiten por ganar la partida.

Abstract

This project comes from the idea of seeking new challenges for the team members, both individually and as a group.

We were looking for something that would allow us to grow individually, developing new skills and abilities, but also involved coordination to enhance our teamwork capabilities.

Furthermore, we wanted to integrate it within the scope of the Entrepreneurship and Business Initiative module, by developing the necessary studies and business plan for a potential commercial product launch.

After considering different possibilities, we decided to develop a video game. This development involved learning a new programming language (C#) and multiple associated tools such as Unity or Blender.

To coordinate the teamwork, we used git (specifically GitHub), the most popular tool for source code management and development work coordination, something that we will undoubtedly need in our future work as developers.

The developed product is a multiplayer video game in which two asymmetrical teams, with different numbers of players and abilities per team, compete to win the game.

Ilustración 1 Esquema de equipos

ALCANCE DEL PROYECTO

El objetivo de este proyecto es crear un videojuego multijugador online 3D para Windows o para PC qué funcione en la plataforma de Steam.

Ilustración 2 Controles de consola

Este desarrollo pretende ir más allá que simplemente la superación de los requisitos necesarios para aprobar este módulo y en definitiva la obtención de la titulación en el grado.

Con este trabajo, que se viene realizando desde un tiempo bastante largo, septiembre de 2022. Se pretende conseguir un producto de calidad rentable, es decir, un producto vendible que nos permita sacar un rendimiento mayor que el ya mencionado simple aprobado.

Un rendimiento económico que produzca un retorno satisfactorio para todas las horas de trabajo que se han puesto detrás de este trabajo.

Es por esto por lo que este trabajo también cuenta con un estudio de mercado detrás. Un estudio que llevó a tomar la decisión de la temática que sigue el juego (Horror) y sobre el que se aporta gran parte del estudio de viabilidad que se desarrolla a continuación, en cuanto a aspectos económicos se refiere.

Así mismo, es parte del alcance de este proyecto el conseguir un producto reconocible en el mercado que pudiera servir como carta de presentación para los desarrolladores de este, es decir, nosotros. Una insignia muestra de nuestra juventud, pero no carente de grandes capacidades y habilidades

ESTUDIO DE MERCADO Y VIABILIDAD

Macroentorno

El mercado actual se encuentra copado de videojuegos principalmente con “Contenido sexual”, “FPS” (First Person Shooter), “Novelas Visuales” (Aventuras Gráficas). Tanto las grandes empresas como los pequeños productores de videojuegos centran todas sus fuerzas en explotar géneros comunes que terminan saturando. Haciendo que las ganancias potenciales se terminen distribuyendo tanto que hace que los ingresos sean cada vez menores.

Aún peor, que terminan concentrándose solo en las grandes producciones con mayor capacidad para llegar a más clientes.

Factores económicos: Nuestra falta de recursos en la actualidad, nos presenta en desventaja frente a empresas con mayor experiencia y recursos qué desarrollan videojuegos. Sin embargo, el desarrollo de videojuegos tampoco requiere de grandes inversiones para obtener resultados de éxito. Al decir esto nos basamos en juegos como el “Among us” o “Cube world” que con su reducido presupuesto y poco personal llegaron al éxito entre los usuarios.

En España la industria de los videojuegos equivale al 0,11% del PIB español mientras que en Europa el sector facturó 1.795 millones de euros lo que equivale a un 2,75% más que el año anterior.

Nos basamos en juegos como el “Among us” o el “Cube World” que incluso tras años de su salida siguen siendo tendencia apesar de su sencillez. (https://www.3djuegos.com/juegos/among-us/noticias/among-us-mantiene-su-exito-a-pesar-del-descenso-de-usuarios-210125-110965#:~:text=Y%20es%20que%2C%20a%20pesar,a%20su%20estudio%2C%20explica%20SuperData.)

(PIB: http://www.aevi.org.es/la-industria-del-videojuego/en-espana/)

Factores políticos: A nivel político, al crear nuestra empresa desde cero, nos beneficiamos de la ley “Crea y Crece”, una de las principales reformas del Plan de Recuperación, Transformación y Resiliencia que nos permitirá crear nuestra empresa a partir de un solo euro y en menos tiempo mediante la reducción de obstáculos regulatorios.

Factores demográficos: Nuestro producto va destinado a un público generalmente cada vez más joven, de entre unos 15 y 25 años, aun así, cada vez son mayores las edades que están empezando a disfrutar de esta industria. Esto significa que nuestro marketing y todas las decisiones que tomemos en general, serán orientadas a un público objetivo juvenil.

Microentorno

El microentorno al que nuestro proyecto se ajusta, “Online Party Game”, es un mercado que cuenta con muy pocas copias. Es un mercado con una oferta muy pequeña que genera gran parte de los ingresos anuales. Una oportunidad que explotar al contar además con técnicas innovadoras en un mercado tan poco explotado.

Como se ha mencionado antes uno de los principales objetivos de este desarrollo es conseguir cierto retorno económico, así como crédito por la calidad del producto desarrollado, haciendo de este algo de lo que cuando se menciona, ya sea el título del videojuego o el nombre de la empresa desarrolladora, no se necesiten introducciones ni explicaciones.

Para lograr esto se debe realizar un estudio de mercado para ver qué tipo de videojuego está demandando el mercado. Este estudio debe de poder dar pistas del camino a seguir en relación con la temática del juego, su jugabilidad, aspecto visual, ...

Estudio del consumidor

El estudio llevado a cabo empieza por la realización del lienzo del modelo de mercado “Business model canvas”, con esta herramienta se dan unos primeros pasos en las decisiones que se deben ir tomando para ir encauzando la búsqueda, sobre todo para ir dilucidando qué tipo de preguntas debemos hacernos como, ¿A quiénes nos interesa vender nuestro producto?, ¿A qué problemas nos vamos a enfrentar?, ¿Cuáles pueden ser los puntos fuertes y débiles?, ¿Posibles aliados en el camino?

Ilustración 3 Bussines Model Canvas

Una vez se dieron los primeros pasos y se vieron los problemas y soluciones, se vio que era viable la posibilidad de desarrollar un videojuego exitoso.

Ahora el estudio debía dirigirse hacia exactamente que estaba demandando el mercado, que ideas escaseaban y cuáles eran las más rentables. Qué ámbitos se estaban dejando sin explotar.

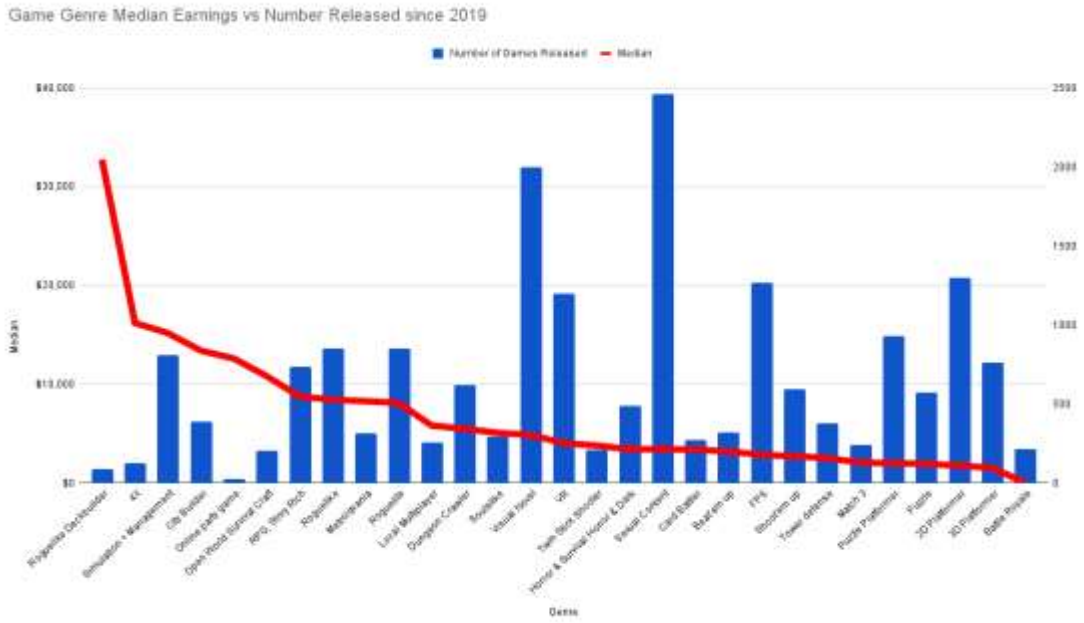

Ilustración 4 Grafica Ingresos medios por genero de videojuego vs número de lanzamientos 2019 (https://howtomarketagame.com/)

En base al anterior gráfico, extraído del siguiente estudio en donde se compara la producción de videojuegos según su género y los ingresos que generan desde 2019 hasta 2022, podemos sacar las siguientes conclusiones:

Nuestro juego (Online Party Game) es el género menos representado, sin embargo, es el 5º qué más ingresos genera, por lo tanto, estamos en un mercado con poca competencia y con mucha demanda. Esto nos facilitará gran parte del marketing, ya qué solo por el hecho de representar este género tan poco visto, muchos jugadores querrán probarlo.

El siguiente paso es analizar al potencial consumidor de nuestro producto, para ello realizamos el denominado “mapa de empatía”. Respondiendo estas preguntas

Responderemos a 6 preguntas:

- **¿Qué piensa y siente?** → Qué es lo que verdaderamente le importa
- **¿Qué ves?** → Cuál es su entorno y como es
- **¿Qué dice y hace?** → Qué aspecto tiene, cómo se comporta
- **¿Qué oye?** → Que dice su entorno, desde que canales multimedia le llega la información
- **¿Qué esfuerzos realiza?** → A que tiene miedo, riesgos que debe asumir
- **¿Qué beneficios espera obtener?** → Cuales son sus deseos reales

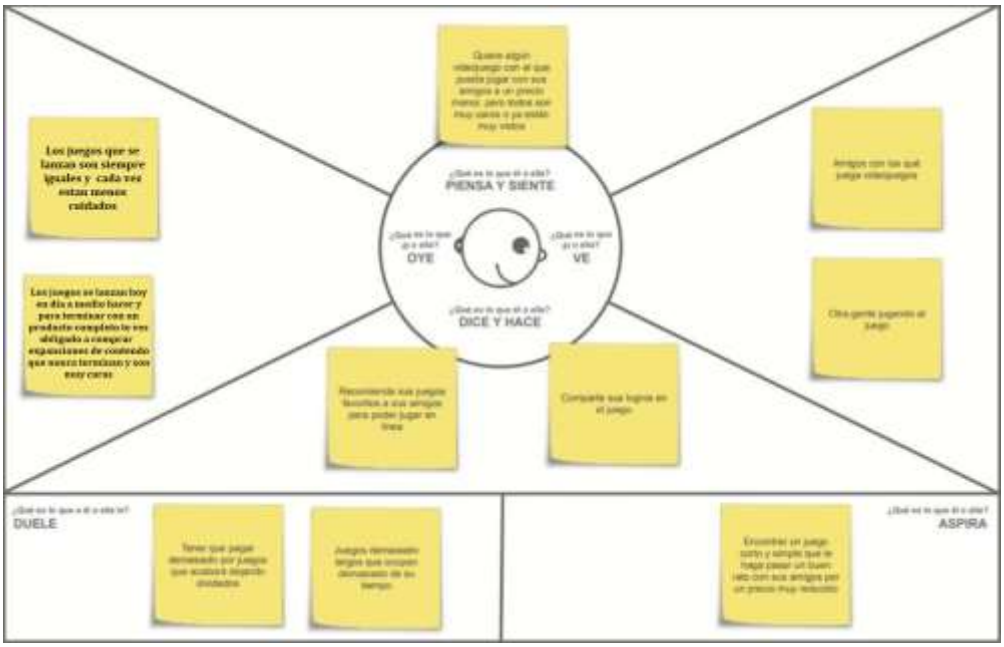

Ilustración 5 Mapa de empatía

Tras haber realizado el mapa de empatía, decidimos salir a la calle y hacer una serie de preguntas gracias a la herramienta “Google forms” (Formulario: https://forms.gle/LZbGSi9En5RQd2CL6) para recabar aún más información.

Para que esta encuesta fuese un muestreo real del mercado al que nos dirigimos, se realizó a personas que son usuarios de videojuegos de forma activa, es decir, nuestro mercado objetivo o target. La primera pregunta fue acerca de la edad, y lo qué buscábamos con esta pregunta era intentar que gente de todas las edades responda a este formulario. Obtuvimos variedad de respuestas, pero la gran mayoría

tenía entre 18 y 25 años. Entendemos qué esto es por la edad media de los “gamers” ya que han sido muchas las personas mayores de 25 años a las qué se les ha preguntado si juegan a videojuegos, pero pocas son las que disfrutan de este sector. Esto confirma lo mencionado anteriormente, nuestro público objetivo es generalmente joven.

Con la siguiente pregunta queríamos medir cuántos de los encuestados han comprado alguna vez un videojuego indie. Esto nos da una idea de qué porcentaje de personas disfrutan del mercado de los videojuegos indie, obteniendo una cantidad del 46%. Nos pareció una estadística muy buena, ya que casi la mitad de los encuestados, tendrán tendencia a ver qué juegos indie han salido en el mercado.

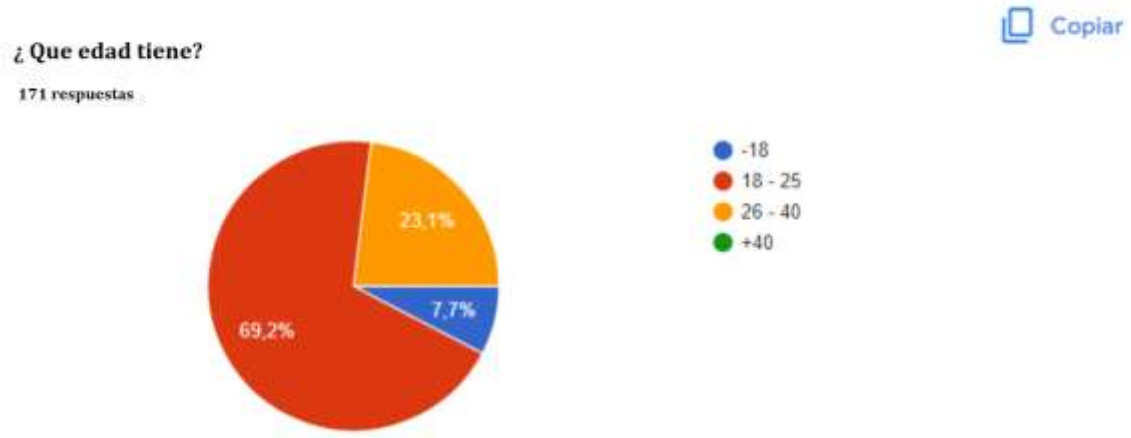

Ilustración 6. Grafica resultado pregunta Google forms

La siguiente pregunta iba para hacernos una idea de cuánto pagarían los encuestados por un videojuego indie, viendo así, si el precio qué teníamos en la cabeza es realista o no. Cabe decir qué la cantidad qué pague una persona depende de muchos factores, pero basándonos en la calidad de otros videojuegos indie, y suponiendo qué el juego tendrá la calidad qué tenemos en la cabeza, el precio qué habíamos pensado para nuestro juego rondaría los 4 euros. Más del 92% de encuestados pagarían más de 5 euros, lo qué nos hace pensar qué vamos por el camino correcto.

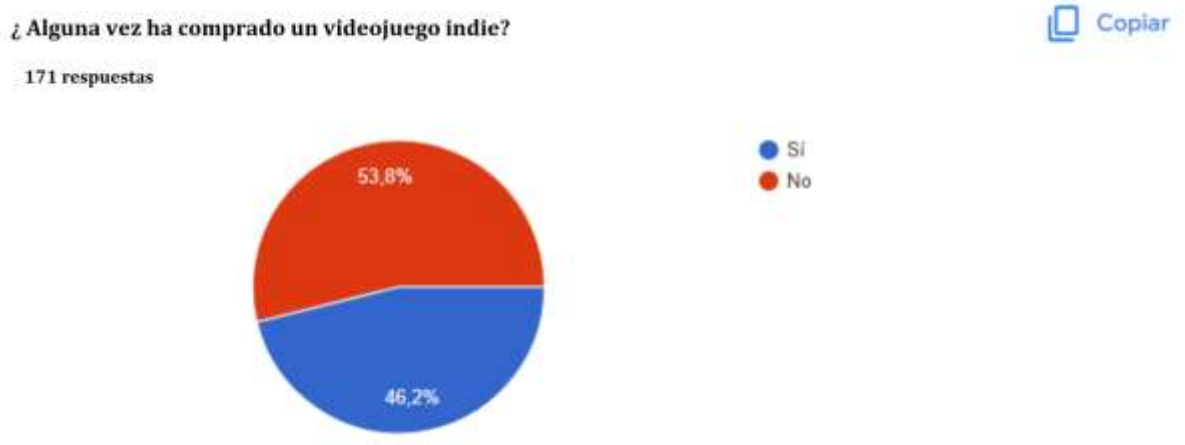

Ilustración 7 Grafica resultado pregunta Google forms

La siguiente pregunta iba para hacernos una idea de cuánto pagarían los encuestados por un videojuego indie, viendo así, si el precio qué teníamos en la cabeza es realista o no. Cabe decir qué la cantidad qué pague una persona depende de muchos factores, pero basándonos en la calidad de otros videojuegos indie, y suponiendo qué el juego tendrá la calidad qué tenemos en la cabeza, el precio qué habíamos pensado para nuestro juego rondaría los 4 euros. Más del 92% de encuestados pagarían más de 5 euros, lo qué nos hace pensar qué vamos por el camino correcto.

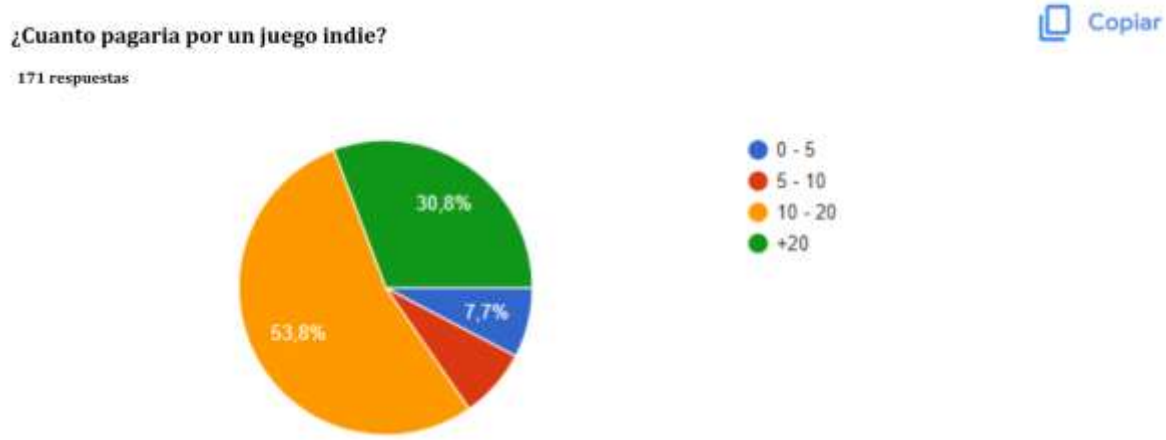

Ilustración 8 Grafica resultado pregunta Google forms

Finalizando con el cuestionario queremos averiguar si la gente prefiere jugar solo o con amigos. Más del 84% prefiere jugar a juegos online lo que concuerda con

nuestras suposiciones y con el estudio de mercado realizado anteriormente, acerca de la preferencia al juego multijugador de los jugadores.

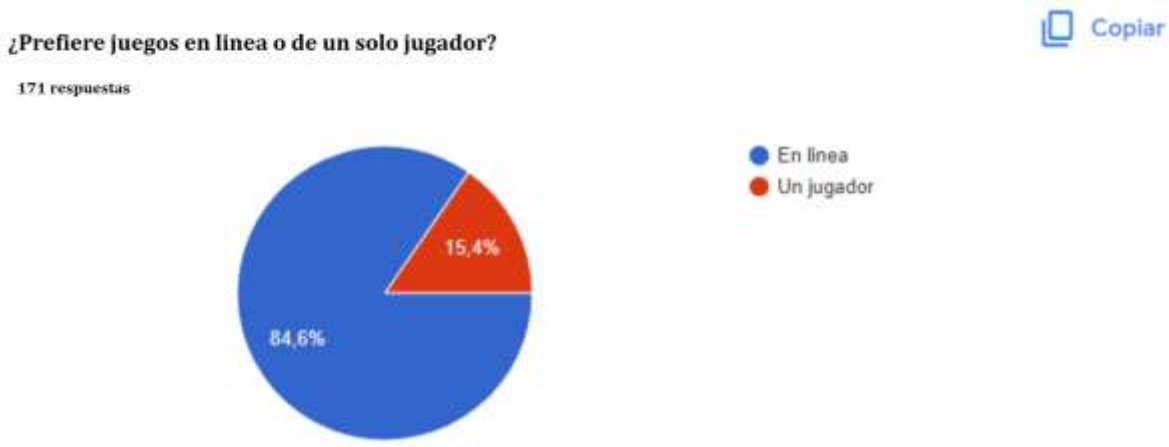

Ilustración 9 Grafica resultado pregunta Google form

ESTUDIO DE VIABILIDAD TÉCNICA

Para llevar a cabo el desarrollo de este proyecto nos surgen varias posibilidades a la hora de escoger qué herramientas son las que más nos van a facilitar el trabajo y nos van a dar un resultado más profesional y acorde a lo que tenemos en la cabeza. Esto se tiene que decidir desde el día uno, ya que una vez empezado, volver atrás podría suponer una pérdida grande de tiempo y dinero.

Motor gráfico

Surgen 3 grandes candidatos a la hora de escoger el motor gráfico: Unity, Unreal Engine y Godot. Todos son opciones muy viables ya qué son gratis y bastante populares en el mundo del desarrollo de videojuegos, pero nos decantamos por Unity, ya que era el qué mayor número de usuarios tenía, y por tanto el qué más documentación iba a tener.

Ilustración 10 Logos tres principales herramientas para desarrollo de videojuegos

¿Qué es Unity?

Unity es una plataforma de desarrollo de videojuegos que permite a los desarrolladores crear juegos en 2D y 3D para múltiples plataformas, como PC, móviles, consolas y realidad virtual. En esencia, es una herramienta que te permite construir el mundo virtual de un juego y crear las reglas y comportamientos que los jugadores experimentarán.

Es muy popular en la industria del videojuego debido a su facilidad de uso y su capacidad para generar resultados impresionantes. Los desarrolladores utilizan una interfaz gráfica intuitiva que proporciona Unity para crear mundos virtuales, insertar modelos de personajes y objetos, y aplicar efectos visuales y sonidos para crear una experiencia completa.

Además, Unity cuenta con una gran cantidad de recursos y herramientas qué facilitan todas las etapas del proceso de creación de un videojuego, desde el diseño

y la planificación, hasta la programación y la publicación. Por ejemplo, Unity proporciona una amplia biblioteca de recursos gráficos, herramientas de animación, herramientas de programación y soporte para múltiples plataformas, qué nos permite a los desarrolladores crear juegos de alta calidad en un tiempo y coste razonables. Todos estos recursos son accesibles desde la "***Asset store***" propia de Unity, donde los creadores de contenido pueden compartir su trabajo ya sea con la búsqueda de beneficio o publicitarse como artistas gráficos.

¿Crear o comprar assets?

Otro de los dilemas qué surgió antes de empezar fue si íbamos a crear nuestros propios modelos 3D o los íbamos a comprar. Al investigar un poco sobre el tema nos dimos cuenta rápidamente de la alta complejidad qué puede tener el diseño 3D, y lo mucho qué nos podría llevar aprender a modelar lo suficientemente bien como para obtener el resultado qué buscábamos.

Por ello investigamos un poco en la Unity Asset Store, y nos dimos cuenta de qué había infinidad de modelos qué podíamos comprar por un módico precio y tenían un acabado mucho más profesional de lo que nosotros podíamos optar a hacer en este periodo de tiempo. Es por esto qué finalmente decidimos hacer una pequeña inversión para poder disfrutar de unos recursos mucho más competentes y ahorrarnos todo ese tiempo qué nos hubiese llevado aprender a modelar en 3D.

Lenguaje de programación

El lenguaje de programación escogido viene determinado por el lenguaje que utiliza Unity y en parte por ser algo nuevo para todos los integrantes, este es C#.

C# es un lenguaje orientado a objetos desarrollado por Microsoft en 2000. Es un lenguaje moderno y versátil que se utiliza en una amplia variedad de aplicaciones, desde la creación de aplicaciones de escritorio hasta la programación de aplicaciones web y móviles.

¿Pero por qué usar C# para el desarrollo de videojuegos?

Bien, C# es una excelente opción para la programación de videojuegos porque es un lenguaje rápido, seguro y fácil de usar. También cuenta con una gran cantidad de recursos y bibliotecas de código abierto que los desarrolladores pueden utilizar para acelerar el desarrollo de videojuegos.

Sin embargo, C# no se ha vuelto famoso en el desarrollo de videojuegos por sus características, no. Mejor dicho, se ha convertido en una de las principales herramientas en el desarrollo de videojuegos debido al éxito de Unity como plataforma de desarrollo en la que se usa C#

Trabajo en equipo

Al tratarse de un proyecto de cierto tamaño, se debe hacer un reparto de tareas efectivo y eficiente, que permita identificar los problemas rápidamente y darles solución aún más rápido. Para hacer de este proyecto algo viable para el equipo.

"Se comenzó con un reparto de tareas inicial, y se fueron añadiendo tareas y asignándole a los distintos miembros del equipo a medida que se avanzaba en el desarrollo."

Una vez hecho el reparto se necesitaba una herramienta eficaz que hiciera del trabajo en equipo algo dinámico, algo que aportase y no un lastre. Se decidió usar Git, un estándar de facto en el control de código fuente, y en concreto GitHub.

GitHub es una plataforma de desarrollo colaborativo que permite a los equipos de programadores trabajar juntos en proyectos de software. Funciona como un sistema de control de versiones y proporciona herramientas para el seguimiento de cambios, la gestión de problemas y la colaboración en el desarrollo de software.

En GitHub, los desarrolladores pueden alojar y compartir repositorios de código, que son colecciones de archivos que componen un proyecto. Estos repositorios permiten a los equipos colaborar de manera eficiente, ya que varias personas pueden trabajar en el mismo proyecto al mismo tiempo, fusionando sus cambios de manera ordenada. También se pueden rastrear los problemas y errores, y se puede discutir y resolver estos problemas directamente en la plataforma.

En resumen, GitHub facilita la comunicación entre los miembros del equipo, permite la revisión de código y ofrece una visibilidad clara de las contribuciones individuales. Además, cuenta con una amplia gama de herramientas integradas y servicios complementarios que mejoran la eficiencia del desarrollo, como integración continua, pruebas automatizadas y despliegue en la nube.

DESARROLLOS INDISPENSABLES PARA LA CONSECUCIÓN DE UN PRODUCTO JUGABLE

Diseño y desarrollo del mapa

El mapa de un juego pese a normalmente pasar desapercibido es una faceta en el desarrollo de videojuegos muy importante.

En un juego de este estilo el diseño del mapa es uno de los principales engranajes para que el mecanismo de diversión de una partida tenga éxito o no. Sin embargo, no es lo único difícil en relación con el" campo de recreo" de la partida, ya que una vez se ha diseñado, hay que convertirlo en algo real.

Ilustración 11 Perspectiva superior del mapa del videojuego

Si primero nos centramos en el diseño del mapa debemos conseguir que este genere situaciones que hagan al jugador sentirse dentro de la partida, en peligro junto con su personaje. Para lograr esto buscamos hacer del mapa tanto una experiencia claustrofóbica dentro de la casa con pasillos y escaleras estrechas, puertas secretas y escondites. Como una experiencia de semi-exploración al exterior de la casa, contado con otros edificios anexos y laberintos los cuales el jugador debe también explorar para obtener recursos, así como exponerse también en los espacios abiertos que unen estas zonas.

En el diseño del mapa también debemos tener en cuenta que se debe de poder jugar la partida, ¿Esto qué significa? Esto significa que tanto el monstruo debe ser capaz de arrinconar a los niños, como los niños deben ser capaces de zafarse del monstruo sin que se genere una situación de desequilibrio.

Ilustración 12 Conjunto de estancias del mapa

Para lograr un diseño equilibrado y funcional no hay otra solución que el ir probando escenarios y situaciones de juego para ir moldeando el mapa según se encuentren fallos, así como mantener un constante seguimiento en el producto final ya que serán los propios jugadores los que terminarán poniendo a prueba el diseño del mapa con tantas tácticas para el uso de este en su beneficio como variedad de jugadores exista.

Poco a poco se fue ideando un mapa y poniéndolo a prueba accediendo a assets en la tienda de Unity que proporcionaban tanto buenas ideas como materiales 3D que podíamos usar en nuestro beneficio.

Finalmente se alcanzó el mapa que tenemos hoy en día partiendo de la adquisición de un asset con la temática “Horror Mansion”. Este asset proporcionó

tanto un diseño bastante bueno como los modelos 3D que permiten modificarlo para terminar de perfeccionarlo.

También forman parte del diseño del mapa las puertas y objetos que podemos encontrar por él, así como la ambientación que se le da y la iluminación jugando un gran papel en la partida, sin embargo, estos aspectos se desarrollarán más adelante en sus respectivos apartados

Iluminación del mapa

La iluminación desempeña un papel crucial en un juego de terror, ya que puede establecer el tono, la atmósfera y la sensación de inmersión para los jugadores. La forma en que se ilumina un mapa puede influir en la tensión, el miedo y la anticipación que los jugadores experimentan al explorar el entorno del juego. Aquí hay algunos puntos clave qué queríamos conseguir:

- Creación de ambiente: La iluminación adecuada puede establecer el estado de ánimo y crear una atmósfera opresiva y siniestra. Los contrastes de luz y sombra, la utilización de colores fríos o cálidos, y la intensidad de la luz pueden influir en cómo los jugadores se sienten mientras navegan por el mapa. Una iluminación tenue y sombría puede generar una sensación de peligro inminente y misterio.

- Guía y dirección de la atención: La iluminación puede usarse estratégicamente para guiar la atención del jugador hacia áreas clave del mapa o elementos importantes de la historia. Al resaltar ciertas áreas con luz más brillante o contrastes de iluminación, se pueden enfocar la mirada y la exploración del jugador, generando momentos de tensión y revelaciones impactantes.

- Crea anticipación y tensión: La iluminación puede ayudar a construir la tensión en un juego de terror. Al utilizar sombras, luces parpadeantes, efectos de luz intermitente o cambios de iluminación repentinos, se pueden generar momentos de anticipación y susto para los jugadores, aumentando la sensación de amenaza y el temor a lo desconocido.

Para lograr estos objetivos, usamos las diferentes luces qué nos proporciona Unity, ajustándolas según el objetivo qué queríamos conseguir, buscando siempre crear ambientes oscuros, ya que el uso de una linterna era otra mecánica qué queríamos implementar en el juego

Ilustración 13 Imagen representativa inmersión producida por la iluminación

Movimiento básico y animaciones de los personajes

Lograr el movimiento de los personajes fue relativamente sencillo, especialmente el del personaje principal, pues al tratarse de un movimiento en primera persona, el jugador siempre se moverá de manera relativa a dónde está mirando, es decir, si pulsa la tecla de moverse hacia delante, se moverá en la dirección en la que está apuntando la cámara, mientras que, si pulsa la tecla de moverse a la derecha, se desplazará hacia el lado derecho desde la dirección en la que está apuntando la cámara.

El único reto qué podríamos encontrar con este movimiento sería al pulsar dos teclas simultáneamente, es decir, para los movimientos diagonales, pero al recibir el input horizontal y el input vertical independientemente, es tan fácil como hallar el vector resultante de las teclas qué se están pulsando y mover al jugador en esa dirección.

Animar a este personaje fue algo más complejo qué el movimiento, ya que, al tratarse de un movimiento en primera persona, el personaje no puede tener la misma animación para moverse hacia delante qué para moverse hacia atrás, hacia un lado, o incluso en diagonal, lo qué quiere decir qué necesitaremos una animación diferente para cada dirección en la que se mueva el jugador. Además, añadimos la funcionalidad de qué el jugador se pueda agachar (de este modo podrá hacer menos ruido y moverse de manera sigilosa por el mapa) y correr (para escapar del Alien) lo qué en resumidas cuentas nos obliga a añadir más animaciones para cada dirección, pero esta vez agachado y corriendo.

Esto significa qué necesitaremos un total de 19 animaciones solo para el movimiento del personaje, y estas las sacamos de Mixamo, una página web en la que podemos encontrar miles de animaciones gratuitas qué facilitan su uso en Unity.

Una vez descargadas usaremos los “Blend Trees” de Unity, qué lo qué hacen es crear transiciones de animaciones según la dirección en la que se está moviendo el personaje, es decir, si el personaje se está moviendo hacia delante, se activará la animación de moverse hacia delante, y si de repente se mueve hacia la derecha, se creará una transición de la animación de moverse hacia delante y de la de moverse a la derecha. Esto es simplemente para qué el movimiento se vea natural y no parezca qué el juego va a tirones.

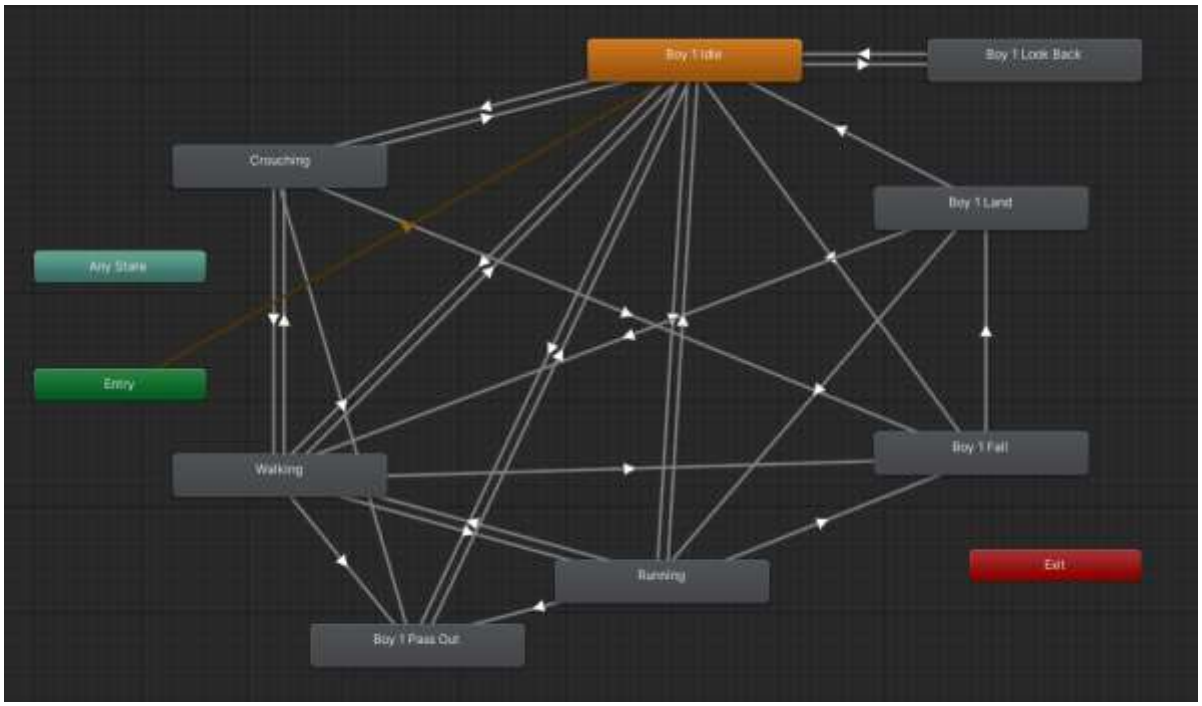

Ilustración 14 Esquema de animaciones del personaje "Niño"

En esta figura se muestra el “Animator” finalizado del personaje principal, donde se define qué animaciones se podrán activar en cada momento y las condiciones para qué estas se activen. Cada rectángulo representa una animación o un “árbol de animaciones” en el caso de las animaciones del movimiento, y cada flecha indica qué se puede pasar de de la animación desde la que proviene a la animación hasta la que apunta si se cumple cierta condición, qué varía dependiendo de cada caso.

Podemos ver también en la figura como además de las animaciones de movimiento tiene alguna más, como la de “idle” qué se ejecuta siempre qué el jugador está quieto, para qué parezca qué tiene vida el personaje, o la de “pass out” qué se ejecutará siempre qué dañen a este personaje.

El movimiento del Alien fue algo más complejo qué el del personaje principal, ya que, al ser en tercera persona, no solo se tiene que mover en la dirección relativa a la cámara, sino que también tiene qué rotar para apuntar en la dirección en la que se está moviendo. Para lograr esto, hallamos la dirección, relativa a la cámara, a la que se quiere mover el jugador y rotamos el personaje a cierta velocidad, ya que al igual que con el personaje principal, queremos buscar naturalidad en los movimientos. Esto es algo más complicado de lo que parece, ya qué Unity no facilita de ninguna forma el hallar la dirección a la que queremos mover el personaje, tomando como punto de referencia la dirección de la cámara. Por tanto, tenemos qué hallar esto de manera matemática mediante otros recursos.

```
float targetAngle = Mathf.Atan2(direction.x, direction.z) * Mathf.Rad2Deg + cam.eulerAngles.y;
float angle = Mathf.SmoothDampAngle(transform.eulerAngles.y, targetAngle, ref turnSmoothVelocity, turnSmoothTime);
transform.rotation = Quaternion.Euler(0, angle, 0);
```

Ilustración 15 Extracto Código control del "Alien"

Por otra parte, las animaciones del Alien fueron bastante sencillas, ya que al rotar sobre su propio eje cada vez qué quiere cambiar de dirección, solo se necesita una animación qué es la de moverse hacia delante. Aunque en este caso fueron dos, una para correr y otra para caminar, y ambas fueron sacadas de Mixamo.

En esta figura vemos el "Animator" del Alien finalizado, mucho más simple qué el del personaje principal, por lo explicado anteriormente. Podemos ver qué también tiene alguna animación a parte de las de movimiento básico, como la de "roar" para hacer un rugido o "attack" para atacar.

Efecto de visión para personaje ciego

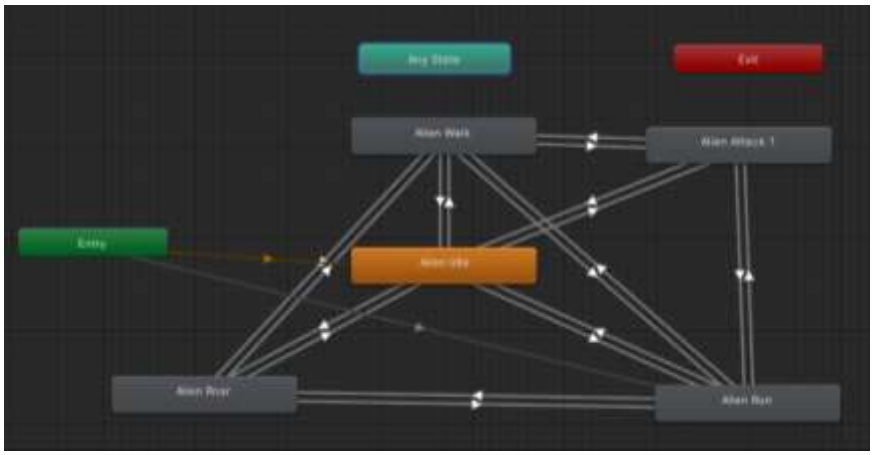

Ilustración 16 Esquema animaciones del personaje "Alien"

El "villano" de nuestro juego iba a ser un personaje ciego, por lo tanto, queríamos implementar una mecánica que dificultara su visión sin que resultara aburrida o demasiado difícil de manejar. Para lograrlo, planeamos crear un efecto de "sonar" en el que el personaje pueda percibir su entorno a través de pulsaciones que se expanden desde su posición.

Para crear este efecto, utilizamos el ShaderGraph de Unity, que es un programa que describe cómo se renderiza un objeto en una escena gráfica, controlando aspectos como el color, la textura, la iluminación y los efectos especiales en tiempo real.

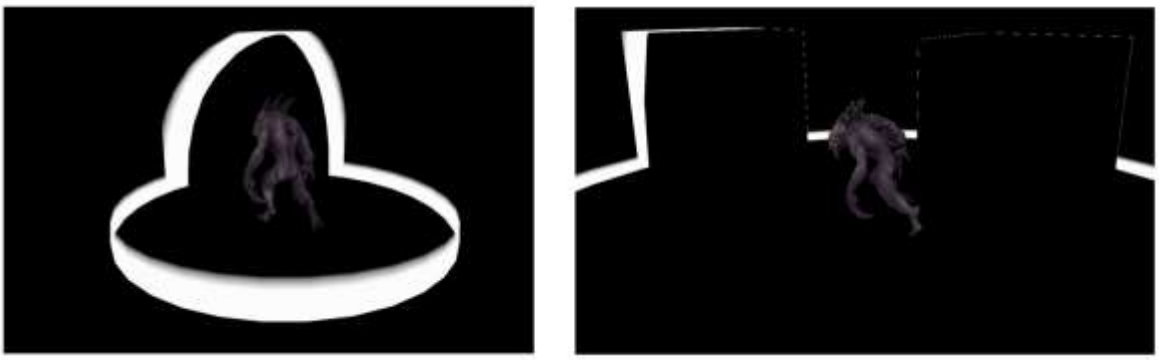

Ilustración 17 Representación visión sonar del Alien

Sin entrar demasiado en detalle, ya que la creación de estos efectos puede llegar a ser muy compleja, lo que buscábamos era aplicar sobre una esfera qué se va agrandando un efecto qué ilumine todo aquello con lo qué interseca.

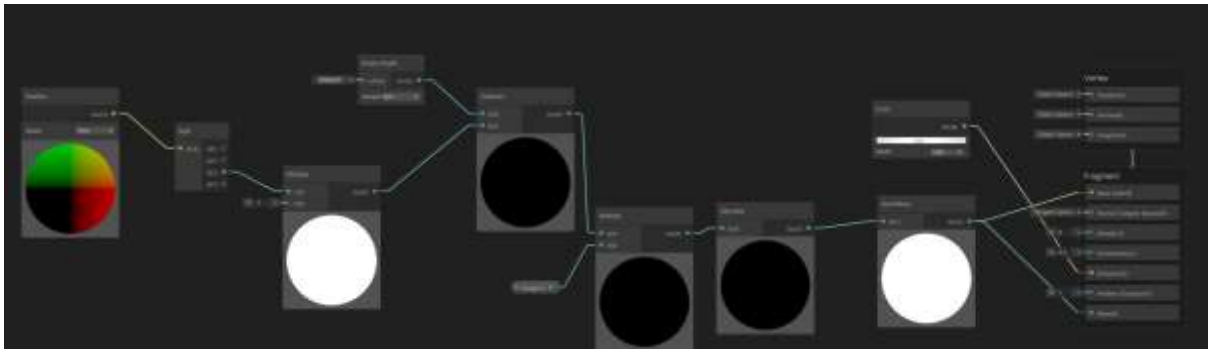

Ilustración 18 Esquema funcionamiento componentes sonar

En la imagen adjunta, se muestra el ShaderGraph finalizado. Básicamente, reduce la opacidad del objeto a cero y colorea de blanco todo lo que interseca con él. A medida que nos alejamos de la intersección, el blanco se vuelve más transparente. Además, aplicamos un efecto de brillo al blanco para que sea más visible, incluso en ausencia de luz en la escena.

Interacción del jugador con el mapa

En la interacción del jugador con el mapa encontramos una serie de aspectos que deben ser abordados de una manera efectiva para conseguir una experiencia atractiva y cómoda de jugar.

a) Interacciones cámara - mapa

La cámara en Unity es un objeto más que interactúa con su entorno, salvo por los elementos propios que la caracterizan que permiten una visión de jugador, tiene los mismos elementos y problemas que un objeto normal.

Sin embargo, al empezar las pruebas con los personajes dentro del mapa surgió un problema específico de la cámara. Este era que mientras el personaje no atravesaba la pared, la cámara si y permitía ver lo que se encontraba al otro lado, y ver a través de las paredes no es algo muy bueno para un juego de miedo.

Tras perder mucho tiempo intentando controlar las colisiones de la cámara para evitar que esta atravesase las paredes se decidió que ese no era el camino correcto pues además daba lugar a muchos problemas para mantener un buen control del personaje.

Investigando se llegó a la conclusión de que la solución reside en el enfoque y campo de visión de la cámara.

Una vez se descubrió esto el problema se solucionó con relativa facilidad. Simplemente había que saber que campos modificar y qué valores concretos ponerles.

El valor que había que cambiar era "Near Clipping Plane" o "Plano de Recorte Cercano". Este concepto se puede explicar como la distancia que va a definir el tamaño del área en el que los objetos son visibles.

Para intentar visualizarlo imagina un cilindro con una altura variable, esa altura es el "Near Clipping Pane". solamente los objetos que están dentro de ese cilindro se van a renderizar y se van a poder ver por el jugador, es por esto que, si el cilindro atraviesa la pared y capta los objetos de detrás, al no poder controlar eficazmente las colisiones de la cámara, está en distancias cercanas atravesaría la pared y dejaría ver lo que hay detrás.

La solución era controlar la altura de ese cilindro para que en distancias cortas disminuyera su altura lo justo para que no alcanzase a ver más allá de la superficie

que tiene delante, de esta manera poco importaría que la cámara atravesase paredes, pues el cilindro solo permitirá ver lo más cercano.

Para controlar la distancia a las paredes una vez más se usó "RayCast", pero esta vez se necesitó usar otra herramienta "OverLapSphere". Esto es porque se debía tener un control muy preciso de los objetos y paredes qué hay cerca, pues variar la altura de este cilindro de manera indiscriminada generaba errores de visión, provocando que si por algún casual la altura del cilindro es la incorrecta mientras por ejemplo el personaje corre, la visión de la cámara empezaría desde dentro de la textura de la cabeza del niño generando una mala visión.

Para esto se usó "OverLapSphere". Esta es una herramienta que está constantemente calculando los objetos que se encuentran dentro del volumen de una esfera virtual con unas dimensiones establecidas según necesidades.

```
Unity Message | 0 references
void FixedUpdate()
{
    Debug.Log("Longitud: " + nearItemViewSphere.Count);
    nearItemOverlapSphere = Physics.OverlapSphere(this.transform.position, 2f);

    foreach (var item in nearItemOverlapSphere)
    {
        if (nearItemViewSphere.Contains(item))
        {
            cam.nearClipPlane = 0.01f;
        }

        if (nearItemViewSphere.Count <= 0)
        {
            cam.nearClipPlane = 0.3f;
        }

    }


}
```

Ilustración 19 Extracto de Código para el control de objetos cercanos

Combinando estas dos herramientas tenemos un control muy preciso sobre todas las posibilidades que se pueden dar al acercar la cámara a una superficie. Ya sea de frente o desplazándose pegado a la pared podemos controlar la altura de ese cilindro de manera precisa para que de ninguna manera se vea a través de las paredes.

b) Apertura de puertas

La apertura de puertas fue un proceso que tomó bastante tiempo. Esto por ser el primer reto al que nos enfrentamos programando en C#, teniendo por primera vez que trabajar con aspectos como “BoxCollider”, “SphereCollider”,” OnTriggerExit” y muchas otras funcionalidades que Unity proporciona y explota.

La primera problemática era evitar que el personaje atraviese las puertas sin más, así como las paredes y demás ítems. Este problema se solucionó fácil pues Unity proporciona una herramienta de cálculo de impacto entre objetos del mapa llamada “Mesh Collider”. Esta herramienta realiza un cálculo sobre la textura del objeto y calcula cuando la textura de otro objeto entra en colisión con ella, pero sin más funcionalidad que esa, evitar el desplazamiento entre objetos.

Bien, una vez evitamos atravesar las puertas, debemos de saber que nos encontramos frente a una puerta. Para ello entran en juego las herramientas de “SphereCollider” y “BoxCollider”. Estos son objetos invisibles que se ponen encima de los objetos reales y visibles, que, si bien no evitan de manera predefinida el traspaso de objetos entre sí, si cuentan con métodos que de manera programática permiten al desarrollador saber cuándo un objeto entra en contacto con otro, que objetos son y demás funcionalidades. Estos métodos unidos a la posibilidad de poner “Tags” a los objetos dentro del juego, permite identificar accediendo a esos Tags que objetos están colisionando.

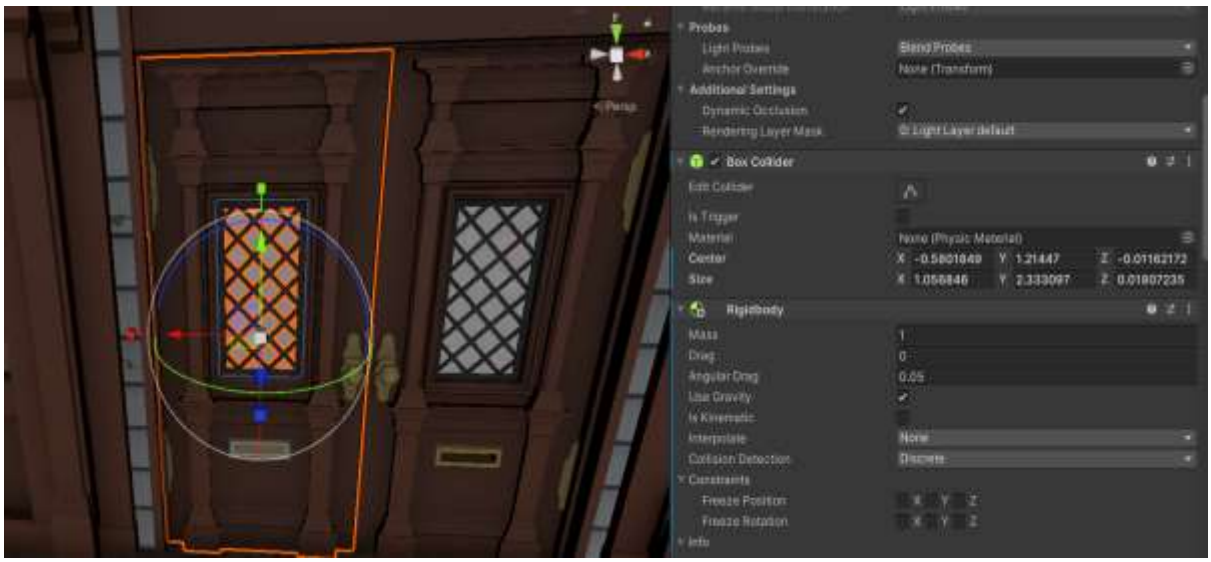

Ilustración 20 Puerta en el editor Unity y sus componentes "BoxCollider" y "RigidBody"

En pocas palabras, cuando la “SphereCollider” del personaje chocha con el “BoxCollider” de la puerta, se devuelve un evento del que podemos acceder al Tag

de los objetos involucrados y así comprobar que estamos delante de una puerta o de una pared, o lo que sea.

Una vez sabemos que estamos delante de una puerta también queremos saber si el jugador quiere abrirla o simplemente está ahí delante indeciso pues no sabe si el monstruo está al otro lado. Esto se soluciona controlando los eventos que son registrados desde los dispositivos externos (ratón y teclado). Si coinciden los colliders “Player” y “Door” y además al mismo tiempo se pulsa la tecla concreta que abre las puertas, entonces podemos decir que si, el jugador quiere efectivamente abrir la puerta, ya solo falta abrir la puerta en sí.

Para conseguir este último aspecto se pensó que la manera más fácil para lograrlo sería simplemente realizar una animación para la apertura de la puerta que se ejecutase en el momento de abrir, y otra animación invertida para el momento de cerrar. Sin embargo, además de quedar muy poco natural y generar un movimiento muy rígido, resultó que debería hacerse una animación para cada puerta pues la animación no podía transferirse completamente de una puerta a otra. Y esto no es una solución ni viable ni efectiva ni bonita.

Por lo tanto, se dejó el camino fácil y se pasó a una solución que aplicaba físicas a través de vectores sobre los objetos, tratando así cada objeto de manera única y más real.

Esta solución se consiguió sumando a cada puerta el componente “RigidBody” lo cual es una herramienta que permite a los objetos dentro del juego comportarse de manera que les afectan las físicas. Y un “HingeJoint”, tal y como su traducción indica, bisagra. Siendo esto un componente virtual que se integra en la puerta y le proporciona un eje de rotación y unos rangos de movimiento máximo y mínimos variables. La puerta ya se comporta como una puerta real, ahora solo falta ejercer sobre ella una fuerza que nos permita moverla. Esto se consigue trabajando con vectores

Ilustración 21 Puerta en el editor Unity y su componente "HingeJoint"

Para el cálculo del vector que se utilizara en la aplicación de la fuerza sobre la puerta, utilizamos una herramienta llamada “RayCast”. Esta herramienta emite un rayo desde el punto en donde se le dice y en la dirección que se le indica, devolviendo información variada, siendo para nosotros la más importante el objeto sobre el que impacta y el punto concreto en el que impacta.

De esta manera si ubicamos el rayo en el centro de la cámara del jugador, podremos obtener usando ese rayo el punto del objeto que tenemos delante en el que impacta directamente el rayo. Y una vez obtenemos el punto de impacto podemos calcular el vector resultante entre ese punto y el centro de nuestro personaje. Al tener este vector podemos utilizar sus componentes como herramientas de trabajo.

El módulo del vector es la distancia entre nuestro personaje y la puerta, según su valor podemos controlar cuán cerca debemos estar para poder abrir la puerta.

```
nearItem = Physics.OverlapSphere(cam.transform.position, 0.5f);
ray = new Ray(cam.transform.position, cam.transform.forward);
Debug.DrawRay(ray.origin, ray.direction * distance, Color.black);
RaycastHit hitInfo;
if (Physics.Raycast(ray, out hitInfo, distance, mask))
{

    objectOnRay = hitInfo.transform.gameObject;

    if (hitInfo.transform.gameObject.layer == LayerMask.NameToLayer("Interactable"))
    {
        if (gameObject.CompareTag("Door"))
        {
            if (Input.GetKey(KeyCode.Mouse0))
            {
                OpenCloseDoorServerRpc(ray.direction * 10, gameObject.name);
            }
            if (Input.GetKey(KeyCode.Mouse1))
            {
                OpenCloseDoorServerRpc(ray.direction * -10, gameObject.name);
            }
            if (Input.GetKey(KeyCode.E) && Input.GetKeyDown(KeyCode.Mouse0))
            {
                OpenCloseDoorServerRpc(ray.direction * 1000, gameObject.name);
            }
            if (Input.GetKey(KeyCode.E) && Input.GetKeyDown(KeyCode.Mouse1))
            {
                OpenCloseDoorServerRpc(ray.direction * -1000, gameObject.name);
            }
        }
```

Ilustración 22 Extracto Código control apertura puertas

La dirección del vector es la dirección en la que vamos a ejercer la fuerza, si bien debemos controlar el sentido de esa fuerza para evitar que la fuerza aplicada sobre cada puerta sea irregular dependiendo del punto del mapa en que nos encontremos. Pero esto es fácil de solucionar si en vez de calcular ese vector en relación con el mapa, lo hacemos contado como eje de coordenadas a nuestro personaje, de esta manera el sentido siempre será positivo en la dirección a la que mira nuestro personaje.

Ya tenemos una puerta funcional, la capacidad para detectarla y el punto, dirección y sentido en el que la queremos empujar. Ya solo falta empujar, fácil también de solucionar ya que el componente “RigidBody” que previamente hemos incluido en la puerta, cuenta con un método “addForce()”, al cual si se le pasa un vector y una magnitud de aceleración hará de la puerta algo al que nuestra interacción le afecte.

Tenemos el vector solo falta decidir cuánta aceleración queremos imprimirle a la puerta, y esta aceleración es variable si queremos abrir la puerta sigilosamente para entrever que hay al otro lado o si queremos dar un portazo pues el monstruo nos pisa los talones. Esto se controla definiendo una combinación de controles para cada situación que el jugador usará según le convenga.

c) Apertura de cajones

Tras la experiencia adquirida con el problema de las puertas se pensó que abrir los cajones ya sería pan comido, sin embargo, no fue del todo así, pues los cajones al estar dentro de otro objeto con sus propias colisiones trajeron otros problemas nuevos.

La manera de interactuar con las puertas utilizando la herramienta de “Raycast” y los “BoxCollider” y “SphereCollider”, resultaron ser una muy buena solución para que el usuario pudiera interactuar con el mapa de manera bastante efectiva. Sin embargo, en este caso se planteaba un problema respecto a la identificación del cajón como un objeto separado del mueble en el que estaba insertado.

Ilustración 23 Mueble y cajones en el editor Unity

No se podía identificar el cajón dentro del mueble ya que los “colliders” del mueble en general absorben al del cajón identificando al mueble antes que al cajón. Para solucionar esto hubo que modificar los campos de colisión del mueble en general para dejar un campo efectivo tanto para la identificación del cajón como un elemento separado, como para evitar la no identificación del mueble desde ángulos y aproximaciones en las que el usuario pretende interactuar con el mueble en vez de con el cajón. Modificando a mano los modelos 3D de los muebles utilizados se solucionó este problema con cierta rapidez, pero algo de trabajo “manual”.

Ya se puede identificar al cajón como un elemento libre dentro del mueble, ahora hay que moverlo. Sabemos que no se puede hacer a través de las animaciones, pero tampoco podemos hacer igual que con las puertas, y tratarlos como objetos libres con sus propias físicas, pues las colisiones de objetos dan bastantes problemas cuando son texturas bastante ceñidas. Sin embargo, para mover objetos en el mapa no es necesario que estos tengan sus propias físicas, se puede hacer que se muevan

de manera programática simulando un movimiento suave modificando sus coordenadas en relación con su posición en el espacio virtual.

C# lleva de la mano de Unity desde sus inicios, esto significa que durante todos estos años han surgido problemas a los que ya se les han puesto solución antes de que nosotros lleguemos a encontrarlos en nuestro camino.

Al trabajar Unity en gran medida con vectores se ha desarrollado con el tiempo en C# librerías y otras herramientas que brindan una gran cantidad de soluciones para los problemas que trae la matemática en este campo. Es por esto que tras un poco de investigación se identificó un método al que cuando le pasamos dos coordenadas en el espacio, y un tiempo de duración para el desplazamiento entre estos dos puntos. Interpola entre estos dos puntos tridimensionales un vector resultante en función al tiempo, esto es, según la variable tiempo que le pasemos nos dará un punto dentro del vector que une linealmente estos dos puntos en el espacio. Esta funcionalidad se llama “Lerp”, y la encontramos dentro del componente “Vector3”, el cual encontramos en todos los objetos dentro de nuestro mapa.

```
2 references
private IEnumerator MoveDrawer(Vector3 targetPosition)
{
    while (Vector3.Distance(transform.position, targetPosition) > 0.01f)
    {
        transform.position = Vector3.Lerp(transform.position, targetPosition, Time.deltaTime * moveSpeed);
        yield return null;
    }
    transform.position = targetPosition;
}
```

Ilustración 24 Extracto de código para el cálculo del vector de desplazamiento del cajón

Bien, ya tenemos la posibilidad de identificar los cajones y sabemos que podemos utilizar para desplazarnos en el espacio sin necesidad de tratarlos como objetos libres y evitando todos los problemas que ello trae. Pero para mover el cajón necesitamos saber de dónde partimos y a dónde queremos llegar.

Cada cajón dentro del mapa tiene una longitud diferente, por lo tanto, no se puede diseñar un recorrido genérico aplicable a todos los cajones, pues mientras que unos se abrirían perfectamente, otros se quedarían a medias y otros saldrían del mueble y se quedarían flotando en el aire. Esto se soluciona desarrollando un código que al pasarlo a cada cajón recoge de este sus dimensiones y realiza los cálculos necesarios para obtener una longitud igual a la mitad de su profundidad y así obtener para cada cajón una distancia de desplazamiento ajustada a sus dimensiones. La obtención de los datos necesarios para estos cálculos es posible gracias a la funcionalidad “GetRenderBounds” del componente “Bounds” dentro de todo objeto del mapa.

```
1 reference
Bounds getRenderBounds(GameObject objeto)
{
    Bounds bounds = new Bounds(Vector3.zero, Vector3.zero);
    Renderer render = objeto.GetComponent<Renderer>();
    if (render != null)
    {
        return render.bounds;
    }
    return bounds;
}
```

Ilustración 25 Extracto de código para el cálculo de dimensiones del cajón

Pero esto no es suficiente porque mientras que ya somos capaces de mover los cajones de un punto a otro a través de este vector y en relación con su profundidad, estos cajones se mueven hacia delante según los vectores generales del mapa y no en base a la propia orientación que tienen los cajones dentro del mapa. Es decir que según la orientación que tenga el cajón dentro del mapa podemos pensar que, al activar el movimiento del cajón para abrirlo, este se moverá hacia delante, hacia donde identificamos que debería ser “hacia delante” del cajón pues el tirador de este se encuentra mirando en esa dirección, pero el cajón termina por moverse en cualquier otra dirección atravesando incluso otros objetos pues su “hacia delante” no es el mismo “hacia delante’ que el mapa le dice de seguir.

Este último problema se soluciona realizando antes de definir el punto de inicio y punto final del movimiento del cajón una serie de cálculos que nos permitan identificar la orientación del cajón respecto al mapa. Una vez obtenemos su orientación utilizando su componente “Transform” como elemento que devuelve la información espacial de un objeto, podemos hacer que ese vector resultante entre el punto de inicio y punto final del movimiento tenga una dirección adecuada y modificada para la disposición en el mapa de cada cajón. De manera que al tirar de un cajón este se mueva “hacia delante” de manera intuitiva, es decir, donde está su tirador, y no “hacia delante” de lo que las coordenadas generales del mapa indican.

d) Recogida e interacción con objetos del mapa

Llegados a este punto, la interacción con los objetos del mapa no fue ningún problema a solucionar, ya sabíamos tratar con los problemas de las texturas, las colisiones y la identificación de objetos. Solo surgieron dos problemas fáciles de solucionar.

El primero sería introducir objetos dentro de los cajones y que estos se comporten como objetos reales dentro de los mismos, moviéndose con estos. Fácil de solucionar al establecer a los objetos de dentro del cajón como elementos relacionados con el cajón, y enlazar sus posiciones en el mapa, esto es que cuando cambie la posición del cajón en el mapa, lo hagan en las mismas condiciones los objetos dentro de este.

Ilustración 26 Item revolver dentro de un cajón en el editor Unity

La segunda es establecer una distribución en el mapa de los objetos para hacer un desarrollo de una partida divertido e interesante, así como establecer un sistema de relación a través de tags que permitan la utilización de objetos de manera combinada, por ejemplo, puertas y llaves que las abran. Pero este problema es algo que se soluciona probando el juego y pensado que distribución es la mejor, sin mucho más trabajo detrás.

Servidores y juego multijugador

Por último, discutimos qué sistema utilizar para implementar el componente multijugador que dará vida a nuestro juego, y nuevamente surgen varias opciones. Inicialmente, consideramos la posibilidad de desarrollar un juego P2P (peer to peer), en el cual uno de los jugadores actuaría como servidor de la partida, mientras que los demás se conectarán a él para jugar. Para esto teníamos pensado utilizar Mirror, un complemento de Unity que nos proporciona una arquitectura cliente-servidor para sincronizar la información del juego entre varios dispositivos.

Sin embargo, en la actualidad, la implementación de un juego P2P sin la ayuda de terceros se ha vuelto extremadamente difícil, si no imposible, debido a diversos desafíos que plantea este modelo.

En términos de seguridad, establecer conexiones directas entre jugadores en un juego P2P puede generar un riesgo considerable de ataques maliciosos y exposición de datos. Proteger la integridad de la partida y garantizar la privacidad de los usuarios se convierte en una tarea compleja que requiere la implementación de medidas adicionales de seguridad.

El rendimiento del juego P2P también puede verse afectado debido a la variabilidad de las conexiones de los jugadores. La calidad de la experiencia de juego puede depender en gran medida de la velocidad y latencia de la conexión de cada participante, lo que puede generar problemas de sincronización y retrasos que afectan negativamente la jugabilidad. Estos desafíos se vuelven más complejos a medida que aumenta el número de jugadores y la interacción simultánea.

Además, se suma la dificultad adicional de atravesar el NAT (Network Address Translation) de cada jugador. Esto implica configurar los routers y abrir los puertos necesarios para permitir la conectividad requerida por el juego, lo cual puede ser complicado. Además, algunos proveedores de servicios de Internet pueden imponer restricciones que dificultan aún más la conexión directa entre jugadores.

Tras mucha investigación descartamos esta idea y nos adentramos en el ámbito de los servidores dedicados.

Una de las principales diferencias entre el enfoque de los servidores dedicados y nuestra idea inicial de P2P radica en cómo se maneja la transferencia de información. En lugar de que los clientes se comuniquen directamente entre sí, ahora todos los datos pasan a través de un servidor central antes de ser enviados a cada cliente individualmente.

Esta nueva estructura nos ofrece varias ventajas. Al utilizar servidores dedicados, podemos tener un mayor control sobre la sincronización y la seguridad de los datos del juego. Además, al centralizar la comunicación, nos permite implementar lógicas de juego más complejas y realizar un seguimiento más preciso de la actividad de los jugadores.

Nos llamaron la atención los servidores de Unity Gaming Services, debido a que ofrecían soporte para hasta 50 jugadores concurrentes al mes de forma gratuita, lo cual nos pareció perfecto, considerando que durante los primeros meses de vida del juego no esperamos tener una gran cantidad de jugadores. Además, esto nos permitirá acumular fondos para cubrir el costo de 0.16 euros por cada jugador adicional.

Después de implementarlo, nos encontramos con un nuevo desafío. Como mencionamos anteriormente, planeamos lanzar nuestro juego en Steam, por lo que es fundamental garantizar una experiencia de usuario lo más agradable posible. Para lograr esto, necesitamos que el juego sea completamente compatible con las funciones de Steam, como la posibilidad de agregar amigos de la lista de amigos a tus partidas, unirse directamente a la sala de un amigo y poder visualizar partidas de otros jugadores.

Tras intentar implementar estas funcionalidades de Steam en nuestro juego, nos encontramos con Steamworks, una plataforma que ofrece una amplia gama de herramientas y servicios para desarrolladores que desean integrar sus juegos con la plataforma Steam.

Streamworks no solo nos proporciona las funcionalidades básicas que buscamos, sino que también nos permite utilizar los servidores de Steam para implementar partidas multijugador donde permitamos a los jugadores unirse a partidas privadas, públicas o protegidas por contraseña.

Esta integración con Steamworks nos permite lograr una sinergia perfecta entre nuestro juego y la plataforma Steam. Al utilizar los servidores de Steam, evitamos los inconvenientes y desafíos de crear una arquitectura P2P desde cero. Pudiendo así aprovechar la infraestructura existente de Steam para manejar la comunicación y la sincronización de los datos del juego, lo que nos brinda una solución más robusta y confiable.

Tras haber tomado una decisión y seguir adelante con los servicios de steam, toca implementarlos.

a) Implementación del multijugador

Para el multijugador utilizaremos, aparte de Steamworks, Netcode for Game Objects el cual es un sistema desarrollado por Unity que nos permite sincronizar el estado de los objetos del juego en el entorno multijugador.

Para poder usar todos estos componentes deberemos primero instalar los respectivos paquetes. Para que todo funcione primero deberemos de, en el “Package Manager” de unity, instalar Multiplayer tools y Netcode for GameObjects. Después tendremos que ir al siguiente enlace para poder instalar Steamwork Facepunch Transport.

Steamworks Facepunch Transport es un complemento crucial que nos proporciona las capacidades necesarias para el transporte de información en la red dentro de nuestro juego. Está diseñado específicamente para integrarse perfectamente con Steam, lo cual nos brinda numerosas ventajas al utilizarlo. Podemos aprovechar al máximo las características y funcionalidades que ofrece Steam, mejorando significativamente nuestra experiencia de juego multijugador en términos de conectividad y rendimiento.

Una vez instalados es momento de aplicarlos en nuestro juego. Primero de todo tendremos que agregar en nuestro proyecto un objeto vacío al cual le añadiremos el script “Network Manager” el cual nos lo proporciona Netcode for GameObjects. Este script es la base de todo nuestro multijugador ya que es el que nos facilitara la conexión de los clientes al servidor (que en nuestro caso es un jugador más).

El script proporcionado por Netcode for Game Objects en Unity nos brinda varias variables personalizables para nuestro juego multijugador. Sin embargo, nos enfocaremos en dos de ellas: "Player Prefab" y "Network Transport". La variable "Player Prefab" nos permite especificar el objeto de jugador que se creará para cada jugador que se conecte al juego. Por otro lado, la variable "Network Transport" nos permite elegir el método de transporte de red utilizado para el intercambio de datos entre los jugadores. En este caso, seleccionaremos Steamworks Facepunch Transport como el método de transporte de red.

Una vez completada esta etapa, estamos listos para crear nuestro primer host y comenzar la primera partida. Sin embargo, para permitir que otros jugadores se unan a nosotros, todavía debemos implementar todas las características que nos ofrece Steamworks Facepunch. Para lograrlo, nos hemos basado en la documentación oficial de Steamworks Facepunch y hemos llevado a cabo una exhaustiva investigación, lo que nos ha permitido desarrollar desde cero los siguientes scripts necesarios para habilitar y gestionar la conectividad en nuestro juego.

b) Experiencia del usuario

Antes de abordar los aspectos técnicos, es importante describir la experiencia que deseamos brindar a los usuarios en nuestro modo multijugador. Al inicio, los jugadores tendrán la opción de crear o unirse a una partida. Si un jugador decide crear una partida, será redirigido a una sala de espera o "lobby" donde esperará a que otros jugadores se unan para comenzar a jugar. Aquí es donde entran en juego las interesantes funcionalidades que nos ofrece Steam, las cuales debemos investigar y aprender cómo implementar. Con estas funcionalidades, desde el lobby podremos invitar a cualquier persona conectada a la plataforma de Steam a unirse a nuestra partida, incluso si no han iniciado el juego. Además, nuestros amigos tendrán la capacidad de unirse automáticamente si nos ven conectados en la misma partida. Por otro lado, si un jugador decide unirse a una partida existente, deberá ingresar un código (generado automáticamente al iniciar una partida por el anfitrión) para unirse a dicha partida.

Dentro del lobby, se mostrarán las imágenes de perfil y los nombres de usuario de todos los jugadores presentes. Para iniciar la partida, los jugadores deberán presionar el botón "Ready" una vez estén preparados. Cuando todos los jugadores estén listos, el anfitrión verá un botón "Start game" que, al presionarlo, redirigirá a todos los jugadores al mapa donde comenzará la partida.

Antes de comenzar la partida, se seleccionará aleatoriamente a uno de los jugadores para que asuma el rol de monstruo, apareciendo con el correspondiente prefab. A partir de ese momento, todos los jugadores podrán disfrutar plenamente de la partida.

c) Estructura del Lobby

Un lobby es un entorno virtual en el que los jugadores se encuentran y se preparan antes de iniciar una partida en un juego en línea. En nuestro juego, el lobby desempeña un papel fundamental, ya que brinda a los usuarios un espacio donde pueden reunirse y organizarse antes de comenzar una partida.

Comencemos con la implementación de los servicios de Steam en nuestro juego mediante el script "SteamManager".

El código se basa principalmente en el uso de manejadores de eventos para diversos eventos de Steam, como la creación de un lobby, la incorporación o salida de miembros del lobby, las invitaciones de amigos, y más. Además, cabe destacar el uso del método "Awake()", el cual nos asegura que solo haya una instancia de "SteamManager" en la escena.

```
Mensaje de Unity | 0 referencias
private void Start()
{
    transport = GetComponent<FacepunchTransport>();


    SteamMatchmaking.OnLobbyCreated += SteamMatchmaking_OnLobbyCreated;
    SteamMatchmaking.OnLobbyEntered += SteamMatchmaking_OnLobbyEntered;
    SteamMatchmaking.OnLobbyMemberJoined += SteamMatchmaking_OnLobbyMemberJoined;
    SteamMatchmaking.OnLobbyMemberLeave += SteamMatchmaking_OnLobbyMemberLeave;
    SteamMatchmaking.OnLobbyInvite += SteamMatchmaking_OnLobbyInvite;
    SteamMatchmaking.OnLobbyGameCreated += SteamMatchmaking_OnLobbyGameCreated;
    SteamFriends.OnGameLobbyJoinRequested += SteamFriends_OnGameLobbyJoinRequestedAsync;
}
```

Ilustración 27 Extracto de Código con la creación de los manejadores de eventos necesarios

Aquí podemos apreciar todos los manejadores de evento, con sus respectivas funciones, que usaremos en nuestro juego.

A modo de ejemplo, vamos a mostrar cómo implementar la creación de un nuevo lobby en nuestro juego. Primero, vamos a llamar al método "SteamMatchmaking.OnLobbyCreated" y asignar la función "SteamMatchmaking_OnLobbyCreated".

```
2 references
private void SteamMatchmaking_OnLobbyCreated(Result result, Lobby lobby)
{
    if (result != Result.OK)
    {
        Debug.Log("Lobby was not created");
        return;
    }
    lobby.SetPublic();
    lobby.SetJoinable(true);
    lobby.SetGameServer(lobby.Owner.Id);
    Debug.Log($"Lobby was created by {lobby.Owner.Name}");
    NetworkTransmission.instance.AddMeToDictionaryServerRPC(SteamClient.SteamId, lobby.Owner.Name, NetworkManager.Singleton.LocalClientId);
}
```

Ilustración 28 Extracto de código para la creación del "Lobby"

En el caso de este ejemplo, la función "SteamMatchmaking_OnLobbyCreated" creará un lobby público al que cualquier jugador puede unirse sin necesidad de una invitación. Además, el servidor actuará como el host del lobby. Posteriormente, se añadirá el lobby a un diccionario para poder listar los lobbies activos en un momento posterior, si así lo deseamos.

Una vez completada la implementación de todos los servicios que consideremos necesarios, es hora de gestionar el estado y la lógica dentro del juego. Para llevar a cabo esta tarea, creamos el script "GameManager".

El script GameManager se encarga de mantener un registro de los jugadores conectados y sus respectivos estados. Cuando un jugador se desconecta, el script se encarga de limpiar la información relacionada con ese jugador, eliminando su tarjeta de jugador y actualizando el diccionario de jugadores.

Cuando un jugador se une al lobby, se agrega su información al diccionario de jugadores. El script crea una tarjeta de jugador para mostrar la información del jugador en la interfaz del lobby.

El script también incluye la función ReadyButton(), que se activa cuando un jugador presiona el botón de "listo" en el lobby. Esta función actualiza el estado de preparación del jugador y verifica si todos los jugadores están listos para comenzar la partida. En caso afirmativo, se activa el botón de inicio de partida para que el anfitrión pueda iniciar el juego gracias al método “StartGame()”.

Este método tiene una particularidad: no funciona de manera independiente. Depende de otro método y de ciertos atributos específicos que han sentado las bases para muchas implementaciones futuras.

```
[ServerRpc(RequireOwnership = false)]
0 referencias
public void StartGame()
{
    RpcChangeScene();
}

[ClientRpc]
1 referencia
private void RpcChangeScene()
{
    // Cambiar a la escena de juego
    NetworkManager.Singleton.SceneManager.LoadScene("Casa Niño", LoadSceneMode.Single);
}
```

Ilustración 29 Extracto de código encargado de establecer la escena para los jugadores de la partida

Como se muestra en la imagen, el método "StartGame()" cuenta con el atributo "ServerRpc(RequireOwnership = false)" y llama al método "RpcChangeScene()", el cual tiene el atributo "[ClientRpc]" y se encarga de cambiar la escena para los jugadores. Estos atributos determinan qué métodos se ejecutan en el servidor y cuáles son ejecutados por los clientes. En este caso, para que todos los jugadores cambian de escena, es necesario que el servidor ejecute "StartGame()", lo que a su vez hará que cada jugador individualmente ejecute el método "RpcChangeScene()" y cambie de escena.

En resumen, estos atributos se utilizan para sincronizar la ejecución de métodos entre el servidor y los clientes en una red de juego, permitiendo la comunicación y la actualización del estado en tiempo real.

Este script utiliza “PlayerInfo”, otro script el cual es el responsable de gestionar la información de cada jugador, como su nombre, imagen y estado de preparación.

El aspecto más destacado de este script es el método "GetTextureFromImage(Steamworks.Data.Image image)". En este método, se itera sobre cada píxel de la imagen para asignar su valor correspondiente a una nueva textura llamada "avatar". Esta textura se aplica y se devuelve, permitiendo que sea visualizada por el resto de los jugadores en el juego.

```
1 referencia
private Texture2D GetTextureFromImage(Steamworks.Data.Image image)
{
    avatar = new Texture2D((int)image.Width, (int)image.Height);
    for (int x = 0; x < image.Width; x++)
    {
        for (int y = 0; y < image.Height; y++)
        {
            var p = image.GetPixel(x, y);
            avatar.SetPixel(x, (int)image.Height - y, new Color(p.r / 255.0f, p.g / 255.0f, p.b / 255.0f, p.a / 255.0f));
        }

    }
    avatar.Apply();
    return avatar;
}
```

Ilustración 30 Extracto de código encargado de la sincronización

Finalmente, para lograr la sincronización de información entre los jugadores, utilizamos el script "NetworkTransmission" como una herramienta fundamental.

En este script, hemos empleado los atributos ServerRpc y ClientRpc como fundamentos para asegurar la comunicación adecuada entre todos los jugadores. Estos atributos nos permiten sincronizar eventos como la conexión de nuevos jugadores y la desconexión de jugadores existentes, así como también actualizar y compartir el estado de los jugadores entre todos los participantes.

Gracias a todos estos scripts, hemos logrado implementar la funcionalidad necesaria para que nuestros usuarios puedan crear y unirse a lobbies sin problemas. Sin embargo, todavía nos enfrentamos al desafío de sincronizar todos los aspectos de una partida para cada jugador.

d) Estructura de la partida

Como habíamos mencionado previamente, una vez que todos los jugadores estén listos y el anfitrión inicie la partida, los usuarios serán redirigidos a la pantalla del juego donde comenzará la sesión. Sin embargo, en este punto debemos abordar la gestión de la posición de aparición (spawn) y el prefab con el que cada jugador comenzará. Para esto, implementaremos un script llamado "PlayerManager" que se encargará de controlar estas situaciones.

La lógica del script es bastante sencilla. Utilizaremos el método "OnNetworkSpawn()" para controlar cada vez que un objeto de tipo "network" aparezca en la escena. Dentro de este método, aplicaremos la lógica necesaria para asignar a cada jugador el prefab correspondiente y su respectivo punto de aparición (spawn). De esta manera, garantizamos que cada jugador tenga la configuración adecuada al inicio de la partida.

En el estado actual del juego, si intentamos probarlo, nos encontraríamos con un problema importante. Cuando los jugadores aparecen en la escena, el juego no funcionará correctamente, ya que todos compartirán la misma cámara y el movimiento de los jugadores no se sincronizará entre ellos, entre otros problemas. Esto se debe a que antes de utilizar el juego en red, es necesario agregar ciertos componentes a nuestros prefabs en Unity para que puedan sincronizarse adecuadamente.

Ilustración 31 “Prefab” del personaje "Niño"

Estos componentes esenciales proporcionados por Netcode for Game Objects desempeñan un papel fundamental en la sincronización de los jugadores en la red. El componente Network Object permite que el objeto sea gestionado y reconocido por el sistema de red, facilitando la comunicación y actualización de estado. Por otro lado, el componente Client Network Transform asegura la sincronización del movimiento y la posición del objeto entre los jugadores conectados. Finalmente, el componente Owner Network Animator se encarga de sincronizar las animaciones del objeto entre los jugadores, garantizando una experiencia de juego coherente y consistente.

Para asegurar un control adecuado del movimiento de los jugadores en la red, además de los componentes mencionados anteriormente, es necesario realizar una modificación en el script que controla el movimiento de cada prefab. ¡Se agrega una condición simple “if (!IsLocalPlayer) {return;}" dentro de los métodos Start() y Update(). Esta condición permite que únicamente el jugador local controle el movimiento de su personaje.

Para abordar el problema de la sincronización de la cámara entre los clientes, identificamos que la raíz del problema era la activación predeterminada de la cámara en el prefab al momento de su aparición, lo cual genera interferencias con las cámaras de los demás jugadores. La solución consistió en desactivar la cámara por defecto en el prefab y agregar una línea de código en el script "PlayerControllerFirstPerson" para activar la cámara al momento de su aparición.

```
void Start()
{
    if (!IsLocalPlayer) { return; }

    playerAnim = GetComponent<Animator>();
    characterController = GetComponent<CharacterController>();

    if (!cam.enabled) { cam.enabled = true; }  // Manejo de la camara
    Cursor.lockState = CursorLockMode.Locked;


}
```

Ilustración 32 Extracto código encargado del control exclusivo de la Cámara por cada jugador

Con esta modificación, logramos que cada jugador tenga el control exclusivo sobre su propia cámara.

Con todo esto faltaría un último detalle fundamental para la jugabilidad dentro de una partida. Además de agregar los componentes necesarios a los prefabs de los jugadores, también es fundamental convertir todos los objetos dentro del mapa que interactúan con los jugadores en "Network Objects". De esta manera, garantizamos que las interacciones con dichos objetos se compartan entre todos los jugadores en la red, evitando que las interacciones sean únicamente locales.

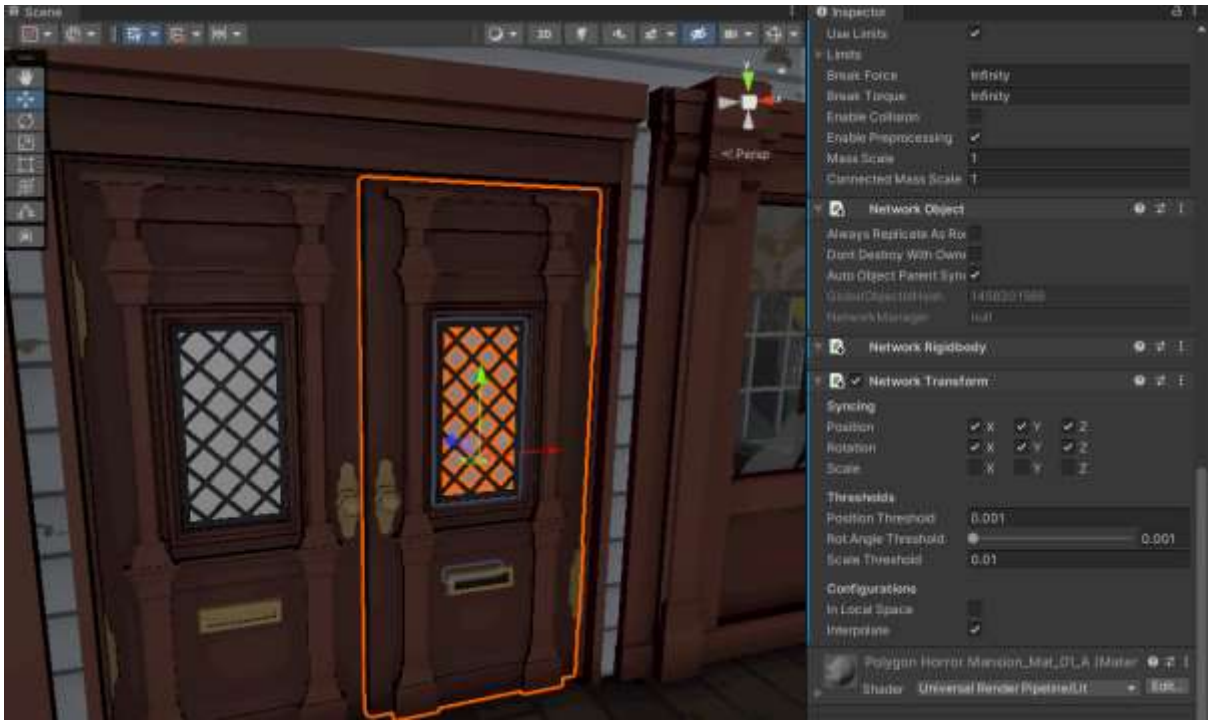

Ilustración 33 Puerta en el editor de Unity con los componentes necesarios para el multijugador

Como podemos ver, para que el movimiento de las puertas se comparta con el resto de los jugadores es indispensable el componente “Network Transform”.

Para concluir, a través de la combinación de diversos scripts y componentes, hemos establecido la infraestructura necesaria para que los jugadores puedan crear y unirse a lobbies, sincronizar su estado, acciones y objetos en tiempo real, control de cámaras y la interacción con elementos del mapa. El resultado final es un entorno de juego dinámico y colaborativo que permite a los jugadores compartir un mismo mundo y una misma experiencia.

CONCLUSIONES

En resumen, el proyecto de desarrollo de un videojuego en Unity abarca tanto el aspecto de estudio de mercado y viabilidad económica como la resolución de desafíos técnicos. Esta combinación de enfoques permite tomar decisiones fundamentales sobre el tema y los aspectos generales del videojuego, así como garantizar una implementación exitosa.

El estudio de mercado y la viabilidad económica son etapas cruciales para determinar la demanda potencial del videojuego y evaluar su rentabilidad. En este proceso, se recolecta y analiza información relevante sobre las preferencias de los jugadores, las tendencias del mercado, la competencia y los modelos de monetización. Las conclusiones extraídas de este análisis son clave en la toma de decisiones posteriores y guiarán el diseño general del juego.

En cuanto a los aspectos técnicos, el proyecto busca resolver desafíos relacionados con el desarrollo del videojuego en Unity. Esto incluye la implementación de mecánicas de juego complejas, la integración de efectos visuales y de sonido, y por supuesto el desarrollo del multijugador con los desafíos que esto conlleva respecto a la sincronización de los eventos del juego, la gestión de la comunicación en red y la resolución de problemas de latencia, aspectos clave para ofrecer una experiencia multijugador compartida.

Todos estos puntos fueron abordados con éxito, pero no todos con sencillez. Se tuvo que invertir mucho tiempo en aprender a utilizar nuevas herramientas y sobre todo a saber adaptarse a la situación, probando una solución y si no tenía éxito pasar rápidamente a otra perspectiva que nos permitiera dar otro enfoque, sin perder mucho tiempo ya que este ha sido un aspecto que tener muy en cuenta durante el desarrollo.

Así mismo el trabajo en equipo resulto ser una pieza indispensable en el desarrollo del proyecto. Para un trabajo en el que se ven involucradas varias personas es fundamental saber identificar los puntos fuertes y débiles de cada integrante para poder realizar un reparto eficiente de las tareas. También es muy importante saber delegar y confiar en el buen trabajo de tus compañeros, pedir ayuda cuando se necesita y saber apoyar y ser apoyado por tus compañeros.

Sin embargo, pese a tener un producto bastante pulido y terminado como resultado de este proyecto, no damos por terminado aun el trabajo. Como ya se ha dicho se pretende hacer de este videojuego algo que pueda producir cierto retorno económico, así como una muestra de nuestras capacidades. Es por esto el equipo va a seguir trabajando en el desarrollo tanto técnico y jugable, desarrollando más mapas, personajes y mecánicas que ofrezcan una experiencia completa al jugador, como en la parte comercial, estudiando las distintas posibles figuras jurídicas que nos permitan empezar una actividad económica a través de Steam, siendo la “Comunidad de Bienes” la más recomendable para empezar este camino.

En conclusión, creemos que fue un completo acierto la decisión tomada para este proyecto final. El desarrollo de videojuegos nos ha permitido aprender mucho sobre otros lenguajes y herramientas, y sobre nosotros mismos y la resolución de problemas. Queda lejos esa idea de sentarse delante de un ordenador con una idea presuntamente exitosa, trabajar unas horas y lograr el éxito de la noche a la mañana. El trabajo debe ser sólido desde su base, deben cometerse errores y aprender de ellos

BIBLIOGRAFÍA

[1] ***Code Monckey*** (YouTube: @CodeMonkeyUnity)

[2] ***Zyger*** (YouTube: @ZygerGFX)

[3] ***Dapper Dinosaur*** (YouTube: @DapperDinosaur)

[5] ***Shrine App*** (YouTube:@ShrineApp)

[6] ***Pixelfizz1718*** (YouTube: @pixelfizz1718)

[7] ***JayCode_dev*** (YouTube: @JayCode_dev)

[8] ***Llam Academy*** (YouTube: @LlamAcademy)

[10] ***Unity Asset Store*** (Unity Asset Store)

[11] ***Unity Forum*** (Unity Forum)

[12] ***Stack Overflow*** (Stack Overflow)