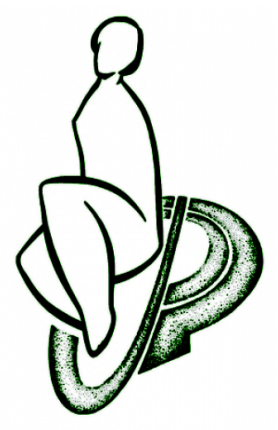


INSTITUTO DE EDUCACIÓN SECUNDARIA JOSÉ PLANES

Departamento de Informática y Comunicaciones

Técnico Superior en Desarrollo de aplicaciones Web

C/ Maestro Pérez Abadía, 2

30100 Espinardo – Murcia

T. 968 834 605

30010577@murciaeduca.es

www.iesjoseplanes.es

Memoria del proyecto

Desarrollo de aplicaciones web

Type Space

Autores/as:

Alberto González Melul

David Pastor López

Profesor/a-coordinador/a:

Luis Miguel Fernández Costa

Murcia, Junio de 2024

Enlace: Type Space

Queremos agradecer a nuestros tutores por el apoyo que nos han dado estos últimos meses por ir más allá de sus obligaciones como profesores y darnos consejos para hacer el proyecto que **queríamos** hacer.

Y a nuestros amigos que nos han ayudado a probar el juego y mejorarlo.

Contenido

1 Resumen extendido

Super TypeSpace es un juego de navegador creado usando Javascript y la librería 'Phaser 3' en el que el usuario debe de **escribir** lo más rápido posible para zafarse de hordas de naves espaciales y acumular puntos durante el mayor tiempo posible.

El juego se plantea como un modo 'supervivencia' donde la dificultad incrementa gradualmente con el tiempo y solo termina cuando el jugador pierde todas sus vidas.

La aplicación también cuenta con una tabla de puntuaciones donde el jugador puede registrar su nombre junto con su puntuación si esta entra dentro de un 'Top Jugadores' global.

Al ser un juego de **mecanografía,** es actualmente exclusivo para escritorio.

2 Palabras clave

Phaser 3, Trello, proyecto, Juego, mecanografía, naves, espacio, HTML5, Firebase, CSS, Javascript, Github, IDX, Vite.

3 Introducción

La idea del proyecto surgió cuando David buscó opiniones de otros profesionales sobre buenos proyectos, listando pros y contras al respecto:

Captura de pantalla de las tablas de brainstorming de Figma realizadas por David.

Entre esas ideas se encontraba la de hacer un juego de mecanografía; tenía bastantes ventajas; la lógica de la mecánica principal no es demasiado compleja, se entiende rápido y suele tener un diseño límpio.

Alberto: "Yo siempre he querido hacer un videojuego así que metí un poco de presión para hacer esta idea. En retrospectiva esto ha complicado otros aspectos del proyecto, pero estoy muy contento de que nos hayamos quedado con esta idea."

3.1 Decisiones del diseño inicial.

Lo que determinó en gran medida que 'pinta' iba a tener nuestro juego fueron principalmente dos factores.

- De qué recursos gráficos y de audio disponíamos.
- Qué suponía un mayor reto mecánicamente.

En nuestro caso un juego en el **espacio** implica no preocuparse tanto de cosas como la **gravedad** y sus gráficos suelen ser sencillos. Además los assets gráficos de naves espaciales eran muy accesibles y fáciles de utilizar.

4 Estado del arte/trabajos relacionados

A la hora de investigar sobre proyectos parecidos nos encontramos con que hay una abundancia **abrumadora** de proyectos similares; esto no nos preocupaba porque las muchas posibilidades de diseño daban hueco a innovar y darle personalidad a nuestro proyecto, sin embargo hizo que la labor de investigación fuera muy caótica inicialmente.

ZTYPE fue uno de los proyectos en los que más nos fijamos al principio. La idea principal es prácticamente idéntica, si bien obviamente más pulida que nuestro resultado final. Actualmente ZTYPE se puede jugar de forma gratuita en https://zty.pe

Captura de pantalla del juego ZTYPE

También encontramos bastantes repositorios públicos en github, aunque la mayoría utilizaban tecnologías ya deprecadas o funcionalidades que no nos interesaban.

Las cosas que principalmente nos llamaron la atención de estos proyectos fue su implementación de la dificultad; usando letras sueltas para enemigos diminutos, palabras largas y complicadas para enemigos grandes, etc.

5 Análisis de objetivos y metodología.

5.1 Objetivos

Con la idea del proyecto en mente, determinamos los siguientes objetivos:

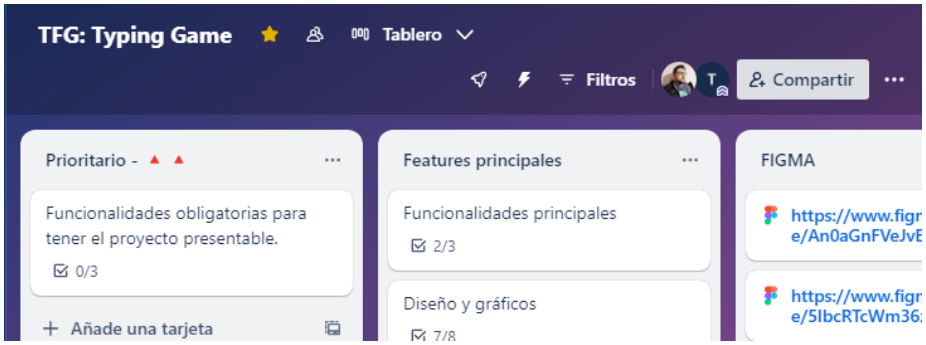
- Crear y gestionar una base de datos
- Estudiar sistemas para implementar físicas y animaciones
- Centrar el diseño en torno a la experiencia en escritorio
- Buscar e implementar ideas para todas las posibles mecánicas del juego.
- Implementar gráficos y efectos de sonido
- Implementar alguna manera de que el usuario interactuara con la base de datos.

El **grueso** de estos objetivos consistía en estudiar qué mecánicas y funcionalidades eran las mejores para nuestro proyecto, de ahí que los objetivos iniciales no fuesen muy concretos. Hablamos más sobre esto más adelante en el apartado de **diseño y resolución**.

5.2 Metodología

Para llevar a cabo los diferentes objetivos hemos utilizado las siguientes metodologías de trabajo:

- Un tablero de trello con listas de funcionalidades, enlaces de interés, archivos compartidos...



Captura de pantalla de nuestro tablero de Trello.

Al ser un proyecto muy centrado en un solo apartado (lógica de programación) y al ser solo dos integrantes de equipo el reparto de tareas ha sido menos esquematizado.

- Un repositorio de Github donde guardar cada funcionalidad en ramas diferentes para luego ir integrándose al proyecto principal.
- Reuniones bi-semanales por Discord para determinar
    - Qué funcionalidad o mecánica de juego se está programando.
    - Qué dificultades se están encontrando.
    - Qué posibles soluciones se van a implementar o investigar al respecto.
- Grupo de telegram con nuestro tutor donde comentar tecnologías, dudas, avisar de retrasos, etc.

6 Diseño y resolución del trabajo realizado.

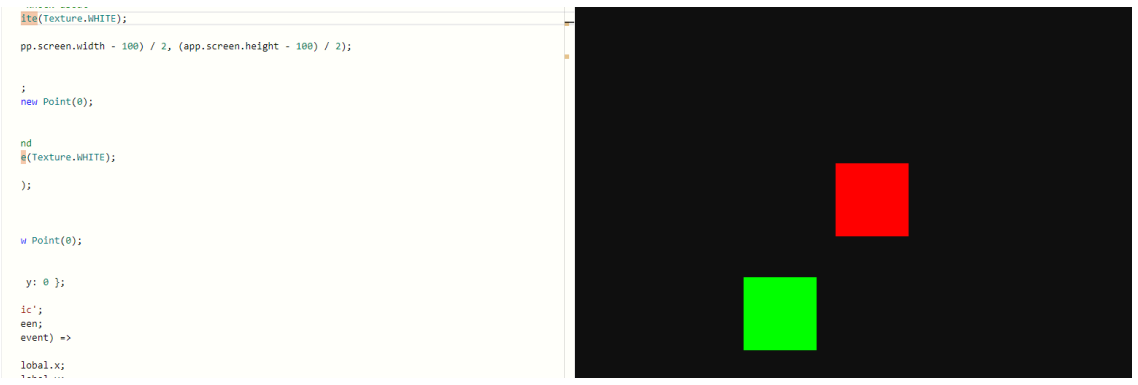
La principal dificultad de este proyecto ha sido trabajar en algo que no hemos hecho nunca, así que una gran porción del trabajo ha sido estudiar tecnologías que no hemos utilizado antes y decidir qué funcionalidades podíamos integrar y cuáles no.

6.1 Investigación inicial y lluvia de ideas.

6.1.1 Investigación de lenguajes y estructura.

El primer periodo de investigación se centró, principalmente, en pensar 'Cómo se hace un videojuego' en javascript. Inicialmente estudiamos el motor gráfico **PixiJS**. Este permite integrar un nodo 'especial' en nuestra página web que mueve sprites al recibir ciertas instrucciones.

Tras realizar algunos prototipos con figuras simples, vimos que requería de mucho código y costaba mucho implementarlo.



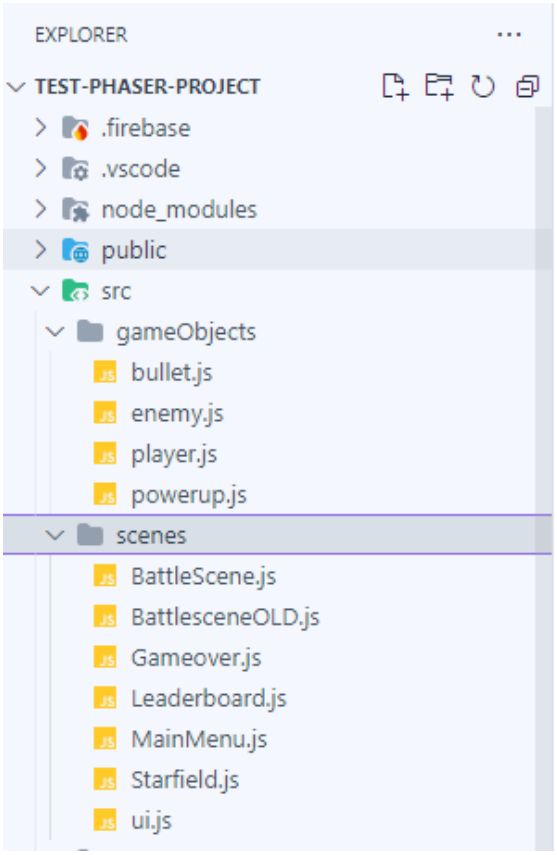
Página de documentación de PixiJS donde te muestran código junto con un ejemplo interactivo de cada funcionalidad. Esta pantalla muestra la colisión entre dos elementos.

PixiJS cuenta con muchos tutoriales sobre cómo realizar muchas mecánicas diferentes (gravedad, colisiones, mover

sprites) y soporta juegos en 3D pero, justamente por eso, era 'demasiado' para lo que teníamos en mente. Implementarlo en una página web también era bastante complicado y perdimos varios días solo intentando integrarlo en una página de Angular.

Eventualmente nos quedamos con una librería de Javascript llamada **Phaser 3.** (Agradecimiento especial a Luismi por la recomendación.)

Ésta, al ser una librería y no un motor gráfico, proporciona una serie de métodos que controlan factores como la gravedad, velocidad, zona del juego... pero sigue siendo compatible con programación Javascript convencional y da más control sobre lo que hace el programa sin tener que estudiar tanto.



Aquí un ejemplo de cómo en Phaser 3 todos los elementos del juego se pueden separar en archivos JS diferentes (Balas, enemigos, jugador, mejoras...).

Este entorno de trabajo es IDX; visualmente es casi idéntico a Visual Studio Code y cuenta con su propio asistente de inteligencia artificial: ***Gemini.***

La página web también pasó a ser un HTML5 simple y, como Phaser 3 se puede desglosar en muchos archivos diferentes y requiere de un servidor web para hacer pruebas, también estudiamos e implementamos **Vite**.

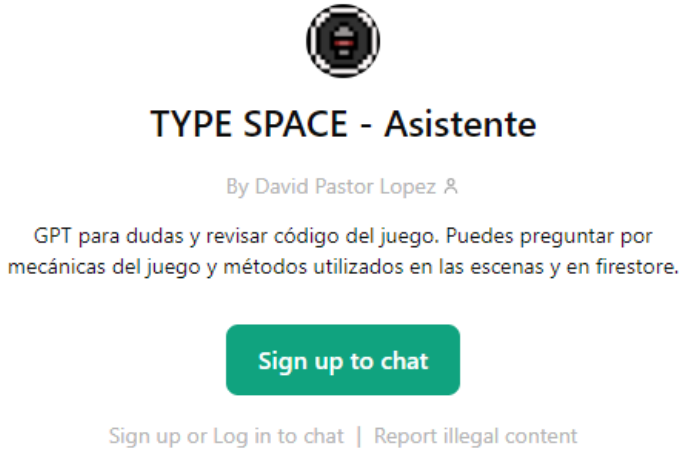
Vite es un servicio que ofrece herramientas de desarrollo para facilitar el sistema de carpetas, compilación, comandos para 'construir' el proyecto en un servidor local con mucha facilidad...

Además de usar Github a mitad de proyecto le dimos algo de uso a la herramienta de google **IDX,** un nuevo entorno de desarrollo que se guarda en la nube y que nos permite trabajar en el mismo entorno de trabajo e incluso programar de forma conjunta al mismo tiempo en el mismo código.

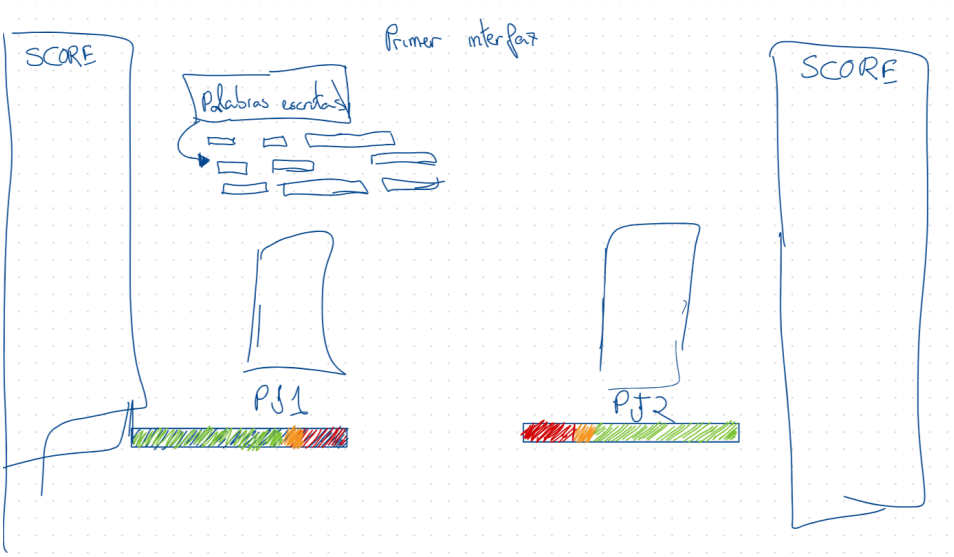
Para la **base de datos** y **despliegue** hemos optado por **Firebase.** Firebase es una plataforma de desarrollo de Google que proporciona herramientas como **Firestore**; Una base de datos NoSQL que, si bien difícil de entender al principio, resulta muy ligera y fácil de utilizar con algo de práctica, es muy escalable y sincroniza los datos a tiempo real.

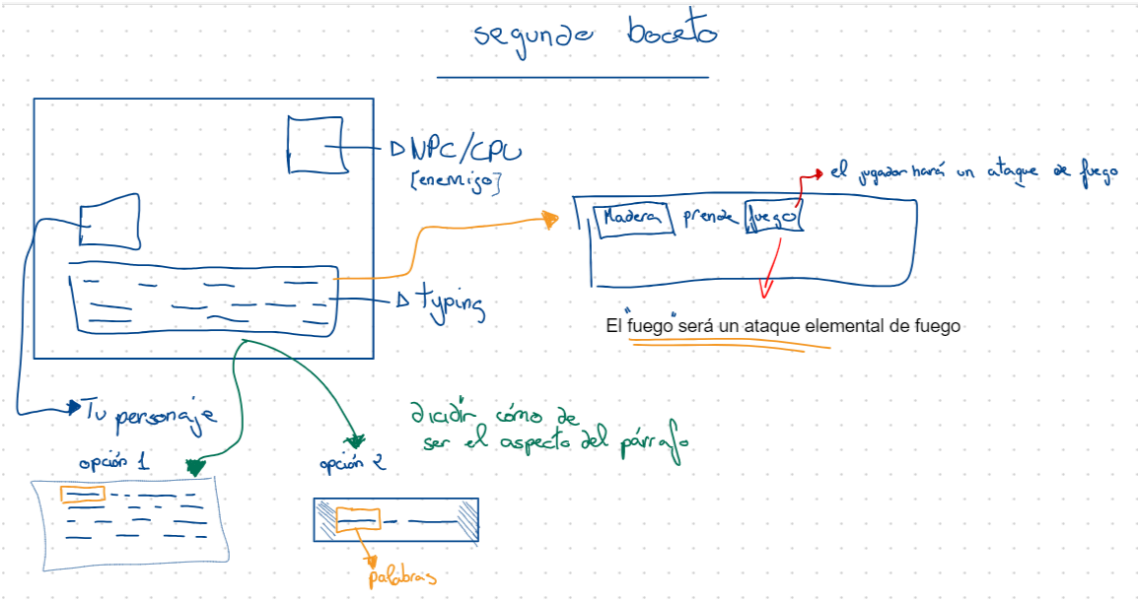
Sopesamos el almacenar ciertos datos en archivos JSON pero al final muchos de estos datos los hemos almacenado en bases de datos ya que Firestore puede traer y sincronizar todos estos datos rápidamente. También permite editar los datos con mucha facilidad.

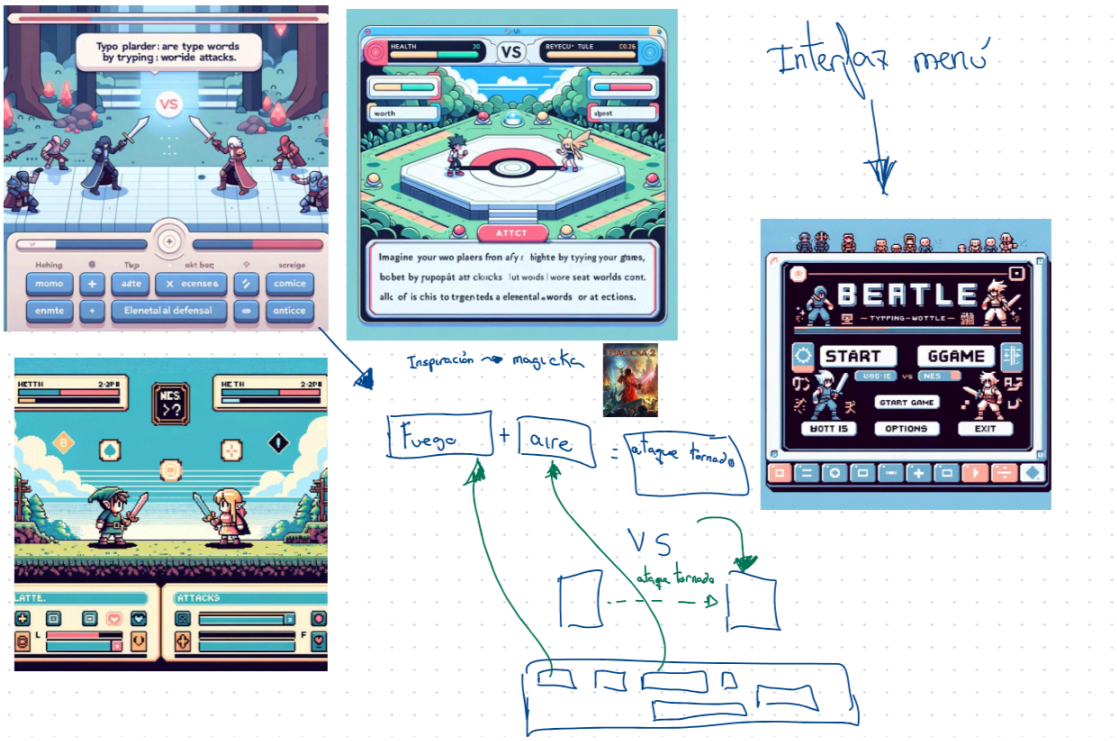
Otra herramienta que nos resultó muy interesante fue la creación de un modelo de **inteligencia artificial** especializado concretamente en nuestro proyecto. Le puedes pedir que te de información sobre las diferentes funciones, te hable de las diferentes mecánicas o te ayude a buscar errores ...



6.1.2 Investigación de mecánicas de juego y funcionalidades.







Bocetos iniciales e imágenes generadas por IA para investigar posibles ideas. La primera idea que se nos pasó por la cabeza era darle la estructura de un juego de lucha.

Decidir **qué** va a ser nuestro juego y **cómo** se va a jugar ha supuesto gran parte del trabajo del proyecto. Aquí hay una lista de las muchas mecánicas de juego que hemos estudiado, para las que hemos visto decenas de tutoriales al respecto y que hemos ido implementando o descartando por complejidad/falta de tiempo.

Aquí dejamos una lista de las funcionalidades que hemos investigado. Hablaremos de algunas de ellas más adelante:

- Que el jugador pueda escoger entre distintas naves.

- Que el juego tomara la estructura de un juego de pelea y que el jugador luchase contra oponentes en un 1 vs 1.
- Que el jugador avance por distintos planetas.
- Un sistema de login.
- Vista con funcionalidades especiales en el móvil.
- Multijugador y modo Vs.
- Moverse por los menús teniendo que escribir las palabras de este.
- 'Powerups'.
- Poderes activables.
- Enemigos formados por distintos segmentos.
- Cinemática inicial con historia.
- Palabras separadas por dificultad.
- Distintos tipos de armas.
- Distintos tipos de enemigos.
- Una tabla de puntuaciones con un 'Top 10 Jugadores'.
- Enemigos que reaccionan al ser impactados por proyectiles.

Alberto: *"Podría parecer que nos estábamos pasando de ambiciosos, pero la idea no era implementar TODAS estas funcionalidades, más bien tirar cosas a la pared hasta tener la visión de 'algo', además, si tuviese que repetir este proyecto con lo que he aprendido hasta ahora podría implementar una buena porción de esta lista."*

Finalmente, la versión final del proyecto cuenta con las siguientes características:

- Los enemigos muestran una palabra y, al ser escrita por el usuario, la nave del jugador dispara un proyectil a esa nave.
- Los enemigos pierden vida y son aturdidos hacia atrás al recibir el primer disparo.
- Los enemigos se ‘inmolan’ contra el jugador para reducir su salud.
- La frecuencia con la que los enemigos aparecen en pantalla crece con el tiempo.
- Las palabras tienen un rango de ‘dificultad’ que determina lo comunes que son y cuantos puntos otorgan.
- Las palabras no se pueden repetir y el jugador puede escribir cualquier palabra que se muestre en pantalla. Escribir una palabra que no puede acertar a ningún enemigo borra el progreso del jugador y le otorga un ‘fallo’ así como una bajada de puntos.
    - Por ejemplo, si el jugador escribe ‘pistr’ intentando escribir “pista”, “pistola” o “pistacho” la aplicación calcula que ‘pistr’ no puede ‘llegar’ a ninguna palabra activa, penalizando al jugador. El campo de texto del jugador se vacía automáticamente.
- Al terminar la partida, el juego permite introducir el nombre del jugador si su puntuación es apta para estar en el top 10.

6.2 Recursos gráficos y de audio.

Los recursos gráficos disponibles también determinan la forma final del proyecto, en nuestro caso hemos utilizado los siguientes recursos:

Imágenes (assets)

- PIXEL SPACESHIPS por Medimon Games bajo licencia CC-BY 4.0

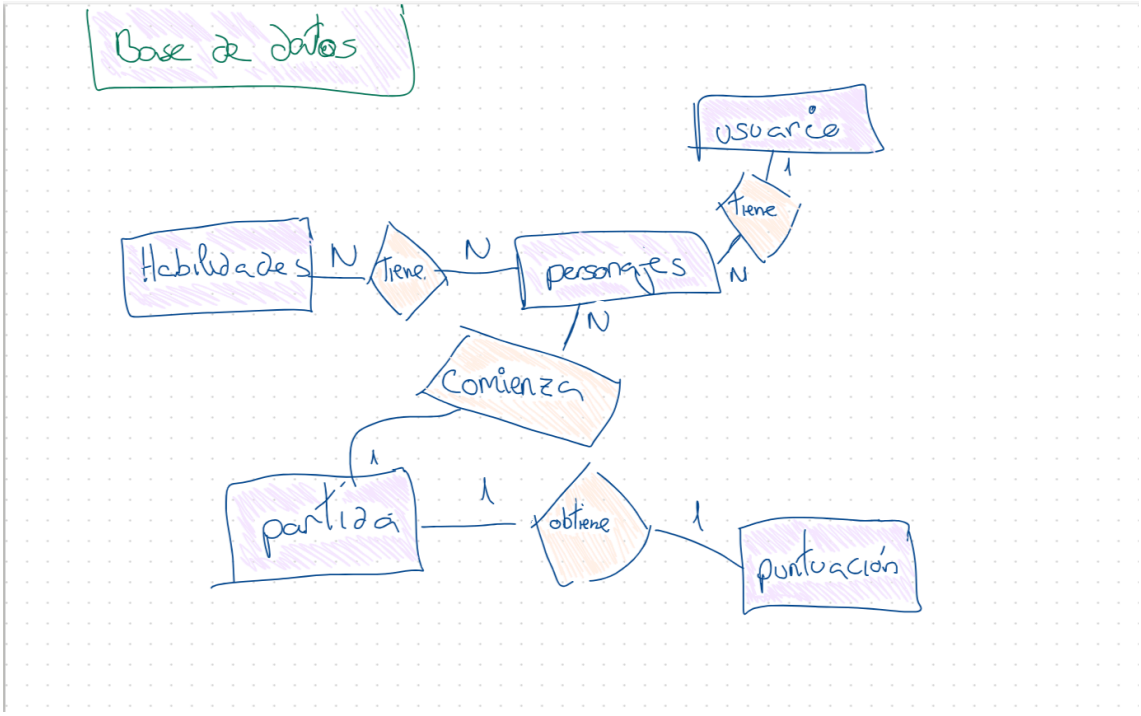
Efectos de partículas y colisiones

- Battle VFX: Hit Spark por Pimen bajo licencia CC0
- Gothicvania Magic Pack 9 por Ansimuz bajo licencia CC0

Música y efectos de sonido

- Música: Tallbeard Studios: Album: Three Red Hearts - Pixel War 1 y Pixel War 2, bajo licencia CC0
- Efectos de disparo y colisiones de las naves Brackeys' Platformer Bundle bajo licencia CC0
- Efectos de sonido de menú: Autor Noahkuehne - pack sonidos bajo licencia CC-BY 4.0

6.3 Esquemas iniciales de la base de datos.



Bocetos iniciales de posibles bases de datos.

Al principio complicamos en exceso la base de datos al no estar acostumbrados a un proyecto así. Estábamos aplicando la lógica que aplicaríamos a un proyecto más convencional donde los datos se tratan de maneras diferentes.

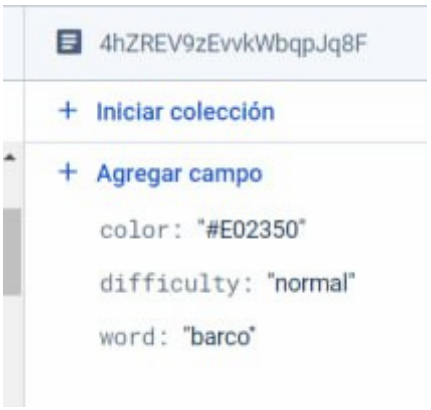
Fue después que nos dimos cuenta que al crear un videojuego es mucho más fácil y rentable crear archivos de configuración para almacenar todos los parámetros que sean variables y almacenar los datos con los que el jugador va a interactuar.

No solo eso sino que, al no necesitar relaciones entre tablas, una base de datos **NoSQL** como es la de **Firebase** nos venía muy bien.

Firebase es un servicio de Google que ofrece herramientas como una base de datos llamada **Firestore** y un servicio de hosting con opciones gratuitas llamada **Firebase Hosting**.

Cargar datos desde **Firestore** es muy rápido, tiene una interfaz web muy cómoda y es el equivalente a guardar datos en archivos **JSON.**

Aunque lo intentamos poner en práctica al principio, nos acabó pareciendo excesivo implementar un sistema de inicio de sesión. Actualmente la base de datos solo almacena la **tabla de puntuaciones y las palabras de los enemigos**. Firestore es muy maleable así que si queremos cambiar la configuración de las palabras solo tenemos que acceder al gestor.

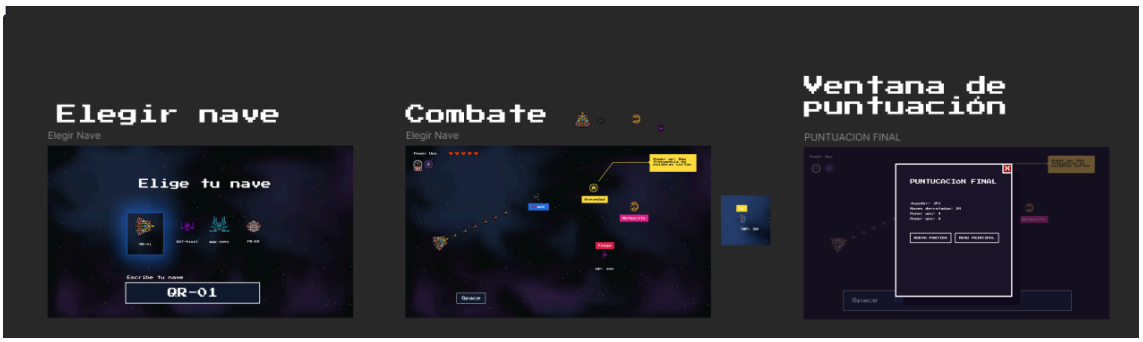


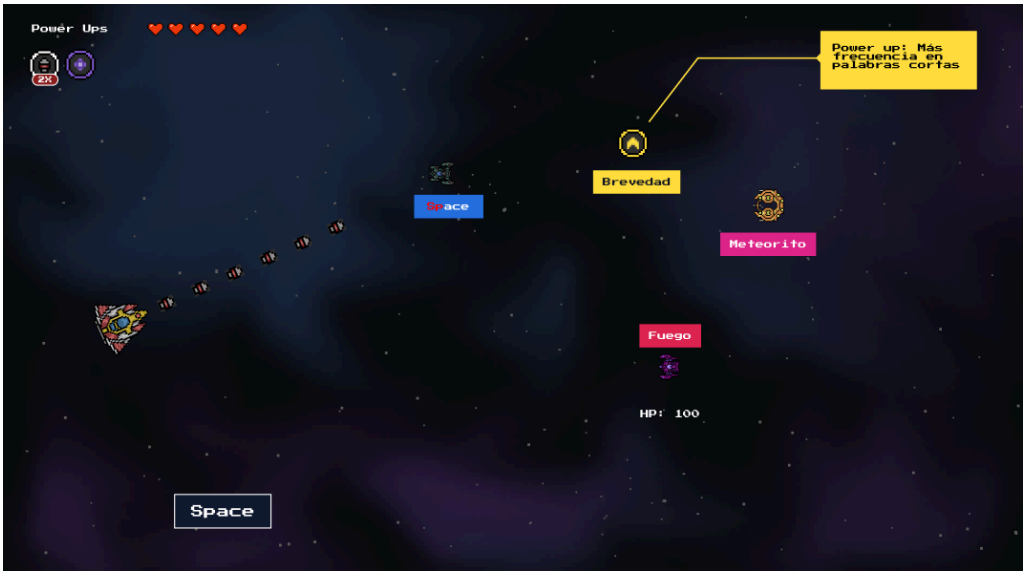
Imágen de un elemento dentro de la base de datos de firestore. Podemos cambiar los parámetros libremente y, a efectos prácticos, funciona igual que un archivo json al solo necesitar una llamada inicial.

6.4 Prototipos de bajo nivel.

Durante los prototipos de bajo nivel soltamos en Figma todos los recursos gráficos así como imágenes creadas por IA que suelen dar ideas sobre potenciales diseños, carátulas, etc.

Aquí dejamos ejemplos de estas imágenes generadas por IA, posibles paletas de colores por si hacíamos una carátula, diseños iniciales de la zona de juego, etc.





Un ejemplo inicial de cómo queríamos que se viese el juego.

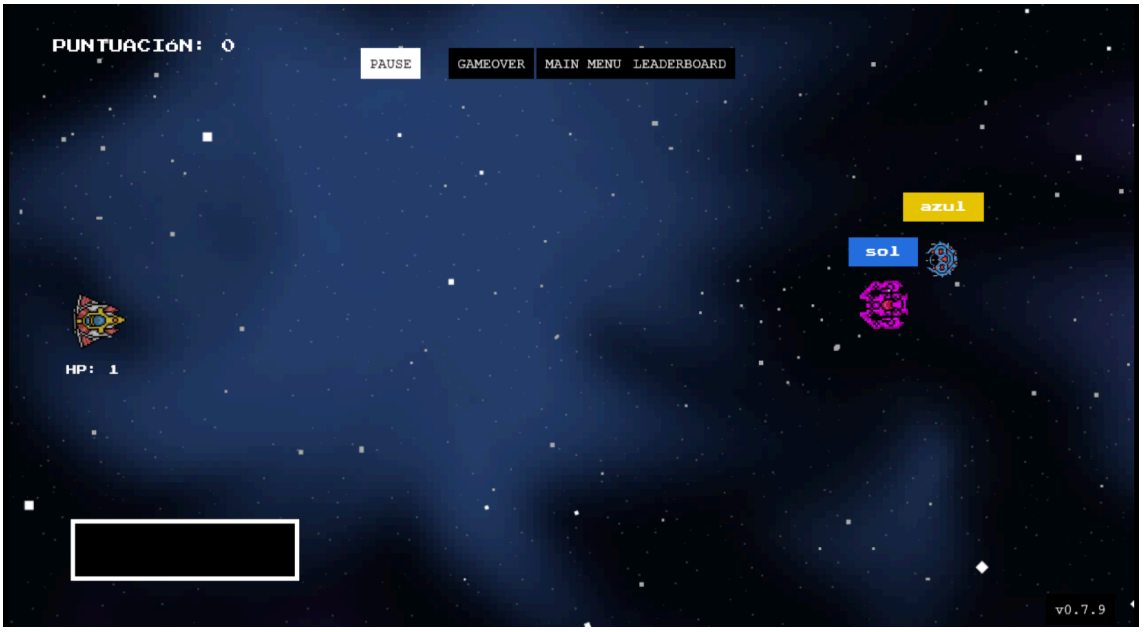


Imagen actual del juego. v0.7.9

6.5 Partes más interesantes del desarrollo.

Parte de lo que ha hecho que el desarrollo fuese tán complejo inicialmente fue entender las físicas y líneas de código que rigen el movimiento y lógica de los elementos dentro de un juego.

Aquí hablamos de un par de ejemplos muy notables.

6.5.1 Velocidad, aceleración y ángulos.

Los enemigos tienen que, principalmente:

1. Perseguir al jugador independientemente de su posición
2. Moverse a una velocidad ajustable de forma sencilla.
3. Morir al impactar contra el jugador.

Los enemigos también tienen que **girar** de forma realista y **virar** a la dirección a la que estén mirando. Aunque el jugador no se mueve su nave si que tiene que virar cada vez que dispara.

Phaser 3 permite añadir velocidad X y velocidad Y a un objetivo, entonces, ¿Cómo le decimos a un objetivo que persiga una coordenada?

```
const tx = target.x;
const ty = target.y;
const x = enemy.x;
const y = enemy.y;
const angleToPlayer = Phaser.Math.Angle.Between(x, y, tx, ty);
if (enemy.rotation !== angleToPlayer) {
  var delta = angleToPlayer - enemy.rotation;
  if (delta > Math.PI) delta -= Math.PI * 2;
  if (delta < -Math.PI) delta += Math.PI * 2;
  if (delta > 0) {
    enemy.angle += enemy.turn_rate;
  } else {
    enemy.angle -= enemy.turn_rate;
  }
}
enemy.body.setVelocityX(enemy.speed * Math.cos(enemy.rotation));
enemy.body.setVelocityY(enemy.speed * Math.sin(enemy.rotation));
```

En Phaser 3 el método 'Update' ejecuta código a cada 'tic' del juego. Esto suele ser cada **16.7 milisegundos**. En este caso, los enemigos:

- Comprueban si existe un elemento 'objetivo'
- Marcan sus propias coordenadas así como las de su objetivo
- Rotan hasta encararse al objetivo
- Si tienen que girar, determinan en qué dirección tardan menos
- Limita el ángulo al que los enemigos pueden girar para que se comporten de manera natural.
- Haciendo un cálculo de (velocidad * coseno) y (velocidad * seno) determina a qué dirección se mueve.

*Alberto: "Si, se que podéis pensar al leer esto. El código no se está aprovechando en el proyecto final. Las naves no realizan giros dramáticos ni **necesitan** comprobar las coordenadas del jugador a cada momento si está quieto. Pero tenéis que comprender que estas características hubiesen venido muy bien si hubiésemos podido integrar otras funciones (como multijugador, misiles teledirigidos, enemigos*

*que actúan de diferentes maneras.) y otras como que los enemigos comprueben las coordenadas todo el rato **si** son necesarias (explicado más adelante). No considero que haya sido una pérdida de tiempo porque, para cuando ya sabía hacer lo básico, había aprendido lo suficiente como para hacer el resto rápidamente. Empezar fue lo difícil."*

6.5.2 Balas y colisiones.

Los proyectiles y colisiones fueron un desafío mecánico muy interesante. Al ser la primera vez que trabajamos en un videojuego surgen muchos comportamientos que uno no espera. Comencemos con la función 'DealDamage()' que simplemente daña a la entidad con la que impacte una bala, esto genera muchos problemas iniciales.

- La bala sale del jugador, impactando directamente en el jugador y matándolo.
- Si la bala no hace daño al jugador, lo empuja.
- La bala puede golpear a un enemigo no intencionado que estaba en medio del objetivo original.
- La bala puede desviar la trayectoria del enemigo (de ahí que los enemigos comprobando su velocidad constantemente no sea un desperdicio de recursos.)
- La bala se choca contra otras balas (En caso de que haya enemigos que disparan. Actualmente no los hay.)

Para ocuparnos de todo esto, hemos desarrollado un sistema que **almacena** información dentro de las balas.

```
dealDamage(bullet, object) {
  if (bullet.type !== object.texture.key) {
    if (object.texture.key === "Enemy") {
      if (object.wordText.text === bullet.currentWord) {
        bullet.destroy();
        object.health -= bullet.damage;
        object.setTint(0xff0000);
        this.time.addEvent({
          delay: 40,
          callback: () => {
            object.clearTint();
          },
          callbackScope: this,
        });
        const originalSpeed = object.speed;
        object.speed = -(originalSpeed * 2);
        this.time.addEvent({
          delay: 200,
          callback: () => {
            this.tweens.add({
              targets: object,
              speed: originalSpeed,
              ease: 'Linear',
              duration: 500,
            });
          },
          callbackScope: this,
        });
        this.randomizarPalabra(object);
        // object.healthText.text = "Health: " + object.health;
        if (object.health <= 0) {
          // object.healthText.destroy();
          object.wordText.destroy();
          object.destroy();
          this.enemiesKilled += 1;
          this.scorePlayer += 5;
          this.scoreText.setText("Score: " + this.scorePlayer); // Actualizar el texto del puntaje
          console.log(this.enemiesKilled);
        }
```

Enorme fragmento de código que determina que ocurre cuando una bala impacta contra un objetivo. Este es solo un fragmento de la serie de cosas que ocurren cuando se dispara una bala.

La lógica que engloba el funcionamiento de una bala funciona tal que así:

1. La bala, al ser disparada, almacena datos (Cuanto daño hago, a qué objetivo voy dirigido…)
2. Cada 16.7ms comprueba si está superpuesta con su objetivo. (¿Está mi objetivo al lado mío? No, ¿Está mi objetivo al lado mío? No, ¿Está mi objetivo…?)

3. Si lo está, reduce su vida, lo destruye si es necesario, lo noquea en dirección contraria y, si tiene una palabra, cambia su palabra a una palabra que no se encuentre actualmente en pantalla. Si el enemigo no muere también lo 'coloreo' de rojo durante una fracción de segundo.
4. Una vez ha terminado su trabajo o está demasiado lejos de la pantalla, la bala deja de existir.

7 Presupuesto

Siendo el salario promedio de un desarrollador de videojuegos Junior alrededor de 24000€ al año o 11.54€ la hora **y** siendo dos trabajadores en el proyecto, podemos estimar un presupuesto con las horas que hemos invertido.

Horas	Tarea	Coste total.
6	Prototipos de bajo nivel	138,48€
12	Investigación	276,96€
4	Desarrollo base de datos	92,32€
60	Programación de lógica principal.	1.384,80€
20	Pruebas de errores, debug y mejoras.	461,60€

5	Documentación	115,40€
Total:		2.469,60€

8 Conclusiones y vías futuras

Este proyecto ha sido muy ambicioso y a causa de eso muchos apartados se han quedado a medias tintas. Tanto que nos sentimos algo tentados de seguirlo post-graduación.

Antes he dejado una grán lista de 'características' que hemos sopesado y, si bien algunas han sido ideas, muchas de ellas han llegado a desarrollo solo para ser descartadas por falta de tiempo.

Nuestros objetivos principales si hubiésemos tenido tiempo hubiesen sido mejorar la experiencia principal. Las características que teníamos a mitad de producción o que introduciríamos después son las siguientes:

- Objetos y botiquines que el jugador puede destruir para curarse u obtener mejoras.
- Una mejora que permite al jugador escribir 'Misil' para arrojar misiles teledirigidos a los enemigos cercanos.
- Modo multijugador en línea donde cada jugador solo le puede disparar a los enemigos de cierto color.

Todas estas mecánicas (excepto el modo multijugador el cual está a medio investigar es muy extenso y complejo.) son completamente

implementables con nuestras habilidades actuales y sólo requerirían más tiempo.

Un problema que sin embargo no habíamos sopesado al principio y que está llevando al juego al límite es el tema de los **recursos**. El juego, si bien se ve fluido y no suele generar errores, ha dado errores en ocasiones contadas que no hemos podido replicar. Es completamente posible que añadir funciones extra abra puertas a muchas otras dificultades y errores.

La experiencia que nos llevamos, a pesar de estas dificultades, ha sido muy positiva. Pone en perspectiva lo difícil que es hacer un videojuego y la cantidad de pequeños detalles que hay que tener en consideración ha sido abrumador pero ver a familiares y amigos probar nuestro juego y 'engancharse' con ganas de jugar otra partida también ha sido muy refrescante.

Alberto: "Yo tengo claro que en el futuro quiero hacer videojuegos. El mercado, si bien está saturado, tiene muy buenas herramientas para aprender, lanzarse y que los demás prueben tus juegos. De momento me lo tomo como algo personal; un proyecto que hago por mi mismo."

9 Bibliografía/Webgrafía.

Atlassian. (2011). *Trello*. Recuperado de https://trello.com

Goodwin, M., & Colaboradores. (2024). *Documentación de PixiJS*. Recuperado de https://pixijs.download/dev/docs/index.html

Google. (2024). *Entorno de desarrollo IDX y documentación*. Recuperado de https://idx.dev/

Mozilla Developer Network. (2023). *Documentación de JavaScript*. Recuperado de https://developer.mozilla.org/es/docs/Web/JavaScript

MedimonGames. (2022). *pixel spaceships hd*. Recuperado de https://medimongames.itch.io/pixel-spaceships-hd

OpenAI. (2023). *ChatGPT* (GPT-4). Recuperado de https://www.openai.com/chatgpt

Richardson, R. (2024). *Documentación de Phaser 3*. Recuperado de https://newdocs.phaser.io/docs/3.80.0

You, E. (2023). *Documentación de Vite*. Recuperado de https://vitejs.dev/guide/

analogStudios. (2024) *Música*. Recuperado de

https://brackeysgames.itch.io/brackeys-platformer-bundle

RottingPizels. (2024) *Efectos de sonido varios*. Recuperado de

https://rottingpixels.itch.io/

Pimen. (2022) *Efectos visuales*. Recuperado de

https://pimen.itch.io/battle-vfx-hit-spark

Ansimuz. (2023) *Efectos visuales*. Recuperado de

https://ansimuz.itch.io/gothicvania-magic-pack-9

Google (2022) *Servicio y documentación de Firebase*. Recuperado de

https://firebase.google.com/docs?hl=es-419 y

https://firebase.google.com/?hl=es