Escuela
Politécnica
Superior

Sobreviviendo como desarrollador indie en Unity

Grado en Ingeniería Multimedia

Trabajo Fin de Grado

Autor:

Pedro Julián García Ruiz

Tutor/es:

Carlos José Villagra Arnedo

Julio 2020

Universitat d'Alacant
Universidad de Alicante

Justificación y objetivos

Desde que era pequeño siempre me ha fascinado el mundo de los videojuegos. Podría decirse que no he tenido una infancia fácil, pero nunca me importó porque cuando jugaba me sumergía en nuevos mundos, quizá mejores. Muchos de estos me han influido de gran manera, e incluso he aprendido lecciones muy valiosas. Con *Sobreviviendo como desarrollador indie en Unity* pretendo devolver un poco a los videojuegos de todo lo que me han dado.

Además, siento que quiero dedicarme a esto, a ganarme la vida desarrollando videojuegos. De momento veo lejos el momento en el que pueda dedicarme completamente a este maravilloso hobby, pero es un paso en el camino correcto.

Con el proyecto pretendo obtener una experiencia completa del proceso de desarrollar un videojuego, mejorar mi técnica de dibujo y solidificar mis habilidades de programación. También quiero adaptarme a una metodología de trabajo que sea eficaz y me funcione, quizás una de las cosas más difíciles de conseguir. Por último, quiero obtener más conocimientos sobre los videojuegos en sí y de cómo funciona el mercado, porque no solo basta con terminar un videojuego, hay que conocer como encaja en él y los pasos para una publicación relativamente exitosa.

Dedicado a Carlos, por aguantarme más de lo que yo podría.

A Fran, por enseñarme la importancia de las bases.

A Alexei, por los grandes momentos durante la universidad.

Y a Olga, por motivarme a terminar lo que empecé.

”I believe it’s called roleplaying”.

Bisqwit *- Programador y speedrunner finlandés.*

Índice de contenidos

1. Introducción

Sobreviviendo como desarrollador indie en Unity trata sobre una persona que quiere acceder a la industria del desarrollo de videojuegos. Tras varios intentos fallidos, se da cuenta de que acceder a un puesto de trabajo en una gran empresa, con un sueldo justo, resulta en un camino lleno de obstáculos, siendo así que pocos son los que consiguen el puesto de sus sueños.

Tras un análisis del mercado de los videojuegos actual y evaluando la opinión de profesionales de la industria, además de los datos de ventas y éxito que tienen los videojuegos del género indie, se plantea la posibilidad de desarrollar un título propio para poder acceder a este mercado.

Sin embargo, la realidad de los títulos de este género es que son muchos los que fracasan o no llegan a la remuneración necesaria para que estos estudios puedan continuar el desarrollo de otros proyectos, y los que consiguen salir adelante llevan historias de sacrificio a sus espaldas.

Pero no todo está perdido. Eduardo Fernández, desarrollador de videojuegos y graduado en ingeniería informática por la Universidad de Alicante, cuenta su carrera a través de diferentes empresas, concluyendo que para acceder a la industria del desarrollo de videojuegos, hay que desarrollar videojuegos, independientemente de si acaba siendo un fracaso, un proyecto terminado a la espalda otorga una valiosa experiencia que sin duda es valorada en esta industria.

Con la idea en mente de publicar un título, se estudian diferentes herramientas y metodologías, decidiéndose utilizar Unity como motor de videojuegos para implementar y desarrollar *Cross Element*, un videojuego multijugador local de peleas y plataformas al estilo *Super Smash Bros*.

Finalmente, se describe el proceso de conceptualización y desarrollo de *Cross Element*, comentando las diferentes características de los elementos del videojuego así como su funcionamiento e implementación. Además, se pretende analizar los componentes básicos que conforman el motor de Unity y la composición de estos que forman las entidades del videojuego.

A continuación, en el capítulo 2, *Marco teórico*, se sintetiza la información recabada acerca del mercado actual de los videojuegos, y del género indie. En el capítulo 3, *Objetivos*, se listan una serie de metas que se pretende alcanzar con este proyecto. Seguidamente, el capítulo 4, *Metodología*, describe una serie de técnicas a utilizar durante el desarrollo del videojuego, con el fin de optimizar el flujo de trabajo. El capítulo 5, *Desarrollo e implementación*, es el culmen de la memoria, que abarca desde el concepto de *Cross Element* hasta su implementación final. Por último, el capítulo 6, *Conclusiones*, reúne las ideas y sensaciones *a posteriori* tras la finalización del videojuego.

2. Marco teórico

2.1 Mercado actual

En los últimos años la industria del videojuego ha experimentado un pico de crecimiento constante, atrayendo a miles y miles de nuevos adeptos a este maravilloso *hobby*, traduciéndose en miles de millones en ingresos, superando a la industria tradicional audiovisual, como el cine y la música. No cabe duda de que es un sector muy rentable.

De acuerdo al análisis del mercado de los videojuegos realizado por Newzoo, y compartido con GameIndustry para un artículo de ventas de videojuegos escrito por James Batchelor, en 2018, la industria del videojuego generó alrededor de 135 billones de dolares a finales de este año, un incremento de casi el 11% en comparación con el año 2017.

Curiosamente, el mercado móvil, *smartphones* y *tablets,* está generando gran parte de estos ingresos, abarcando el nada despreciable 47% de los beneficios totales en 2018, ninguna sorpresa para algunos expertos. Los datos recogidos estos últimos años indican que esta tendencia seguirá creciendo más y más. No es de extrañar que titanes de la industria como Activision Blizzard, tradicionalmente dedicada al sector de PC, estén tratando de acercarse a este mercado, por ejemplo, con el anuncio del *port*[1] de Diablo 3 a dispositivos móviles, bajo el nombre de Diablo Inmortal, saga que vio su nacimiento en la plataforma de PC, en la BlizzCon de 2018 dirigido exclusivamente a dispositivos Android e iOS, aunque sus *fans* no vieron con buenos ojos dicho anuncio, cabe decir que no deja de ser una decisión estratégica.

[1] Es el proceso de adaptar software con el objetivo de conseguir su ejecución en un entorno de computación distinto para el que fue diseñado originalmente.

Actualmente, los mercados de consola y PC siguen manteniéndose, aunque con el tiempo se verá que plataforma predomina predomina el mercado.

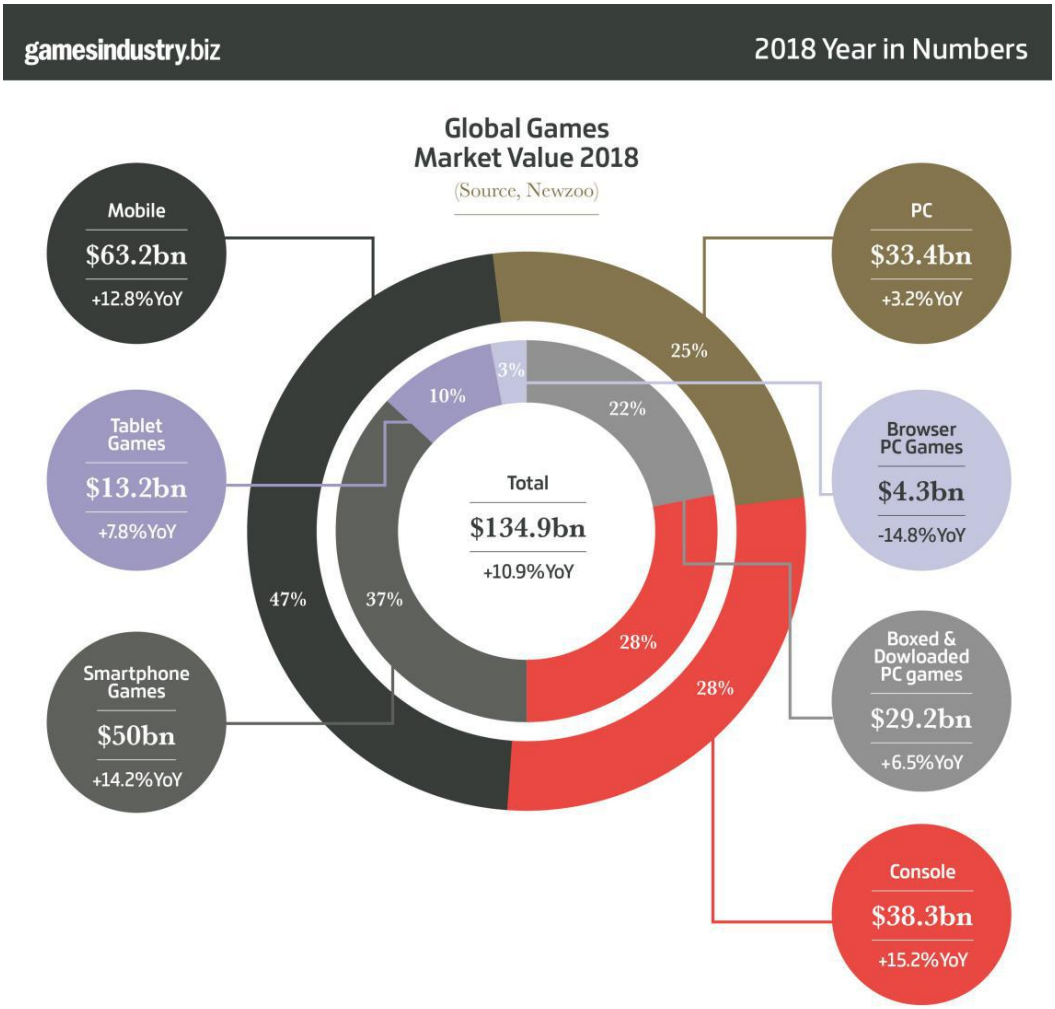

Figura 2.1. Ventas de videojuegos en 2018[2].

[2] Fuente: https://www.gamesindustry.biz/articles/2018-12-17-gamesindustry-biz-presents-the-year-in-numbers-2018

2.2 Puestos de trabajo en estudios de videojuegos

No obstante, aun con el auge de los beneficios de la industria del videojuego, conseguir un puesto de trabajo en estudios de desarrollo AAA[3] es muy complicado. Las cualificaciones para trabajar en uno de estos estudios a menudo suele ser haber trabajado en otro estudio de las mismas dimensiones. Básicamente hay más personas cualificadas que puestos de trabajo disponibles en estos estudios, de hecho, pueden estar durante años con una pequeña plantilla de *juniors*, y cuando ofertan nuevas plazas, generalmente se ocupan todas incluso antes de haber sido ofertadas públicamente, y aun así exigir cualificaciones desorbitadas para estos puestos.

Citando a Mike Prinke (2016), diseñador de videojuegos con más de 6 años de experiencia y diversos méritos en este campo;

"If you want to make video games I recommend it as a hobby before I recommend it as a career. If the upward climb is going to be "slowly but surely," so to speak, you might as well do it in such a way that you aren't risking your entire livelihood with each step and where you can insure satisfaction with your own work".

[3] Se le llama estudio AAA de manera informal a distribuidores de videojuegos o editores con altos presupuestos para invertir tanto en *marketing* como en el propio desarrollo del videojuego.

2.3 El género indie

Abrirse paso en la industria del videojuego desarrollando títulos propios es una opción. Sin embargo, parece que intentar competir contra mega éxitos de ventas como las sagas de Assasin Creed y Call of duty, es una batalla perdida incluso antes de haberla empezado.

En la pasada década, el número de videojuegos en el mercado se ha visto incrementado exponencialmente. Steam ha dejado de ser la empresa humilde que empezó con sus títulos de Valve, como Counter-Strike o Half-Life, con recursos limitados, y se ha convertido en un gigante económico, además de ser la referencia número uno en cuanto a videojuegos en la plataforma de PC se refiere. Además, otras tiendas virtuales como la eShop de Nintendo tampoco andan muy por detrás de esta.

Sin embargo, este crecimiento exponencial se debe también en gran medida al crecimiento de videojuegos indies en el mercado (sí, hay un montón), lo que ha supuesto una presión para que la industria AAA no dé por supuesto su posición consolidada en el mercado, el cual está más saturado que nunca. Es un hecho que la cantidad de juegos del género indie es abrumante, tan solo en la plataforma virtual de Steam existen alrededor de 26000 títulos de este género (a día de 09/05/2020).

Figura 2.2. *Consulta de videojuegos del género indie en la plataforma Steam*[4].

Cabe pensar que este dato supone una buena razón por la cual pasar directamente de largo la idea de entrar en este mundo. Existe demasiada competencia, aunque también es sinónimo de la buena acogida que está recibiendo este género entre los aficionados.

2.4 La verdad incierta

Una entrevista realizada a diversos desarrolladores de videojuegos independientes, realizada por Steven T. Wright en 2018 y publicada en la web de Polygon, es un perfecto reflejo de la realidad de esta industria, y es que independientemente de las experiencias obtenidas por cada estudio indie, todos coinciden en una cosa, y es que ninguno sabe realmente qué esta pasando.

[4] Fuente: https://steamspy.com/genre/Indie

Los siguientes fragmentos corresponden a una conversación de Dave Crooks y recogido por el propio Steven en el mismo artículo, donde se habla de la incertidumbre del éxito que puede llegar a tener un videojuego.

"He visto muchos juegos que pensaba que serian auténticos éxitos masivos y que finalmente acabaron siendo un fracaso", dice Dave Crooks, de Dodge Roll. *"Cada que vez que ocurre, me preocupa más y más que la próxima cosa que decida hacer seguirá ese camino. Yo hubiera apostado por esa gente, hubiera comprado acciones en esas compañías de tan solo dos personas."*

"La idea que prevalece es, que nadie sabe qué va a ser un éxito, qué es lo que la gente quiere. La respuesta aburrida es que nadie sabe para nada que está ocurriendo realmente. Es por esto que alguna gente juega al fantasy football con los juegos que salen en Steam, por que es divertido hacer un juego sobre adivinar cuál lo hará bien. Todos hacen el juego que más les entusiasma, y rezan por que sea un éxito."

2.5 Entrevistando a un desarrollador de videojuegos

En una entrevista realizada a Eduardo Fernández García, ex estudiante de Ingeniería Informática en la Universidad de Alicante, y desarrollador de videojuegos con más de ocho años de experiencia, cuenta su trayectoria desde que empezó a trabajar para una empresa de desarrollo de videojuegos hasta crear el suyo propio, *Elliot*.

- ¿Como empezaste tu carrera como desarrollador de videojuegos?

- Terminé la carrera de ingeniería informática en la Universidad de Alicante, y después hice un máster en videojuegos en la politécnica de Barcelona. Pensé que sería fácil encontrar un

trabajo como desarrollador, y sin cortarme nada, empecé a aplicar para grandes empresas cuyos videojuegos había jugado y me gustaban. La realidad es que en España habían pocas ofertas y al final acabaron respondiéndome de una empresa en Mallorca llamada Tragnarion Studios, y a la isla que me fui, en la que estuve trabajando durante más de año y medio..

- Me has comentado que luego empezaste a trabajar para otra empresa en Valencia, llamada Gameloft. ¿Cómo sucedió el cambio?

- Bueno, principalmente empecé a buscar otra empresa donde trabajar, porque en la que estaba, decidió tomar otro rumbo, dejar de hacer videojuegos, y yo tenía muy claro que quería seguir en esta carrera. Durante ese tiempo, la empresa internacional Gameloft, decidió abrir un nuevo estudio en Valencia, en la que apliqué y me escogieron. Tenían un primer test online que envían a los candidatos, y luego, si les gustas pasas a una entrevista personal con ellos.

- Tengo entendido que estuviste trabajando para Ubisoft Bulgaria. ¿Qué pasó en Gameloft y cómo fue el proceso?

- Bueno, esta industria es complicada, porque hay estudios que van bien y otros que no. Aquí en España, la verdad, es que montar un estudio de videojuegos y que salga bien, además de rentable, es complicado, porque el mercado es muy inestable. Este estudio de Gameloft se montó con una idea, que duró aproximadamente 2 años, y debido a que la empresa sufrió una crisis, empezó a cerrar estudios en todo el mundo. Nuestro proyecto tampoco salió todo lo bien que se esperaba, y fue uno de los estudios que se cerró, entre varios.

- Cada vez me veía con mas experiencia, pero claro, es un mercado muy competitivo y aquí en España hay pocas opciones, aunque cada vez hay más, no es un sector muy desarrollado. En Bulgaria me contestaron para empezar el proceso de selección con ellos. Este proceso era mas

largo que otros que había realizado, en la que te pasan varias pruebas. Al final, me hicieron una entrevista por Skype y les guste.

- Si que es verdad que cada vez hay mas puestos de trabajo en desarrollo de videojuegos, pero cada vez hay más programadores en general, y desarrolladores de videojuegos en específico. También ocurre que los videojuegos se han convertido en fenómeno *mainstream*, y no como era hace 12 o 15 años, que no estaba del todo bien visto por la sociedad jugar a videojuegos, y ahora cada vez hay más y más jugadores. ¿Qué opinas tú?

- Sí, es una industria que ha crecido muchísimo, y la oferta no crece a la par. Hay muchas academias, universidades y lugares de formación que ofrecen estudios relacionados con los videojuegos, y que lo abarcan de diferentes perspectivas. Algunos son estudios generalizados y otros más específicos, y hay mucha gente formada, aunque no sé si de la mejor manera para poder pelear con la oferta que hay en el mercado. Es una industria que mueve más dinero que el cine y la música, y con el auge de los deportes electrónicos están a la orden del día.

- Cuéntame cómo fue tu trayectoria en Ubisoft Bulgaria.

- En Bulgaria acabamos nuestro proyecto, que colaborábamos con otros estudios de Ubisoft para un DLC de Assasin Creed Origins. No tenía pensado pasarme la vida en Bulgaria, y como no tenia muy claro por donde ir, y mi pareja encontró una buena oportunidad en Mallorca, decidí volverme. Se me ocurrió hacer algo por mi mismo, ya que tenía algo de ahorros.

- Entonces decides hacer un videojuego por ti mismo, ¿cómo nace la idea para desarrollar *Elliot*? y, ¿qué expectativas tienes con su lanzamiento?

- Desde pequeño tenía claro que quería desarrollar videojuegos, y empecé a pensar qué podía hacer. Yo crecí en el 88, y mi primera consola fue la Megadrive. Muchos de los juegos eran

de plataformas, y he jugado mucho a este tipo de videojuegos y los conozco bien, y podía asumir el género. Estuve 7 meses trabajando a tiempo completo en el proyecto para sacar un prototipo, además de formar un equipo que me ayudase, ya que yo soy programador y diseñador, pero no soy artista ni sonidista, se me escapa, por eso tenía que buscar ayuda para suplir estas carencias, gente que confiase en el proyecto, porque yo no tengo capacidad económica para pagar a nadie, por tanto ¿como convences a gente para que trabaje sin cobrar?

- He conocido otros equipos que desarrollan videojuegos, y que estaban en una situación parecida. Al final lo que hicimos era firmar un contrato de colaboración sobre futuros beneficios. Así fue entrando gente el equipo, y saliendo también, hasta formar un equipo de 4 personas, incluyéndome a mi, que aceptaron trabajar bajo estas condiciones y le pusieron muchas ganas. Ahora llevamos 2 años y medio seguidos con el desarrollo del videojuego, por que además cada miembro trabaja en otras empresas y solo pueden dedicarle horas de tiempo libre, al igual que yo, que empecé a trabajar para otra empresa, y solo podía desarrollar Elliot después de mi jornada laboral.

- Al final esto es complicado, yo porque llevo años trabajando en el sector e intento plasmar un proyecto comercial de la mejor manera que sé. Quiero un cierto tipo de proyecto con unos mínimos. Tenemos la suerte de que ningún miembro del equipo necesita que el juego tenga ciertas ventas para comer. Pero sí que queremos que lo juegue cuanto más gente posible, que tenga algo de repercusión, al final quieres que tu trabajo se valore, como un músico o una película, te gustaría que la viese cuanta más gente posible y que tenga relevancia. Aunque es complicado, porque tienes que mantener la ilusión durante mucho tiempo por un proyecto del que no estás ganando nada, además de compaginar el trabajo y el desarrollo del videojuego con tu vida, y sobre todo tener mucha fe. Al final tienes menos tiempo para hacer cosas, menos ocio, y sólo tienes ilusión por que vaya bien. Si quieres sacar un producto competitivo con esta estructura, trabajo a tiempo completo, y luego tiempo libre dedicándoselo a un proyecto así, tienes que compaginarlo bien y sacrificar muchas cosas.

- Respecto a la parte comercial, me interesa conocer más tu opinión, porque la perspectiva que yo tengo, es que mucha gente cuando se quiere iniciar en el mundo del desarrollo indie, quieren hacer el juego de sus vidas, por ejemplo, pues me encantan los RPG, pues un RPG pequeño de 12 horas ya es lo suficientemente pequeño para implementarlo, y claro, luego se pegan contra la pared. El juego que estoy desarrollando sí que le estoy dando un enfoque más comercial. Utilizar mecánicas que sean agradables para todo el mundo.

- Me parece bien lo que haces. Una cosa es el tamaño del proyecto, no hay que volverse loco. No tiene sentido planear un juego tan grande porque se te va a escapar, aunque hay gente que le ha funcionado, como el creador de Stardew Valley, que estuvo desarrollando el título durante más de 4 años. Ha sido capaz y ha tenido éxito, pero para mí es extremo. Tienes que pensar en un tamaño de proyecto que puedas afrontar, y eso sólo lo puedes aprender mediante la experiencia. Otra cosa es pensar en el jugador, como se va a consumir tu videojuego. La verdad es que eso no lo hemos pensado mucho, nos hemos enfocado más en qué tipo de proyecto podíamos afrontar, y cómo encaja eso en el mercado. Queríamos crear algo diferente, para que no sea igual que otros proyectos que existen y que se sienta también diferente. Aunque ya nos han dicho varias veces que las estadísticas de los videojuegos plataformeros en Steam no venden nada. Pero nos ha dado igual, porque si te pones a ver estadísticas, no sacas el juego. Aunque si tienes que sacar el juego para pagarte la comida, el enfoque cambia.

- Lo mas importante que miran las empresas es que tengas un proyecto desarrollado. Es muchísimo mejor que haber estudiado 2 o 3 años sobre esto. Te lo digo en confianza, pero valoran mucho que hagas proyectos por tu cuenta y los termines.

- Respecto a las herramientas de desarrollo, ¿utilizáis algún motor de videojuegos de terceros?

- Nosotros utilizamos Unity. Es una decisión que tuvimos que tomar, sí que planteamos hacer un motor por nuestra cuenta por aprender, pero ya te digo que es una mala decisión a nivel comercial, y a nivel de todo. Si me hubiera tocado hacer uno, me hubiera vuelto loco. Aunque sí que me parece una buena manera de aprender como funcionan todas las capas de un videojuego, porque vas a entender como funciona un motor desde cero. Pero si quieres hacer un videojuego, quieres hacer un videojuego.

- *Elliot* saldrá a la venta el 20 de Julio de 2020, por tanto aún es pronto para saber si será un éxito, tan solo espero que tengan suficiente repercusión para que este pequeño estudio pueda continuar su camino en esta industria.

2.6 Preparado para agotar mi tiempo y salud

Depende de donde vivas, trabajar en una empresa de videojuegos puede llegar a ser complicado, hay pocas ofertas de puestos de trabajo en comparación con la demanda. Sin embargo, tras conocer la opinión de varios desarrolladores de videojuegos, se podría concluir que no todo está perdido, y que hacer un videojuego por uno mismo es un posibilidad muy válida, tanto para comenzar una carrera como desarrollador indie, como para construir un porfolio que hable por si mismo de los proyectos en los que uno ha participado.

Desarrollar un videojuego mientras se mantiene una jornada laboral completa como hobby, recurriendo al tiempo libre, no es un camino fácil y exige sacrificios a nivel personal. También es un movimiento arriesgado, ya que puede llegar a ser un completo fracaso, pero no hay mejor manera de aprender, que perder. Preparado para agotar mi tiempo y mi salud, me dispongo a desarrollar mi propio videojuego.

Antes de comenzar con el desarrollo e implementación de *Cross Element*, en el siguiente capítulo se describe una lista de objetivos que se pretende conseguir con *Sobreviviendo como desarrollador indie en Unity*.

3. Objetivos

3.1 Objetivo principal

Con los capítulos anteriores se ha estudiado las situaciones por las que una persona, que quiere acceder a la industria del videojuego, puede pasar, evaluando el mercado, posibilidades de acceder a un estudio AAA y el estado del género indie como tal además de estudiar la posibilidad de acceder a este último mercado mediante la publicación de un título propio.

A continuación, el objetivo es describir el proceso involucrado en el desarrollo de *Cross Element*, utilizando un motor de videojuegos como lo es Unity, que permiten acelerar el proceso, teniendo en cuenta que el tiempo que se va a invertir es literalmente dinero que se está invirtiendo en el proceso, partiendo de un presupuesto inicial bajo o cero.

3.2 Desglose de objetivos

A continuación, una lista de los objetivos específicos que se pretende conseguir con *Sobreviviendo como desarrollador indie en Unity*:

1. Analizar oportunidades de acceso en el sector del desarrollo de videojuegos.
2. Evaluar el mercado de los videojuegos, especialmente el género indie.
3. Desarrollar nuevas habilidades con herramientas de trabajo, en específico Unity.
4. Estudiar y aplicar metodologías de trabajo.
5. Desarrollar un videojuego, *Cross Element*.

6. Redactar un documento de diseño de videojuego (GDD).
7. Aprender de los errores comunes durante el desarrollo de un videojuego.
8. Aumentar la experiencia en el desarrollo de proyectos, en específico de videojuegos.

4. Metodología

En el desarrollo de *Cross Element,* se ha utilizado el sistema Kanban, en el cuál se basa la aplicación web Trello, también utilizada en el proyecto y complementada con la técnica del Pomodoro, quizás no las mas apta para todo el mundo pero sí la que ha resultado ser (para mí), la más efectiva y merece al menos una mención.

4.1 El sistema Kanban

Kanban es una palabra de origen japonés que significa tarjeta y puede definirse como un sistema de estas que permite la organización del trabajo enfatizando llegar a tiempo o JIT (*just in time*). En la mayoría de implementaciones del sistema Kanban en desarrollo de software, estas tarjetas son virtuales y representan los elementos de trabajo, los cuales se distribuyen en un gran tablón dividido en columnas que pueden variar dependiendo del nivel de complejidad o de las fases del proceso (Willy-Peter Schaub, 2020). Las columnas más utilizadas suelen ser las siguientes:

- Lista de tareas o *TO DO*[5].
- En desarrollo.
- Pruebas.
- Despliegue.
- Terminado.

[5] TO DO significa, de manera coloquial, tareas que están pendientes de realizar y que se pueden afrontar inmediatamente.

4.2 Trello

Trello se basa en el sistema Kanban para el registro de actividades, pudiendo añadir tarjetas virtuales que permite organizar tareas, agregar ideas, o enlaces. Destacando principalmente por su fácil uso, Trello puede utilizarse en cualquier tipo de tarea que requiera organizar información.

Imagínese una pizarra en blanco, llena de notas adhesivas, cada una representando una tarea. Ahora imagíneselas pudiendo además, añadir imágenes, archivos adjuntos, y comentar cada nota y ahora imagina que puedes llevarte esa pizarra a cualquier lugar (Trello, 2017).

4.3 La técnica del Pomodoro

El flujo de trabajo utilizando la técnica de Pomodoro es muy simple. Según su creador en un artículo a modo de invitado para Trello (Francesco Cirillo, 2012), la técnica del Pomodoro se divide en cinco pasos:

1. Elige una tarea a completar.
2. Establece una alarma de 25 minutos.
3. Trabaja en la tarea hasta que la alarma del Pomodoro suene.
4. Descansa 5 minutos, ya has completado un Pomodoro.
5. Repite hasta completar 4 Pomodoros, entonces puedes descansar un poco mas.

Un Pomodoro es una unidad de 25 minutos, seguido de un descanso de 3-5 minutos. Este proceso es continuo, pero teniendo descansos más largos, 15-30 minutos, cada 4 Pomodoros.

4.3.1 Aplicando la técnica del Pomodoro en Trello

Para aplicar la técnica del Pomodoro en Trello, se utiliza un sistema de etiquetado por colores que permite distinguir rápidamente las tareas más densas. Se codifica un sistema de colores que se asigna a cada tarjeta del tablón de Trello, a distinguir:

· Color verde: equivale a un *Pomodoro* (25 minutos).

· Color amarillo: equivale a dos *Pomodoros* (50 minutos).

· Color rojo: equivale a tres *Pomodoros* (75 minutos).

Si una tarea requiere más de 3 Pomodoros, entonces se debe especificar la tarea en subtareas más pequeñas. Para medir el tiempo existen diversas herramientas, incluso puedes usar un reloj de cocina con forma de tomate, aunque se sugiere encarecidamente utilizar una aplicación web, como Toggl (entre muchas más).

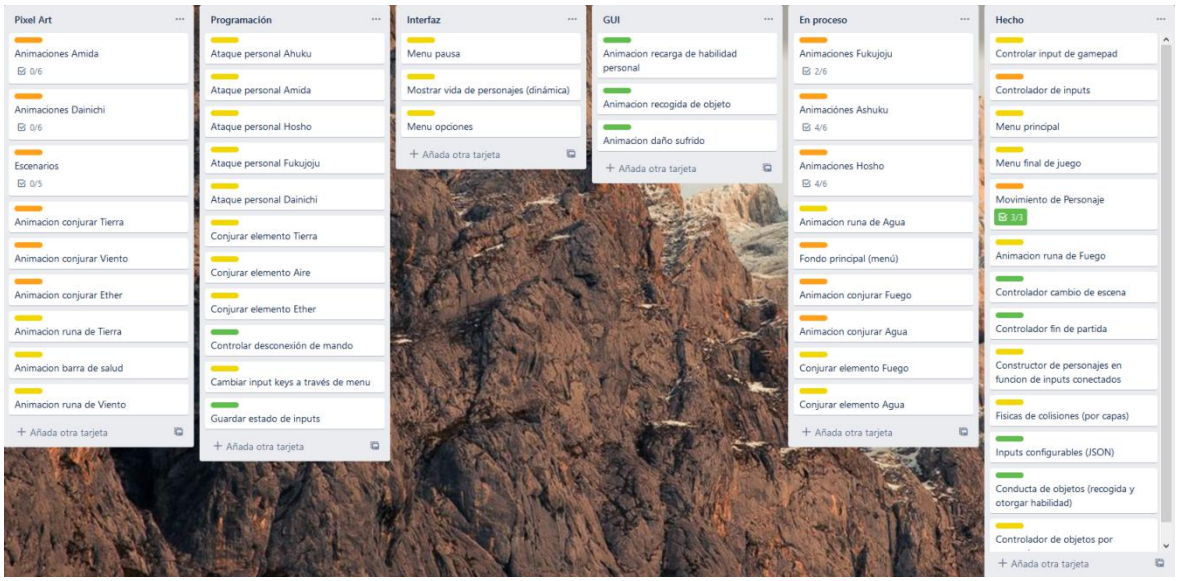

Figura 4.1. *Tablero en Trello, con el estado de desarrollo de Cross Element.*

5. Cuerpo del trabajo

5.1 Análisis, explicación y elección de motor

A la hora desarrollar un videojuego, es vital decidir qué motor de videojuegos va a utilizarse, pero, ¿qué son? Un motor de videojuegos es una arquitectura que los desarrolladores utilizan para implementar sus videojuegos. Típicamente, un motor de videojuegos provee sistemas básicos para el desarrollo como por ejemplo:

- Sistema de físicas.
- Sistema de entrada (teclado y/o mando).
- Motor gráfico (renderizado).
- Modelo de *scripting*[6].
- Sistema de detección de colisiones.
- Inteligencia artificial (algunos).

Si bien puedes crear uno por ti mismo, es importante pensar en el tiempo del que dispones para realizar tal inversión, y si realmente merece la pena invertir ese tiempo para llevar a cabo tu proyecto.

Para desarrollar *Cross Element* se ha utilizado un motor de videojuegos comercial, Unity. A continuación se detallan algunos de los motores más populares que existen en el mercado, aunque hay muchos más, es cuestión de buscar el que me mejor se adapta a las necesidades del proyecto, y que las condiciones que exige sean razonables.

[6] Script en informática es un término informal para definir un programa con una funcionalidad relativamente simple.

5.1.1 Unreal Engine

Unreal Engine es un motor de videojuegos muy potente. Aunque se desarrolló principalmente para los *shooters* en primera persona, se ha utilizado con éxito en una variedad de otros géneros, incluyendo videojuegos del género de sigilo, lucha, MMORPG y otros RPG. Con su código escrito en C++, Unreal Engine presenta un alto grado de portabilidad y es una herramienta utilizada actualmente por muchos desarrolladores de juegos. Algunos videojuegos creados con Unreal Engine; la saga Gears of War, la saga Mass Effect o la saga de Bioshock (GameDesigning, 2020).

Figura 5.1. *Logo de Unreal Engine 4.*

Se dice que es uno de los motores más fáciles de utilizar en manos de un profesional, aunque resulta confuso para principiantes. Unreal Engine 4 es gratuito, sin embargo a partir de los 3000 dolares de ganancias, debes pagar un 5% a modo de canon cada trimestre.

5.1.2 Unity

Unity es un motor de videojuegos multiplataforma, el cual utilizan muchos desarrolladores *indies* debido a su excelente funcionalidad y para poder crear, prácticamente, cualquier tipo de videojuego. Algunos juegos exitosos desarrollados en este motor son: Pillars of Eternity y Hearhstone: Heroes of Warcraft, entre otros (GameDesigning, 2020).

Figura 5.2. Logo de Unity.

Unity ha resultado ser una herramienta eficaz a la hora de implementar gráficos en 2D, además de tener una comunidad enorme en la que apoyarse cuando un proyecto se atasca, estando prácticamente todas las respuestas en Internet.

Actualmente Unity es gratuito si ganas menos de 100 mil dolares anuales, a partir de ahí cuesta 25 dolares mensuales hasta los 200 mil dolares anules, y finalmente 125 dolares mensuales al pasar esa barrera.

5.1.3 Godot

Godot es un motor perfecto para hacer tanto videojuegos en 2D como en 3D. El motor provee de muchas herramientas para que puedas centrarte en el desarrollo sin tener que reinventar la rueda.

Uno de los mejores beneficios al usar Godot, es que es completamente *open-source* y por tanto gratuito independientemente de las ganancias que se obtengan con el videojuego (GameDesigning, 2020).

Figura 5.3. Logo de Godot.

Una característica positiva de Godot es la propia comunidad, que está constantemente arreglando *bugs* del motor y desarrollando nuevas funcionalidades, lo que es sin duda una buena señal. En este sentido, la comunidad es lo mejor que tiene este motor.

5.1.4 Conclusión

Para desarrollar *Cross Element* se va a utilizar el motor de Unity. Tiene perspectivas implementadas para el desarrollo de videojuegos 2D, asi como varios complementos 2D implementados de serie. Además, su sistema de componentes es muy intuitivo.

5.2 Documento de Diseño de Videojuego

Los siguientes apartados describen la conceptualización inicial para *Cross Element;* la historia, sus personajes, tema del videojuego y mecánicas jugables.

5.2.1 Historia

Se cuenta que hace milenios en las sagradas tierras de Vajra, el dios de la tierra Gōzanze, el dios del fuego Gundari, el dios del agua Daiitoku y la diosa del viento Kongōyasha convivían en armonía, hasta que un día un barco llegó a la costa...

Los dioses, confundidos ante esta especie desconocida decidieron observarlos, y tras unos días los exploradores llegaron a las prohibidas aguas del manantial de Éter. Ignorantes y exhaustos por la travesía, bebieron de él sin dudar provocando la ira de los dioses por beber del líquido incluso prohibido para ellos.

Los dioses disputaron entre sí sobre el destino de los insolentes humanos, y tal fue la contienda que llegaron a enfrentarse entre si. Descomunales fuerzas elementales se desataron en el continente, provocando un cataclismo que cambió la geografía de Vajra para siempre. Desde entonces se dice que los dioses regresaron a su plano elemental, formando las regiones de Vajra.

A día de hoy, las tierras de Vajra se dividen en cinco grandes regiones; las áridas tierras del este, las ardientes arenas del sur, las místicas aguas del oeste, los riscos ventosos del norte y la ciudad flotante Fudō.

Cada 20 años y con motivo de este suceso, la ciudad flotante de Fudō celebra el gran torneo de los elementos donde cada región elige a su campeón para enfrentarse al resto, donde el victorioso conseguirá el beneplácito elemental para los suyos durante los próximos 20 años.

5.2.2 Personajes

Cross Element está formado por cuatro personajes jugables, y uno extra opcional que podría formará parte de una futura expansión, cada uno de ellos representando un elemento.

- *Chirin* (**tierra**): predominan los colores cálidos, como marrones, amarillos y rojos. Es un joven de piel clara, que utiliza una espada y lleva un vestido de samurai.

- *Suirin* (**agua**): predominan los colores fríos, principalmente azules y celestes. Es un hombre de avanzada edad que utiliza un sombrero y viste una toga.

- *Karin* (**fuego**): mezcla de colores cálidos y fríos, como rojos y verdes. Es un duende de edad indefinida de color oscuro que lleva puesta una capucha y de sus manos emanan llamas.

- *Fuurin* (**viento**): mezcla de colores cálidos y fríos, como azules, violetas y rosas. Es una mujer de mediana edad de piel clara que lleva puesta una capa y vuela.

5.2.3 Tema

Cross Element es un juego de acción multijugador local inspirado en los elementos del Budismo tántrico, **tierra, agua, fuego, viento y éter**. La figura 5.4 muestra un *gorintou*, que significa literalmente cinco anillos, una estructura formada por varios pisos de origen japonés, cada uno representando uno de los elementos.

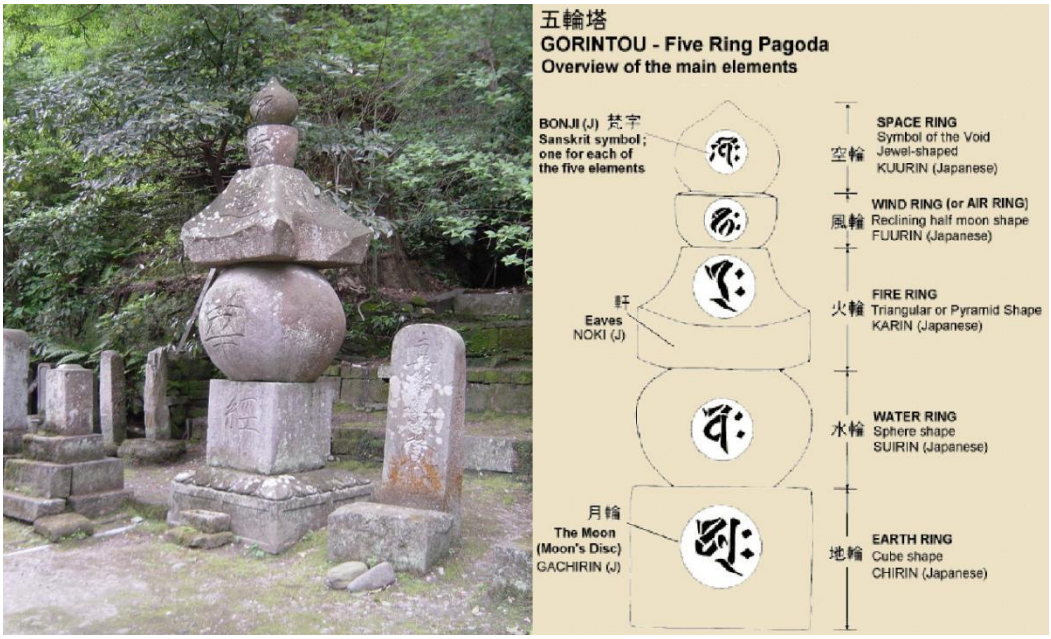

Figura 5.4. Gorintou en un jardín[7].

Cada personaje, objeto y habilidad tiene que ser visualmente identificable con el elemento que representa, además de que las mecánicas propias también deben estar acordes tanto con el estilo como con el ritmo rápido del juego. Los escenarios de juego deben propiciar situaciones estratégicas y/o divertidas con elementos propios de la región que representa, por ejemplo, arenas movedizas en el nivel del desierto o ráfagas de viento en los riscos ventosos.

5.2.4 Objetivos

El objetivo del juego es esquivar las diferentes habilidades que se usarán contra él al mismo tiempo que hace uso de las mismas para derrotar a los adversarios. La ronda finaliza cuando quede un único jugador con vida recibiendo un trofeo. El primer jugador en conseguir el número de trofeos configurado gana la partida.

[7] Fuente: http://www.aisf.or.jp/~jaanus/deta/g/gorintou.htm

5.2.5 Mecánicas de juego

Las mecánicas de juego son similares a las de un juego de lucha y plataformas, tomando como referencia *Super Smash Bros* (Nintendo, 1999) y *Contra* (Konami, 1987). La figura 5.5 muestra un boceto de una supuesta partida de *Cross Element*.

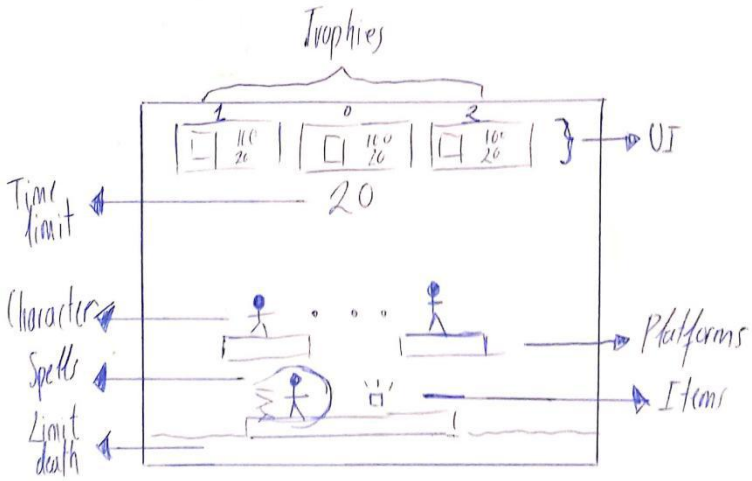

Figura 5.5. *Boceto de una partida de Cross Element.*

5.2.5.1 Movimiento y cámara

El juego utiliza una cámara con vista de perfil, donde los personajes pueden moverse en dos dimensiones a una rápida velocidad para favorecer el ritmo de juego. Además estos pueden saltar y son afectados por la gravedad, por tanto pueden caer.

5.2.5.2 Bola de energía

Cada personaje tiene la habilidad para materializar su energía elemental en forma de proyectil y lanzarlo, el cuál avanza a una velocidad constante en una dirección, siendo para todos igual pero cambiando el arte utilizado para representar cada uno de los elementos.

5.2.5.3 Objetos y poderes

Cada uno de los personajes tiene la habilidad para canalizar el poder de los elementos. Para ello, el jugador debe recoger unas rocas con los símbolos de los elementos de la alquimia grabados en estas, que aparecen aleatoriamente en el escenario, cada una representando un elemento y que otorga una habilidad del mismo tipo.

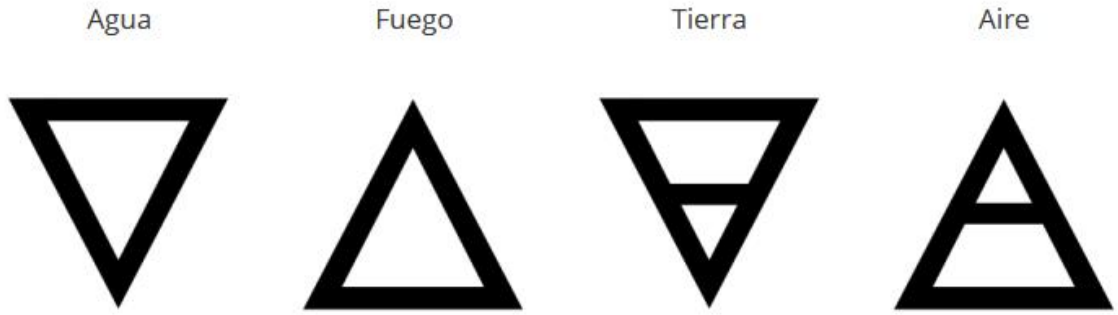

Figura 5.6. *Símbolos de los elementos de la alquimia*[8].

- **Habilidad de agua:** el personaje se convierte en una masa de agua durante un periodo de tiempo, volviéndose inmune a las habilidades de los adversarios y destruyendo las bolas de energía. Además, el personaje avanza rápidamente dañando a los enemigos que se encuentre a su paso.

[8] Fuente: https://www.simboloteca.com/simbolos-alquimicos/

- **Habilidad de fuego:** dispara una bola de fuego que acelera rápidamente hasta golpear un personaje enemigo o superficie, donde explota. Los adversarios golpeados son dañados y empujados en dirección opuesta.

- **Habilidad de tierra:** el personaje hunde el terreno, golpeando a los adversarios cercanos e inutilizando cada uno de ellos por un corto periodo de tiempo.

- **Habilidad de viento:** el personaje sacude los vientos cercanos en un área circular, dañando y empujando en direcciones opuestas a los adversarios alcanzados.

5.2.5.4 Habilidad definitiva

Cada personaje tiene la capacidad de utilizar una habilidad definitiva única, que puede utilizar cada vez que recoge tres runas de su mismo elemento. Los efectos de estas habilidades son los mismos que los proporcionados por las runas, pero a mayor escala en todos los ámbitos, como daño infligido, área de efecto o fuerza de empuje.

5.2.6 Arte

Los gráficos a utilizar en *Cross Element* se realizarán mediante la técnica de *pixel art*[9], inspirados en los elementos. Para ello se utilizará el programa *Aseprite* (David Capello, 2014). La tabla 5.7 muestra las características de los diferentes elementos del juego.

[9] El *pixel art* es una forma de arte digital, creada a través de una computadora donde las imágenes son editadas al nivel del píxel.

Elemento	Tamaño (*píxeles*)
Fondo de escenario	320 x 180
Plataformas	Según necesidad
Personaje	32 x 32
Habilidad de tierra	32 x 32
Habilidad de agua	32 x 32
Habilidad de fuego	32 x 32
Habilidad de viento	144 x 144
Bola de energía	16 x 16

Figura 5.7. *Tabla de tamaños para los gráficos de Cross Element.*

5.3 Desarrollo e implementación

En los siguientes apartados se describen los componentes básicos con los que trabaja Unity, comenzando con los *Assets*, que es la unidad básica multimedia con la que importar ficheros de sonido, imágenes, etc...

5.3.1 *Assets* y componentes básicos de Unity

Un *Asset* es, generalmente, un fichero multimedia que se importa en el editor de Unity, ya sea un audio, un *sprite*[10], una textura o una malla 3D entre muchas otros, aunque existen algunas

[10] Se refiere comúnmente a un mapa de bits bidimensional que se integra es una escena mayor, frecuentemente en un videojuego 2D.

excepciones, por ejemplo, también se pueden crear ciertos tipos de *Assets* especiales, como un controlador de animaciones o un *Prefab* (véase apartado 5.3.1.5 *Prefab*).

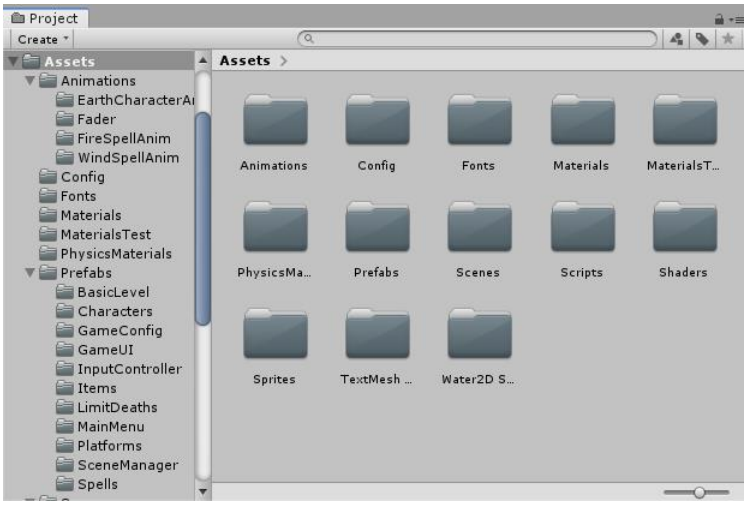

Figura 5.8. *Ejemplos de Assets organizados por carpetas en Cross Element.*

Unity se basa en un sistema de componentes. Existen una gran cantidad de estos incorporados de forma nativa en el propio motor, los cuales se van a agregando a una entidad básica llamada *GameObject* (véase apartado 5.3.1.1 GameObject). Mediante un conjunto de estas entidades se construye una escena (*Game Scene*), que es donde se diseñan y construyen los niveles, y es a partir de un conjunto de escenas qué se construye un videojuego.

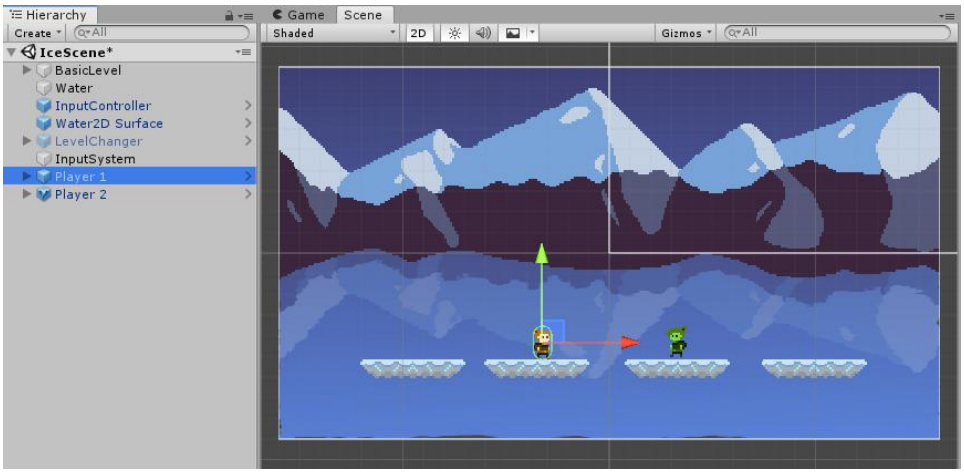

Figura 5.9. *Ejemplo de una escena de Cross Element.*

Antes de empezar a comentar la implementación de *Cross Element*, es necesario conocer como funcionan los componentes básicos de Unity, que generalmente son utilizados en casi todos los videojuegos hechos con el mismo motor. La figura 5.10 ilustra la construcción de entidades (*GameObjects*) en Unity y lista los componentes básicos que suelen formar una de estas entidades.

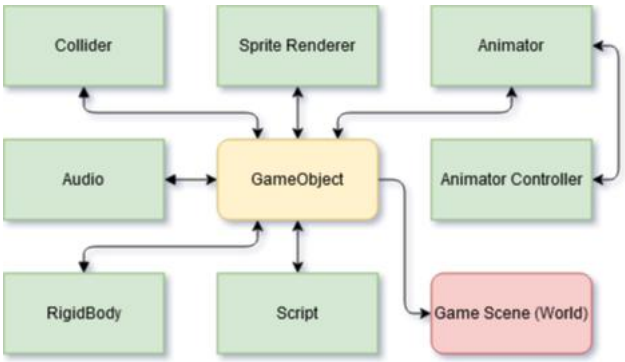

***Figura 5.10**. Esquema de los componentes básicos de una entidad en Unity*[11].

5.3.1.1 GameObject

Como se comentaba anteriormente, es la entidad primitiva con la que trabaja Unity. Todo *GameObject* contiene siempre un componente *Transform.* Es a través de este componente por el cual se controla la posición, rotación y escala de un *GameObject*, cada uno identificado por un vector de dimensión tres, como se ve en la figura 5.11.

[11] Fuente: basado en https://koenig-media.raywenderlich.com/uploads/2011/08/unity14_diagram.png

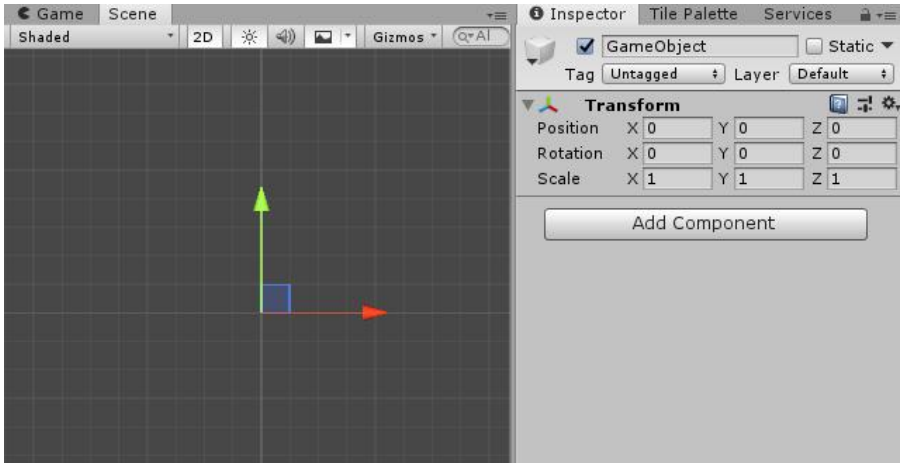

Figura 5.11. *Ejemplo de un GameObject con su componente Transform.*

5.3.1.2 Collider y RigidBody

Son los componentes básicos de físicas, tanto en 2D como 3D. El *RigidBody* controla cómo afectan los cálculos internos de físicas de Unity a un *GameObject*, por ejemplo para determinar colisiones entre *GameObjects* o la aceleración de la gravedad. Hay que tener en cuenta que, además, hay que añadir un componente *Collider*, que se encarga de determinar el área de colisión del *GameObject*, y generalmente suelen ir de la mano (figura 5.12). Existen diferentes tipos de *Colliders*; cuadrado o cúbico, circular o esférico, incluso en forma de cápsula, según las necesidades del desarrollador.

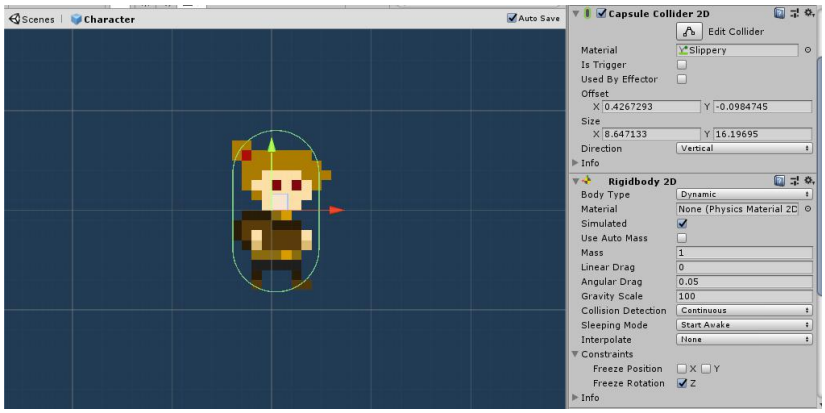

Figura 5.12. *GameObject con un RigidBody y un Collider de tipo cápsula.*

Comentar como excepción, que en algunos casos no es necesario añadir un *RigidBody* a un *Collider*, por ejemplo, si se implementa un objeto que el jugador puede recoger, seguramente no sea necesario, ya que esto causaría una colisión y aplicaría una fuerza que repele a ambos, por lo que en este caso es necesario activar la casilla *IsTrigger*, permitiendo así colisiones sin fuerza. Para este último caso, es interesante conocer que puedes acceder a la función *OnTriggerEnter* a través de un *Script* (véase apartado 5.3.1.4 *Script*), para controlar la lógica cuando colisiona el objeto con otro. Para implementar una lógica personalizada cuando colisiona un *RigidBody*, se sobrescribe el método *OnCollisionEnter*.

5.3.1.3 Animator

Es el componente básico para asignar una animación a un GameObject en escena. Este componente, además, requiere de una referencia a un *Animator Controller*, que define qué animación utilizar (en caso de tener más de una), además del cómo y cuándo debe transicionar una animación a otra. Además, requiere que el *GameObject* tenga un componente *Sprite Renderer*, que se encarga de renderizar los sprites en pantalla.

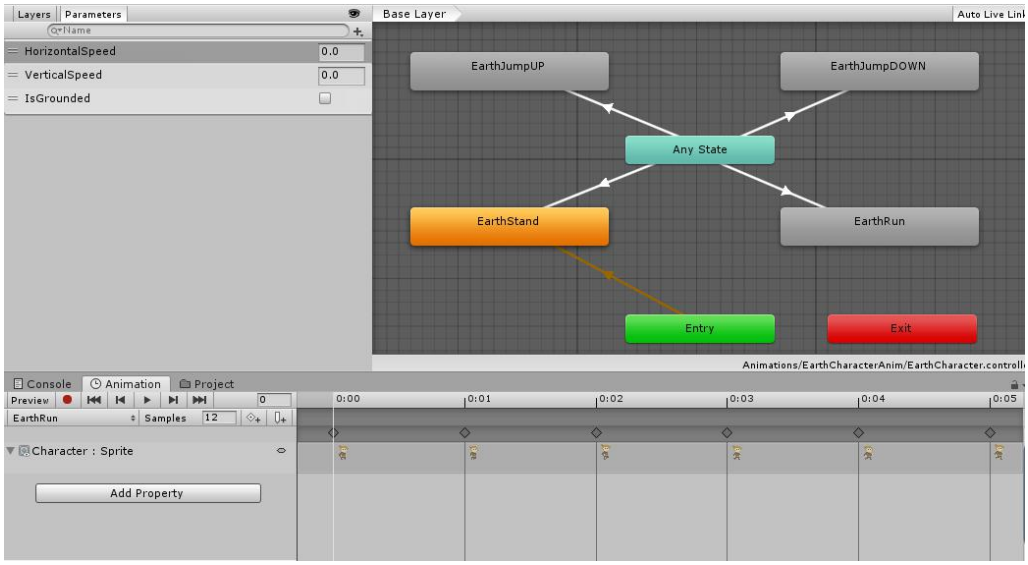

Figura 5.13. *Animator Controller con transiciones.*

5.3.1.4 Script

Un *Script* es una pieza de código, generalmente escrita en C#, que puede añadirse a un *GameObject* para definir una lógica de éste, por ejemplo, para que se mueva en una dirección cuando el sistema de entrada detecte una presión de tecla definida o instanciar un *Prefab*, entre otros muchos.

Una particularidad de los *Scripts* es que heredan de un clase base implementada por Unity, llamada *MonoBehaviour*, permitiendo sobrescribir funciones implementadas por esta clase para definir una lógica propia, por ejemplo, el método *Start* o *Awake*, que son llamados internamente por Unity cuando se inicia un escena, o los métodos *Update* o *FixedUpdate*, que son llamados cada *frame* (*FixedUpdate* se llama menos veces, por eso es ideal para hacer cálculos de físicas o trabajar con variables de tiempo).

Por último, cuando se desarrolla un *Script* heredando de la clase *MonoBehaviour*, permite acceder a los componentes del *GameObject* a través de código, por ejemplo, para modificar el componente *Transform* y así poder rotar o mover dicho objeto, entre otros.

```
using UnityEngine;

0 references
public class ItemController : MonoBehaviour
{
    3 references
    public Transform[] item_spawns;
    2 references
    public GameObject[] item_prefab;
    2 references
    private const float time_spawn = 5f;
    3 references
    private float spawn_timer = time_spawn;

    0 references
    private void FixedUpdate()
    {
        if (spawn_timer <= 0)
            SpawnItem();
        else
            spawn_timer -= Time.deltaTime;
    }

    1 reference
    private void SpawnItem()
    {
        int random_spawn = Random.Range(0, item_spawns.Length);
        int random_item = Random.Range(0, item_prefab.Length);

        Instantiate(item_prefab[random_item], item_spawns[random_spawn].position, item_spawns[random_spawn].rotation);

        spawn_timer = time_spawn;
    }
}
```

Figura 5.14. *Script sobrescribiendo la función FixedUpdate en C#.*

También existe la posibilidad de crear Scripts que no hereden de *MonoBehaviour*, aunque no se pueden añadir directamente como componente a un *GameObject.* Sí que puede ser útil, por ejemplo, para definir estructuras de datos que sean utilizadas por otro *Script* que herede de *MonoBehaviour*.

Por último, comentar que cuando se declara una variable como pública en un *Script* que forma parte de un *GameObject*, se puede añadir una referencia desde el editor, por ejemplo, para acceder a la posición de otro *GameObject* mediante su *Transform*.

Figura 5.15. *Referencia pública de un Script a un componente Transform, entre otros.*

5.3.1.5 Prefab

Un *Prefab* es un *GameObject* hecho *Asset*, y es de mucha utilidad, ya que te permite reutilizar GameObjects ya definidos, por ejemplo, para utilizarlos como plantilla de un personaje ya construido, o a partir de la plataforma de una escena, insertar varias iguales.

5.3.2 Implementación de *Cross Element*

Al inicializar *Cross Element* se configura por defecto el control de los mandos conectados. Seguidamente se muestra un menú principal desde el cual se puede realizar una configuración avanzada, empezar una partida o cerrar el juego. La figura 5.16 define el diagrama de flujo que recorre el videojuego, detallándose cada proceso en los siguientes capítulos.

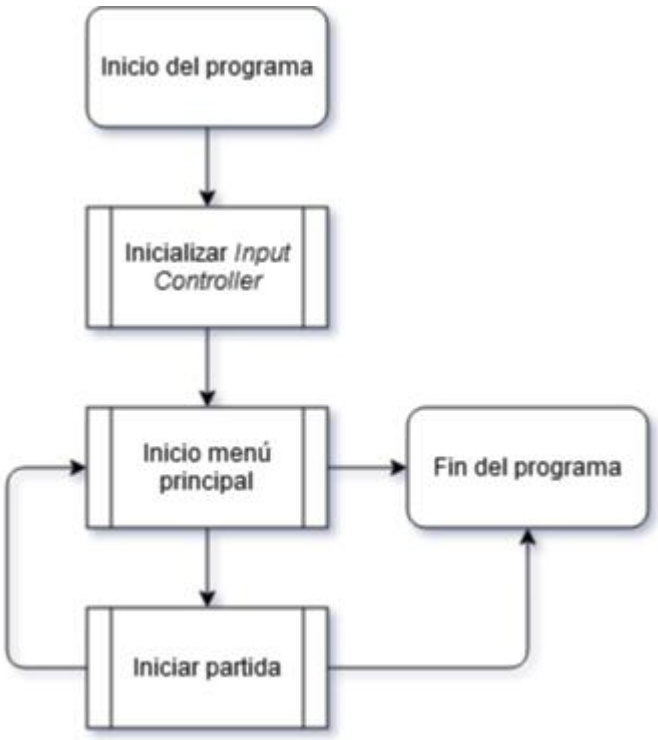

Figura 5.16. *Diagrama de flujo de Cross Element.*

5.3.2.1 Input Controller

Antes de mostrar el menú principal, se carga un *Script* llamado *InputController*, que se encarga de detectar cuantos mandos están conectados al ordenador, y asigna para cada uno de ellos un código identificativo (ID). Además, contiene un diccionario de pares de número entero y un objeto de tipo *CustomInput*, el cuál es una estructura de datos que define por defecto qué botones disparan las diferentes acciones de los personajes. Por cada mando conectado, se crea una entrada en este diccionario con el ID de mando asignado y un *CustomInput* por defecto, hasta cuatro mandos, habilitando la detección de entrada de botones de estos. Además, cada *CustomInput* lanza una rutina en intervalos de un segundo, para detectar si el mando al que está vinculado con su ID se ha desconectado, y en tal caso lanzar un mensaje de aviso en la interfaz. Estos *CustomInput* pueden habilitarse o deshabilitarse desde la interfaz de juego. A continuación se describe el flujo de un *InputController*.

Figura 5.17. *Diagrama de flujo de Input Controller.*

Este controlador está implementado siguiendo el patrón *Singleton*[12], de manera que siempre existe una única instancia de este *Script*. Esto será de utilidad más adelante, ya que los personajes en escena lanzan peticiones constantemente a este *Script* para detectar si se está pulsando un botón del mando con su mismo ID.

5.3.2.2 Menú principal

Es el primer menú que se visualiza al inicializarse el videojuego. Está compuesto por varios botones que muestran cada uno de los diferentes paneles a través de los cuales se navega por las opciones de *Cross Element*. La figura 5.18 muestra el flujo de este panel.

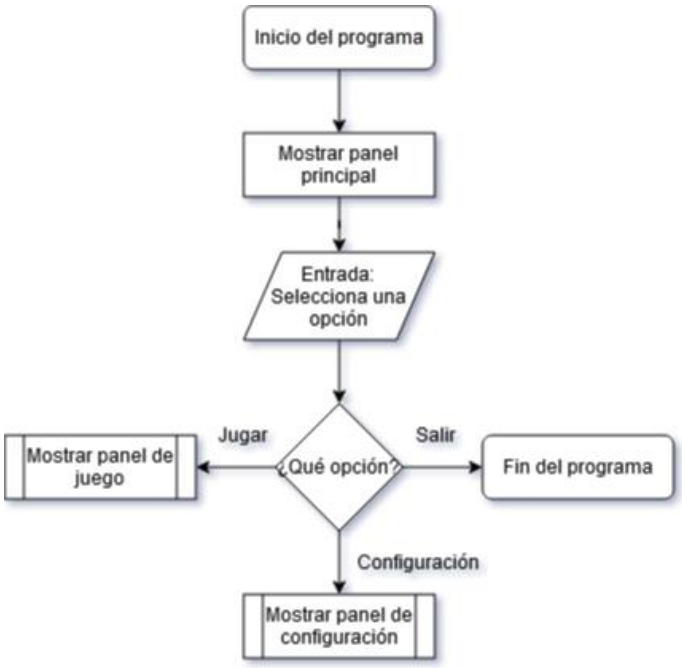

Figura 5.18. *Diagrama de flujo del menú principal.*

[12] Singleton o instancia única, es un patrón de diseño que permite restringir la creación de objetos pertenecientes a una clase o el valor de un tipo a una única instancia.

El panel de configuración permite cambiar algunos ajustes por defecto; la asignación de botones, el volumen de los efectos de sonido y música ambiental, y la resolución de pantalla. Además es posible guardar estas configuraciones para próximas sesiones.

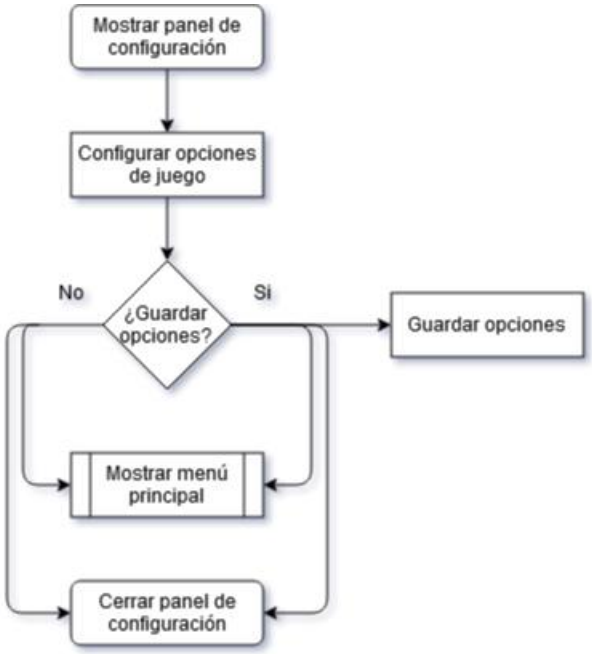

Figura 5.19. *Diagrama de flujo del menú de configuración.*

Finalmente, se juega una partida de *Cross Element* a través del panel de juego, donde se configuran las opciones de la partida, como el tiempo máximo de duración y el número de rondas a jugar, además de la selección de personajes y del nivel en el que se desarrollará dicha partida. Las opciones configuradas de esta manera son guardadas en el *Script CharacterController*, que compone junto el *InputController* un *GameObject* que persiste hasta que se cierra el programa, y donde se registra la relación entre el código ID de mando y el personaje escogido, para cada jugador.

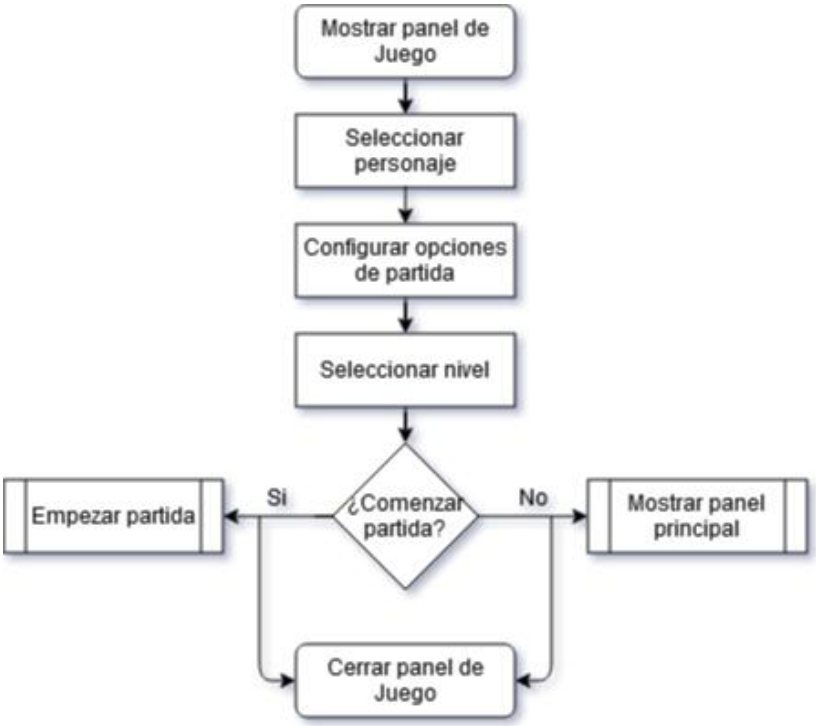

Figura 5.20. *Diagrama de flujo del panel de juego.*

5.3.2.3 Inicio de una partida

La lógica de la partida está controlada por un *Script* llamado *GameStart*, implementado utilizando el patrón Singleton. Al comenzar la partida, genera una instancia de cada personaje escogido en el panel de juego y los añade a una lista, y seguidamente los sitúa en posiciones aleatorias (dentro de un conjunto no aleatorio) del nivel seleccionado. Finalmente, se les asigna el código ID del jugador que lo ha escogido a cada uno de los personajes, respectivamente.

Cada personaje, cuando muere, avisa al *GameStart* (al ser único, no requieren de una referencia pública) utilizando el código ID que se le ha asignado, y este *Script* lo elimina de la lista inicial. La partida termina cuando la lista únicamente posea un elemento, mostrando un panel indicando el ganador de ese juego (el último elemento en la lista). En caso de haberse

configurado varias rondas de juego, la partida terminará cuando un jugador alcance el mismo número de puntos de victoria.

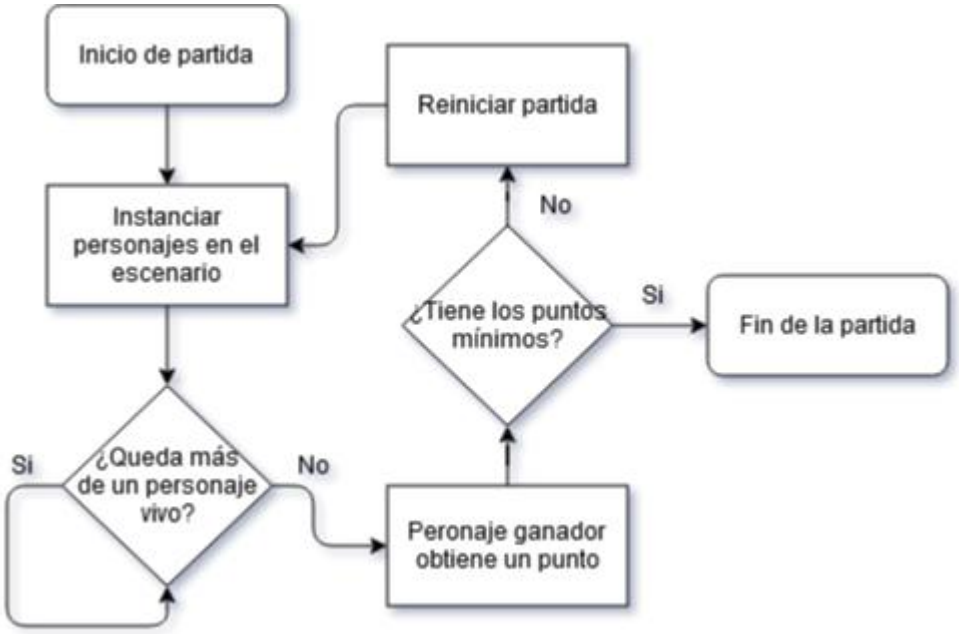

Figura 5.21. *Diagrama de flujo de una partida.*

5.3.2.4 Cuadro de colisiones

Unity permite configurar *Layers*[13] que se pueden asignar a un *GameObject*. Esta característica es muy útil, ya que permite identificar a que *Layer* pertenece un *GameObject* cuando colisiona con otro, controlando que lógica se ejecuta cuando pertenece a un tipo u otro.

A continuación se muestra el cuadro de colisiones de *Cross Element* (figura 5.22), indicando qué entidades colisionan entre si sirviendo de referencia para los siguientes apartados donde se describe la lógica de colisión con otras entidades.

[13] *Layer*: en Unity, es una capa que se asigna a un GameObject, para controlar qué entidades pueden colisionar entre si. También se utiliza para generar un orden de renderizado (por capas).

Layers	Tierra	Agua	Fuego	Viento	Energía	Personaje	Plataforma	Muerte	Limite	Objeto
Tierra					X	X				
Agua		X			X	X	X	X	X	
Fuego			X		X	X	X	X	X	
Viento					X	X				
Energía	X	X	X	X	X	X	X	X	X	
Personaje	X	X	X	X	X	X	X	X	X	
Plataforma		X	X		X	X	X		X	
Muerte		X	X		X	X				
Limite		X	X		X	X	X			
Objeto						X	X	X	X	X

Figura 5.22. *Cuadro de colisiones de Cross Element.*

5.3.2.5 Personajes

Los personajes de *Cross Element* están construidos a partir de un mismo *Prefab* base, variando valores de algunos de los componentes, además de los sprites y las animaciones que utilizan.

Figura 5.23. *Personajes de Cross Element.*

En la figura 5.24 se puede observar los componentes de este *Prefab*, y a continuación se detalla la funcionalidad de cada uno de estos.

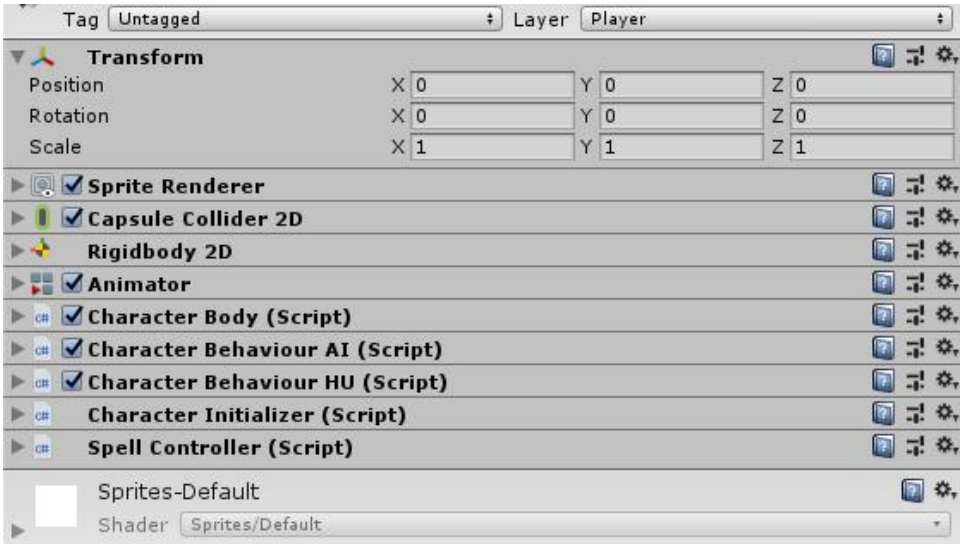

Figura 5.24. *Componentes de un personaje.*

El componente principal es el *Script CharacterBody*, el cuál se encarga de llevar a cabo las diversas acciones que puede realizar un personaje, además de registrar la vida, energía total, velocidad de movimiento y fuerza de salto, que varían según el personaje siguiendo la siguiente tabla (figura 5.25).

Personaje	Vida máxima	Energía máxima	Velocidad de movimiento*	Fuerza de salto*
Tierra	100	5	100	100
Agua	120	5	90	90
Fuego	90	6	110	110
Aire	80	6	120	120
* Porcentaje de una fuerza y velocidad base.				

Figura 5.25. *Valores de CharacterBody pora cada personaje.*

Para que un personaje realice un acción, el *CharacterBody* llama a la instancia del *InputController*, enviando su código ID, para detectar si el mando al que está asociado está presionando algún botón configurado en su *CustomInput*, y de ser así realiza la acción respectiva. Las acciones que puede llevar a cabo un personaje son:

1. **Mover:** un personaje se mueve evaluando si el mando que lo controla esta presionando los botones de movimiento horizontal, y aplicando un vector de movimiento al componente *RigidBody2D*. Por ejemplo, mediante *Script*:

```
GetComponent<RigidBody2D>().velocity = new Vector2(float, float);
```

2. **Saltar:** un personaje puede saltar. Esta acción se lleva a cabo aplicando una fuerza vertical en el componente *RigidBody2D*. Una manera de aplicar esta fuerza en Unity a través de un *Script* es utilizando la siguiente función:

```
GetComponent<RigidBody2D>().AddForce(new Vector2(float, float));
```

3. **Bola de energía:** puede disparar un proyectil rectilíneo, que se destruye cuando impacta con un elemento del escenario. En caso de ser otro personaje con el que colisiona, le aplicará una cantidad de daño leve. Esta acción se lleva a cabo a través del *SpellController*, que es llamado desde el *CharacterBody* utilizando una referencia pública (véase apartado 5.3.2.8 Bola de energía). Cada disparo consume 1 punto de energía, de la cual el personaje regenera 1 punto cada 2 segundos, hasta un máximo de 5.

4. **Habilidad especial:** cada personaje tiene la posibilidad de convocar un elemento cuando recoge uno de los objetos elementales (véase apartado 5.3.2.7 Objetos). Esta acción se lleva cabo a través del *Script SpellController*.

Además, el *Prefab* de personaje posee dos *GameObject* hijo; ambos son utilizados por su componente *Transform*, cuya posición del primero, *ProyectileOrigin*, sirve como punto de partida de los proyectiles (bola de energía y la habilidad de fuego). El segundo, *GroundCheck*, se utiliza junto a un *Collider* para detectar si está colisionando con el suelo, con el fin de controlar que solo se pueda saltar de nuevo cuando el personaje pise el terreno.

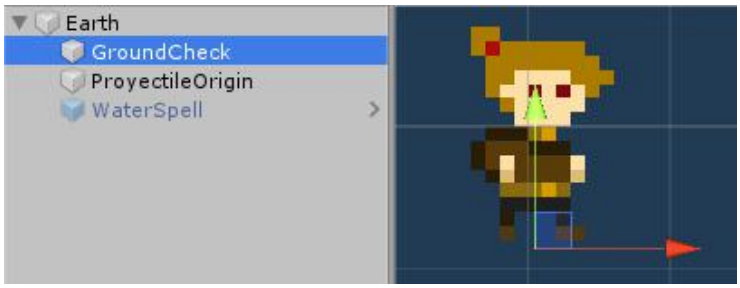

Figura 5.26. *Posición del GameObject GroundCheck.*

Los personajes poseen un Collider de tipo cápsula. De esta manera se consigue generar unas colisiones más jugables, generando deslizamientos en ciertas superficies. Además, el RigidBody puede colisionar con diferentes entidades de *Cross Element*. El comportamiento de dichas colisiones está definido por la figura 5.27.

Tipo de entidad	Comportamiento
Personaje	Un personaje no puede traspasar otro personaje.
Plataforma	Un personaje no puede traspasar una plataforma.
Muerte	El personaje muere.
Límite	Un personaje no puede traspasar un límite.

Figura 5.27. *Tabla de comportamiento ante colisiones de un personaje*[14].

[14] Existen otras entidades con las que un personaje colisiona, pero el comportamiento de estas colisiones esta definida por las otras entidades.

Finalmente, en el siguiente diagrama (ver figura 5.28) se puede visualizar el flujo de comportamiento de un personaje.

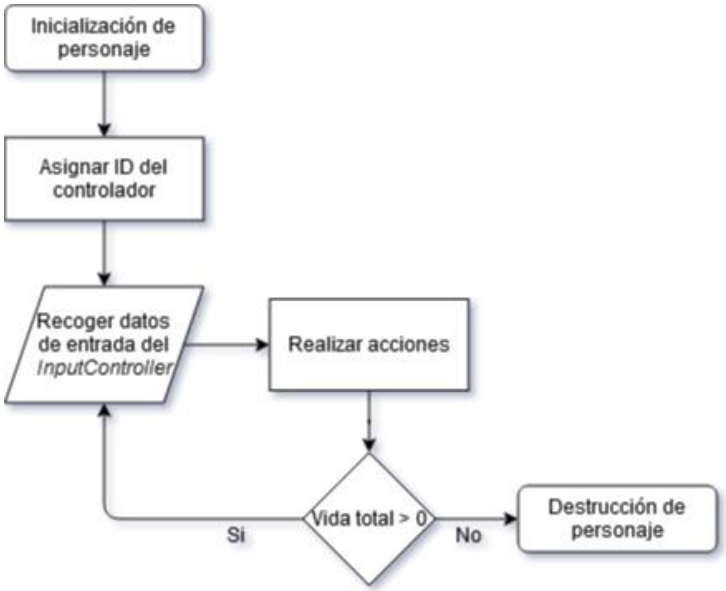

Figura 5.28. *Diagrama de flujo de un personaje.*

5.3.2.6 Controlador de objetos

En cada escena, el *GameObject* que tiene el componente *GameStart*, contiene además un Script *ItemController*, que contiene una referencia a un grupo de *GameObjects* vacíos (solo contienen el componente *Transform*), que sirven para indicar la posición donde van a salir los objetos.

Cada 5 segundos, en una posición aleatoria de las anteriores mencionadas, se instancia un objeto aleatorio que desciende. Continúa de esta manera hasta que termina la partida.

5.3.2.7 Objetos

Los objetos están construidos a partir de un mismo *Prefab*, ya que su comportamiento es igual para todos, diferenciándose los sprites para cada uno de ellos y la habilidad que asignan al *SpellController* del personaje que recoja uno de ellos.

Figura 5.29. *Sprites de los objeto.*

La figura 5.30 muestra la composición del *Prefab* base utilizado para los objetos, y se explica el funcionamiento de cada uno de estos componentes a continuación.

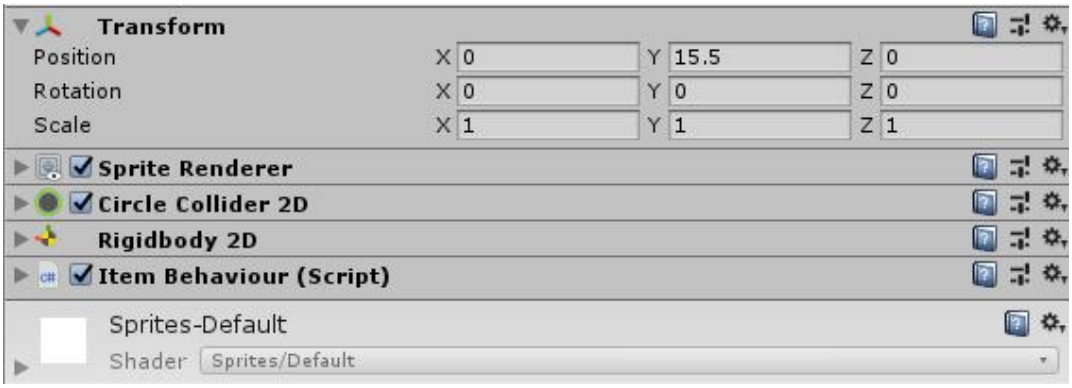

Figura 5.30. *Componentes de un objeto.*

Los objetos se componen de un *Collider* de tipo círculo, además de un *RigidBody2D*, de manera que son afectados por la gravedad, es decir, que caen. Sin embargo, el *Script ItemBehaviour* desactiva este *RigidBody2D* cuando se detecta que el objeto ha colisionado con el terreno, y activa la función *IsTrigger* del *Collider*. De esta manera, cuando colisiona con un

personaje, no se genera una colisión con fuerzas. La figura 5.31 describe el comportamiento de un objeto cuando colisiona con otra entidad.

Tipo de entidad	Comportamiento
Personaje	Otorga al personaje una habilidad elemental.
Plataforma	Desactiva el RigidBody y estaciona.
Muerte	Se destruye el objeto.
Límite	Se bloquea (no puede atravesar un límite).
Objeto	Si el otro objeto estaba estacionado, el nuevo se destruye.

Figura 5.31. Tabla de comportamientos ante colisiones de un objeto.

Por último, la figura 5.32 describe el flujo de comportamiento de un objeto, durante su ciclo de vida en la partida.

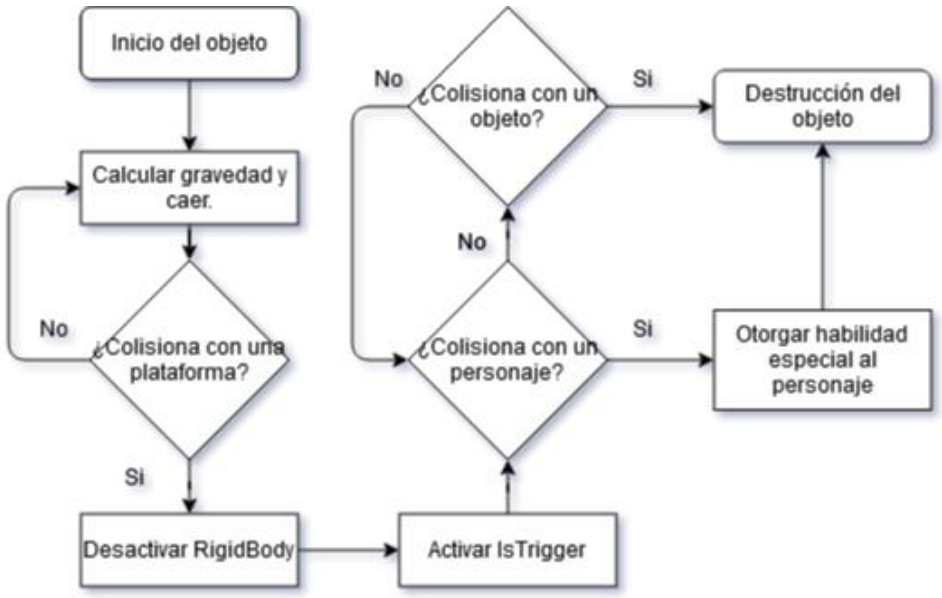

Figura 5.32. Diagrama de flujo de un objeto.

5.3.2.8 Bola de energía

Las bolas de energía están construidas a partir del mismo *Prefab*, *NormalSpell*, variando el color de cada una según el personaje que la utiliza.

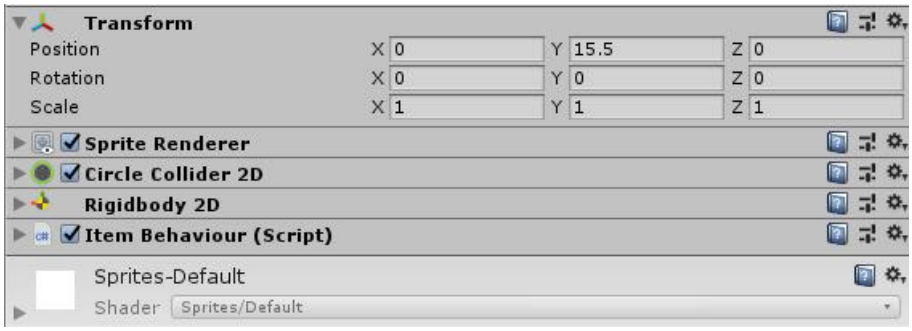

Figura 5.33. *Sprites de las bolas de energía.*

La figura 5.34 muestra la composición del *Prefab* base utilizado para las bolas de energía, y se explica a continuación el funcionamiento de cada uno de estos componentes.

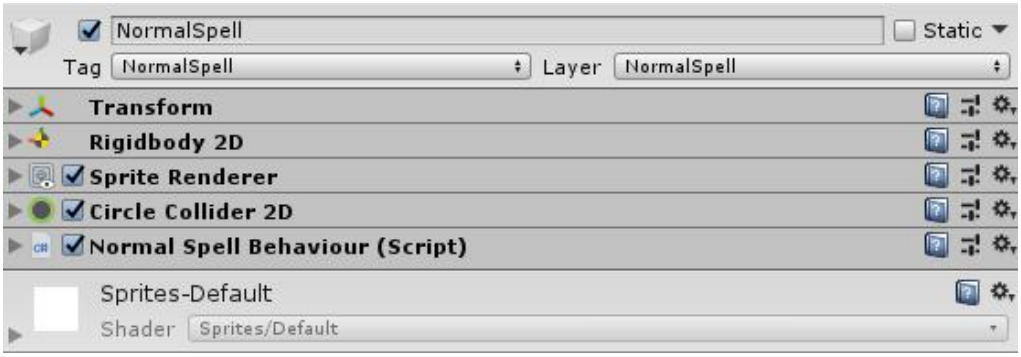

Figura 5.34. *Componentes de una bola de energía.*

El componente principal es el script *NormalSpellBehaviour*. Cuando se instancia una bola de energía, este script controla que avance en dirección rectilínea desde el componente hijo *ProyectileOrigin* del personaje hasta que colisiona con otra entidad, siguiendo el

comportamiento en función de la figura 5.35. Además, tiene un *Collider* de tipo circulo y un *RigidBody2D*, sin ser afectado por la gravedad.

Tipo de entidad	Comportamiento
Personaje	Se destruye y hace 10 puntos de daño al personaje.
Plataforma	Se destruye.
Muerte	Se destruye.
Límite	Se destruye.

Figura 5.35. *Tabla de comportamientos ante colisiones de una bola de energía*[15].

La figura 5.36 muestra el flujo de comportamiento de una bola de energía durante su ciclo de vida.

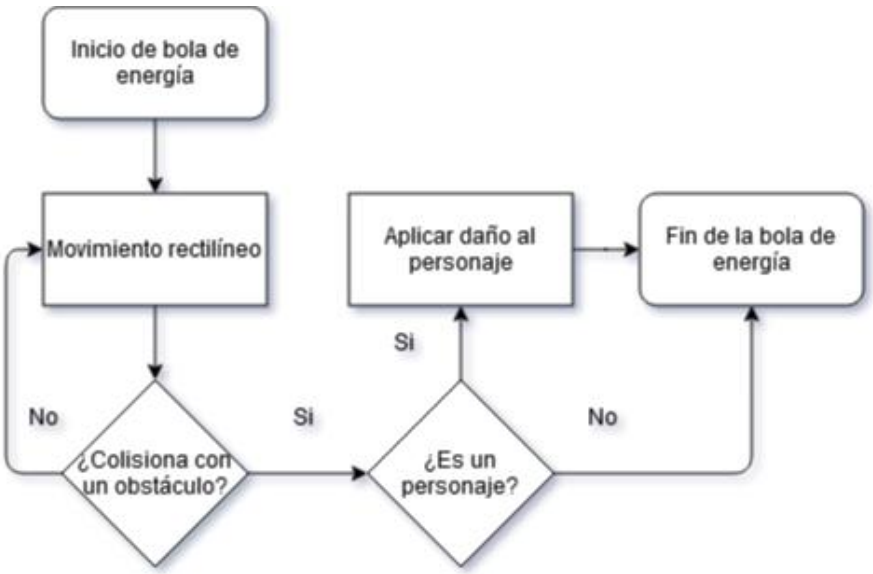

Figura 5.36. *Diagrama de flujo de una bola de energía.*

[15] Existen otras entidades con las que una bola de energía colisiona, pero el comportamiento de estas colisiones está definida por las otras entidades.

5.3.2.9 Habilidades

Las habilidades se manejan a través de un script, *SpellController*, que compone al personaje. Cuando un personaje recoge un objeto, este *script* asigna la habilidad especial correspondiente a dicho objeto.

Cuando se utiliza una habilidad especial, empieza una cuenta atrás de 3 segundos, impidiendo volver a utilizar la misma habilidad hasta que no termina esta cuenta atrás. Si después de utilizar la habilidad especial, el personaje recoge un objeto que otorga otra habilidad distinta, la cuenta atrás se termina inmediatamente, permitiendo al personaje utilizar la nueva habilidad sin tener que esperar. Los siguientes apartados describen el comportamiento de cada una de estas habilidades.

5.3.2.10 Habilidad de tierra

La habilidad de tierra genera un área alrededor del personaje que la utiliza, dañando a los personajes enemigos que alcanza, además de dejarlos inutilizados por un breve periodo de tiempo.

Figura 5.37. *Sprite de la habilidad de tierra.*

Para implementar la habilidad de tierra, se ha implementado un *Prefab* llamado *EarthSpell*, que está compuesto por una serie de componentes básicos; un *SpriteRenderer*, un *Animator* y un *Collider* de tipo círculo con la opción de *IsTrigger* activada. Además, destaca un componente principal de tipo *script*, *EarthSpellBehaviour*, que controla el comportamiento de la habilidad. La figura 5.38 muestra todos los componentes, y a continuación se describe el funcionamiento del *script*.

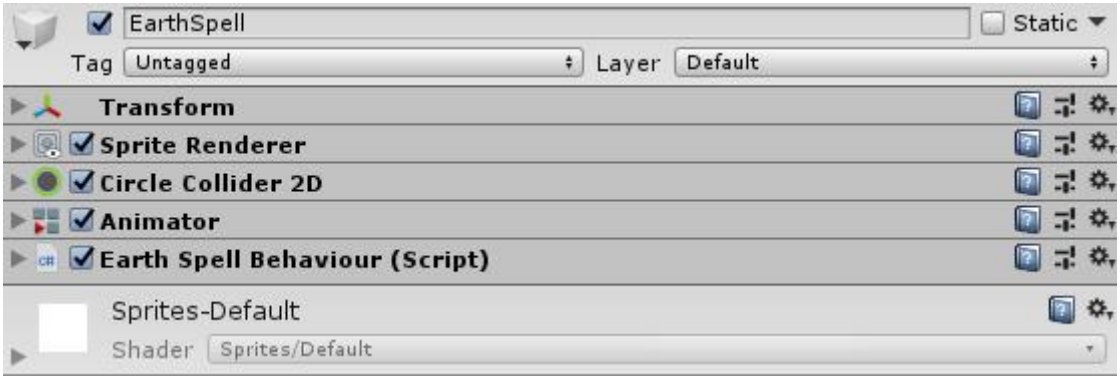

Figura 5.38. *Componentes de la habilidad de tierra.*

El *script EarthSpellBehaviour* contiene una lista de los ID de los personajes. Cuando la habilidad colisiona con uno de estos personajes, se añade su código ID a esta lista con el objetivo de bloquear posibles colisiones adicionales, además de infligir a cada uno de estos personajes una cantidad moderada de daño, además de dejarlos inutilizados. Estos valores se pueden consultar a continuación.

Valores	
Variable	Valor
Daño (entero)	15
Tiempo de inutilización (segundos)	0.75
Comportamiento ante colisiones con entidades	

Tipo de entidad	Descripción
Personaje	Inutiliza e inflige daño al personaje.
Bola de energía	Destruye la bola de energía.

Figura 5.39. Tabla de valores y comportamientos ante colisiones de la habilidad de tierra.

Por último, la figura 5.40 muestra un diagrama de flujo mostrando el ciclo de vida de la habilidad de tierra.

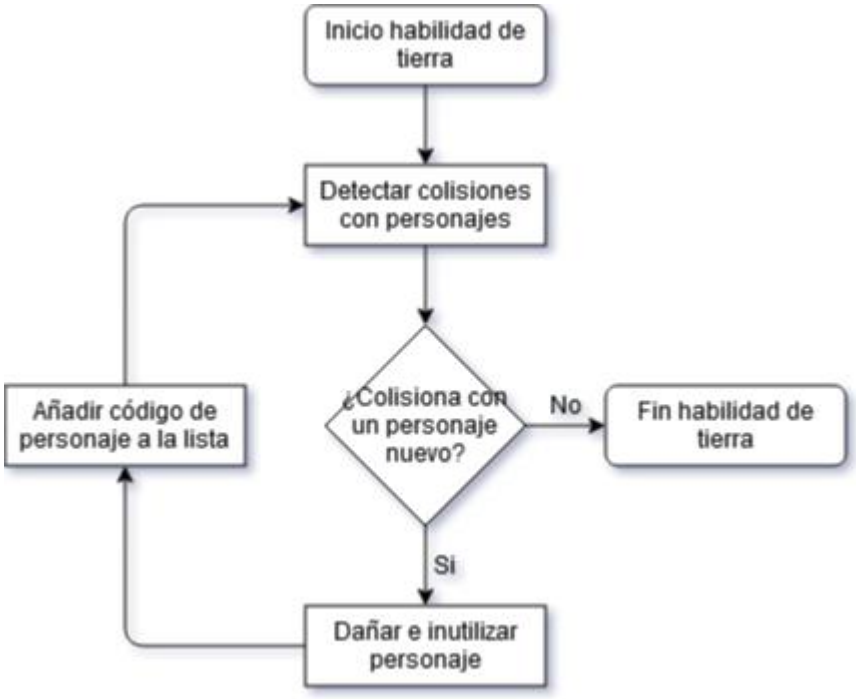

Figura 5.40. Diagrama de flujo de la habilidad de tierra.

5.3.2.11 Habilidad de agua

El personaje que utiliza esta habilidad se convierte en una masa de agua que avanza rápidamente en la dirección que se desea, permitiéndole elevarse en el aire y volver inmune al resto de ataques durante un breve periodo de tiempo.

Figura 5.41. Sprite del hechizo de agua.

La implementación de la habilidad de agua ha sido realizada mediante la construcción de un *Prefab*, llamado *WaterSpell*, el cual está compuesto por una serie de componentes básicos; un *SpriteRenderer*, un *ParticleSystem* y un *Collider* de tipo círculo con la opción de *IsTrigger* activada. También tiene un componente *script IceSpellBehaviour*, que controla la lógica de la habilidad. A continuación se observa una figura con los componentes de este *Prefab*.

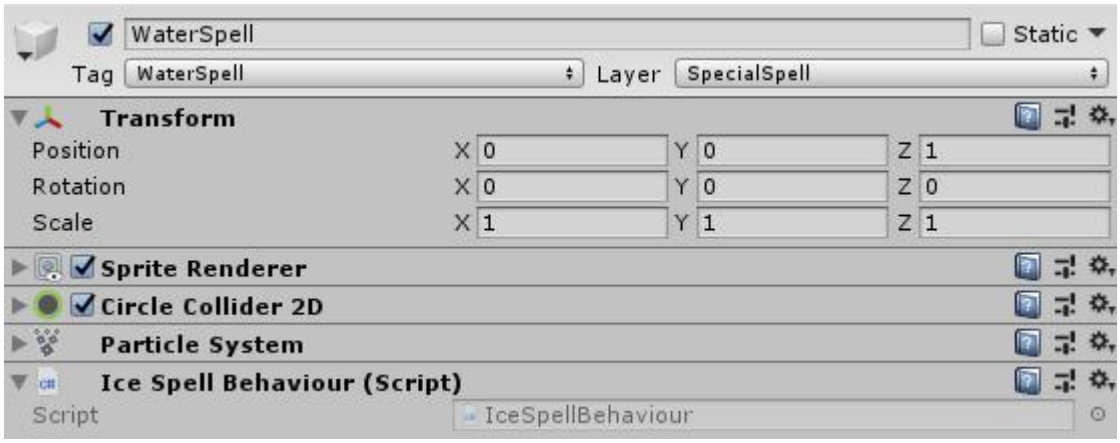

Figura 5.42. Componentes de la habilidad de agua.

Cuando un personaje utiliza la habilidad de agua, este desaparece momentáneamente del juego, siendo el *script IceSpellBehaviour* quien toma el control del *CharacterBody* del personaje y moviéndolo en la dirección que el jugador decida. Esto permite al personaje ser inmune a los ataques de los enemigos, además de destruir las bolas de energía y dañar a los enemigos con los

que colisione. La figura 5.43 muestra los valores de duración y daño infligido, además del comportamiento ante colisiones con otras entidades.

Valores	
Variable	Valor
Duración (segundos)	0.75
Daño (entero)	10
Comportamiento ante colisiones con entidades	
Tipo de entidad	Descripción
Personaje	Empuja e inflige daño al personaje.
Plataforma	Se bloquea (no puede atravesar una plataforma).
Muerte	El personaje que utiliza la habilidad muere.
Límite	Se bloquea (no puede atravesar un límite).
Bola de energía	Destruye la bola de energía.
Habilidad de agua	Cuando colisiona con otra habilidad de agua, se repelen mutuamente.

Figura 5.43. *Tabla de valores y comportamientos ante colisiones de la habilidad de agua.*

Por último, la figura 5.44 muestra un diagrama de flujo mostrando el ciclo de vida de la habilidad de agua.

Figura 5.44. *Diagrama de flujo de la habilidad de agua.*

5.3.2.12 Habilidad de fuego

La habilidad de fuego dispara un proyectil, que avanza en la dirección del personaje. La primera entidad que colisiona con este proyectil hace que explote generando un área que daña y empuja a los personajes dentro de esta. En la figura 5.45 se muestra la animación de esta habilidad.

Figura 5.45. Animación de la habilidad de fuego.

El *Prefab FireSpell* implementa la lógica de la habilidad de fuego. Está construida por un conjunto de componentes básicos; un *SpriteRenderer*, un *Animator*, un *Collider* de tipo cuadrado y un *RigidBody2D*. El componente principal es el Script *FireSpellBehaviour*, que se encarga de la lógica de esta habilidad. A continuación se muestra una figura con todos los componentes de la habilidad de fuego.

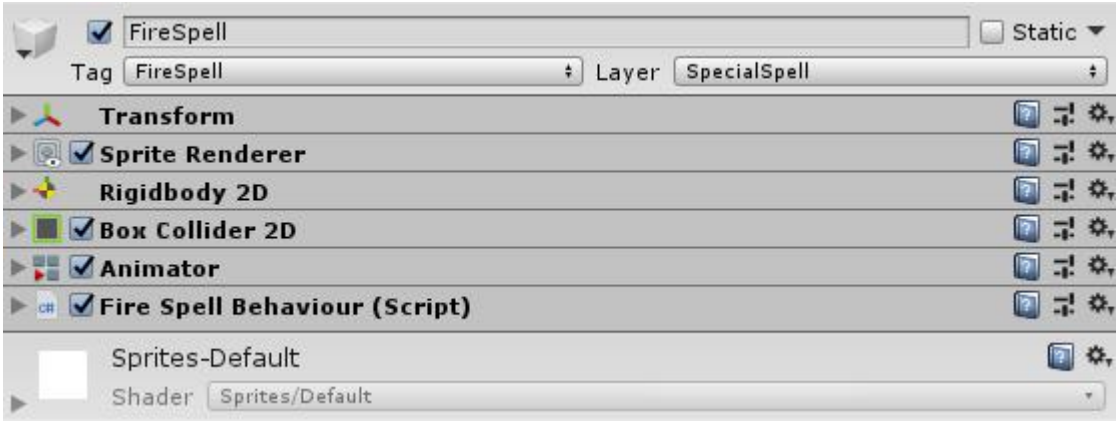

Figura 5.46. Componentes de la habilidad de fuego.

Cuando un personaje utiliza la habilidad de fuego, se instancia el *Prefab FireSpell*. El componente *FireSpellBehaviour* controla la velocidad de este, haciendo que acelere en la dirección que mira el personaje. Cuando colisiona con una entidad, este explota en un área alrededor, dañando severamente a los personajes alcanzados además de empujarlos en dirección opuesta. La figura 5.47 muestra los valores de la habilidad de fuego, así como la descripción del comportamiento ante colisiones con otras entidades.

Valores	
Variable	Valor
Velocidad (unidades de Unity)	10000
Daño (entero)	25
Comportamiento ante colisiones con entidades	
Tipo de entidad	Descripción
Personaje	Daña al personaje y lo empuja. Además, el proyectil explota generando el mismo efecto.
Plataforma	El proyectil explota generando un área que daña y empujar a los personajes alcanzados.
Muerte	El proyectil desaparece.
Límite	El proyectil explota generando un área que dañando y empujando a los personajes alcanzados.
Bola de energía	Destruye la bola de energía y sigue avanzando.
Habilidad de fuego	Ambas habilidades explotan, dañando y empujando a los personajes alcanzados.

Figura 5.47. *Tabla de valores y comportamientos ante colisiones de la habilidad de fuego.*

Por último, la figura 5.48 muestra un diagrama de flujo mostrando el ciclo de vida de la habilidad de fuego.

Figura 5.48. *Diagrama de flujo de la habilidad de fuego.*

5.3.2.13 Habilidad de viento

Cuando un personaje utiliza la habilidad de viento, genera un área de fuertes vientos que daña y empuja a todo personaje enemigo que se encuentre dentro de esta área. La figura 5.49 muestra la animación de la habilidad de viento.

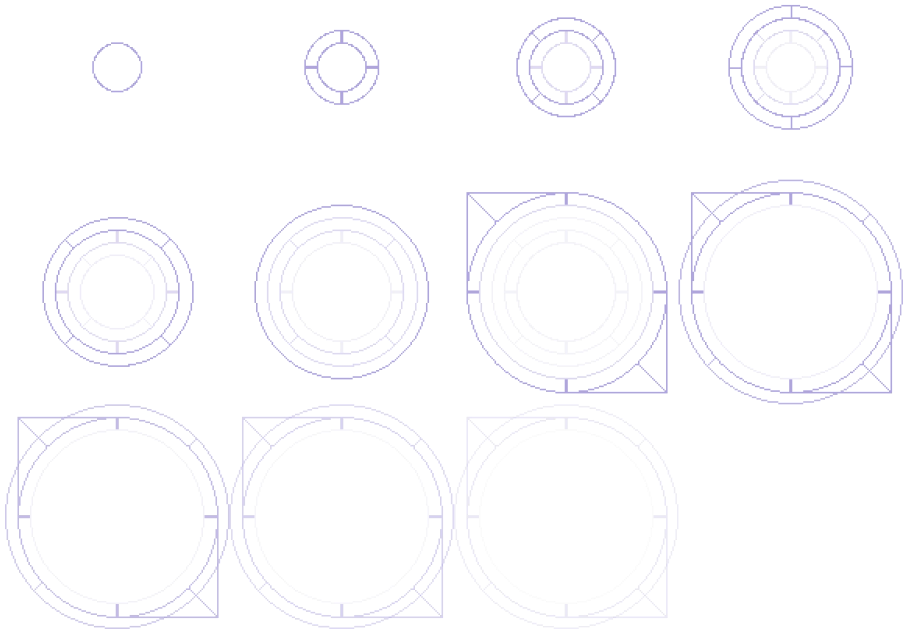

Figura 5.49. Animación del hechizo de viento.

Esta habilidad está implementada mediante un *Prefab*, *WindSpell*, que utiliza varios componentes básicos de Unity como; un *SpriteRenderer*, un *Collider* de tipo círculo con la opción de *IsTrigger* activada y un *Animator*. También forma parte de este *Prefab* el componente *script WindSpellBehaviour*, el cual se describe más adelante. La figura 5.50 muestra los componentes de un *WindSpell*.

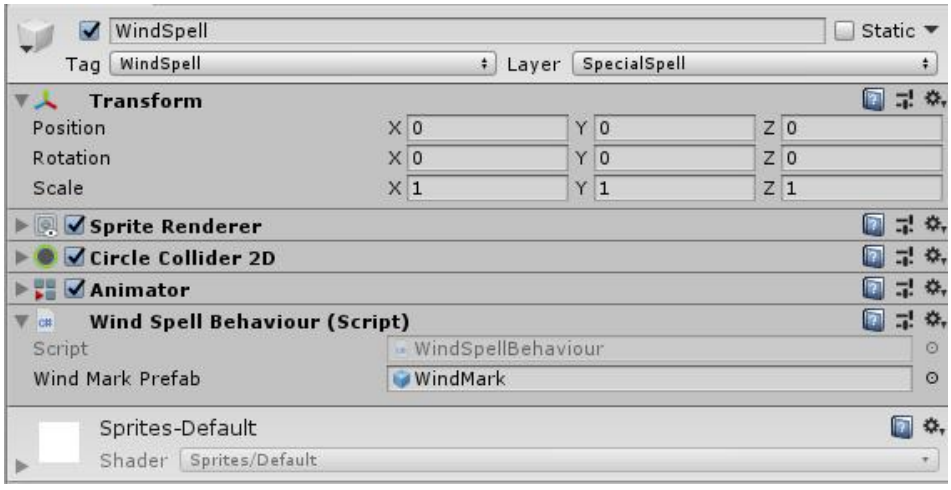

Figura 5.50. Componentes de la habilidad especial de viento.

Al utilizar la habilidad de viento, se instancia el *Prefab WindSpell* en la ubicación del personaje, y su *Collider* se expande con el tiempo. El *script WindSpellBehaviour* controla que la habilidad colisione una sola vez con cada personaje, almacenando en una lista el código ID de cada personaje alcanzado. Cuando colisiona con un personaje, calcula el vector dirección entre el centro del personaje que utiliza la habilidad y el primero, y en función de su magnitud, aplica un porcentaje proporcional de la fuerza y daño, siendo máximos cuando la distancia es mínima, y mínimos cuando la distancia es máxima, siguiendo los valores de la figura 5.51.

Valores	
Variable	Valor
Fuerza de empuje mínimo (unidades de Unity)	10000
Fuerza de empuje máximo (unidades de Unity)	16000
Daño mínimo (entero)	11
Daño máximo (entero)	22
Duración (segundos)	1
Comportamiento ante colisiones con entidades	
Tipo de entidad	Descripción
Personaje	Daña al personaje y lo empuja, en función de la distancia.
Bola de energía	Destruye la bola de energía.

Figura 5.51. *Valores de daño y fuerza de empuje y comportamiento ante colisiones con entidades.*

Por último, la figura 5.52 muestra un diagrama de flujo mostrando el ciclo de vida de la habilidad de viento.

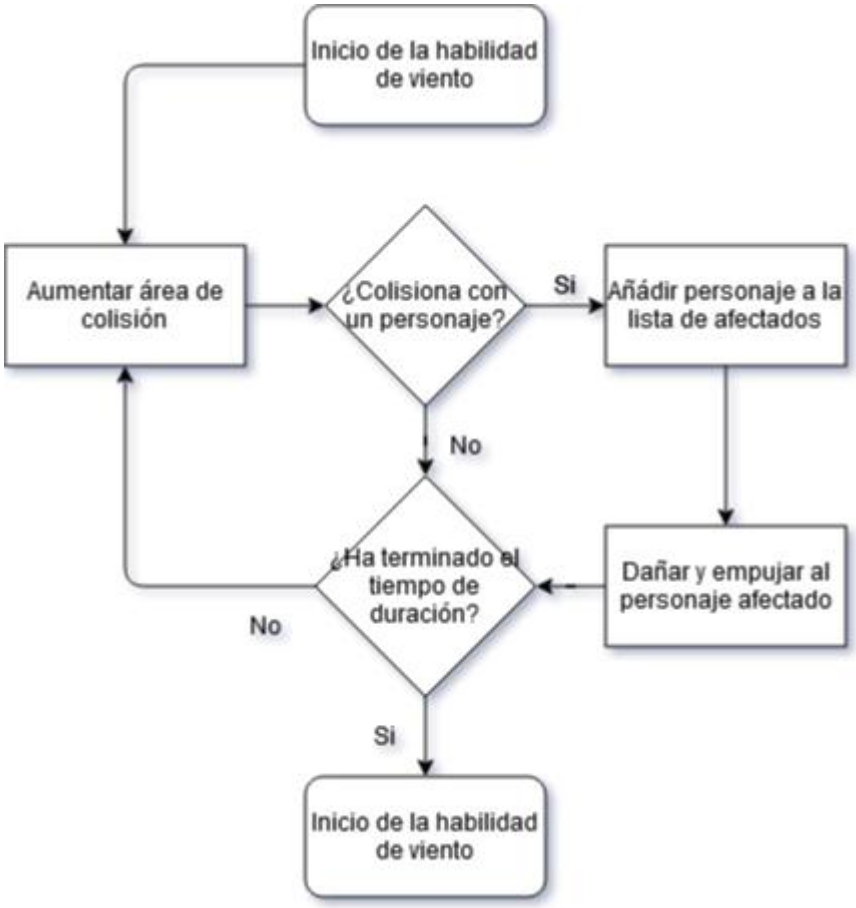

Figura 5.52. *Diagrama de flujo de la habilidad de viento.*

5.3.2.14 Escenario de agua

El escenario representa el elemento de agua, un lago rodeado por montañas nevadas donde se han formado unas pequeñas plataformas de hielo, las cuáles se inclinan en función del peso de los personajes que se sitúen encima, donde lucharan por conseguir el trofeo.

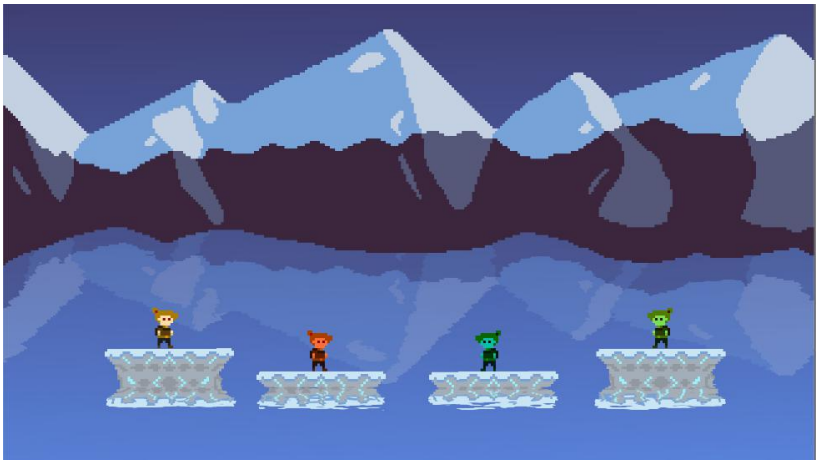

Figura 5.53. Escenario del elemento del agua.

Este escenario ha sido construido a partir de una escena *Prefab* llamado *BasicLevel* que contiene las entidades básicas configurables, para construir un nivel, como la cámara, las posiciones de inicio de los personajes o las posiciones donde aparecen los objetos, los cuales se van a comentar a continuación de la figura 5.54.

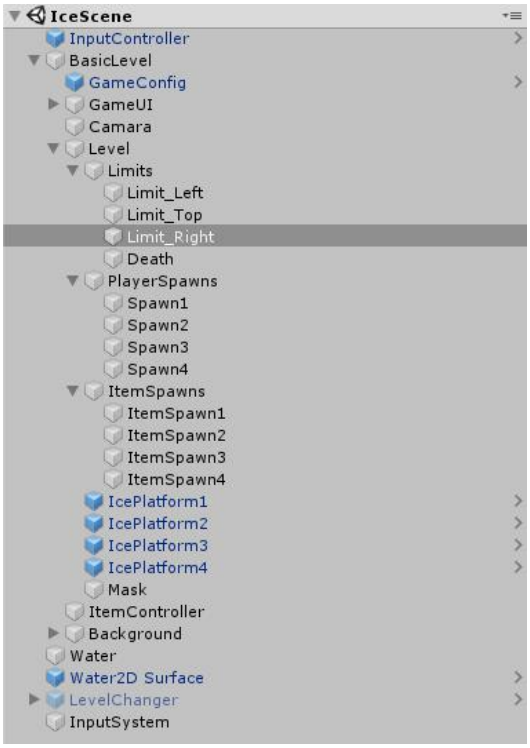

Figura 5.54. Entidades del escenario del agua y de un BasicLevel.

Un *BasicLevel* contiene un *GameObject GameConfig*, que es la entidad que contiene los *scripts* de *ItemConfig*, donde se almacenan las posiciones de salida de los objetos, y *CharacterConfig*, el cuál contiene una lista con las posiciones de salida de los personajes, de tal manera que cuando se inicia la partida a través del *script GameStart* (véase apartado 5.3.2.3 Inicio de una partida) llama al *CharacterConfig* para recoger la posición correspondiente de cada personaje.

Los elementos pertenecientes a la entidad *PlayerSpawns* son *GameObects* vacíos, importando únicamente la posición donde se sitúan, ya que sera los lugares donde se instancian los personajes. Al igual que los *PlayerSpawns*, los *ItemSpawns* sirven para indicar la posición donde se van a instanciar los objetos.

Todos los escenarios tienen una entidad *Background*, el cuál sirve para añadir el fondo de pantalla del escenario, siendo estos de 320 x 180 píxeles. También tienen la entidad *ItemController*, que se encarga de instanciar los objetos (véase apartado 5.3.2.6 Controlador de objetos).

La entidad *Limits* controla los límites del escenario los cuáles no pueden ser traspasados por ninguna entidad de *Cross Element*, cómo los personajes, las habilidades o la bola de energía. La entidad *Death* es en realidad un límite, pero tiene un componente *script DeathTouch*, el cuál detecta si un personaje colisiona con esta, lo destruye, al igual que ciertas habilidades y la bola de energía.

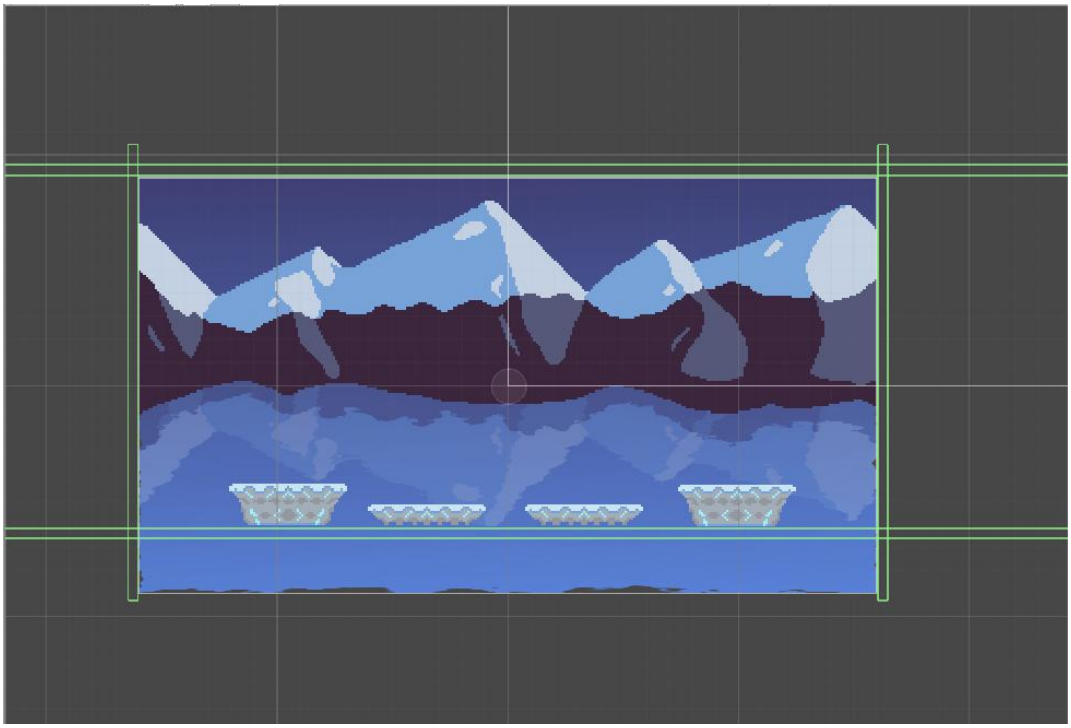

Figura 5.55. *Límites y zona de muerte (rectángulo inferior) del escenario de agua.*

Por último, las entidades *IcePlatform* son las plataformas que se observan en la figura 5.55, que son propias de este escenario, además del *Water* y *Water2DSurface*, que se encargan de reflejar estas plataformas, simulando ser agua.

6. Conclusiones

Al finalizar el proyecto he llegado a la conclusión de que para acceder al mundo del desarrollo de videojuegos, es importante involucrarse en el desarrollo de varios proyectos y tener un buen porfolio que refleje la experiencia obtenida. He obtenido un mayor conocimiento sobre el estado del mercado de los videojuegos, y realmente creo que un título indie es una idea perfectamente válida para hacerse un nombre en este mercado, si bien no es un camino fácil, te aporta nuevas perspectivas sobre el desarrollo de videojuegos y te enseña a valorar el duro trabajo que hay detrás de un videojuego, valiosas experiencias que no podría haber aprendido de otra manera.

Con el desarrollo de *Cross Element,* he desarrollado nuevas habilidades que serán de utilidad en próximos proyectos. He aprendido mucho sobre *pixel art* y aunque me quede mucho trabajo por delante para obtener una técnica pulida, es un gran paso. Respecto a las metodologías aplicadas, aun tengo trabajo pendiente, pues la estimación del tiempo de desarrollo no ha sido tan precisa como hubiera querido. Sin embargo, he mejorado mucho en el análisis de requisitos, y estoy muy satisfecho con el código implementado, además de haber aprendido muchísimo sobre Unity, herramienta que volveré a utilizar en futuros proyectos. *Cross Element* ha sido una gran experiencia, y tengo pensado seguir su desarrollo y poder publicar el videojuego próximamente.

Por último, creo haber aprendido la lección sobre que tipo de proyectos se pueden afrontar en un espacio de tiempo. Sin duda el concepto del proyecto es demasiado grande y algunos elementos han tenido que ser descartados. Inicialmente se pensó incluir cuatro escenarios jugables, y entre animaciones y personajes no ha habido tiempo para finalizarlos todos. La clave para desarrollar un videojuego es tener en cuenta el tiempo de desarrollo, y es una tarea

increíblemente difícil de estimar. A continuación, expreso una reflexión que he aprendido tarde durante *Sobreviviendo como desarrollador indie en Unity*.

6.1 Un juego pequeño es un juego terminado

Hacer un juego pequeño es la clave, es muy importante estudiar los requerimientos de un proyecto y sobre todo ser realistas con el tiempo que llevará su desarrollo. En el mundo indie, cuanto más pequeño mejor, y quiere decir que si un videojuego AAA del género RPG puede llegar a durar 60 horas de juego, entonces un RPG pequeño podrían ser tan solo 10 horas. Esto es un error común, un juego pequeño se refiere a que no tenga una duración de más de una hora, es más, que dure 5 minutos, que esté bien acabado y sea divertido de jugar.

La mayoría de las personas quieren empezar por desarrollar el videojuego de sus sueños, es entendible pero también es la fórmula del fracaso. Ocurrirá que a la hora de desarrollar una tarea estimada para una semana, se acabará convirtiendo en dos semanas y así sucesivamente, y aun peor si no se tiene experiencia en el desarrollo de videojuegos o no se conocen bien las herramientas que se van a utilizar.

Si se consigue terminar el videojuego de dicho calibre, la probabilidad de que sea bueno es escasa, simplemente porque hacer algo bien a la primera es muy difícil. Incluso si acaba siendo un buen videojuego, eso tampoco asegura el éxito. Existe la idea que si un juego es bueno, acabará triunfando, bien por las recomendaciones de sus jugadores o bien el tiempo acabará haciendo justicia y el este acabará vendiendo lo suficiente, pero no es verdad.

El mercado esta saturado y lleno de buenos videojuegos con buen arte que no han vendido nada, incluso ganando premios como Nohamund, un JRPG creado por el estudio español

Ábrego, un videojuego ambicioso con traducciones al español, ingles y japones, con buena banda sonora y un exitoso *Kickstarter*[16], desarrollado durante varios años. A *priori* da la sensación que cumple todas las reglas del éxito en el mundo de los videojuegos, pero la realidad es cruel, y al final del videojuego ha tenido muy poca repercusión.

Figura 6.1. *Reseñas de Noahmund en Steam*[17].

[16] Kickstarter es una plataforma para publicar proyectos en desarrollo donde los usuarios pueden volverse patrocinadores apoyando económicamente al creador.

[17] Fuente: https://store.steampowered.com/app/752560/Noahmund.

Una de las ventajas de un juego pequeño es que es más fácil de mejorar y pulir, lo cual aumenta las posibilidades de que la gente lo juegue, y aunque nadie lo juegue al menos no se habrán invertido varios años. La gente estará dispuesta a darle una oportunidad si sabe que es una experiencia corta, que si tiene que invertir varias horas. Además es más fácil que terminen el videojuego, y así dejar una buena experiencia que los pueda convertir en *fans* del desarrollador. Por último, será útil para calcular mejor el tamaño de próximos proyectos.

7. Bibliografía

- James Batchelor, *Global games market value rising to $134.9bn in 2018*, 2018. [En línea]. Recuperado el 3 de Mayo de 2020: https://www.gamesindustry.biz/articles/2018-12-18-global-games-market-value-rose-to-usd134-9bn-in-2018

- James Batchelor y Newzoo, Gameindustry.biz presents... The Year In Numbers 2018, 2018. [En línea]. Recuperado el 25 de Julio de 2020: https://www.gamesindustry.biz/articles/2018-12-17-gamesindustry-biz-presents-the-year-in-numbers-2018

- Ben Kuchera, *Diablo: Immortal broke the unspoken rules of Blizzard, and BlizzCon*, 2018. [En línea]. Recuperado el 13 de Mayo de 2020: https://www.polygon.com/2018/11/5/18064290/blizzard-diablo-immortal-reaction-explainer-blizzcon

- Mike Prinke, *How hard is it to get a job in game design at a AAA studio?,* 2016. [En línea]. Recuperado el 13 de Mayo de 2020: https://www.quora.com/How-hard-is-it-to-get-a-job-in-game-design-at-a-AAA-studio/answer/Mike-Prinke

- Steven T. Wright, *There are too many video games. What now?*, 2018. [En línea]. Recuperado el 13 de Mayo de 2020: https://www.polygon.com/2018/9/28/17911372/there-are-too-many-video-games-what-now-indiepocalypse

- Willy-Peter Schaub, *How does kanban relate to DevOps?*, 2020. [En línea]. Recuperado el 13 de Mayo de 2020 en: https://opensource.com/article/20/4/kanban-devops

- Trello, *What is Trello?,* 2017. [En línea]. Recuperado el 13 de Mayo de 2020: https://help.trello.com/article/708-what-is-trello

- Francesco Cirillo, *The Pomodoro Technique for better productivity*, 2012. [En línea]. Recuperado el 13 de Mayo de 2020: https://blog.trello.com/the-pomodoro-technique-for-better-productivity

- GameDesigning, *The top 10 videogame engines*, 2020. [En línea]. Recuperado el 13 de Julio de 2020: https://www.gamedesigning.org/career/video-game-engines/

- Mark Schumacher, GORINTŌ / GORINTO / GORINTOU / 五輪等, fecha de publicación desconocida. [En línea]. Recuperado el 27 de Julio de 2020: https://www.onmarkproductions.com/html/5-elements-pagoda-gravestone.html

- Desconocido, *Singleton pattern*, 2013. [En línea]. Recuperado el 13 de Julio de 2020: http://design-patterns-with-uml.blogspot.com/2013/02/singleton-pattern.html