
# Trabajo Fin de Grado

Ingeniería Informática

# Demo técnica de videojuego 2.5D en Unity

Post Mortem y Memoria Técnica del desarrollo, en colaboración con Gambusino Labs


Autor

Miguel Medina Ballesteros

Director

Marcelino Cabrera Cuevas


Granada, Septiembre de 2018

# Demo técnica de videojuego 2.5D en Unity

Post Mortem y Memoria Técnica del desarrollo, en colaboración con Gambusino Labs.

Miguel Medina Ballesteros (alumno)

Marcelino Cabrera Cuevas (director)

**Resumen**

Este TFG describe el desarrollo de una demo técnica de un videojuego de plataformas en 2.5D en Unity, exponiendo las etapas tales como el documento de diseño de juego, gestión y planificación temporal del proyecto, análisis de requisitos y diseño software y arquitectónico del producto. El código y ejecutables de la demo producida también constan como parte del TFG. Además, también expone una investigación y discusión subyacente sobre la aplicación de la Ingeniería del Software al Desarrollo de Videojuegos.

**Keywords**: Unity, Game Design, SCRUM, Software Engineering, Game Development, Software Architecture, Software Design, Design Patterns, Game Engineering.

# Introducción

Desde que empecé la carrera hace cinco años he aprendido muchísimo y he profundizado en muchísimas áreas del conocimiento, sobre todo en el ámbito de la Ingeniería Informática, tanto práctica como teóricamente, y aún así he sabido reconocer que esta es tan solo la punta del iceberg. Desde pequeño me han gustado los videojuegos, embarcarme en sus mundos y dejarme llevar por sus historias, pero llegó un momento en mi vida, alrededor de bachillerato, en que ya no tenía tiempo para dedicarle a ese hobby. Cuando entré en la carrera no lo hice porque quería hacer videojuegos. Amaba y amo todo lo demás: la complejidad y grandiosidad de la historia de la informática, el buen gusto necesario para hacer páginas webs, sistemas operativos y demás sistemas complejos que acabarán siendo usados por usuarios muy diversos, la mente de estratega que hay que tener para enfrentarse a tales desarrollos, tanto a la hora de escribir código como a la de organizar y analizar el trabajo… Pero en algún momento del camino, en el tercer año, con las asignaturas de Sistemas Gráficos e Informática Gráfica, me enamoré.

Desde entonces no he parado de ser autodidacta y de esforzarme cada día para aplicar todo lo que he aprendido de la ETSIIT en este campo. Este trabajo es el culmen del aprendizaje de todos estos años y el principio de lo que está por venir.

El proyecto que presento es el desarrollo de la demo técnica de un videojuego con mecánicas en 2.5D, programado en C# con el motor gráfico Unity. Hago hincapié en el término “demo técnica” porque el ejecutable que se entrega son tan solo unos minutos de juego en que se ilustran las mecánicas centrales en torno a las que girará el videojuego. Si bien este proyecto de demo está finalizado, la creación del videojuego en sí no ha hecho más que comenzar.

En los siguientes apartados describiré y analizaré los puntos nucleares del desarrollo de esta demo. Iré escalando desde los apartados menos técnicos: game design document, gestión del proyecto y coordinación con los artistas 3D, ingeniería de requisitos; hasta los más técnicos: arquitectura del software, sistemas programados y patrones de diseño utilizados.

## Objetivos del proyecto

La demo técnica que nos ocupa es parte de un proyecto más grande y ambicioso: Both, que se trata del videojuego completo. Es importante entender esta diferencia antes de continuar con la lectura, pues los apartados que siguen referencian a veces a una parte del trabajo, y a veces a otra.

Como se describe en el título, este proyecto ha sido desarrollado en colaboración con el equipo de Gambusino Labs, compuesto por tres artistas 3D que han dado forma a ideas, concepts, escenarios y personajes desde hace varios años. Yo soy la incorporación más reciente.

Para entender bien los objetivos del proyecto, se deben distinguir, por una parte los objetivos perseguidos con el desarrollo de la demo técnica que nos ocupa; y por otra, los objetivos del proyecto que la engloba, que es Both.

### Objetivos del proyecto que engloba esta demo

En el momento de la escritura, ya nadie discute el asombroso crecimiento que ha sufrido la industria de los videojuegos en los últimos años, superando incluso a la del cine. España ha tardado un poco más que otras industrias en batir este récord, pero en 2017 se hizo oficial: la industria del videojuego en España supera en facturación a la del cine y la de la música y se sitúa como primera opción de ocio audiovisual. [1]

El equipo de Gambusino Labs aprovecha este contexto para profesionalizarse en un campo que hasta ahora había sido solo un hobby. Con una pasión y curiosidad infinitas por crear y aprender, hace un año nos marcamos como objetivo trabajar constantemente en este y otros proyectos hasta abrirnos un hueco en la industria del videojuego.

Al principio, los objetivos eran crecer y mejorar nuestras habilidades como equipo, generando prototipos pequeños y jugables del videojuego mientras se iban desarrollando también los apartados artísticos, y seguir esforzándonos hasta alcanzar un ritmo de desarrollo estable. Consideramos ese objetivo cumplido hace algunos meses, y por esa razón surgió la necesidad del desarrollo de la presente demo técnica.

Actualmente, el objetivo de Gambusino Labs es terminar de perfeccionar esta demo durante el mes de Septiembre, y vestirla del arte que ha generado el equipo artístico en paralelo al desarrollo de la demo. A finales de septiembre, Gambusino Labs asistirá al evento “Level Up! Granada”, donde presentará al público el resultado de estos meses de trabajo.

Los objetivos a futuro son seguir asistiendo a eventos para darnos a conocer, generar una base de seguidores en las redes para diseñar una campaña de Crowdfunding. Paralelamente también se buscarán otras vías de financiación, como publishers o mediante medios propios; con el fin último de continuar con el desarrollo del videojuego durante los próximos años hasta publicarlo.

## Objetivos del TFG y de la presente demo técnica

Si bien los objetivos antes descritos no me eran exclusivos, pues los compartía con mis compañeros de Gambusino Labs; el desarrollo de esta demo técnica, así como su presente documentación, y por tanto sus objetivos sí que me son exclusivos a mí. Es importante diferenciar ambas competencias, pues mientras que yo no tengo la autoría de los recursos artísticos creados por ellos; todo lo que contiene esta memoria, así como el código y el proyecto de Unity son exclusivamente de mi autoría. Aclarado esto:

El objetivo inicial del proyecto era la construcción de esta demo técnica como la base sobre la que después se desarrollaría junto a Gambusino Labs la demo abierta al público que se mostrará al público en “Level Up! Granada”. Sin embargo, para potenciar mi aprendizaje como Ingeniero Software Desarrollador de Videojuegos, además de la productividad a largo plazo, desde el principio del desarrollo he planeado enfocar el trabajo en distintos aspectos teóricos relacionados con el Desarrollo de Videojuegos y la Ingeniería Software, que son los apartados que componen la estructura de esta memoria: diseño, planificación, análisis de requisitos, arquitectura y diseño del software.

Por tanto, el objetivo de este TFG queda definido no solo como la consecución de la demo jugable final, sino principalmente por la investigación teórica que expone sobre cómo adaptar lo aprendido en el grado de Ingeniería del Software al Desarrollo de Videojuegos

de la mejor manera posible, potenciando la productividad y la mantenibilidad del desarrollo en el tiempo; y por tanto potenciando también, en última instancia, la creatividad; que se vería coartada si fallaran el resto de pilares.

# Game Design: Both

## *Pitch*, o descripción breve

Both es un videojuego mainstream que apunta hacia donde otros títulos indies han triunfado ya. Con mecánicas sencillas en dos dimensiones pero con la belleza de un mundo tridimensional, dos personajes no antropomorfos, con muchos ojos y tentáculos, ven como su planeta es invadido por un alienígena gelatinoide y un instinto innato los lleva a enfrentarse a él para salvar su planeta. Con un ritmo tranquilo y un diseño de puzles basados en la lógica, el jugador viajará por los distintos biomas de ese extraño planeta, y en su camino descubrirá los vestigios de una antigua civilización que dejó las pistas necesarias para expulsar a la gelatina. [2]

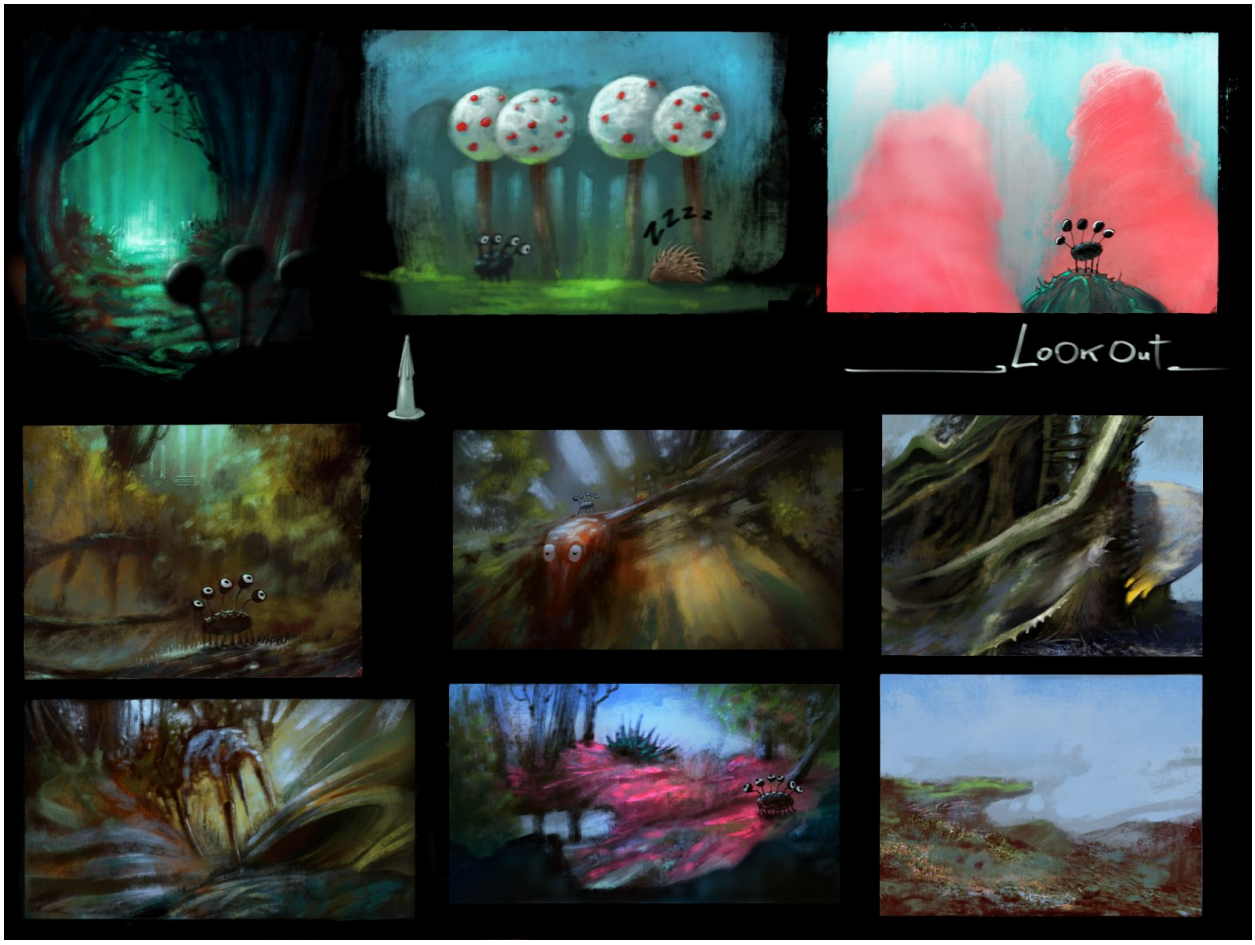


Figura 1. Concept art de Both.

## *Referencias*

Las referencias de Both son indies de éxito como Inside, Unravel o Brothers: A Tale of Two Sons, a pesar de que el proyecto comenzó a fraguarse antes del lanzamiento de estos, irónicamente. Como se demuestra en los siguientes apartados, Both implementa mecánicas que han triunfado en títulos anteriores, lo que aumenta su potencial de ganar adeptos en el mercado. Aunque pueda parecer poco importante, o incluso un punto negativo, no es así en absoluto, ya que para atraer publishers e inversores Gambusino Labs tiene que esgrimir eichas similitudes con casos de éxito para probarles que vale la pena confiar en nuestra idea.

## Inside

Desarrollado por Playdead, estudio indie danés y creador del videojuego Limbo, este es su segundo juego lanzado al mercado. Es un plataformas de puzles en 2.5D en el que manejamos a un niño en un mundo post apocalíptico en el que parece que ya no queda libertad. [3]

Figura 2. Inside.

Both toma como influencias de este título su ritmo pausado y su diseño de narrativa, contada mediante escenarios y sin mediar palabra con el jugador. Pero, sobre todo, toma de él un elemento narrativo muy interesante: el de hacer pensar al jugador durante toda la historia que tiene el control y que va a salvar al mundo, para después revelar que no todo es lo que parece, y que tan solo somos un engranaje más en la cadena.

## Unravel

Sus desarrolladores,  Coldwood Interactive, son de Suecia. A diferencia de Limbo, el ritmo de Unravel es más acelerado, y aunque también tiene puzles que resolver, no son su principal baza. [4]

Figura 3. Unravel.

En lo que más se inspira Both en Unravel es en su apartado artístico, con escenarios muy detallados y un efecto constante de profundidad de campo que te hace parecer pequeño, y dando así un aire épico a la aventura. Pero de Unravel también llaman la atención las mecánicas de su simpático personaje, que puede utilizar el hilo que arrastra para resolver puzles y moverse por el escenario. En este caso, con Both también se quiere conseguir transmitir el mismo encanto con los personajes de Din y Vlinky, ya que ambos cuentan también con mecánicas muy originales.

## Brothers: a Tale of Two Sons

También de Suecia, el equipo de Starbreeze nos sorprendió en 2013 con esta interesante propuesta en la que dos hermanos parten en un viaje para salvar a su padre en un mundo de fantasía. [5]

Figura 4. Brothers: a Tale of Two Sons

El parecido de Both con Brothers es más obvio: radica en la colaboración entre el dúo protagonista, y en que ningún personaje podría superar la aventura sin las habilidades del otro, lo que obliga al jugador a pensar constantemente en ellos como un equipo.

# Visión general

## Sentimiento buscado en el jugador

El jugador es un observador en tercera persona de la historia del mundo, que se cuenta a través del viaje de los protagonistas y los escenarios que recorren. No habrá líneas de diálogo ni textos inteligibles, y todo se contará mediante la experiencia jugable, al igual que hacen videojuegos AAA recientes como Dark Souls, Bloodborne; o, en el caso de los indies, el propio Inside. [6]

Superficialmente, se busca enganchar al jugador con las mecánicas de los puzles, siendo del grado justo de desafío como para que supongan un reto y despierten un sentimiento de satisfacción al completarlos.

Conforme avance la historia, no será suficiente con el interés del jugador por conocer el desenlace, pues todo el mundo sabe cómo acaban las misiones salvaplanetarias. Se irá más allá, dando pistas al jugador de que hay algo que desconoce, de que no se trata simplemente de una invasión alienígena.

El final dejará muchas dudas en el aire, pero dará la información clave para hacer al jugador consciente de todas las pistas que se ha ido perdiendo desde el principio, incitándole así a rejugar el juego para desentrañar el misterio.

Además, los escenarios tendrán recovecos por los que no será obligatorio pasar para llegar al final. En ellos se esconderán coleccionables y pistas sobre la historia del mundo. De esta forma se consigue premiar la curiosidad y se promueve la rejugabilidad.

Finalmente, también se persigue el ingrediente viral de las teorías, ofreciendo la información justa para que el jugador ate los hilos con ayuda de la comunidad, en las redes y en los foros. [7]

## Plataformas objetivo

PC primeramente, por la facilidad de publicar en steam y de evitar el trabajo de portabilidad. Una vez conseguido ese objetivo se perseguirá la publicación en las plataformas de sobremesa de Xbox, PlayStation y Nintendo. [8]

## Características multijugador

No existe intención de implementar multijugador en línea.

En cuanto al multijugador local, si bien se ha considerado, se ha optado por no desarrollarlo para esta demo. Permitir o no dos jugadores, uno controlando a cada jugador, dota al juego de un componente social pero no se adapta al diseño de puzles del

juego completo, ya que existirán puzles en los que un personaje tenga que estar esperando mientras el otro desbloquea su camino, por ejemplo.

En el videojuego completo, Gambusino Labs se ha reservado la posibilidad de diseñar fases específicas para multijugador en el futuro, en las que ampliar la historia, y que sí estarían diseñadas para ser jugadas en multijugador local. Por esa razón, el software que se exponse en los próximos apartados ha sido diseñado para ser mantenible y extensible y así poder implementar con facilidad la inclusión de un segundo jugador. Para implementar el multijugador local hay una técnica que me interesa mucho, llamada “voronoi splitscreen” (pantalla dividida en voronoi), que divide la pantalla con una línea perpendicular a ambos personajes y que se une en una sola pantalla cuando se acercan. Algunos videojuegos de la saga de Lego han utilizado dicho sistema. [9, 10]


Figura 5. Lego Voronoi Splitscreen

## Motor gráfico

El motor gráfico en que se ha decidido implementar la demo es Unity, actualmente en su versión 2018.2.6. La decisión no atiende a una comparativa exhaustiva de los motores disponibles, simplemente a la circunstancia de que los primeros prototipos se comenzaron realizando en dicho motor, y que los miembros del equipo tienen en general más soltura en él que en el resto. Como Ingeniero, soy consciente de sus limitaciones, y de la facilidad de Unreal Engine 4 para superarlo en gráficos, pero en los últimos meses he juzgado una mejoría en Unity en ese sentido, por lo que estoy seguro de que se podrán superar esas barreras para conseguir en Unity unos gráficos tan cuidados como los que ellos mismos publican en sus demos. [11]

## Monetización

1. El método para obtener beneficios será el más tradicional: la venta de la copia del juego y la publicidad aprovechando descuentos de la plataforma Steam. [12]
2. Merchandising: se venderán camisetas y figuras del videojuego en la web, además de versiones especiales.

## Resumen del desarrollo y plan futuro

1. **Pasado incierto - 2018.** Preproducción: el equipo original de Gambusino Labs ha estado fraguando las ideas del proyecto desde hace varios años.
2. **Junio - Septiembre de 2018.** Desarrollo de una demo técnica “Test Room”: presente proyecto.
3. **Septiembre - Octubre de 2018.** Continuación del desarrollo de la demo técnica hasta convertirlo en una demo para el público general que **no** será publicada. Dicha demo se mostrará en las redes, en ferias y eventos, siendo “Level Up! Granada” el primer evento al que se llevará, el 29 y 30 de Septiembre de 2018.
4. **Octubre 2018 - Primavera 2019.** Continuar el desarrollo con nuevas milestones y ferias, así como intensificar la presencia en redes sociales para generar una base de seguidores.

5. **Primavera 2019 - Futuro.** Buscar financiación con el contenido generado en los meses pasados para poder terminar con el desarrollo. Ver apartado siguiente, “Financiación”.

## Financiación

Tras perfeccionar la jugabilidad y estética de la demo protagonista de este TFG durante los próximos meses, y haber generado una base de usuarios en las redes demostrable y activa, se perseguirá la financiación del resto del desarrollo mediante cuatro vías principales:

- **Crowdfunding.** Ya entrado 2019, y contando previamente con una base de seguidores en las redes, se estudiarán casos de éxito y se generará un plan, unas fechas, y un diseño de campaña lo atractivos de cara a los mecenas. [13]
- **Publishers e inversores.** Gambusino Labs asistirá a eventos y moverá la demo por diferentes ferias usando los fondos personales de los miembros del equipo. En dichos eventos existe la posibilidad de contactar con publishers y potenciales inversores, así que se irá preparado para ello.

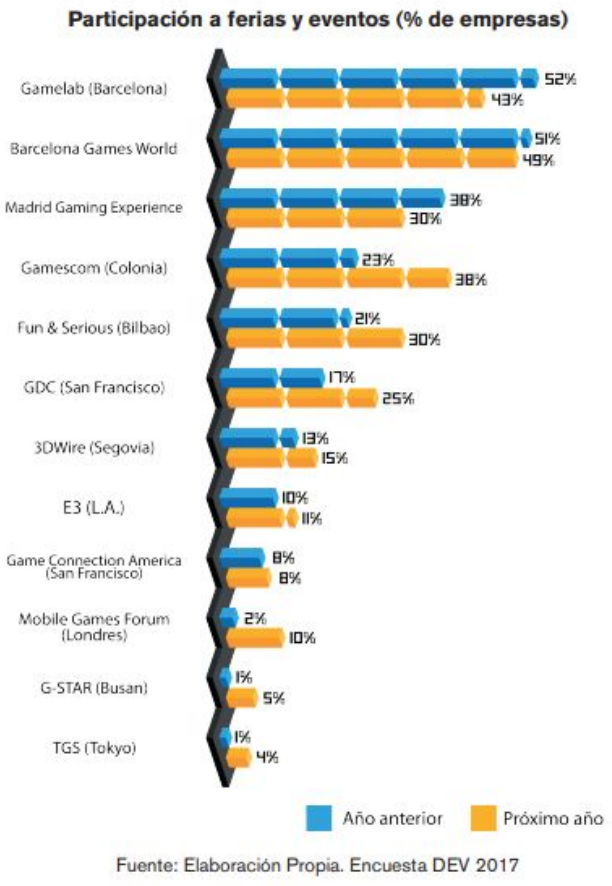


Figura 6. Participación de empresas en ferias y eventos.

- **Autofinanciación.** Dado que Gambusino Labs tiene un perfil que casa muy bien con aplicaciones punteras como realidad aumentada y virtual, ya que consta de dos ingenieros informáticos y tres animadores 3D en el equipo, paralelamente se están desarrollando otros proyectos de ese ámbito. Con el beneficio de dichos proyectos será financiado el presente desarrollo y la asistencia a eventos y ferias. [12]
- **Subvención del "Programa de Impulso al Sector del Videojuego" de Red.es.** En 2018 se ha abierto la primera convocatoria a dicho programa, dirigido a estudios independientes en la modalidad de micropymes para el desarrollo y la comercialización de videojuegos, perfil con el que Gambusino Labs coincide. El principal obstáculo es que es necesario estar constituido como empresa seis meses

antes de la convocatoria, que es en Abril. Además, no es válida cualquier modalidad de empresa: las empresas constituidas como sociedad civil no optan a esta ayuda, siendo esta la modalidad más barata y fácil de realizar. [15 ,16]


Figura 7. Fuentes de financiación de estudios.

## Elementos de juego

- **Vlinky** (personaje).
- **Din** (personaje).
- Din en forma de globo (de ahora en adelante, **Din Globo**).
- **Cubos**, que pueden ser empujados por Vlinky o cogidos por Din Globo.
- **Botones grandes**, que se accionan al pasar sobre ellos a personajes u objetos.
- **Botones pequeños**, que se accionan al dispararles un ojo.
- **Puertas**, que pueden ser abiertas por botones.
- **Cosquilleadores**. Autómata básico que anda por la pared como Din y que si lo toca le obliga a descolgarse.
- **Géiseres de gas**, con los que Din podrá inflarse para transformarse en globo.

- **Gelatinas**, representadas por geometría simple (como cubos) y con un color llamativo rojo. Te matan al tocarlas.


Figura 8. Personajes de Both.

## Sistema de juego: acciones, reglas y mecánicas

*Un grupo de reglas es llamado sistema, y la interacción de dos o más reglas se denomina mecánica. Las reglas limitan la acción del jugador.* [17]

Sin embargo, cabe destacar que el nivel de detallismo que se puede alcanzar en este apartado es muchísimo, y que yo no profundizaré demasiado: expondré las mecánicas a nivel superficial pero no indagaré en las reglas formales que se deberán cumplin a escala de frames. Es decir, explicaré los componentes que compondrán los puzles, pero no desgranaré, por ejemplo, las leyes físicas a las que se someterán el movimiento de los personajes y objetos, o todos los estados posibles de cada elemento. [18]

### Acciones

En común con todos los personajes: se debe poder cambiar a otro personaje.

### Vlinky

- Andar por el suelo.
- Empujar objetos.
- Disparar ojos.
- Coger a Din.
- Disparar a Din.

### Din

- Andar por suelo y paredes, pegando sus ventosas.
- Pegarse a una superficie superior, si está cerca.
- Descolgarse.
- Rodar en caída libre.
- Subirse sobre Vlinky.
- Inflarse para convertirse en globo si está sobre un géiser de gas.

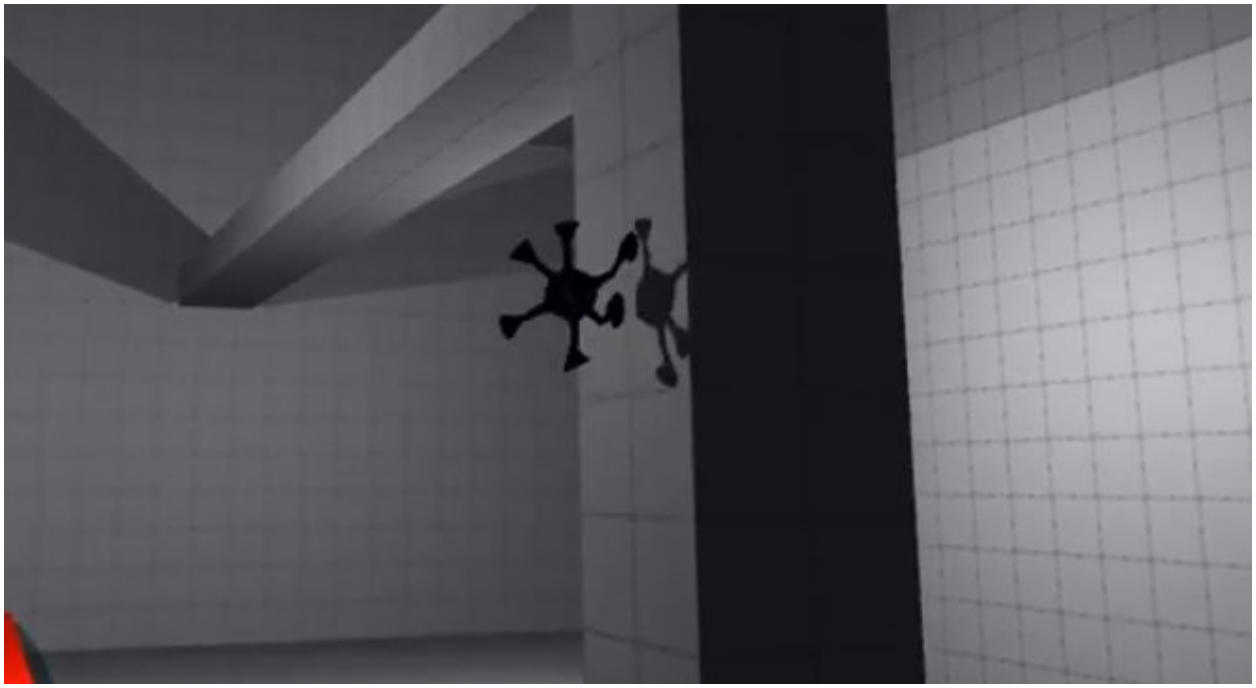

Figura 9. Habilidad de Din de pegarse a las paredes.

### Din Globo

- Flotar libremente por el escenario.
- Coger objetos.
- Coger a Vlinky.
- Desinflarse.

## Reglas

### Vlinky

- No puede disparar ni empujar en el aire.
- No puede saltar si está empujando un objeto, disparando o cogiendo a Din.
- No puede disparar, saltar, empujar ni moverse si está siendo transportado por Din Globo. Por tanto, no puede tomar el control.
- Si lanza a Din, el jugador pasa a controlar a este último.

### Din

- No puede transformarse en globo si no está sobre un géiser de aire.
- Si se sube sobre Din, el jugador pasa a controlar a este último.

### Din Globo

- Solo puede coger un objeto al mismo tiempo.
- Mientras Din Globo esté transportando un objeto, su movimiento se verá limitado por el peso del objeto.
- El estado de Din Globo se gastará con el movimiento, ya que se desinfla, obligándole a repostar de nuevo en otro géiser.

**Regla común a ambos**: si cualquiera de ellos toca la gelatina, muere y es teletransportado al último punto de control.

## Mecánicas emergentes

- Las puertas se abrirán al pulsar el botón asociado a ellas.

- Los botones grandes se despulsarán si dejan de tener peso sobre ellos, y no solo podrán ser accionados por los personajes sino también por el peso de un cubo, por lo cual dará lugar a puzles en los que el jugador tenga que buscar un objeto para mantener abierta una puerta, o elegir qué personaje se quedará esperando sobre el botón para mantener la puerta abierta y superar el desafío concreto que se esconde tras de ella.

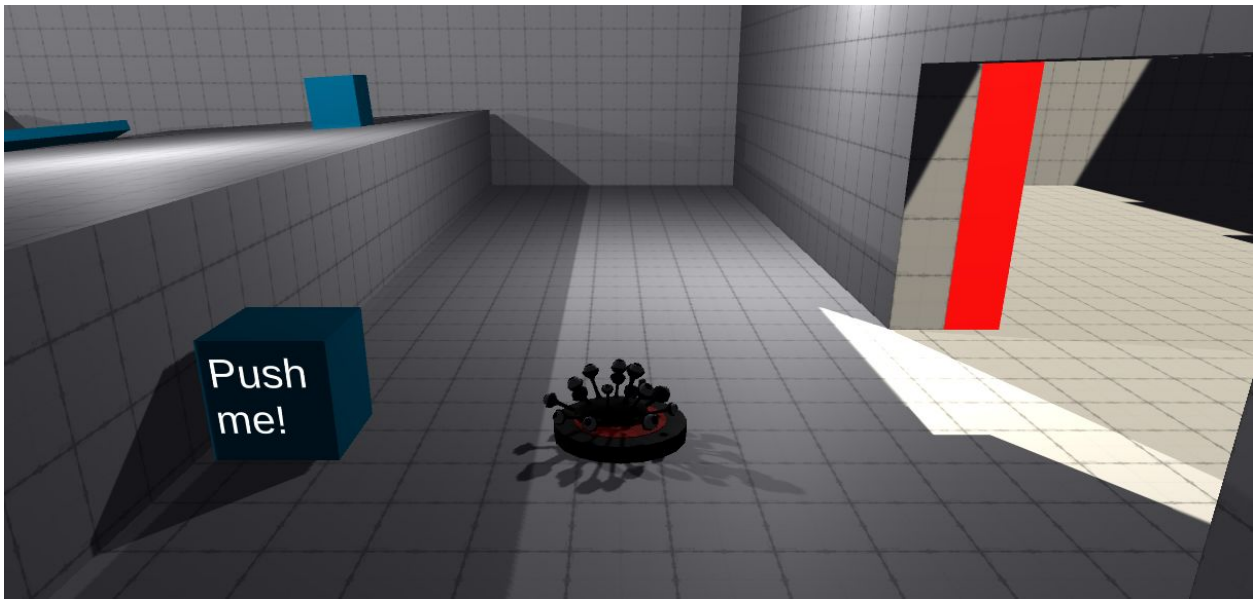


Figura 10. Mecánicas de botones y puertas.

- Las cubos pueden estar en lugares a priori inaccesibles, por ejemplo colgando de un poste en el techo, y podrán ser tirados al suelo lanzándoles un ojo.

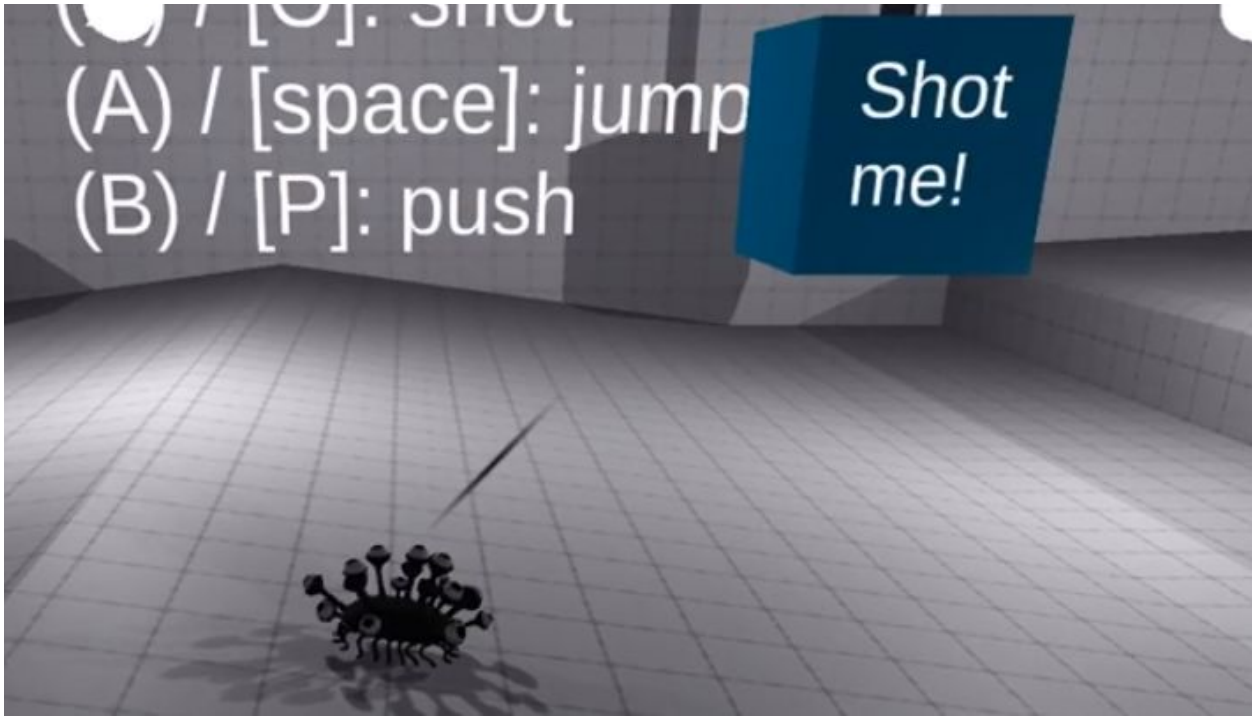


Figura 11. Habilidad de disparo

- Asimismo, podrán existir otros objetos en el escenario que reaccionarán a las físicas del impacto del ojo de Vlinky. Por ejemplo, Vlinky puede disparar a un tablón inalcanzable para que caiga y usarlo de pasarela, o también puede, simplemente, disparar a Din para molestarle.
- En ocasiones Vlinky lo tendrá imposible para avanzar porque la distancia vertical será demasiada para él. En esos casos Din podrá transformarse en globo para actuar como ascensor.
- Din Globo también podrá actuar como medio de transporte para ir más rápidamente de un punto a otro en escenarios muy amplios.
- Puede darse el caso de un precipicio en el camino que ninguno de los dos personajes puede sortear. Entonces Vlinky podrá coger a Din y lanzarlo para que llegue al otro lado y ayude a cruzar a su compañero.

Estas son las mecánicas que emergen a un nivel básico con los primeros componentes del juego, y son las que se ilustran en la demo técnica. Pero no se quedan solo ahí.

Combinándolas entre ellas dan lugar a sistemas más complejos, y Gambusino Labs continuará el desarrollo del videojuego en base a esos sistemas. Por ejemplo, combinando esas mecánicas se pueden crear puzles más complejos que satisfagan más al jugador al completarlos.

### Otros factores

Antes de cerrar el apartado de Game Design Document, cabe mencionar que, en cuanto a diseño general del videojuego, esta es solo la punta del iceberg. Se ha hablado principalmente la jugabilidad, y se ha obviado la estética, el sonido y la música, los escenarios, la narrativa, el control, aspectos como el márketing o la internacionalización. Son apartados que alargarían demasiado el TFG y que se alejan de las materias de la Ingeniería Informática. [19]

Aún así, también se han dejado por el camino aspectos relacionados con la jugabilidad que, si bien están más relacionados con el ámbito de la programación del videojuego, no tienen cabida para una demo técnica con una duración de dos minutos. Estos aspectos son:

- Condiciones de victoria y de derrota.
- Diseño de la curva de dificultad.
- Variables que determinan el estado del juego.
- Así como la superficialidad de las reglas antes descritas, etc.

## Dirección y gestión del proyecto

### Manual de coordinación

La siguiente tabla fue elaborada al comienzo del desarrollo del videojuego, antes de que comenzara a desarrollar la demo que nos ocupa. Concretamente en el contexto de estar yo estudiando la asignatura de Dirección y Gestión de Proyectos. De hecho, esta plantilla ha sido directamente tomada de esa asignatura. En aquel momento consideré útil aplicar lo aprendido en dicha asignatura a la coordinación con el equipo artístico. [20]

La organización y planificación del equipo de desarrollo es una parte crucial en el Desarrollo de Videojuegos, así como en el de cualquier software; y además dicha gestión forma parte importante de la Ingeniería del Software también. Durante el desarrollo de los primeros prototipos, este documento fue útil para dejar por escrito las metodologías por las cuales el equipo artístico se coordinaría conmigo. Por esas razones he decidido incluir en la memoria el manual de coordinación que escribí a tal efecto.

Cabe mencionar que el documento ha sido adaptado al contexto de la demo técnica para que se entienda bien en el contexto del TFG. Mientras que se han excluido secciones que no proceden, relativas al desarrollo del videojuego completo o al de Gambusino Labs como empresa, y aunque el desarrollo de la demo y la memoria dependen exclusivamente de mí, se han conservado las partes relacionadas con la coordinación y organización del equipo de trabajo fuera del contexto de la demo, ya que la información que aporta es relevante para la investigación y perfectamente aplicable al de un desarrollo software cualquiera.

| **GAMBUSINO LABS: miembros** |
| --- |
| ● Miguel Medina Ballesteros, autor del TFG.<br>● Fernando Montero Valdivieso<br>● Daniel Montero Valdivieso<br>● David Sánchez de la Torre<br>● Javier Izquierdo Vera, Ingeniero Informático y compañero de la ETSIIT. No ha participado en el desarrollo de este TFG, ni tampoco en el del videojuego completo todavía. |
| **MÉTODOS DE TRABAJO** |
| ● Miguel Medina Ballesteros, programará las mecánicas definidas para la demo. No necesito nada de los artistas, puesto que los modelos 3D son solo complementarios en este TFG y no son evaluados. Si al final da tiempo a implementar modelos y animaciones, se hará; si no, se presentará un prototipo en que los personajes estarán representados simplemente por cápsulas y esferas.<br>● El equipo artístico, trabajará paralelamente fuera del ámbito del TFG, en tareas relacionadas con el desarrollo del videojuego completo, modelando los escenarios y personajes, así como sus animaciones, e integrándolas en Unity a la espera de que yo decida si tengo tiempo a implementarlas en la demo o no.<br>● Javier, compañero de la ETSIIT, no trabaja en Both porque ha estado ocupado con |

su propio TFG, así como otros proyectos. Está planeado que se incorpore tras la entrega de este TFG.

**CICLO DE VIDA**

Iterativo basado en prototipado. Prototipos evolutivos y revisiones constantes.

- Reuniones cada sábado por la noche en el estudio (el ático de la casa de Fernando).
    - Primero se escribe el orden del día en la pizarra, para no disgregar.
    - Se hace una revisión del trabajo desarrollado de la última semana.
    - Se discute sobre los problemas encontrados y sus posibles soluciones.
    - Se realiza una planificación a corto plazo para la próxima semana.
    - Se revisa y, si procede, se corrige la planificación a largo plazo. En este punto hay que preguntarse si será posible cumplir los plazos.

**METODOLOGÍA DE DESARROLLO**

Se han tomado prestados elementos de la metodología SCRUM para la gestión del proyecto. Los siguientes puntos son aplicables también a proyectos individuales, ya que yo mismo los he utilizado en el desarrollo individual de la demo que nos ocupa, como mostraré a continuación.

- **Prototipos ágiles**: durante el desarrollo del videojuego completo, el equipo en su conjunto se dio cuenta de que estancarse en escenarios y niveles durante semanas sin poder jugarlos desmotiva a los miembros y, por tanto, ralentiza el desarrollo. Por ello se tomará como norma el desarrollo de niveles y mecánicas sencillas pero jugables en espacios cortos de tiempo. En las reuniones de los sábados siempre se deberá aportar algo nuevo que probar.
- **Stand Ups meetings**: en el canal de slack del equipo se ha instalado un bot que realiza a cada miembro 3 preguntas por las mañanas:
    - ¿Qué conseguiste hacer ayer?
    - ¿Qué harás hoy?
    - ¿Qué obstáculos están frenando tu progreso?

De esta forma los departamentos de arte y programación se mantienen actualizados a pesar de no estar trabajando conjuntamente. Además, los reportes generados ayudan a detectar posibles cuellos de botella o fallos en la organización, para así corregirlos en el

futuro.

- **Milestones y sprints.** Se usará HacknPlan y las reuniones semanales de los sábados para dividir el trabajo en objetivos
- **Scrum taskboard.** Cada vez que surja una tarea, aunque no se vaya a realizar en el momento, se apuntará en el Hacknplan en la columna de "pendientes". Al crear nuevas milestones se asigna a ellas una cantidad razonable de tareas para lograr terminarlas todas. Las columnas en las que se dividirán las tareas son:
    - Planned
    - In progress
    - Testing
    - Done

Una vez finalizada una tarea pasará primero por la fase de "Testing", hasta que otro miembro del equipo la examine y la valide y decida pasarla a "Done". No podemos validar nuestras propias tareas. Por razones obvias, durante el desarrollo de la demo y el TFG he tenido que saltarme esta última regla y validar mi propio trabajo. [21, 22, 23, 24]

* HacknPlan es una plataforma para organización de proyectos enfocada al desarrollo de videojuegos, que destaca por ofrecer categorías predeterminadas para las distintas tareas (arte, programación...) y un sistema para escribir el game design. [25]

**RECURSOS SOFTWARE DESARROLLO**

- Unity. Assets utilizados:
    - Pro Buiilder
    - Pro Grids
    - Text Mesh Pro
    - Shader Graph [26]
- Blender
- C#
- HacknPlan
- Git & GitHub

**ROLES ESPECIALES**

Cada miembro del equipo hace un poco de todo, así que aquí solo se nombran los roles extra que ha adoptado, con los que el resto están de acuerdo

- Miguel (Yo):
    - Gestor del proyecto

| |
|---|
| ○ SCRUM master [27]<br>• Fernando:<br>○ Modelador y animador de personajes<br>• David:<br>○ Dirección artística<br>○ Diseño de niveles<br>• Dani:<br>○ Coordinador |
| **HERRAMIENTAS PARA COMUNICACIONES EN EL EQUIPO DE TRABAJO** |
| • Telegram (grupo BOTH_DEV)<br>• Slack (canal gambusinolabs.slack.com)<br>• HacknPlan (tablero SCRUM)<br>• Google Drive (assets gráficos y documentos) |

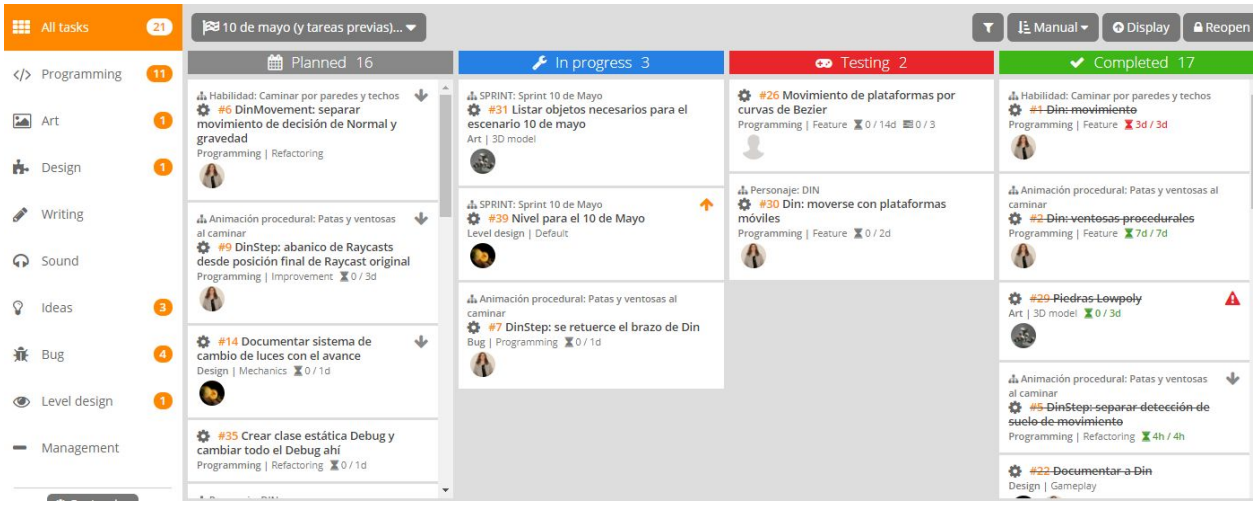


Figura 12. Tablero SCRUM.

# Planificación temporal

## Sprints

A continuación expongo cómo me he planificado mis propias actividades de programación y cómo he ido cumpliendolas y avanzando en el desarrollo.

Nótese que el la mayoría del peso del desarrollo recae en el movimiento de los personajes y en su interacción entre ellos, pero existe una mayor carga de trabajo en el personaje de

Vlinky que en el Din. Esto es debido a que Vlinky comenzó a desarrollarse desde cero, mientras que a Din le había dedicado algo de tiempo anteriormente, y su programación fue más sencilla.

Además del tiempo de programación, estimé alrededor de 1 hora de investigación cada día, refiriéndome a la búsqueda de información y aprendizaje sobre Unity, su documentación, sus patrones de diseño, seguimiento de tutoriales, etc. En definitiva, todo aquello que supusiera aprender y probar cosas que no eran directamente implementables en la demo, pero me servían para aprender cosas necesarias para ésta.

| **Sprint 1: reorganización del proyecto antiguo, refactorización y planificación temporal.** | | **16 Junio → 23 Junio** |
|---|---|---|
| Tarea | Tiempo planificado | Tiempo real |
| Organizar directorios del proyecto de Unity y reunir prototipos antiguos | 1h | 1h 30m |
| Documentar estándar de directorios del proyecto | 1h | 1h |
| Configurar asset XT Utilities para tomar notas en los game objects | 30m | 30m |
| Configurar asset Rainbow Folders para la organización del proyecto | 30m | 30m |
| Planificación y estimación de tiempos | 1h | 1h |
| Investigación sobre organización de proyectos en Unity | 5h | 2h |

| **Sprint 2: movimiento básico de Vlinky.** | | **14 Julio → 21 Julio** |
|---|---|---|
| Tarea | Tiempo planificado | Tiempo real |
| Separar input del jugador de lógica de movimiento | 1h | 1h |
| Explorar posibilidades de movimiento mediante Rigidbody | 2h | 1h |

| | | |
|---|---|---|
| Explorar posibilidades de movimiento mediante Character Controller | 2h | 2h |
| Implementar movimiento mediante Rigidbody | 8h | 11h |
| Vlinky resbala en pendientes | 2h | 5h |
| Investigación sobre movimiento y físicas de personajes | 5h | 5h |

| **Sprint 3: salto de Vlinky.** | | **21 Julio → 28 Julio** |
|---|---|---|
| Tarea | Tiempo planificado | Tiempo real |
| Hacer que Vlinky salte | 2h | 3h |
| Añadir control en el aire | 2h | 4h (no logrado) |
| Arreglar bug en precipicios | 2h | 4h |
| Adaptar movimiento de cámara | 1h | 1h |
| Investigación sobre movimiento y físicas de personajes | 5h | 3h |
| Imprevisto: revisión del movimiento de Vlinky | | 4h |

| **Sprint 4: Sistema de Input y Sistema de Eventos.** | | **28 Julio → 4 Agosto** |
|---|---|---|
| Tarea | Tiempo planificado | Tiempo real |
| Implementar sistema de InputManager e InputListeners | 2h | 4h |
| Refactorizar inputs | 1h | 1h |
| Implementar sistema de eventos mediante scriptable objects | 1h | 4h |
| Refactorizar: eliminar dependencias y comunicar por eventos | 2h | 2h |

| | | |
|---|---|---|
| Investigación<br>• Sistemas de Input<br>• Sistemas de Eventos | 5h | 6h |
| Imprevisto: tiempo extra aprendiendo delegates de C# | | 2h |

| **Sprint 5: Refactorización y scripts de organización + Habilidad de Vlinky de disparar.** | | **4 Agosto → 11 Agosto** |
|---|---|---|
| Tarea | Tiempo planificado | Tiempo real |
| Editor scripts para ordenar jerarquía | 4h | 2h |
| Editor scripts para controlar la posición de los personajes en la jerarquía | 4h | 3h |
| Refactorización del sistema de eventos: implementación mediante C# delegates | 2h | 2h |
| Disparo de Vlinky | 6h | 4h |
| Investigación<br>• Editor scripts<br>• Line renderers para dibujar trayectorias de disparo | 5h | 5h |

| **Sprint 6: habilidad de Vlinky para empujar objetos.** | | **11 Agosto → 18 Agosto** |
|---|---|---|
| Tarea | Tiempo planificado | Tiempo real |
| Skill / Movement architecturing refactor | 2h | 4h |
| Objeto empujable | 2h | 4h |
| Objeto empujable mock prototype | 1h | 1h |
| Habilidad de empujar | 6h | 8h |
| Investigación<br>• Empujar objetos<br>• Unity Joints | 5h | 4h |

| **Sprint 7: Din Globo + Interacciones finales entre personajes.** | | **18 Agosto → 25 Agosto** |
|---|---|---|
| Tarea | Tiempo planificado | Tiempo real |
| Din Globo: movimiento y transformación | 6h | 2h |
| Géiseres de gas | 1h | 30m |
| Vlinky coge y dispara a Din | 4h | 2h |
| Din Globo coge objetos | 4h | 1h |
| Din Globo coge a Vlinky | 4h | 1h |
| Investigación<br>● Animaciones<br>● Cámara | 5h | 2h |

| **Sprint 8: Construcción del escenario de la demo.** | | **25 Agosto → 2 Septiembre** |
|---|---|---|
| Tarea | Tiempo planificado | Tiempo real |
| Storyboard | 1h | 30m |
| Aprendizaje Pro Builder | 2h | 1h |
| Montaje del escenario | 8h | 5h |
| Corrección de bugs | 8h | 6h |
| Investigación<br>● Animaciones<br>● Pro Builder<br>● Text Mesh Pro<br>● Iluminación | 5h | 3h |

## Sprints: resumen

| | | | Horas | |
|---|---|---|---|---|
| Sprint | Fecha | Features implementadas | Netas | Reales |
| 1 | 16 Jun. → 23 Jun. | • Organización<br>• Planificación | 9h | 6h 30m |
| 2 | 14 Jul. → 21 Jul. | • Movimiento de Vlinky | 20h | 25h |
| 3 | 21 Jul. → 28 Jul. | • Salto de Vlinky | 12h | 19h |
| 4 | 28 Jul. → 4 Ago. | • Input System<br>• Event System | 11h | 19h |
| 5 | 4 Ago. → 11 Ago. | • Disparo de Vlinky<br>• Editor scripts | 21h | 16h |
| 6 | 11 Ago. → 18 Ago. | • Objetos empujables<br>• Habilidad de Vlinky para empujar | 16h | 21h |
| 7 | 18 Ago. → 25 Ago. | • Din Globo<br>• Vlinky dispara a Din<br>• Din Globo coge objetos<br>• Din Globo coge a Vlinky | 24h | 8h 30m |
| 8 | 25 Ago. → 2 Sep. | • Escenario de la demo | 24h | 15h 30m |
| | | | | |

| | |
|---|---|
| Netas totales | 137h |
| Reales totales | 130h 30m |

## Conclusiones obtenidas

Cuando di comienzo al desarrollo planeé dedicarme al proyecto como si de un horario laboral de media jornada se tratase. Han sido 8 semanas de trabajo, a 20 horas por semana, deberían haber salido 160 horas totales. Al principio, antes de comenzar el desarrollo, los tiempos eran más difíciles de estimar, pero conforme avancé en los sprints perfeccioné esa habilidad.

A pesar de que la estimación de tiempos no fue del todo exacta debido a mi falta de experiencia en el desarrollo de videojuegos, sí que fue muy positivo contar con esa división de tareas por sprints semanales, ya que me ayudaba a prever a qué me iba a enfrentar y qué prototipo jugable tendría listo para la semana que viene.

## Estimación de costos

Se ha decidido realizar una estimación de costos del desarrollo de la demo abierta al público completa, es decir, de continuar el desarrollo de esta demo técnica durante un mes más haciendo hincapié en el apartado gráfico y trabajando codo con codo con los artistas.

Para ello, he realizado la estimación de costos siguiendo el modelo COCOMO II, basándome en las métricas obtenidas de este último mes de desarrollo y añadiendo un 20% más del trabajo de programador que se ha hecho en esta demo. El volumen de código del próximo mes no será el mismo que el de este desarrollo, en que las mecánicas se han desarrollado desde cero; y, además, el mayor peso del desarrollo de la demo pública recaerá en integrar los gráficos desarrollados en Unity.

Para esta demo técnica se han escrito en total 5.000 líneas de código. Se ha supuesto que para el desarrollo de la demo abierta al público, harán falta un 20% más, es decir, otras 1000.

También cabe mencionar que la razón por la que se han usado como métricas las líneas de código en lugar de los puntos de función, que es para lo que originalmente está diseñado el modelo COCOMO II, es debido a que en estos últimos se consideran múltiples factores típicos de una aplicación software más tradicional, y la mayoría de sus mediciones son

sobre inserciones, consultas y salidas de bases de datos, formularios del usuario, etc. Como el desarrollo de videojuegos no cuenta con estas características, se ha optado por utilizar como input de medida las líneas de código, a pesar de que se trate de un valor más inexacto que los puntos de función para los que fue diseñado. [28, 29]

## COCOMO II: Input

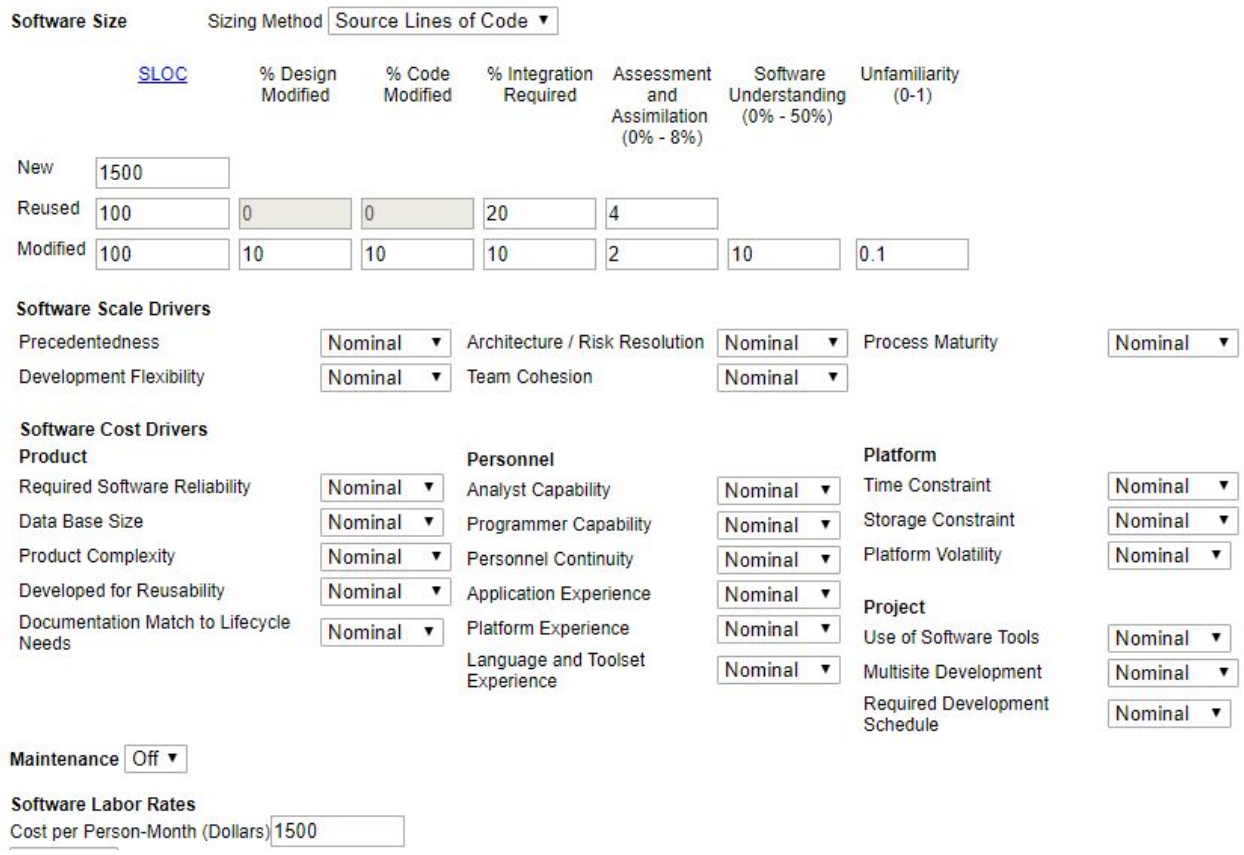


Figura 13. COCOMO II Input.

## COCOMO II: Resultados

**Results**

**Software Development (Elaboration and Construction)**

Effort = 4.7 Person-months
Schedule = 6.1 Months
Cost = $7000

Total Equivalent Size = 1522 SLOC

**Acquisition Phase Distribution**

| Phase | Effort (Person-months) | Schedule (Months) | Average Staff | Cost (Dollars) |
|---|---|---|---|---|
| Inception | 0.3 | 0.8 | 0.4 | $420 |
| Elaboration | 1.1 | 2.3 | 0.5 | $1680 |
| Construction | 3.5 | 3.8 | 0.9 | $5320 |
| Transition | 0.6 | 0.8 | 0.7 | $840 |

**Software Effort Distribution for RUP/MBASE (Person-Months)**

| Phase/Activity | Inception | Elaboration | Construction | Transition |
|---|---|---|---|---|
| Management | 0.0 | 0.1 | 0.4 | 0.1 |
| Environment/CM | 0.0 | 0.1 | 0.2 | 0.0 |
| Requirements | 0.1 | 0.2 | 0.3 | 0.0 |
| Design | 0.1 | 0.4 | 0.6 | 0.0 |
| Implementation | 0.0 | 0.1 | 1.2 | 0.1 |
| Assessment | 0.0 | 0.1 | 0.9 | 0.1 |
| Deployment | 0.0 | 0.0 | 0.1 | 0.2 |

Figura 14. COCOMO II Outputs

## COCOMO II: Análisis de los resultados

Los resultados de COCOMO II estiman un desarrollo de 6 meses de duración para un equipo de unas 5 personas. Sin embargo, estos resultados no se deben ser tomados al pie de la letra debido a varios factores:

- La principal razón es que, a pesar de que COCOMO II es un modelo para estimar el coste de desarrollos software, este último término es demasiado ambiguo. ¿Se pueden considerar los videojuegos un desarrollo software? Sí, pero hay que entender que en el contexto en que se diseñó este algoritmo, en los años 80, los desarrollos software dependían de otro tipo de métricas.

- El tiempo de desarrollo que estima es el de un proyecto completo, lo cual tampoco se ajusta a nuestra definición de demo. En la demo pública de LevelUp! Granada, Gambusino Labs no entregará el producto final al usuario, sino una pequeña parte de éste. Debido a ello, se tiene mucho más control sobre lo que el usuario tiene o no capacidad de hacer, porque se trata precisamente de eso, de una demostración. Por ello, se pueden permitir determinados errores o bugs, o flexibilizar el diseño de niveles para que algo que se encuentra en desarrollo parezca un producto final a ojos del usuario. Mientras que permanezcan ocultos a la experiencia de juego que ofrezca la demo no habrá problema.
- Otro factor de suma importancia es que las variables con las que se calibra el algoritmo son obtenidas directamente de la experiencia en desarrollos previos. Puesto que este es el primer proyecto de Gambusino Labs, y la experiencia obtenida consta tan solo de unos meses en el desarrollo, no es posible calibrar el algoritmo con precisión suficiente.

Al fin y al cabo COCOMO II es un modelo para estimación de costes, y hay que tomarlo como lo que es, una estimación. Por tanto, y teniendo en cuenta los puntos arriba descritos, es de suponer que los seis meses que COCOMO II estima, pueden rebajarse debido a que el proyecto es una demostración. Respecto al precio obtenido, los 7000$ son también sólo una estimación de lo que costaría como contratar a Gambusino Labs como equipo para hacer tal desarrollo, pero una vez más, tampoco se debe tomar al pie de la letra, ya que la demo pública es un producto que desarrollará Gambusino Labs con el fin último de obtener rentabilidad comercial, sí, pero no la venta de la demo, sino de lo que vendrá después.

## Conclusiones

Estimar costos en el desarrollo de videojuegos no es una tarea fácil, sobre todo al nivel de una microempresa como es el caso de Gambusino Labs. Nuestra falta de experiencia en desarrollos previos, así como la inmadurez del sector indie del videojuego, sobre todo en España; y la multitud de facetas profesionales que requiere el desarrollo de un videojuego, hace que sean cifras difíciles de estimar.

Ante esta situación de de evidente falta de herramientas teóricas, lo mejor que puede hacer Gambusino Labs como estudio de videojuegos indie en España es nutrirse de la información empírica que ofrecen organismos fiables como la organización DEV (Desarrollo Español de Videojuegos) en los informes que publican. Por ejemplo, gracias a su publicación más reciente, el "Libro Blanco del Desarrollo de Videojuegos en la Comunidad Valenciana 2018", página 27, se conoce que la necesidad de financiación de un videojuego en España está entre 50.000€ y 150.000€ en el 39% de las empresas encuestadas. [12]


Figura 15. Necesidades de financiación.

Además, Gambusino Labs también coincide con el perfil más común de empresa desarrolladora, el de micropyme, que son el 55%, página 15; a la hora de hacer una estimación de costes del proyecto completo es posible basarse en estos casos.


Figura 16. Distribución de empresas por número de empleados.

# Ingeniería de requisitos

En el caso que nos ocupa, en del contexto de la demo desarrollada para el TFG, yo mismo como Game Designer soy el cliente. Por ende, debo realizarme la extracción de requisitos a mí mismo. Afortunadamente, el Game Design Document aporta la mayoría de información necesaria para definir los requisitos de la demo. Para el resto de requisitos, ya que no tendría sentido que me entrevistara a mí mismo, he utilizado la técnica de observación. Es decir, cuando pruebo y juego los diferentes prototipos que implemento cada semana, me utilizo de beta-testers y tomo nota de mis sensaciones. En base a ello, tanto los requisitos como el Game Design Document han sufrido modificaciones y revisiones.

A continuación se detallan los requisitos extraídos:

## Requisitos funcionales

Principalmente extraídos del Game Design Document

- Vlinky.
    - **RF_0.** Mover al personaje por el suelo, pudiendo caminar en el plano de acción correspondiente a Z = 0.
    - **RF_1.** Saltar, utilizando las físicas del Rigidbody para aplicar una fuerza hacia arriba en el momento del salto.
    - **RF_2.** Disparar ojos, permitiendo apuntar alrededor en el plano correspondiente a Z = 0.
    - **RF_3.** Empujar objetos en el plano correspondiente a Z = 0.
    - **RF_4.** Coger a Din: Vlinky debe poder coger a Din al accionarse el input correspondiente cuando está cerca de él.
    - **RF_5.** Lanzar a Din: con Din sobre él, aplicar las mismas leyes físicas que en el disparo de los ojos y apuntar de la misma manera.
- Din.
    - **RF_6.** Mover al personaje rodando por suelo y paredes, pegándose a la superficie como una lapa si la gravedad lo impide, haciendo así posible andar por paredes y techos. Como con Vlinky, el movimiento se producirá en el eje correspondiente a Z = 0.
    - **RF_7.** Pegarse a paredes y techo. En lugar de la mecánica de salto, será posible para Din pegarse directamente a un techo aunque no esté tocándolo, si está a una distancia mínima configurable desde el inspector.
    - **RF_8.** Dejarse caer. Si Din está pegado a alguna superficie que no sea la del suelo normal, que sea posible despegarlo para dejarse caer de la pared o techo, restaurando en él las físicas normales.
    - **RF_9.** Transformarse en globo. Cuando se esté sobre un objeto de tipo “Géiser de gas”, Din tiene que ser capaz de transformarse en Globo si el jugador ejecuta dicha acción.
    - **RF_10.** Subirse a Vlinky. Además de ser Vlinky el que pueda coger a Din, este último también debe ser capaz de subirse a Vlinky por acción del jugador.
- Din Globo.

    - **RF_11.** Volar. Como con el resto de movimientos, este también se producirá en el eje correspondiente a Z = 0. La característica especial es que debe poderse mover libremente no solo en el eje horizontal, X, sino también en el vertical, Y; haciendo así posible el control volador que la mecánica persigue.
    - **RF_12.** Coger objetos. Si se tiene debajo a un objeto que se pueda coger, el jugador debe ser capaz de pegárselo al Globo y volar con él sujeto. Las físicas del peso del objeto deben afectar a las físicas del globo, haciéndolo más errático debido al peso.
    - **RF_13.** Coger a Vlinky. Caso especial de RF_12: debe ser posible para Din Globo no solo coger objetos, sino también coger al personaje Vlinky.
- **RF_14.** Cambiar de personaje. El jugador debe poder cambiar de un personaje a otro por voluntad, a no ser que lo contrario sea especificado por inspector.
- **RF_15.** Deben poder abrirse puertas pulsando botones.
- **RF_16.** Cosquilleadores: bichos que andan por las paredes y descuelgan a Din si lo tocan.
- **RF_17.** Muerte: objetos rojos indicando peligro que te teletransportan al último punto de control al tocarlos
- **RF_18.** Permitir configurar mediante el inspector eventos ejecutados al ocurrir las habilidades. Por ejemplo, al lanzar Vlinky a Din (RF_5), debe poderse definir que Vlinky pierda el control y lo coja Vlinky.

## Requisitos no funcionales

Principalmente extraídos del desarrollo diario y de la comunicación con los diseñadores.

- Relativos al control y movimiento de los personajes
    - **RNF_0.** El movimiento debe ser fluido y no quedarse atascado en ninguna situación.
    - **RNF_1.** Si se sube a una plataforma móvil debe desplazarse con ella de forma realista.
    - **RNF_2.** Los controles tienen que ser intuitivos para un jugador novato.

    - **RNF_3.** Los personajes deben sentirse ágiles y agradables de manejar para el jugador.
- **RNF_4.** La infraestructura del juego debe estar preparada para soportar multijugador local, aunque no se implemente en la demo.
- **RNF_5.** Optimización.
    - El apartado gráfico de la demo debe estar optimizado para funcionar fluidamente en portátiles modernos sin gráficas dedicadas, por lo menos a 30 FPS por segundo.
    - Los sistemas programados en la demo también deben perseguir la eficiencia, haciendo hincapié en la comunicación de los game objects mediante eventos en lugar de comprobaciones continuas en cada frame.
- **RNF_6.** Los sistemas programados deben ser directamente y fácilmente configurables por los diseñadores desde el inspector de Unity sin tener la necesidad de modificar o entender el código.
- **RNF_7.** Los componentes implementados deben utilizar las herramientas que ofrece Unity, como tooltips y textos informativos, para ofrecer las instrucciones necesarias para su uso.
- **RNF_8.** El sistema de captación de input utilizado será el que viene construido en Unity por defecto, pero se modificarán sus parámetros predeterminados para ajustarse a nuestro esquema de control.
- **RNF_9.** Eventos como saltos, caídas, o habilidades activadas deben ser publicados en el inspector para su fácil integración con las futuras animaciones por parte de los artistas.
- **RNF_10.** Se debe facilitar la fase de testeo exponiendo comportamientos en el editor.

## Requisitos de información

En el caso que nos ocupa no existen requisitos de información, ya que nuestra demo no almacena estado alguno. De haberlos, también podrían haber sido extraídos del Game Design Document, concretamente de un apartado llamado “Estado del juego” del que carece esta aplicación por su naturaleza de demostración.

## Ingeniería del Software en el desarrollo de videojuegos

Este apartado sirve del análisis de los dos anteriores, ya que de ellos se puede extraer una serie de diferencias notables entre la Ingeniería del Software aplicada a aplicaciones más tradicionales, tal y como se enseña durante el grado, y la Ingeniería del Software aplicada al desarrollo de videojuegos.

- La primera diferencia es que la comunicación no se establece tanto entre programador-cliente o programador-usuario, sino entre programador-game designer. Por ello, los requisitos funcionales son extraídos de forma directa del Game Design Document.
    - Puede parecer que la figura del cliente la ocuparían, en este caso, los publishers, y no es así por diversas razones. La primera de ellas es que, mientras que en los contratos de software tradicional, el cliente no tiene por qué tener conocimientos del campo del desarrollo, los publishers sí que tienen conocimientos de márketing y game design. Además, en todo caso los publishers median directamente con el Game Designer y nunca con el Ingeniero Software.
- Los requisitos no funcionales son muchos y evidencian que los objetivos del programador de videojuegos distan de las del desarrollador web o de aplicaciones. Por ejemplo:
    - Requisitos producto de compartir entorno de desarrollo con profesionales de otras disciplinas. Al diseñador de niveles hay que ofrecerle las herramientas necesarias para que no tenga que recurrir al programador a la hora de crear un nivel. Si bien es cierto que en una interfaz de escritorio, móvil o web, la tarea visual también puede recaer sobre el Diseñador Gráfico en lugar del programador, se diferencia de los videojuegos en que el Diseñador Gráfico no es el que después implementará las interfaces. En el caso de los videojuegos, sí que es el diseñador de niveles el que construirá

los puzles y escenarios con los elementos generados por el programador y los artistas.

    - Otra diferencia notable son los requisitos surgidos del motor gráfico en que se desarrolla el videojuego. En este caso, existe la necesidad de ofrecerle al diseñador una interfaz clara donde calibrar las mecánicas del juego, que en Unity se trata de la ventana del inspector.

- De los requisitos de información, es interesante remarcar que los videojuegos raramente utilizan bases de datos SQL, sino archivos de texto o binarios para el almacenamiento de la partida del jugador. Además, otra diferencia es que, mientras que las aplicaciones web, por ejemplo, suelen tener como requisitos de información bases de datos para múltiples usuarios, eso no se suele encontrar en los videojuegos offline, como el caso que nos ocupa. Lo más parecido sería la posibilidad de guardar varias partidas.

Existe un libro, llamado “Computer Games and Software Engineering” que abarca con detalle esta problemática del desarrollo de videojuegos, disciplina que históricamente se ha caracterizado más por otro tipo de documentación más informal, como el Game Design Document o los Post-Mortems, y menos técnica y sistemática que la que produce la Ingeniería del Software. Por ejemplo, en videojuegos raramente se documentan casos de uso o escenarios. [30]

Como conclusión personal, y ocupando en este proyecto tanto el rol de Game Designer como el de Ingeniero Software, he echado en falta las bondades de la Ingeniería del Software en el desarrollo de videojuegos. Sí que he utilizado los conocimientos aprendidos durante la carrera, pero ha requerido de un esfuerzo extra para adaptarlos y conseguir utilidad en el caso concreto del desarrollo de videojuegos.

## Cumplimiento de requisitos

Se han satisfecho con éxito todos los requisitos extraídos en el presente apartado. La mayoría de estos requisitos se han solventado generando un diseño software que facilite la escalabilidad de la demo hacia el desarrollo del videojuego completo, generando sistemas

sólidos y fácilmente mantenibles, como se explicará en el apartado “Diseño Software de Both”. Algunos otros, como el RF_16 o el RF_17, han sido satisfechos mediante la implementación de prototipos ágiles que sirven a la consecución de esta demo, pero que serán revisados durante el desarrollo posterior.

Por tanto, como ya se ha dicho, durante el desarrollo del apartado “Diseño Software de Both” se referencian y justifican los requisitos satisfechos durante el desarrollo.

# Arquitectura Software

## Importancia del Systemic Game Design en la Arquitectura Software de los videojuegos

En el desarrollo de videojuegos el diseño arquitectónico del software juega un papel muy importante, ya que esa faceta influirá fuertemente en lo robusta, extensible, flexible y escalable que acabe siendo nuestra base de código. Comenzar un desarrollo sin explorar ni investigar este aspecto acabará por afectar al desarrollo de una u otra manera, bien generando bugs imposibles de solventar sin generar dependencias innecesarias, bien porque si surgen nuevas especificaciones durante el desarrollo que no encajen en nuestro actual diseño, éste podría haber sido más flexible para estar abierto a este tipo de cambios.

Las aplicaciones tradicionales constan de uno o varios sistemas, y la comunicación y organización entre ellos es lo que determina el diseño arquitectónico del software. Normalmente esta interacción de sistemas es fácil de predecir desde una etapa temprana del desarrollo y suele diferenciar las diferentes capas o niveles de abstracción de una aplicación, por ejemplo diferenciando entre el código responsable de mostrar la interfaz al usuario, el relacionado con los datos almacenados a los que la aplicación accederá, y la capa intermedia que comunica ambos sistemas; tal y como en el Modelo Vista Controlador.

Sin embargo, en los videojuegos existen diferencias sustanciales.

- El número de sistemas suele ser mayor
- La interacción entre los sistemas está ligada al Game Design y, si este cambia y la arquitectura no ha previsto ese cambio, es necesario reestructurarla.

- Los sistemas suelen estar al mismo nivel, es decir, no son diferenciables en cuanto a nivel de abstracción.

Este comportamiento orgánico de los componentes software del videojuego es debido a la naturaleza desestructurada del mundo que los videojuegos pretenden imitar. Con el avance del detalle gráfico de los últimos años, los videojuegos han pasado de ser una abstracción de la realidad a convertirse en prácticamente simuladores. Y en su carrera de parecerse a la realidad no solamente han mejorado los gráficos en pos del fotorrealismo, sino que también han imitado los comportamientos erráticos e impredecibles del mundo real. [32]

En ese tipo de juegos, el Game Designer no diseña una experiencia de juego, ni planea el camino que tomará el jugador para afrontar una situación concreta. Los Game Designers diseñan sistemas que interactúan entre ellos, todos con todos; y los presentan en el mundo del juego como parte de él, y no como parte de un nivel o experiencia aislada. Los sistemas han sido diseñados de forma que todos influyan a todos, dando lugar así a situaciones impredecibles. De esta forma en la que el jugador interactúa con los sistemas le ayudarán o perjudicarán en el desafío al que se enfrenta según los utiliza a su favor o en contra.

El diseño sistémico marca la diferencia entre una experiencia lineal, en que el jugador juega un nivel que siempre trata de quemar un bosque para acabar con unos enemigos, y siempre lo hace de la misma manera; a un juego en que al jugador se le presenta el mundo y su objetivo: acabar con los enemigos. En el bosque hay animales que son mansos (sistema de fauna). El jugador decide quemar el bosque para acabar con sus enemigos (sistema de fuego interactuando con sistema de enemigos), de tal suerte que el juego asusta a los animales y se vuelven agresivos hacia el jugador y hacia los enemigos (sistema de fauna interactuando con el sistema de fuego, con el jugador y con los enemigos). Esto pone al jugador en un nuevo aprieto, huir del incendio y de los animales violentos; pero comienza a llover, el fuego se apaga y los animales se resguardan (sistema de clima interactuando con el resto). Para colmo, la salud del jugador comienza a disminuir por el

frío (sistema de supervivencia) y si corres puedes caerte porque la fricción del suelo ha disminuido (sistema de físicas).

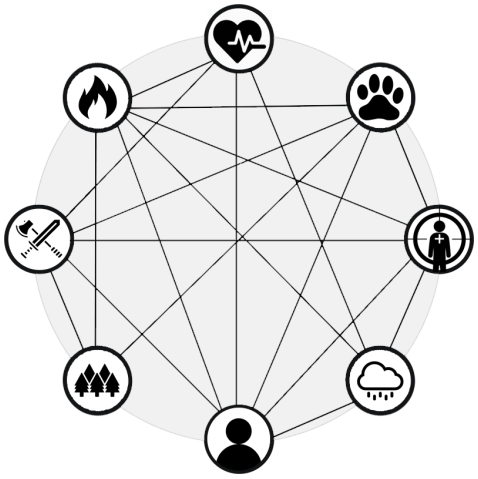

Figura 17. Systemic Game Design.

Y bien, ¿qué conclusión sacamos de esta interrelación entre arquitectura de software y systemic game design? Que surge la necesidad de un diseño arquitectónico más orgánico, más flexible, menos estructurado que el del software tradicional, y abierto a este tipo de cambios. Esta necesidad Unity la solventa mediante una Arquitectura Software Basada en Componentes.

## Unity: Arquitectura basada en componentes

La arquitectura basada en componentes es una alternativa a la orientación a objetos (Object Oriented Design, OOD), pero ambos paradigmas no son excluyentes entre sí. El flujo de trabajo de Unity favorece en gran medida el orientar tu diseño de clases a este nuevo paradigma. Cuando comencé a programar en este motor y comencé aplicar lo que había aprendido de otros desarrollos para generar un software sólido: conceptos más avanzados de programación como patrones de diseño o estructuras de herencia, me di cuenta de que en Unity funcionaban diferente. No quiero decir que dichas técnicas sean imposibles de implementar, de hecho lo conseguí, y funcionaban tan bien como en cualquier otro software; pero constantemente sentía que estaba luchando contra el motor

y contrariando su forma de hacer las cosas. Llegué a pensar que Unity era así de malo y promovía un mal diseño. [31, 33, 34]

Gracias a esta investigación entendí qué era lo que estaba pasando: estaba aplicando los conceptos de OOD a un software cuya arquitectura estaba basada en componentes. No quiero decir que ambos paradigmas de programación sean mutuamente excluyentes, pues la orientación a componentes es un caso concreto dentro de la orientación a objetos, y por tanto necesita de ella y persigue sus mismos objetivos. La arquitectura basada en componentes hace especial énfasis en los principios de responsabilidad única y bajo acoplamiento.

Un componente es pieza pequeña dentro de un sistema mayor que puede cumplir su cometido sin la necesidad de ayuda externa. Y, como las piezas de las máquinas, raramente pertenecen exclusivamente a una sola máquina, sino que se pueden combinar de maneras diferentes para dar lugar a resultados distintos. Esto es gracias a que los componentes se abstraen completamente del conjunto completo. De hecho, ni siquiera saben que éste existe.

Gracias a esta estructura, Unity provee al desarrollador de una serie de herramientas que permiten modificar, añadir, mover o borrar componentes de forma visual e incluso en tiempo de ejecución, mediante su ventana de inspector.

## Ejemplos

- Mientras que en OOD podría darse la siguiente estructura de objetos: “Un cachorro de gato es un tipo de gato, que a su vez es un tipo de cuádrupedo, que a su vez es un tipo de animal. Animal -> FourLegged -> Cat -> Kitten”. En orientación a componentes sería: “La entidad ‘Gatito’ está compuesta por AnimalBehaviour, FourLeggedBehaviour, CatBehaviour y YoungAnimalBehaviour”.
- En videojuegos, podemos pensar en el comportamiento de un enemigo como un mismo sistema. Pero algunos enemigos no pueden andar, solo pueden saltar; o viceversa, y unos pueden disparar y otros no. A lo mejor algunos vuelan y disparan. Así, se pueden combinar los componentes “Walk”, “Jump”, “Fly” y “Fire”, de cualquier

manera diferente para generar nuevos tipos de enemigo. ¿Y si queremos que un enemigo sea totalmente invisible? Se soluciona prescindiendo de los componentes “MeshFilter” y “MeshRenderer”, que en Unity son los responsables de dar forma y visualizar al objeto, respectivamente.

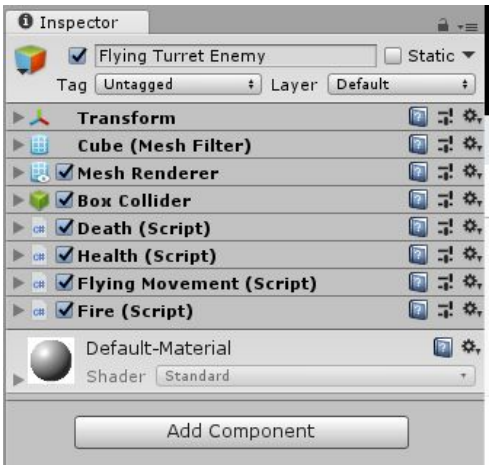


Figura 18. Ejemplo de componentes.

En resumen, mientras que la orientación a objetos descompone el comportamiento de los objetos en base a qué son, la orientación a componentes los descompone en base a qué funcionalidad necesitan. [35]

Este tipo de diseño arquitectónico colabora con que RNF_4 y RNF_6 sean satisfechos.

## Unity: Estructura de clases interna

Todas las clases que componen la estructura de objetos de Unity, y por tanto su API pública en C# a la que se puede acceder, heredan de Object, un tipo de objeto que en esencia está preparado para ser destruido e instanciado en tiempo de ejecución. Por tanto los Components son también un tipo de Object en Unity y todos soportan esa “edición en caliente”, como explicoen el apartado anterior. Unity, concretamente, te obliga a extender todos tus componentes de la clase MonoBehaviour, hija de Behaviour, que es un componente que puede estar habilitado o deshabilitado. Un Behaviour deshabilitado no será llamado por Unity en cada frame en las funciones de Update(), ni en su función de Start(). Sí que recibirá el la llamada de la función Awake() a pesar de ello, a no ser que

también el Game Object que lo contiene esté deshabilitado, en cuyo caso tampoco Awake() será llamada. [36]

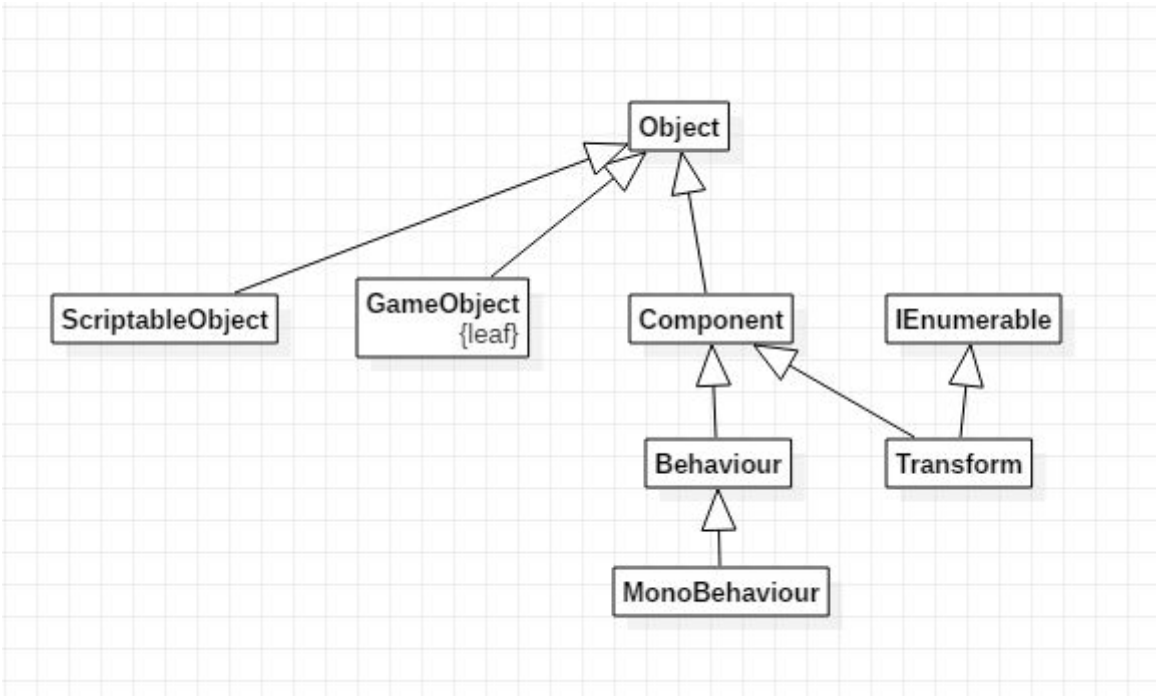


Figura 19. Clases internas de Unity.

Además existe un tipo de objeto preparado para existir en la instancia de juego pero sin estar ligado a ningún GameObject, es decir, sin tener Transform ni estar posicionado en el mundo: el tipo Scriptable Object. Heredando de él he construido el sistema de eventos, que ilustraré a continuación junto al resto de sistemas.

# Diseño Software de Both

## Cumplimiento de requisitos mediante el desarrollo de sistemas sólidos

En los apartados siguientes expondré los componentes software y patrones de diseño que he implementado para generar un código que cumpla los requisitos de manera robusta, extensible y fácilmente mantenible. Desde el principio del desarrollo he tenido especial

cuidado de escribir un código que cumpla con los principios de la programación orientada a objetos, siempre teniendo presentes los cinco principios de la programación SOLID:

1. Single responsibility
2. Open/closed principle
3. Liskov substitution principle
4. Interface segregation principle
5. Dependency inversion principle

Además de esos principios, yo mismo me formulé unos objetivos concretos en base al game design de la demo en los que yo veía peligro de cometer code smells. En la consecución de dichos objetivos, también se cumplen la mayoría de requisitos. Los que faltan por cumplir aquí

- Son dos personajes y ambos se relacionarán entre sí, pero no puede haber dependencias entre uno y otro para hacer posible el control de uno sin el otro y viceversa. **Solucionado mediante el sistema de eventos, en unos casos, e interfaces en otros,** cubriendo así el RNF_09 y colaborando en el cumplimiento del RNF_04. Dicho sistema también satisface RF_18. Además, el desarrollo de un sistema de eventos en una etapa temprana del desarrollo evita que se escriban funciones Update(), ejecutadas en cada frame, pesadas y que están continuamente comprobando si ha ocurrido un cambio en el estado del juego. Por ello, se satisface también RNF_5.

- Los personajes estarán compuestos por un movimiento (clase padre común) que dictará cómo se moverán por el mundo, y por otro lado están sus diferentes habilidades que se activan y desactivan en momentos concretos (o durante un lapso de tiempo). El movimiento y las habilidades deben ser independientes entre sí, pudiendo añadir o restar habilidades a los personajes incluso en tiempo de ejecución, e incluso. **Solucionado mediante las clases abstractas ControllableCharacter y Skill**. En el desarrollo de este módulo se han satisfecho también los RNF_0, RNF_1, y RNF_3, y colaborado al cumplimiento del RNF_2. Además, los sistemas implementados en este apartado cumplen los requisitos que

abarcan dos personajes y sus transformación, ya que engloban tanto su movimiento como sus habilidades. Por tanto, quedan satisfechos también los RF desde el RF_0 al RF_13, éste incluido.

    - Como se puede ver en las capturas de pantalla de las secciones próximas, que ilustran no solo el código de estos componentes sino también su vista de inspector, se satisface también RNF_7 y RNF_10, ya que el desarrollo de estos componentes ha tenido lugar en torno a estas pautas.

- Hay comportamientos que son comunes a objetos y personajes. Por ejemplo, Din Globo es capaz de coger tanto a Vlinky como a otros objetos, al igual que Vlinky es capaz de lanzar tanto ojos como a Din. **Solucionado mediante componentes comunes a personajes y objetos: GlobeCarriable y VlinkyBullet**. La consecución de este objetivo influye positivamente en la limpieza y elegancia con la que se han cubierto los requisitos RF_4, y RF_5, RF_10, RF_12 y RF_13, haciendo que los sistemas programados colaboren también a la escalabilidad del código, colaborando así en el cumplimiento de los RNF_4 y RNF_6.

- Dado que habrá diversos sistemas que dependerán del input del jugador, se deberá evitar repartir el código de input en cualquier clase. Solo las clases registradas deberían poder escucharlo, y para modificar las reglas del input debería hacerse sólo una vez, desde una clase centralizada. **Solucionado mediante la clase singleton InputSystem y la interfaz IInputListener**. Este sistema colabora también en el cumplimiento de los RF_0 al RF_13, ya que comunica dichos sistemas con el input del jugador. Además, también satisface RF_14 y RNF_8.

    - Los Singletons son peligrosos de utilizar si se abusa de ellos y pueden dar lugar a errores difíciles de trazar o a dependencias ocultas. Para una demo es imprescindible que este sea el único Singleton, e incluso para el juego final se deberán buscar alternativas más seguras o usarlos con precaución. **Solucionado limitándome a la utilización de esta única clase Singleton.**

    - Además, los Singletons en Unity tienen otro problema: por culpa de que el todo el mundo y los comportamientos del juego están incluidos en escenas, esto genera el riesgo de que el diseñador olvide incluir el InputSystem en la escena. **Solucionado mediante el método estático Object.DontDestroyOnLoad().** De esta forma el objeto se generará a si mismo cuando sea necesario y no hará falta colocarlo en cada escena, además de que se mantendrá el mismo entre una escena y otra.

- Existirán elementos que, como las puertas, podrán abrirse y cerrarse, subirse o bajarse, o activarse y desactivarse. Del mismo modo habrá otros elementos que además de los botones que podrán activar o desactivar mecanismos, por ejemplo una palanca, o simplemente al pasar por un sitio. **Solucionado mediante el sistema de Activadores / Activables.** Mediante dicho sistema se satisface RF_15, y se implementan menús contextuales para probar los activadores sin jugar al juego, con lo que se colabora en el cumplimiento de RNF_10.

## Cumplimiento de requisitos mediante prototipos para la demo

El objetivo de la presente demo es, recordemos, ilustrar las mecánicas básicas de los personajes. Para ello, era necesario incluir algún tipo de obstáculos que obligaran a la utlización de estas mecánicas. Dichos obstáculos son los referidos en RF_16 y RF_17: los cosquilleadores que despegan a Din de las paredes y la muerte y el checkpoint.

Es obvio que el sistema de muerte y checkpoints, así como el sistema de enemigos, forman parte crucial en el desarrollo de casi cualquier videojuego. En el caso que nos ocupa, sin embargo, no requerimos del funcionamiento de varios enemigos interactuando entre sí y con el jugador, ni tampoco de varios tipos de muerte, ya que no han sido ideados como elementos conductores del gameplay sino ilustrativos.

Es por esta razón que, para la satisfacción de RF_16 y RF_17, se han desarrollado componentes prototipo que cumplen los requisitos para la realización de la demo, pero que serán revisados al escalar el videojuego futuro a nuevos enemigos y formas de morir.

Los componentes implementados han sido:

- Tickler (RF_16): su movimiento es una simplificación del movimiento de Din. Además, en lugar de recibir el input del jugador, lo recibe de una IA simplísima que decide aleatoriamente cuando cambiar de sentido.
- QuickDeath (RF_17): normalmente, la muerte implica restaurar la partida a un estado anterior. En el caso de la demo, que ni siquiera define estado del juego, la muerte ha sido implementada simplemente como un sistema que teletransporta al jugador a un punto anterior.

## Sistema de Eventos

En los próximos dos apartados se ilustran los tipos de eventos que se han utilizado. Mientras que el primero de ellos, Game Events mediante Scriptable Objects, ha sido desarrollado para la ocasión; los eventos de Unity son una característica del motor.

### Game Events mediante Scriptable Objects

| **Directorio** |
|---|
| Both\Assets_Both_Scripts\Event System |
| **Clases desarrolladas** |
| GameEvent |
| IGameEventListener |
| GameEventListener |

Debido a mi experiencia previa en Java, al comienzo del desarrollo comenzé a desarrollar el patrón de diseño Observable Observador. Sin embargo durante su aplicación determiné varias carencias:

- La comunicación entre los objetos se define a nivel de código, impidiendo así su trazabilidad desde el inspector de Unity.

- Por la misma razón que el punto anterior expone, no es posible configurar la comunicación a nivel de diseñador, sin necesidad de escribir código.

Por ello investigué nuevas soluciones y alternativas. El sistema implementado está basado en la conferencia "Game Architecture with Scriptable Objects" de Unite Austin 2017, por Ryan Hipple, el ingeniero principal de Schell Games. [38, 39]

El sistema propuesto por Ryan Hipple implementa un sistema de eventos completamente orientado a los datos, es decir, a la generación de eventos serializables que los diseñadores pueden crear directamente desde la ventana de proyecto en Unity. Además, el componente que utiliza los eventos y que emite la respuesta también está pensado para ser utilizado por diseñadores en el inspector, como se ilustra en la siguiente figura:


Figura 20. Inspector de Game Event Listener.

Yo he mantenido la implementación de Ryan Hipple y el componente que propone, "Game Event Listener", pero lo he extendido un poco más, tal como se ilustra en la figura siguiente:

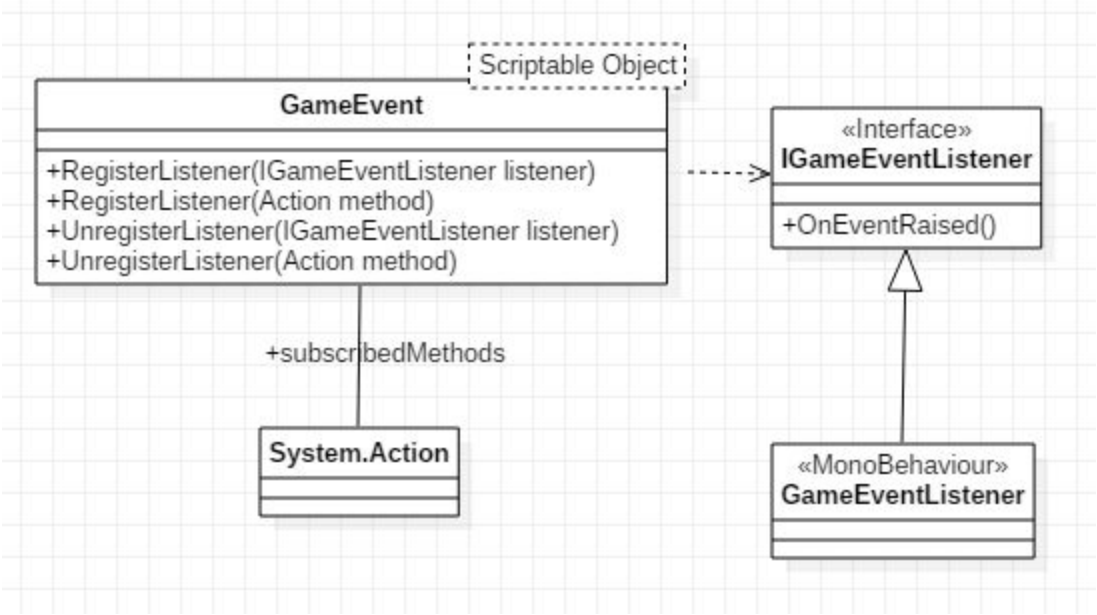


Figura 21. Diagrama de clases del sistema de eventos.

Inclinado por el patrón Observable Observador que tan buenos resultados me ha dado, abstraí el GameEventListener que Ryan propone en una interfaz que cualquier MonoBehaviour podía implementar. De esta forma cualquier clase podía registrarse a sí misma como oyente de un evento.

Pero fui a más, y en el ecuador del desarrollo, fruto de una revisión de sprints anteriores, refactoricé el sistema para hacerlo aún más versátil. Aunque no me fue necesario utilizarlo de esta manera, pero, ¿y si un mismo MonoBehaviour quisiera responder a uno o más eventos, y hacerlo de maneras diferentes? Como no es plausible implementar IGameEventListener dos veces, sobrescribí los métodos de suscripción para que aceptaran también delegates de C#, concretamente Actions callbacks. De esta forma puedo pasar métodos como argumento en el registro de los eventos. Internamente la clase Game Event fue modificada para que almacenara los eventos de esta manera y los métodos que recibían IGameEventListeners como argumento fueron modificados para almacenar el método OnEventRaised, en lugar del objeto en sí. Así también ahorré refactorizar los usos ya hechos del sistema de eventos, además de dotarlo de más flexibilidad.

Además de las características anteriormente mencionadas, se desarrolló también la clase de Editor para visualizar en el inspector de los Game Events qué listeners están suscritos a ese Evento en cada momento. Dicha funcionalidad se ilustra en la figura siguiente: [40]

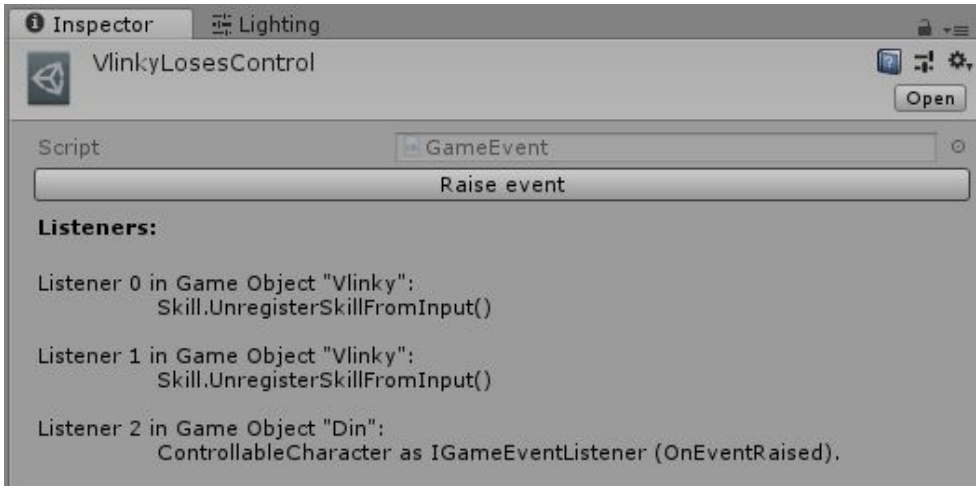


Figura 22. Custom Inspector para Game Event.

Como se puede observar en la figura, hay 3 objetos esperando a que Vlinky abandone el control. Dos de ellos son sus habilidades, empujar y disparar, para desactivarse; y el tercero es el personaje controlable Din, para tomar él el control.

Esta ejemplo es perfecto para ilustrar la filosofía detrás de RNF_10.

| **Custom Inspectors y clases de Editor** |
| --- |
| Si bien Unity ofrece ciertos atributos para personalizar un poco la forma en que el Inspector muestra los objetos, las cosas que no alcanza a hacer mediante atributos pueden ser implementadas libremente extendiendo la clase Editor, con el atributo CustomEditor.<br>• Estas clases no son incluidas en la build final del videojuego.<br>• Para ser detectadas, el archivo que las contiene debe estar dentro de una carpeta llamada “Editor”, que puede estar ubicada en cualquier parte del proyecto e, incluso, pueden existir varias. [41] |

## Paso de mensajes en Unity: Unity Events, GameObject.SendMessage() y Unity's Messaging System

El Game Design Document especifica en sus reglas que Vlinky no puede moverse si está disparando. ¿Debería entonces existir una dependencia en el código que desactivara el movimiento de Vlinky al activarse el disparo? Desde luego que no. Pero, ¿es necesario entonces crear y emitir un Game Event para comunicarse dos componentes que, en los casos definidos en el Game Design, estáran contenidos en el mismo Game Object? Es una posibilidad, pero en mi opinión no merece la pena crear un evento que solo va a tener un oyente. ¿Qué opciones quedan?

Unity tiene tres características que aportan soluciones a dicho escenario: los tipos Unity Events y Unity Actions, y los métodos "Send Message" implementado en los Game Objects y el Messaging System. En el caso que procede, he optado por utilizar la primera opción, tal y como se verá más adelante.

**GameObject.SendMessage("MethodName"):** Existen tres variantes: SendMessage, que envía el mensaje a los componentes del GameObject, BroadcastMessage, que envía el mensaje a los hijos del GameObject, y SendMessageUpwards, que envía el mensaje a los ancestros del GameObject hasta llegar a la raíz. La debilidad de este sistema es que confía en la reflexión para llamar a los mensajes, con lo cual es difícil trazar los errores. Es por esa razón que el siguiente sistema es propuesto en la documentación oficial como una alternativa más fiable. [42]

**Messaging System:** en esencia es muy parecido al implementado anteriormente. Mediante la implementación de una interfaz definida que a su vez hereda la clase IEventSystemHandler, Unity se encarga de marcar ese componente como receptor de mensajes y envía las llamadas a él mediante la clase estática ExecuteEvents, tal y como explica la documentación. Se descartó el uso de este sistema porque no es configurable desde el editor y porque su funcionalidad ya la cubren los Game Events antes explicados. [43]

**UnityEvent class**. la clase Unity Event encapsula los eventos de C# y los hace serializables, es decir, se pueden mostrar y configurar desde el editor, por lo tanto es más amigable para los diseñadores. Es por esa razón que se ha decidido utilizar este sistema. En la siguiente figura se muestra cómo se ve una variable del tipo Unity Event llamada "Response" en el inspector: [44]

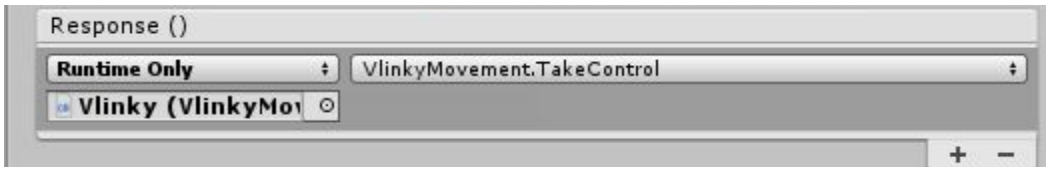


Figura 23. Inspector de Unity Event.

Pulsando en los botones + y - se pueden definir más acciones que ocurrirán al ser ejecutado dicho evento. De esta forma, los diseñadores pueden modelar comportamientos y relaciones sin necesidad de escribir nuevo código, de la misma forma que con los Game Events y Game Events Listeners. De hecho, se puede observar como, en la solución propuesta por Ryan también se utilizan los Unity Events para definir la comunicación a nivel de inspector.

## Personajes

Los personajes Vlinky y Din pueden son representados en el mundo del videojuego mediante una colección de componentes englobados en un Game Object con su mismo nombre y su misma Tag.

Pero antes de continuar con la exposición de los sistemas, cabe mencionar que, debido a que los Game Objects que componen el juego son contenedores de Componentes, serializados y almacenados por Unity y no una clase en sí, no tienen un equivalente en el estándar UML para su documentación.

Para representar los componentes de los Game Objects, me he basado en los diagramas de componentes que exponen unos compañeros de la Universidad Complutense de Madrid en su proyecto "Producción de un Videojuego Multijugador en Unity Combinando

los Géneros MOBA y RTS". En el siguiente apartado se pueden ver las figuras que ilustran esta forma de documentación en un ejemplo práctico. [37]

## Vlinky

| **Directorios** |
|---|
| Both\Assets_Both_Scripts\Characters\Vlinky |
| Both\Assets_Both_Scripts\Characters\Pushable |
| **Clases desarrolladas** |
| VlinkyMovement |
| VlinkyShotMode |
| PushAgent |
| GlobeCarriable |

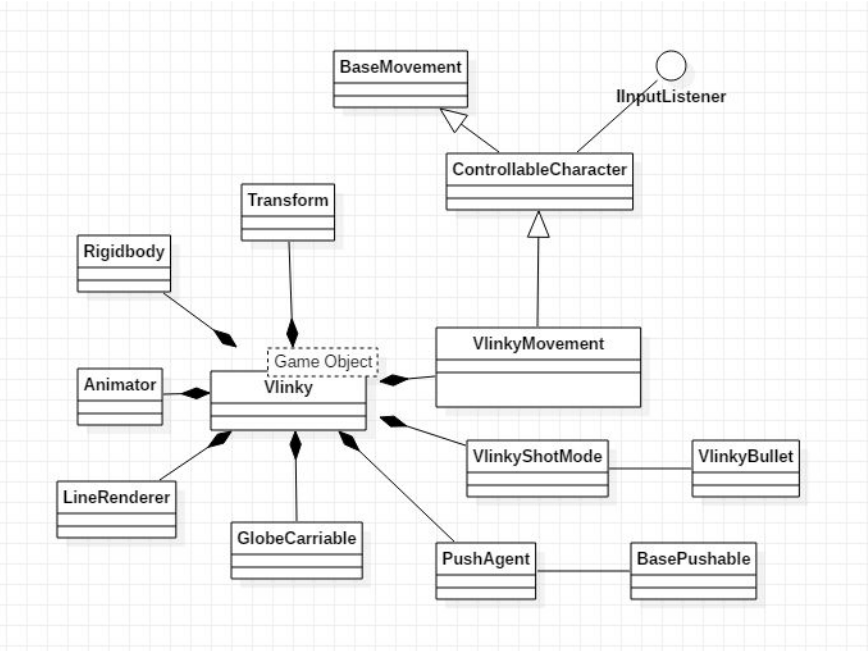


Figura 24. Diagrama de clases de Vlinky.

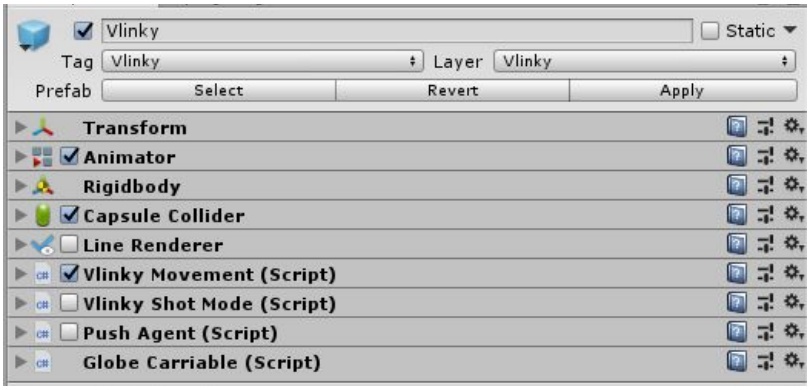


Figura 25. Inspector de Vlinky.

## VlinkyMovement

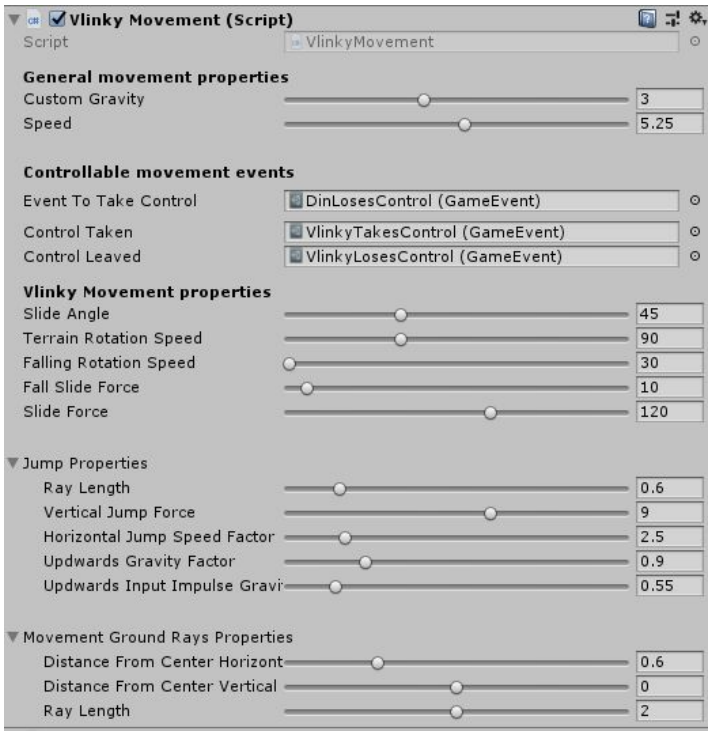


Figura 26. VlinkyMovement.

Como se puede ver en la figura anterior, y al igual que en las siguientes, para cada componente se han exprimido las funcionalidades de Unity para permitir la modificación

de sus valores desde el inspector por parte de los diseñadores. Los puntos de interés que expone el inspector del componente son:

1. Evento que lo activa, cuando Din pierde el control.
2. Eventos que emite al tomar y perder el control.
3. Raycasts para adaptar a Vlinky a la inclinación del suelo.
4. Parámetros del movimiento: velocidad, inclinación máxima que puede subir, fuerza del salto, etc.

Cabe destacar que existen múltiples maneras de mover a un personaje. En este caso se ha optado por realizar el movimiento mediante un Rigidbody y limitar las físicas de la simulación para hacerlo cómodo de manejar. Otra opción sería implementar el componente de Unity CharacterController, sistema en el cual sería necesario programar todas las físicas desde cero, ya que careceríamos de Rigidbody. Mientras que con la solución implementada se gana en rapidez de implementación y físicas realistas, la alternativa del CharacterController ofrece mayor personalización y precisión en los movimientos. [45, 46]

## VlinkyShotMode

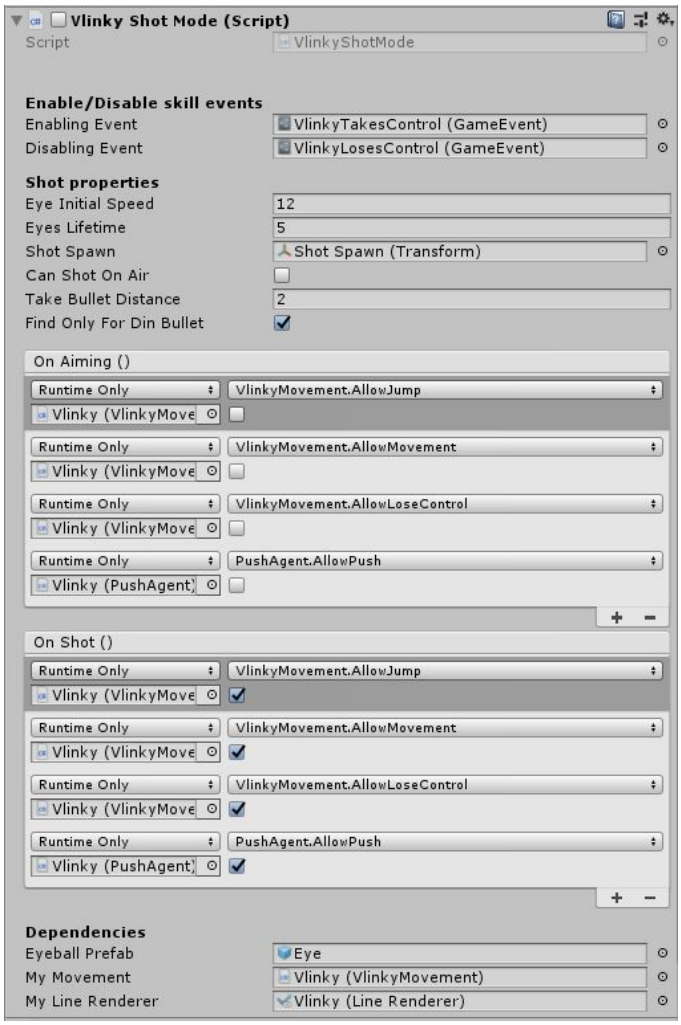


Figura 27. Vlinky Shot Mode.

**Nota:** aunque en la figura anterior parece que el modo disparo depende de VlinkyMovement, no es así, pues esa variable espera un tipo BaseMovement. Esto es debido a que necesita conocer la posición del cuerpo que dispara para buscar VlinkyBullets cercanos, y porque al disparar es necesario conocer el Rigidbody asociado para aplicar la inercia a la bola. Entonces, ¿por qué se ha decidido implementar una asociación con BaseMovement en lugar de dos, una con Rigidbody y otra con Transform? Se ha decidido hacer así porque, en esa ocasión, se quiere limitar la flexibilidad del modo disparo, sin ser

una mala decisión de diseño. Se hace así para evitar que el objeto que dispara aplique al proyectil una inercia que no es la suya sino la de otro rigidbody ajeno asociado, y porque no tendría sentido, al menos en el Game Design Generado, que al coger el objeto VlinkyBullet lo teletransportara a una posición arbitraria que no es la suya.

Los aspectos más destacables de la configuración del modo disparo son:

- Personalización del Shot Spawn: los ojos no tienen por qué salir del mismo centro del cuerpo.
- Depende del componente Line Renderer para pintar la trayectoria en la que se va a disparar.
- Aunque el Game Design especifica que solo puede lanzar a Vlinky, y no a objetos, la estructura de clases está preparada para que cualquier objeto que tenga el componente VlinkyBullet será susceptible de ser cogido por Vlinky. Esta característica se puede desactivar, bien teniendo solo como VlinkyBullet a Din, bien desmarcando la casilla booleana “Find only for Din bullet”.
- Para satisfacer las reglas del Game Design, emite los eventos On Aiming cuando comienza a apuntar, al entrar en modo disparo y On Shot, cuando dispara el ojo o la VlinkyBullet. Gracias a este tipo de eventos el diseñador puede configurar el personaje en base a las reglas del Game Design Document sin necesidad de generar dependencias fuertes en el código. [44]
- Los eventos “On Aiming” y “On Shot” están preparados para comunicarse con la futura capa de game HUD, para aplicar efectos como vibración de cámara al disparar o enfoque de cámara al apuntar.

## Push Agent

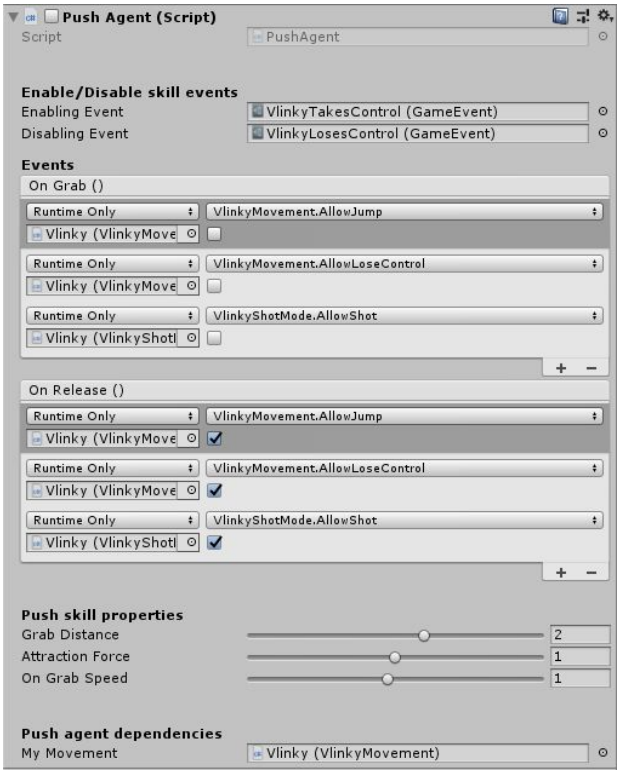


Figura 28. Push Agent.

- Push Agent emite los eventos Unity "On Grab" y "On Release" para su comunicación con el resto de habilidades, y también para una futura capa de Game HUD donde fuera necesaria cualquier interacción con esta.
- El inspector expone también una referencia a un BaseMovement para mover al Push Agent hacia el objeto a empujar si no está completamente pegado a este.

## GlobeCarriable

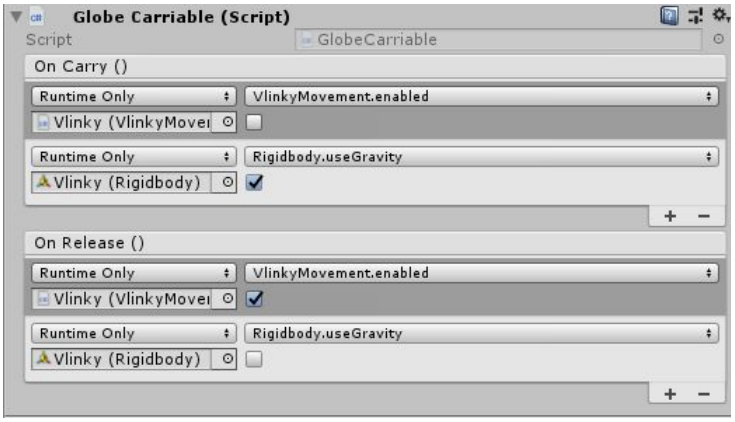


Figura 29. Globe Carriable.

GlobeCarriable es un componente muy sencillo que permite a DinGlobe reconocer los objetos que pueden ser cogidos. En el caso de la demo solo lo implementa Vlinky, pero también respondería a cualquier otro Game Object que lo utilice.

## Din

| **Directorios** |
|---|
| Both\Assets_Both_Scripts\Characters\Din |
| Both\Assets_Both_Scripts\Characters\Shot Din Interaction |
| **Clases desarrolladas** |
| DinMovement |
| DinGlobe |
| VlinkyBullet |
| ClimbOnVlinky |

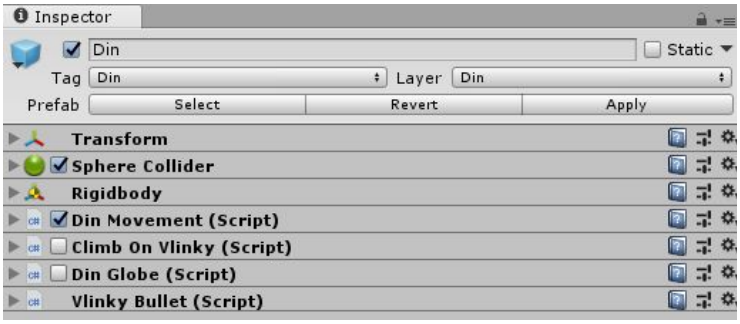


Figura 30. Inspector de Din.

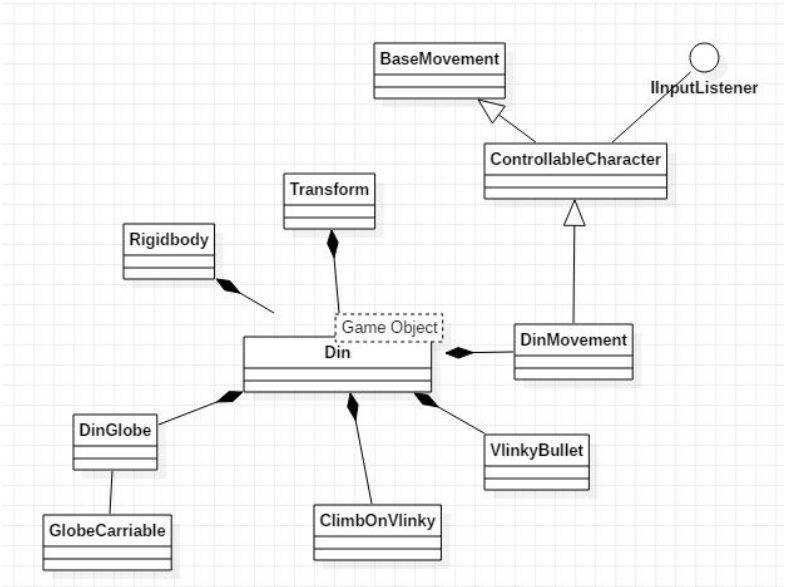


Figura 31. Diagrama de clases de Din

## DinMovement

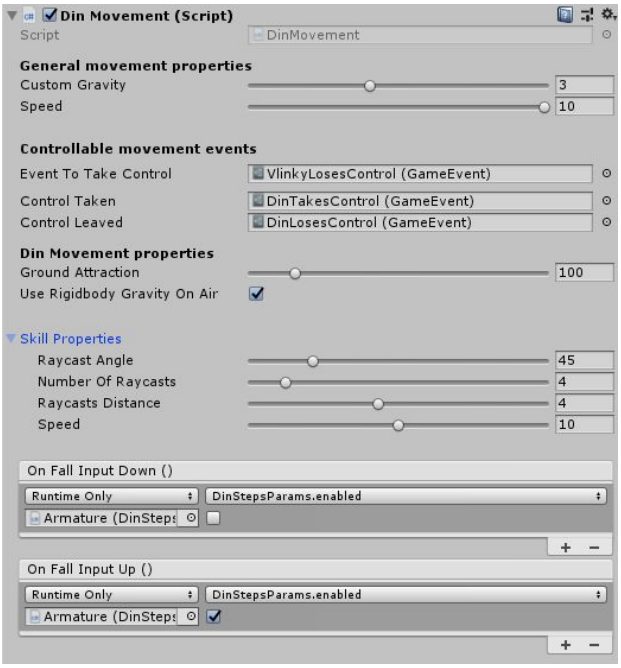


Figura 32. Din Movement.

Mientras que Vlinky sigue un esquema de movimiento mucho más tradicional en videojuegos, el del personaje que se mueve por el suelo, salta, y empuja; el movimiento de Din es el de una lapa que se pega a paredes y techos. Este comportamiento se ha conseguido comprobando en cada frame las superficies de terreno que está tocando. En caso de estar tocando terreno, la dirección de la gravedad se aplica en dirección a esa superficie. En caso de no estar en contacto con nada, se aplica hacia abajo.

Su habilidad, en lugar de salto, consiste en estirar sus patas y pegarse a la superficie que tiene encima de él. Esos parámetros son controlados en el apartado Skill Properties. Para reconocer si tiene o no superficie encima de él lanza un abanico de Raycast que, en caso de colisión, decide el punto objetivo y cambia la gravedad de Din hacia allí.

En la figura que ilustra sus componentes, al principio del apartado, se puede apreciar que no consta del componente Animator. La razón de su ausencia viene dada porque Din, al girar físicamente su cuerpo completo, no responde bien a las técnicas tradicionales de

animación, que no se pueden adaptar con eficacia a tantos posibles ángulos. Para superar este problema se explorarán en el futuro soluciones de Inverse Kinematics (IK) que animen el esqueleto del personaje proceduralmente, adaptándolo así al terreno y no dependiendo de animaciones tradicionales.

En el caso que nos ocupa, esta versión de Din ya tiene implementados unos componentes que generan una animación procedural lo suficientemente vistosa para la demo. Son prototipos a la espera de ser sustituidos por técnicas de IK, pero han ayudado a que Din no parezca un objeto estático. Los componentes que lo animan son:

- **DinLeg.** Componente asociado a cada una de sus piernas, con métodos para moverlas a los puntos indicados.
- **DinStep**. Componente asociado también a cada una de sus piernas, junto a DinLeg. Utiliza Raycasts para escanear el alrededor de la pierna y ver si procede pegarla a alguna superficie. En caso de que sí, el componente responsable del movimiento es DinLeg. No solo se encarga de moverla, sino también de rotarla y escalarla para que quede bien pegada a la pared y algo achatada, simulando el comportamiento de una ventosa de verdad.
- **DinStepParams**. Clase fachada para configurar los parámetros de las piernas hijas.
- **FixedRotation**. Este componente evita que el objeto rote, aunque lo haga su jerarquía. Está asociado al ojo central de Din.
- **Breathe**. Componente que anima el Game Object al que está asociado en base a una función seno que avanza con el tiempo. De esa forma, simula una respiración.

En la siguiente figura se observa claramente la deformación de las ventosas pegadas al suelo, además de que el ojo sigue horizontal a pesar de haber girado al personaje.

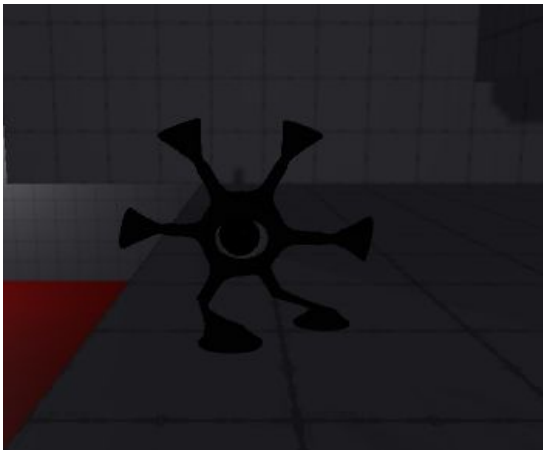

Figura 33. Ventosas de Din.

## VlinkyBullet

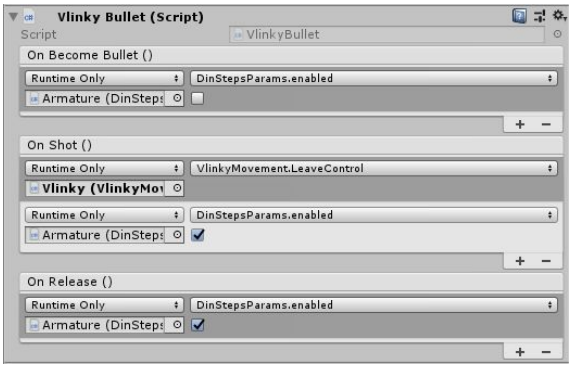


Figura 34. Vlinky Bullet.

VlinkyBullet es el componente que buscará VlinkyShotMode para cogerlo y dispararlo. Además, emite una serie de eventos útiles para definir qué ocurre en qué momentos del disparo. Aunque en VlinkyShotMode ya existían eventos emitidos al disparar y al coger balas, resulta más ordenado si cada objeto también emite los suyos propios. En caso contrario, los diseñadores tendrían que definir los eventos de todos los objetos en el inspector de Vlinky, lo cual extendería la longitud del componente en pantalla muchísimo, y lo haría más difícil de mantener.

## ClimbOnVlinky

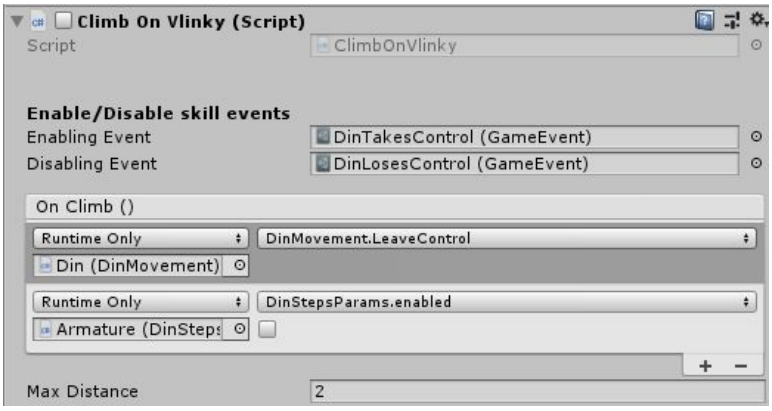


Figura 35. Climb On Vlinky.

Si bien VlinkyBullet ya implementa la funcionalidad necesaria para que Vlinky coja a Din, solo permite a Din interactuar con Vlinky como un elemento pasivo. ¿Qué pasa si queremos que Din también pueda subirse activamente encima de Vlinky, tal y como está definido en el Game Design y en los requisitos funcionales? Entonces emerge la necesidad de una habilidad propia de Din que busque a un componente cercano VlinkyShotMode para solicitarle que lo coja. Este es un caso en que se ha optado por agregación en lugar de herencia, ya que como C# no soporta herencia múltiple era imposible para ClimbOnVlinky ser habilidad y bala al mismo tiempo (es decir, heredar a la vez de VlinkyBullet y de Skill).

## DinGlobe

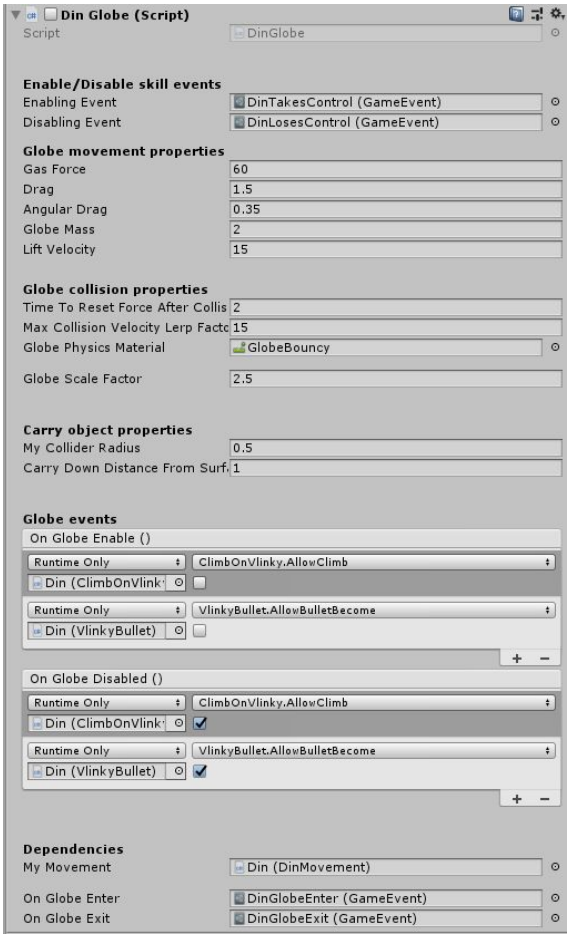


Figura 36. Din Globe.

Aunque el inspector sea largo y Din Globo de la impresión de funcionar prácticamente como un personaje independiente, se ha determinado no implementarlo como tal porque la funcionalidad que implementa el componente es muy sencilla: basta con desactivar el movimiento asociado a la habilidad, que en este caso son las físicas de lapa de Din, y activar los elementos del Rigidbody que no se estaban utilizando, excepto la utilización de gravedad. Es esa configuración tan simple la que provee al globo de un movimiento tan suave y realista, ya que los cálculos físicos de ese movimiento no son responsabilidad suya sino del Rigidbody asociado.

- Como se esperan varios eventos que responderán a la transformación de Din en globo, como cambios en la interfaz, en la cámara, en los comportamientos de los futuros enemigos…; se han creado dos Game Events nuevos que notifiquen cuando la transformación ocurre.
- La lista de parámetros del principio son los parámetros que se le aplicarán al Rigidbody al activarlo. Como Unity no permite que los Game Objects tengan más de un Rigidbody, dichos parámetros se tienen que definir en el componente activador, recordando los anteriores para restablecer el estado.
- Los Unity Events “On Globe Enable” y “On Globe Disabled” comunican la habilidad con el resto del personaje. Así, la escucha de los eventos Game Event se limitará a oyentes de otros sistemas.

# Sistema de Input

## Unity's built-in Input System

| **Directorio** |
| --- |
| Both\Assets_Both_Scripts\Input System |
| **Clases desarrolladas** |
| InputManager |
| IInputListener |
| PlayerInput |

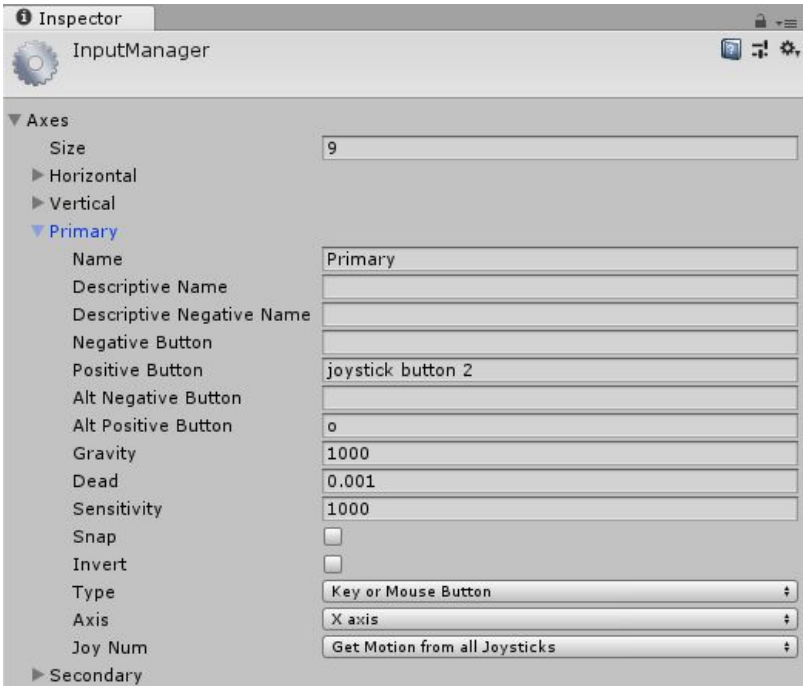


Figura 37. Unity Input Manager.

En Unity, la clase estática Input provee en todo momento del input del jugador. Como se explicó al principio del apartado "Diseño Software de Both", utilizarla sin una clase o sistema wrapper acabará generando un código difícil de mantener. Por ello se ha decidido implementar un sistema propio.

Aún así no todo ha sido construido desde cero. Por debajo de ese sistema se sigue utilizando la clase Input, que a su vez hace uso del Input Manager que los proyectos de de Unity utilizan por defecto.

Este sistema es poco flexible, ya que dificulta demasiado modificar la asignación de las teclas desde dentro del juego; y está escuchando en todo momento a todos los inputs a la vez (por ejemplo teclado, mando del jugador 1, mando del jugador 2, ratón...); lo cual no es deseable, tanto por optimización de recursos como porque puede dar lugar a malas experiencias de juego (como por ejemplo que el jugador mueva el ratón sin querer mientras juega con el mando). La prueba de que el sistema no cumple con las expectativas de los desarrolladores es que Unity lleva desde 2017 desarrollando un nuevo InputSystem, que a fecha de hoy sigue bajo desarrollo activo. [47, 48, 49]

Aún así, se ha optado por sí utilizar esta parte del sistema de input de Unity para ahorrar tiempo de desarrollo, y porque no es necesaria mayor complejidad para la demo.

Otra carencia que tiene el sistema de input de Unity es que promueve malas prácticas de diseño. A saber:

- La explicada anteriormente. Una clase estática, Singleton, usada por todo el proyecto da lugar a errores difíciles de detectar y dificulta el mantenimiento.
    - **Solución**: encapsular Input en un sistema propio diseñado para el escenario concreto.
- Además de obtener el input del usuario mediante los botones definidos en el Input Manager de las opciones del proyecto (figura anterior), también permite obtenerlo directamente del teclado mediante la clase estática KeyCode. Esto incrementa los problemas del punto anterior, sobre todo si se combinan KeyCodes con Botones desestructuradamente.
    - **Solución**: no obtener el Input del usuario mediante KeyCodes directamente , a no ser que se construya un sistema que los encapsule de forma adecuada para evitar la libertad absoluta a la hora de utilizarlos.

- Por último, un último detalle es que el Input Manager, en su configuración por defecto, falsea los ejes de movimiento (Horizontal y Vertical, asignados a WASD y a las flechas) para hacerlos parecer un joystick real. Aunque esa fluidez en el movimiento conseguida sin esfuerzo atrae al desarrollador novato, trae problemas si es usada en conjunto con un sistema de movimiento basado en físicas, como el implementado en nuestros personajes. Si se quiere que el personaje acelere suavemente, se debe implementar las funcionalidades físicas pertinentes (en este caso aceleración y velocidad, que en nuestro caso vienen dadas por el Rigidbody), y no asignar esa responsabilidad al sistema de input.
    - **Solución**. modificar los parámetros “Gravity” y “Sensitivity”, de los ejes Horizontal y Vertical asociados a las teclas, y setearlos a un número muy alto, como 1000 (originalmente tienen un valor por debajo de 1). Esos valores determinan cuánto tarda el botón en alcanzar su valor extremo y retomar su valor por defecto. Con valores altos, es instantáneo, y se comportarán como en el resto de teclas, donde los valores por defecto son también 1000, como se puede observar en la figura anterior.

## Input System de Both

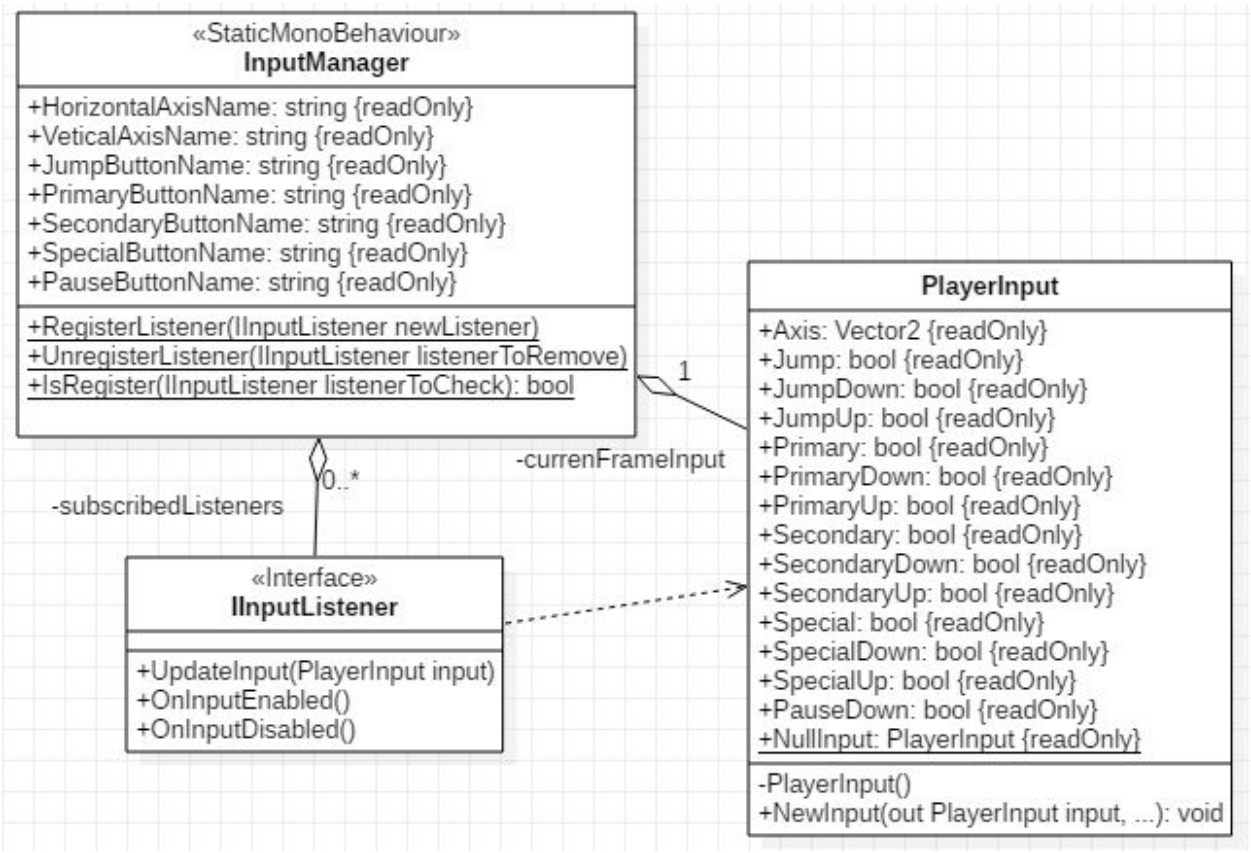


Figura 38. Diagrama de clases de Input System.

Consta de:

- **InputManager**: Singleton estático que persiste entre escenas. Su única interfaz pública es para registrar o desregistrar IInputListeners. Se evita el registro doble de un mismo IInputListener porque puede dar lugar fallos difíciles de detectar. En Update(), lee constantemente la entrada del usuario a través de la clase estática Input utilizando exclusivamente los métodos Input.GetButton e Input.GetAxis, evitando utilizar KeyCodes.
- **PlayerInput**: clase que recopila los estado de cada Input
    - Button: representa si el botón está siendo pulsado en ese frame.

    - ButtonDown: representa si el botón ha sido pulsado ese frame por primera vez.
    - ButtonUp: representa si el botón ha sido despulsado ese frame.
- **IInputListener**: es la interfaz de la que heredan los oyentes de Input, que en el caso que nos ocupan son tan solo los dos personajes mediante la implementación de la clase abstracta ControllableCharacter

**Optimización**: como Unity no provee de callbacks ni eventos que sean llamados en la Input del usuario,  InputManager está ejecutándose siempre cada frame para comprobar dicha input. Como además se trata de una clase central en el diseño, se ha optimizado para evitar reservar memoria con la creación de nuevas PlayerInput en cada frame. Esto se ha conseguido haciendo el constructor privado en PlayerInput y rellenando la PlayerInput pasada por referencia al método NewInput. A su vez, InputManager almacena su propia variable privada PlayerInput que tan solo se inicializa al principio.

## Singleton design pattern en Unity

El patrón Singleton en Unity puede implementarse como se haría fuera de él, pero si en su funcionalidad requiere de eventos de Unity como Awake(), Start() o Update(), entonces será necesario que la clase Singleton herede de MonoBehaviour.

Para su implementación, por tanto, es necesario seguir los mismos pasos que para la de cualquier otro componente: ligarlo a un Game Object en la escena. Para ello, en su función Awake, tan pronto como se carga la escena, la clase crea una instancia de sí misma y la asigna a la variable Singleton. En caso estar ya asignada, no solo no la crea, sino que también destruye su Game Object, ya que significará que existe otro Game Object en la escena con una instancia de la misma clase.

El tener que heredar de MonoBehaviour puede resultar un problema, ya que obliga a publicar al resto de sistemas la interfaz de MonoBehaviour también, es decir, el resto de sistemas podrán acceder a su GameObject, modificar sus componentes o incluso destruirlo. Cuando no se necesite conocer estas interfaces en el resto de clases, lo ideal es

que su variable Singleton quede oculta. Por ello, he ocultado la variable de instancia de la interfaz pública.

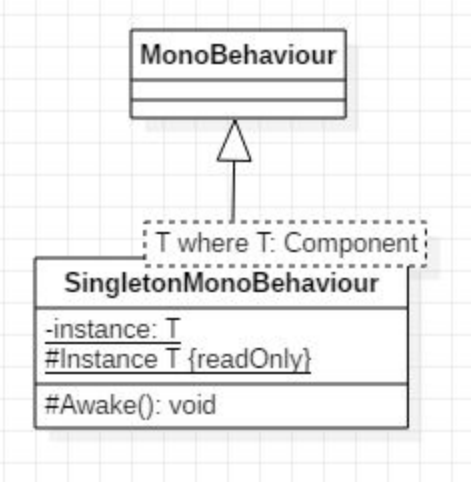


Figura 39. Diagrama de clases del patrón Singleton en Unity.

## Unity callbacks en clases no MonoBehaviour

Si Unity permitiera suscribirse a los callbacks que se disparan en cada frame, como Update(), no sería necesario heredar de MonoBehaviour para obtener esa funcionalidad, y sería más fácil la implementación del caso anterior. Sin embargo no es imposible, existe una manera de implementar esa funcionalidad: mediante un único SingletonMonoBehaviour que gestione dichos callbacks y publique los métodos necesarios para suscribirse a ellos.

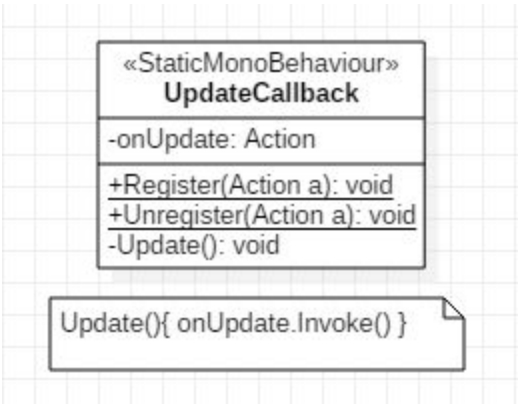

Figura 40. Diagrama de clases de la propuesta Callback Manager.

A pesar de ser una solución válida, no se ha implementado para la demo que nos ocupa porque solo se hubiera utilizado para la clase InputManager. Si en un futuro vuelve a surgir la necesidad de clases Singleton, se estudiará su inclusión más en profundidad, haciendo pruebas de rendimiento y comparando con la solución de usar directamente SingletonMonobehaviours.

## Sistema de activadores y activables

| **Directorio** |
|---|
| Both\Assets_Both_Scripts\Gameplay\ActivatorActivableSystem\ |
| **Clases desarrolladas** |
| IActivable |
| Activable |
| Activator |

Una de las características de la demo, definida en el Game Design Document y en la Ingeniería de Requisitos, es que existan puertas que se abran por botones. Como se dijo anteriormente, al comienzo de la sección “Diseño Software de Both”, este tipo de sistemas son muy comunes en la resolución de puzles en videojuegos y se espera que surjan más mecánicas que encajen en la mecánica de esta misma abstracción.

Una puerta es un objeto activable, que puede ser activada bien por un botón, una palanca, al pasar por un sitio, por la muerte de un enemigo, la resolución de un puzzle, etc.

Un botón es un objeto activador, que puede activar tanto una puerta, como un ascensor, un cofre del tesoro, una pantalla, una luz, etc.

Para que las puertas y botones desarrollados para la demo encajen dentro de esta abstracción, se ha desarrollado el sistema que ilustra la siguiente figura. En la figura se incluyen además los componentes desarrollados que implementan el sistema.

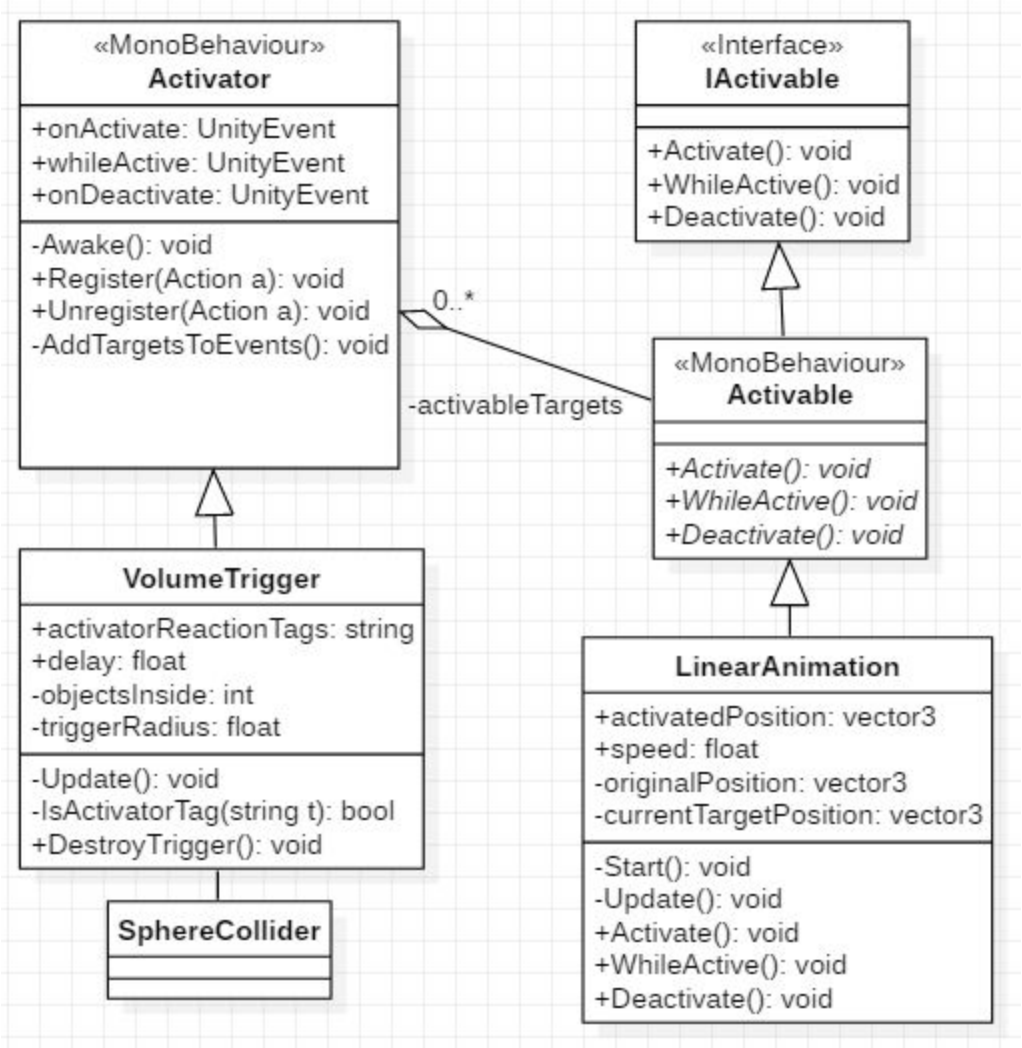


Figura 41. Diagrama de clases del sistema de activables activadores.

- VolumeTrigger será usado por el GameObject “Botón” para detectar cuando alguien se posa sobre él.
- LinearAnimation será usado por el GameObject “Puerta” para definir un movimiento hacia la posición definida como posición activada. Además de para animar la puerta, también es usado por el propio GameObject “Botón” para animar la pulsación del mismo.

# Discusión final y conclusiones

## Conclusiones de la producción de la demo técnica

El desarrollo y la consecución de objetivos del TFG en cuanto al desarrollo de la demo técnica planificada ha sido un éxito rotundo. Se han satisfecho todos los requisitos, gracias a los cuales se ha conseguido una experiencia de usuario fluida durante el desarrollo de los pocos minutos que componen la experiencia. Además, también han sido integrados con éxito el conocimiento teórico de los marcos expuestos en los diferentes apartados que componen el TFG, ayudando así a la productividad, la planificación, la mantenibilidad y la escalabilidad futura del proyecto. De esta manera, se da por cerrada la etapa de producción de la primera demo técnica de Both.

Como se ha mencionado a lo largo del documento, el desarrollo del videojuego completo que engloba a esta demo continuará, tanto por parte del autor como por parte del equipo artístico de Gambusino Labs; siendo el siguiente objetivo más inmediato la extensión de la presente demo hasta la feria Level Up! Granada a finales de Septiembre.

Por último, la producción de este software ha potenciado enormemente los conocimientos del autor sobre el motor gráfico Unity, concretamente los referidos a la generación de una base de código estable y escalable. La condición de estar desarrollando una demo técnica dentro del marco de un desarrollo más grande, ha motivado especialmente el cuidado de la mantenibilidad y extensibilidad del software, llevando los conocimientos aprendidos en el grado de Ingeniería del Software hasta sus últimas consecuencias.

## Conclusiones de la Ingeniería de Software aplicada al Desarrollo de Videojuegos

La consecución de objetivos del TFG en cuanto a la investigación y documentación de los diferentes ámbitos del desarrollo del videojuego también ha sido un éxito rotundo. Como se dijo en la sección de Introducción, la intención del autor no era tan solo la finalización del software, sino la profundización teórica en los pilares en que se apoya tanto el

desarrollo de videojuegos como el desarrollo software en general. Dicha investigación queda resuelta a lo largo de los distintos apartados que se exploran en la memoria.

La conclusión última que obtengo de este trabajo y el fruto de toda la investigación y producción elaboradas, lo expongo a continuación. Como explico en el apartado "Ingeniería del Software en el desarrollo de videojuegos", a pesar de sus puntos en común y de ser ambos compatibles, pues en última instancia se está hablando del desarrollo de un software, la investigación realizada concluye que la utilización de herramientas y estándares difiere en ambos ámbitos mucho más de lo que difieren ambos desarrollos en la práctica. Esta fisura práctica la achaco a los años de ventaja que lleva el desarrollo software tradicional al desarrollo de videojuegos, cuyo crecimiento más grande ha sido visto en los últimos años, mientras que la proliferación de la informática viene ocurriendo desde años atrás.

Por las razones expuestas, puedo concluir que el campo del desarrollo de videojuegos necesita de mayor investigación teórica y estandarización en sus procesos para así acercarse al enfoque práctico de cualquier otro software. Probablemente esto se deba no solo a su vejez, sino también a su historia, ya que hasta hace poco los videojuegos no eran considerados una industria seria; y por ello los profesionales dedicados a su desarrollo no tenían por qué haberse especializado necesariamente en el campo de la Ingeniería. Prueba de ello son los grados específicos en desarrollo de videojuegos, o incluso cruzados entre Ingeniería Informática y Desarrollo de Videojuegos, que están surgiendo poco a poco actualmente, y que empiezan a ser una alternativa para aquellos estudiantes que quieran especializarse exclusivamente en este campo. Gracias a los profesionales surgidos de estos nuevos estudios, el estado del arte de los videojuegos avanzará los próximos años en una dirección más ingenieril y menos desorganizada; lo cual ayudará a que los inversores y la industria vean en estos proyectos apuestas más fiables para su inversión. [50]

A modo de opinión personal, auguro que en un futuro próximo el estado del arte de este ámbito avanzará en la dirección ingenieril antes mencionada hasta el punto de que comenzará a hablarse de ello como una nueva área de conocimiento independiente de las demás: la Ingeniería del Videojuego (Game Engineering). Al fin y al cabo todos los esfuerzos

actuales en ambos ámbitos, así como las competencias específicas de este TFG, no son más que aproximaciones teóricas a dicha nueva Ingeniería; que si ya existiera éstos no serían necesarios. Me parece muy curioso que ni siquiera una búsqueda en Google arroje resultados a este término, así que me aventuro a afirmar que es algo que está por venir.

# Agradecimientos

No puedo dar cierre a este TFG sin antes mencionar a las personas que lo han hecho posible.

Quiero agradecer a  Marcelino Cabrera Cuevas, director del TFG, por guiarme y asistirme en su consecución y ayudarme a conseguir la alta calidad del presente documento, aún cuando el tiempo corría en nuestra contra. Además también quiero agradecerle por algo que se escapa a su competencia de director, y es que durante el año 2016-2017 él tutorizó mi estancia Erasmus en Florencia. Fue durante ese año que yo empecé a aprender por mi cuenta sobre Desarrollo de Videojuegos, y eso fue posible gracias su esfuerzo por librarme del peso de la burocracia académica que este tipo de becas supone. Para mí, has cumplido con todas las expectativas.

Los siguientes son al equipo artístico de Gambusino Labs: Fernando Montero Valdivieso, David Montero Valdivieso y Daniel Sánchez de la Torre, por brindarme la oportunidad de embarcarme con ellos en este viaje y sus esfuerzos constantes por dotar a Both de una calidad artística exquisitas. Esta demo es solo un paso más en el camino, a por todas.

A Javier Izquierdo Vera, compañero en la carrera y en la vida, porque sin él habría tardado más en llegar a donde estoy y habría sido mucho menos divertido. Gracias por aquella última vez que me ayudaste en Informática Gráfica de tercero, precisamente a construir un cubo en OpenGL, porque desde entonces despegué y, probablemente de no ser por aquella noche de asistencia antes de la práctica, no estaría ahora escribiendo este TFG. Así funcionan los efectos mariposa, así que continuemos con la asistencia.

A los profesores de la carrera que me han ayudado a superarme y aprender más y más. Ellos saben quienes son, pues apuesto a que al principio pensaron que suspendería su asignatura, pero luego consiguieron que aprendiera, que es de lo que se trata.

Y por último, como no podría ser de otra manera, a mis amigos, a mi amor, y a mi familia, que son las personas que están ahí constantemente apoyándome incluso cuando estoy fuera del código. ¡Chapó por vosotros!

# Referencias y bibliografía

1. Crecimiento de la industria del videojuego en España.
   https://www.hobbyconsolas.com/reportajes/industria-videojuego-espana-bate-records-facturacion-270081
2. Qué es un pitch en el desarrollo de videojuegos.
   http://www.gamedonia.com/blog/game-pitch-how-to
3. Inside.
   http://www.playdead.com/games/inside/
4. Unravel.
   https://www.ea.com/games/unravel
5. Brothers: a Tale of Two Sons.
   https://www.starbreeze.com/games/brothers-a-tale-of-two-sons/
6. Dark Souls & Bloodborne environmental storytelling.
   https://www.linkedin.com/pulse/minimalist-storytelling-bloodborne-dark-souls-iii-why-nick-jones
7. Teorías en las redes sobre el final de Inside.
   https://kotaku.com/the-wild-theories-behind-insides-secret-ending-1783552341

8. Steam Direct, para publicar videojuegos en Steam.
https://partner.steamgames.com/steamdirect

9. GDC: Voronoi Splitscreen.
https://www.youtube.com/watch?v=tu-Qe66AvtY

10. Voronoi Splitscreen: implementation.
https://www.shadertoy.com/view/4sVXR1

11. Unity: High Definition Render Pipeline.
https://blogs.unity3d.com/es/2018/03/16/the-high-definition-render-pipeline-focused-on-visual-quality/

12. Libro Blanco del Desarrollo de Videojuegos en la Comunidad Valenciana 2018
http://www.dev.org.es/images/stories/docs/libro%20blanco%20comunidad%20valenciana.pdf

13. Diseño de campaña de Crowdfunding
https://vanacco.com/lanzar-crowdfunding/

14. Libro Blanco del Desarrollo Español de Videojuegos 2017.
http://www.dev.org.es/images/stories/docs/libro%20blanco%20dev%202017.pdf

15. Red.es programa de Impulso al Sector del Videojuego.
http://www.red.es/redes/es/actualidad/magazin-en-red/redes-abre-una-convocatoria-que-movilizar%C3%A1-625-millones-de-euros-para-el

16. Tipos de sociedades.
https://infoautonomos.eleconomista.es/tipos-de-sociedades/sociedad-civil-caracteristicas-ventajas/

17. Definición de mecánicas de juego a alto nivel
http://aev.org.es/las-reglas-el-elemento-mas-importante-del-juego-2/

18. Definición formal de mecánicas de juego a más bajo nivel
http://gamestudies.org/0802/articles/sicart

19. Game Design Document Template
http://wwwx.cs.unc.edu/Courses/comp585-s11/585GameDesignDocumentTemplate.docx

20. Guía docente Dirección y Gestión de Proyectos (ETSIIT)
https://grados.ugr.es/informatica/pages/infoacademica/guias_docentes/curso_actual/cuarto/ingenieriadelsoftware/etsiit_gii_dgp_1718_dirgestproy/!

21. SCRUM: Agile prototyping.
https://www.atlassian.com/blog/agile/agile-design-prototype

22. SCRUM: Stand Up Meeting.
https://www.mountaingoatsoftware.com/agile/scrum/meetings/daily-scrum

23. Geekbot.io for standups meetings.
https://geekbot.io/

24. SCRUM: Sprints and Milestones.
https://pm.stackexchange.com/a/22516

25. HacknPlan: project management for videogames.
https://hacknplan.com/

26. Unity official assets.
https://assetstore.unity.com/publishers/1

27. Qué es un SCRUM master.
http://www.ceolevel.com/scrum-master-que-es-y-que-no-es

28. COCOMO Guide.
http://csse.usc.edu/csse/research/cocomoii/cocomo2000.0/cii_modelman2000.0.pdf

29. COCOMO II: aplicación usada.
http://csse.usc.edu/tools/cocomoii.php

30. Computer Games and Software Engineering
https://www.ics.uci.edu/~wscacchi/Papers/New/Intro-ComputerGames+SoftEngr-Chapter-2015.pdf

31. Components architecture
https://gamedevelopment.tutsplus.com/articles/unity-now-youre-thinking-with-components--gamedev-12492

32. The rise of Systemic Games
https://www.youtube.com/watch?v=SnpAAX9CkIc&t=42s

33. Components architecture 1
http://cowboyprogramming.com/2007/01/05/evolve-your-heirachy/

34. Components architecture 2
http://gamearchitect.net/Articles/GameObjects1.html

35. “La orientación a objetos descompone el comportamiento de los objetos en base a qué funcionalidad necesitan, en base a lo que son”.
http://www.laboratoryspokane.com/openchord/2011/09/unity-and-the-component-model/

36. Unity Object
https://docs.unity3d.com/ScriptReference/Object.html

37. Producción de un Videojuego Multijugador en Unity Combinando los Géneros MOBA y RTS
https://eprints.ucm.es/26495/1/Producci%C3%B3n_de_unvideojuegomultijugador.pdf

38. Unite Austin 2017 - Game Architecture with Scriptable Objects
https://www.youtube.com/watch?v=raQ3iHhE_Kk&t=7s

39. Ryan Hipple
https://github.com/roboryantron/Unite2017

40. Unity Editor's API
https://docs.unity3d.com/ScriptReference/Editor.html

41. Unity Custom Editors
https://docs.unity3d.com/Manual/editor-CustomEditors.html

42. GameObject.SendMessage()
https://docs.unity3d.com/ScriptReference/GameObject.SendMessage.html

43. Messaging System
https://docs.unity3d.com/Manual/MessagingSystem.html

44. Unity Events
https://docs.unity3d.com/ScriptReference/Events.UnityEvent.html

45. Unity Character Controller
https://docs.unity3d.com/es/current/Manual/class-CharacterController.html

46. Unity 3D - Rigidbody VS CharacterController [UnityQuickTips]
https://www.youtube.com/watch?v=AEPI5rmg3XY

47. Unity built-in Input Manager
https://docs.unity3d.com/Manual/class-InputManager.html

48. Unity new input system blog
https://blogs.unity3d.com/es/2016/04/12/developing-the-new-input-system-together-with-you/

49. Unity new input system repo
https://github.com/Unity-Technologies/InputSystem

50. Carreras de videojuegos en España
https://www.educaweb.com/carreras-universitarias-de/videojuegos/

# Índice de figuras