UNIVERSITAT POLITÈCNICA DE VALÈNCIA

etsinf

Escola Tècnica Superior d'Enginyeria Informàtica

Escola Tècnica Superior d'Enginyeria Informàtica

Universitat Politècnica de València

AR CATAPULT: DESARROLLO DE UN VIDEOJUEGO PARA DISPOSITIVOS MÓVILES EN UNITY

Trabajo Fin de Grado

Grado en Ingeniería Informática

Autor: Daniel Torres Inglés

Tutor: Antonio Garrido Tejero

Tutor Externo: Javier Giménez Aznar

2019-2020

Resumen

Se desarrollará AR Catapult, un videojuego para dispositivos móviles en el que el usuario hace una foto al rostro de una persona, selecciona una localización cercana a él en un mapa y lanza un avatar con la cara de la persona a la que se le ha realizado la foto. Tras lanzar dicho avatar, el usuario debe conseguir una serie de puntos y esquivar una sucesión de enemigos hasta alcanzar la posición seleccionada previamente en el mapa y caer en una diana que le incrementará su puntuación en función de su distancia respecto al centro. Este videojuego se hará en el motor Unity usando el lenguaje C#.

Palabras clave: Ingeniería del software, Metodologías Ágiles, Unity, C#, Dispositivos móviles, Android, IOS.

Abstract

AR Catapult will be developed, a video game for mobile devices in which the user takes a picture of a person's face, selects a location near him on a map and launches an avatar with the face of the person to whom the photo was taken. After launching said avatar, the user must get a series of points and avoid a succession of enemies until they reach the previously selected position on the map and fall on a target that will increase their score based on their distance from the center. This video game will be made on the Unity engine using the C # language.

Keywords : Software engineering, Agile methodologies, Unity, C#, Mobile devices, Android, IOS.

Tabla de contenidos

1. **Introducción** **9**
1.1. Motivación 9
1.2. Problemática 9
1.3. Objetivo 11
2. **Estado del Arte** **12**
2.1. Contexto 12
2.2. Motores 17
2.2.1. Unreal Engine 4 17
2.2.2. Unity 18
2.2.3. GameMaker Studio 2 20
2.3. Referencias 21
2.3.1. Pokémon Go 21
2.3.2. Filtros de reconocimiento facial 23
3. **Marco Teórico** **24**
3.1. Unity 24
3.1.1. Editor 24
3.1.2. GameObject 25
3.1.3. Prefab 26
3.1.4. Escena 27
3.1.5. Componentes 29
3.1.5.1. Colliders 29
3.1.5.2. Rigidbody 30
3.1.5.3. SpriteRenderer y MeshRenderer 30
3.1.5.4. Animator 31
3.1.5.5. Camera 32
3.1.6. Script 32

3.1.7. ScriptableObject 34

3.1.8. Interfaz gráfica de usuario 34

3.1.9. Time 35

3.2. Patrones de Diseño 36

3.2.1. Comman 36

3.2.2. Pool 36

3.2.3. Inyección de dependencia 36

4. **Desarrollo** **38**

4.1. Planificación 39

4.2. Análisis y diseño Orientado a Objetos 39

4.3. Diseño de Interfaz gráfica de usuario 40

4.4. Implementación 43

5. **Conclusión** **53**

6. **Bibliografía** **54**

1. Introducción

El 1 de abril de 2019 tuve el honor de comenzar mi carrera profesional en Wildframe Media S.L. una empresa dedicada a la creación de videojuegos. Desde sus inicios, bajo la marca Brave Zebra, la empresa valenciana creó juegos para terceros, pero en el año 2018, bajo el nombre de Digital Sun, lanzaron su primer juego propio: Moonlighter.

Más de medio millón de copias avalan Moonlighter como un éxito y la marca Digital Sun supone el futuro de Wildframe Media; relegando a Brave Zebra a su pasado, dado que los estudios de Brave Zebra y Digital Sun están formados por las mismas personas y son más rentables los esfuerzos invertidos en Digital Sun. Durante esta memoria se expondrán los conocimientos obtenidos durante el desarrollo de AR Catapult, el último juego desarrollado por Brave Zebra, el estudio encargado de realizar videojuegos para terceros. AR Catapult supone el cierre de una era y un modelo de negocio. Sin lo aprendido en Brave Zebra el estudio Digital Sun no sería la marca internacionalmente conocida que es hoy en día.

1.1. Motivación

En un sector formado en su mayoría por empresas jóvenes de pequeño tamaño que a su vez requiere de personal muy especializado un proyecto como AR Catapult nos permite aumentar las competencias de la plantilla. Desde poner en práctica conocimientos teóricos relativos a la programación orientada a objetos y los patrones de diseño, hasta la mejora de soft skills como son el trabajo en equipo y la comunicación entre profesionales tan dispares como son programadores, artistas y diseñadores, pasando por las metodologías ágiles, el uso de herramientas como Unity, Rider y Slack, la integración en dispositivos móviles y el diseño de interfaces de usuario.

Además, al ser un proyecto financiado por un cliente, aporta la estabilidad económica de la cual, en líneas generales en España, carece el sector. Y si el cliente queda satisfecho tras la entrega final, aumenta el bagaje de la empresa, lo que facilita la contratación por parte de futuros clientes.

En resumen, un proyecto como AR Catapult permite financiar, con un riesgo mínimo, el aprendizaje de nuevos integrantes y prepararlos para futuros proyectos.

1.2. Problemática

Si bien el desarrollo de un software nunca es sencillo, en el caso de los videojuegos hay un factor más a tener en cuenta: la jugabilidad. La jugabilidad es el conjunto de reglas o sistemas que rigen el videojuego. En un videojuego el software está sujeto a la jugabilidad. Durante la preproducción se define la jugabilidad. Si no se alcanza una jugabilidad que satisfaga, en este caso, al cliente, se deberá alargar el periodo de preproducción, lo que provocaría que o se dediquen más recursos a la producción total del producto, lo que aumentaría los costes, o se reduzca el tiempo

empleado en otras fases del proyecto. El desarrollo de AR Catapult tiene unos plazos de preproducción concretos, en los que hay margen para la corrección de la jugabilidad, pero este margen no es infinito, por lo que existe el riesgo de exceder dicho plazo si no se alcanza una jugabilidad satisfactoria.

Además, AR Catapult es un videojuego para dispositivos móviles, lo que hace más complejo el desarrollo respecto a videojuegos de ordenador o consolas. Un ejemplo de esta complejidad extra lo hayamos en la interfaz, la cual debe ser mucho más legible y simple, sin perder funcionalidad, al haber móviles de solo 3" hasta 10", al no haber un estándar de *aspect ratio* para la pantalla (16:9, 18:9, 4:3, 21:9, 19.5:9) y haber dispositivos con *notch.*

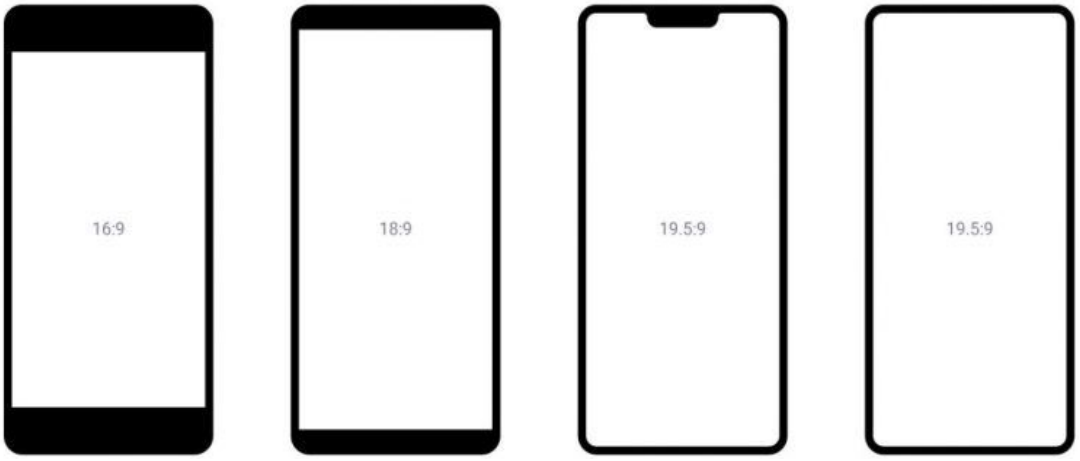


Figura 1. Visualización de aspect ratio y notch en dispositivos móviles.

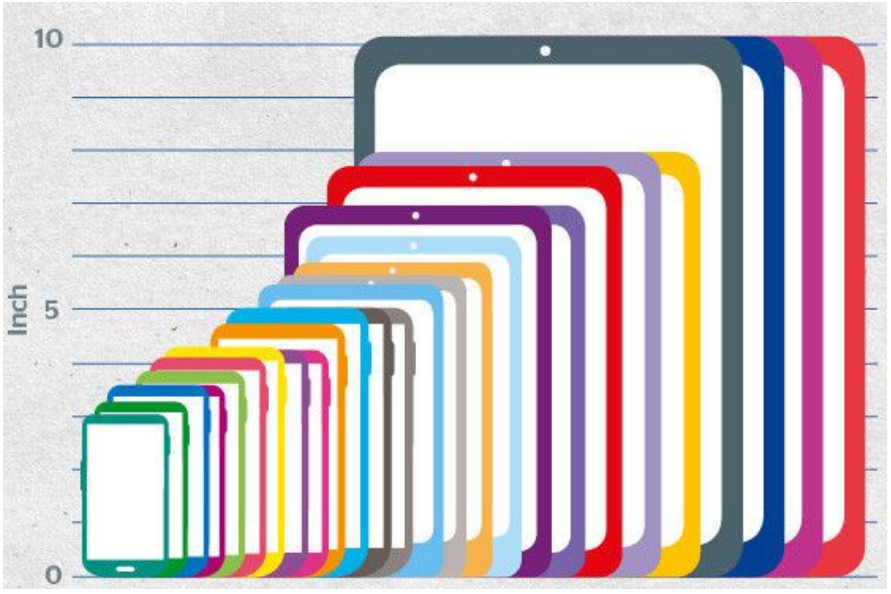


Figura 2. Tamaño de pantalla dispositivos móviles de Samsung.

1.3. Objetivo

El objetivo es la creación, en los plazos estipulados, del videojuego AR Catapult, un videojuego para dispositivos móviles en el que el usuario hace una foto al rostro de una persona, selecciona una localización cercana a él en un mapa y lanza un avatar con la cara de la persona a la que se le ha realizado la foto. Tras lanzar dicho avatar, el usuario debe conseguir una serie de puntos y esquivar una sucesión de enemigos hasta alcanzar la posición seleccionada previamente en el mapa y caer en una diana que le incrementará su puntuación en función de su distancia respecto al centro.

Para lograrlo, debemos cumplir los siguientes subobjetivos:

1. Implementación del software responsable de la jugabilidad.
   - 1.1. Sistema de movimiento del jugador.
   - 1.2. Sistema de movimiento de los enemigos.
   - 1.3. Sistema de colisiones.
   - 1.4. Sistema de puntuación.

2. Integración con dispositivos móviles.
   - 2.1. Sistema de controles del jugador: Pantalla táctil y giroscopio.
   - 2.2. Sistema de retratos: Cámaras trasera y delantera.
   - 2.3. Sistema de localización: GPS y reloj.

3. Implementación interfaz.
   - 3.1. Interfaz requerida para la jugabilidad.
   - 3.2. Interfaz de menús.

4. Implementación arte.
   - 4.1. Integración de imágenes.
   - 4.2. Sistema de gestión de animaciones.

2. Estado del Arte

La realidad del sector del videojuego es compleja y dispar. Para intentar entender el contexto en el que se realizará este proyecto se partirá desde una imagen general del sector con una perspectiva global, siguiendo por los mayores acontecimientos, videojuegos y compañías de los últimos años, hasta llegar a la situación del sector en España.

También se analizarán algunos de los motores de videojuegos más relevantes actualmente en el sector y la influencia de otros videojuegos y aplicaciones a la hora de crear AR Catapult.

2.1. Contexto

Ante todo, la industria del videojuego es una de las mayores industrias dentro del sector cultural y está en pleno crecimiento. En el año 2015, según datos ofrecidos por la ESA [1] en el Video Games in the 21st Century: The 2017 Report , los videojuegos aportaron 11,5 millones de dólares al PIB de Estados Unidos (un 3.68% más que en el año 2013) y en China, el primer país en volumen de mercado, el tamaño del mercado alcanzó los 34.400 millones de dólares [2] (Figura 1).

#	País	Población (millones)	Tamaño del mercado (M$)
1	China	1.415	34.400
2	Estados Unidos	327	31.535
3	Japón	127	17.715
4	Corea del Sur	51	5.764
5	Alemania	82	4.989
6	Reino Unido	67	4.731
7	Francia	65	3.366
8	Canadá	37	2.399
9	España	46	2.202
10	Italia	59	2.168

Fuente: Newzoo

Figura 3. Principales 10 países en tamaño de mercado. Fuente: Libro blanco del Desarrollo Español de Videojuegos 2018 [2]

Este mercado, el chino, es uno de los más cerrados del mundo. Para entrar en este mercado es necesario hacerlo a través de empresas de dicho país, por lo que estas empresas, al ser la única puerta de entrada al mayor mercado del mundo, han crecido exponencialmente junto al tamaño del mercado. La empresa líder del país asiático es Tencent, la cual, según datos recopilados por Statista [3], cuadruplica en tamaño a empresas tan reconocibles como Nintendo.

Otra clave para entender el mercado chino es que la primera plataforma en porcentaje de mercado son los dispositivos móviles [4]. Principalmente los videojuegos gratuitos o *free-to-play* que basan sus beneficios en microtransacciones opcionales.

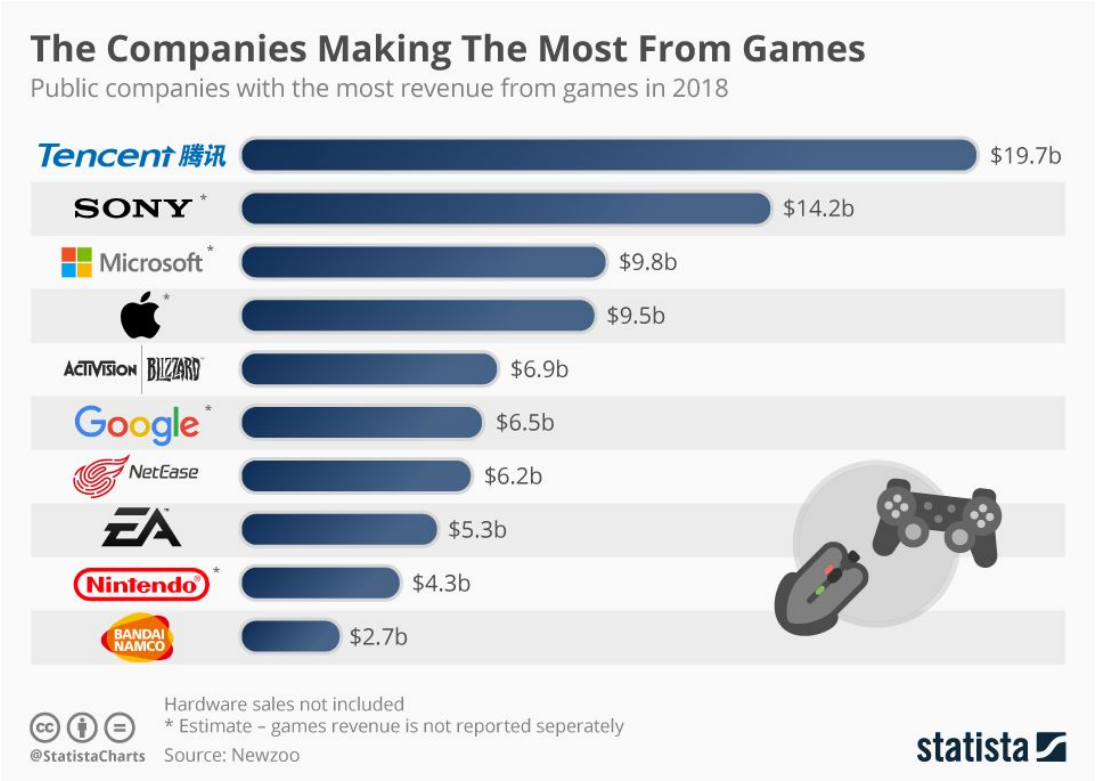


Figura 4. Tamaño Tencent en comparación a las mayores compañías del sector. Fuente: Statista [3]

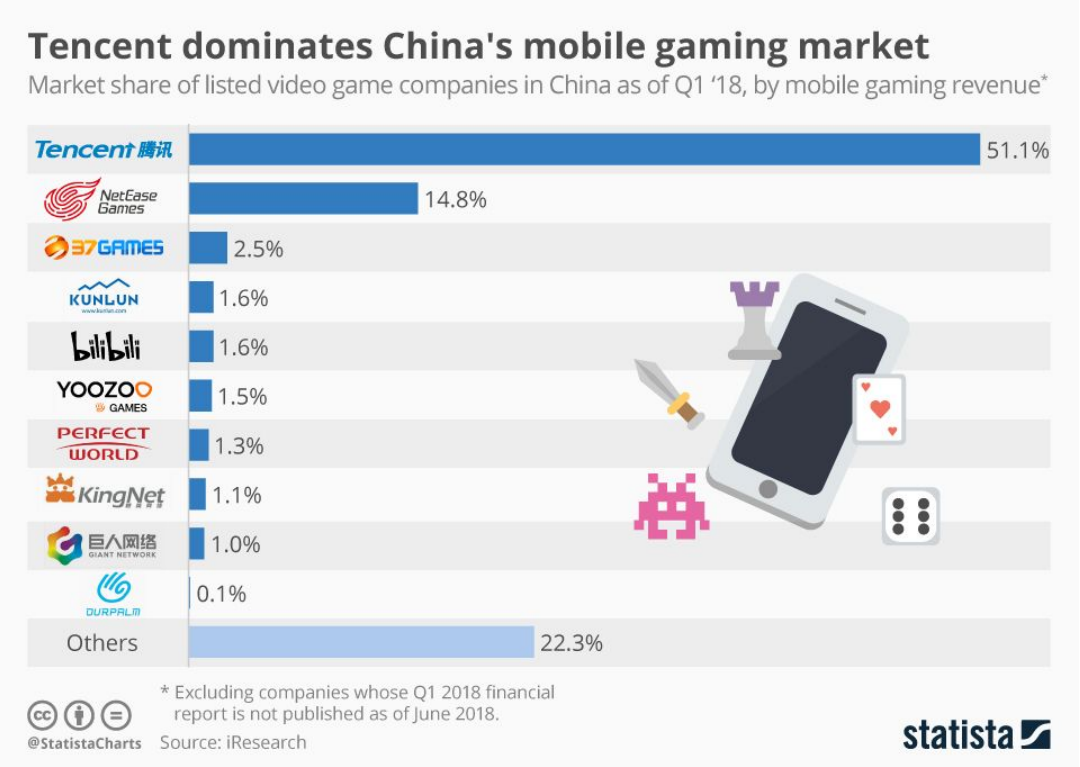


Figura 5. Porcentaje del mercado móvil en china por compañía. Fuente: Statista [4].

Un ejemplo de la envergadura de la industria del videojuego es Fortnite, un videojuego para ordenador, consolas y dispositivos móviles que superó en 2018 los 200 millones de jugadores [5] y recaudó un total de 2.400 millones de dólares según recoge SuperData Digital Games and Interactive Media Year in Review 2018 [6].

Otro caso de éxito es GTA V, el cual costó 265 millones de dólares [7] , siendo la segunda producción audiovisual más costosa de la historia, solo superada por Piratas del Caribe 3, y generó, según un informe de MarketWatch [8], 6.000 millones de dólares, tres veces más que Avatar o Vengadores: Endgame, las películas más taquilleras de la historia.

Sin duda ejemplos como los anteriores son prueba de la fuerza de una industria billonaria y globalizada. Pero también de su inestabilidad y precariedad laboral. En el año 2010 Rockstar, la compañía detrás del desarrollo de GTA V, recibió una carta abierta [9] escrita por las mujeres de los empleados que denunciaban tratos abusivos por parte de la compañía hacia sus empleados. En el año 2018 estas prácticas volvieron a salir a la luz debido a las largas jornadas de trabajo, conocidas como *Crunch*, que los empleados realizaban. Y es que el Crunch también asoló el día a día de los desarrolladores de Fortnite tras su inesperado éxito.

Asimismo, otro caso que repercutió negativamente a la industria fue el cierre de Telltale. Un compañía que tras el éxito alcanzado en un juego basado en la franquicia The Walking Dead y firmar acuerdos de licencia con franquicias tan importantes como Juego de Tronos, Batman, Minecraft, Guardianes de la galaxia o Stranger Things, sin previo aviso, en octubre de 2018 se declaró en bancarrota dejando sin empleo a más de

200 personas en San Francisco, una de las zonas con el nivel de vida más caro de todo Estados Unidos.

Ese mismo año, Activision-Blizzard, una de las mayores compañías del sector, despidió al 8% de su plantilla (800 trabajadores) a pesar de haber alcanzado beneficios históricos según su informe financiero: Activision Blizzard announces fourth-quarter and 2018 final results [10].

Casos como el de Fortnite, Rockstar, Telltale o Activision-Blizzard llevaron a la creación de Game Workers Unite en el año 2018. Un grupo de defensores de los empleados, no se definen como sindicato, del sector del videojuego que espera mejorar sus condiciones laborales.

La situación laboral de los empleados del sector no ha sido la única sombra que ha asolado la industria del videojuego en los últimos años. Las cajas de botín (o *loot boxes*) han sido una de las mayores polémicas. Las cajas de botín son un sistema de micropagos en el que el jugador paga una cantidad de dinero y recibe a cambio un recompensa aleatoria dentro del juego. El motivo de la polémica es la aleatoriedad, pues recrea en funcionalidad a los juegos de azar. Esto, junto al hecho de que muchos de estos juegos carecen de regulación o control y pueden ir dirigidos a niños, despertó las alarmas y llevó a países como Bélgica a prohibirlas. Pero en la mayoría de países siguen sin regulación, y juegos con cajas de botín como FIFA 20 o NBA 2K20 tiene una calificación por edades para mayores de 3 años (PEGI +3).

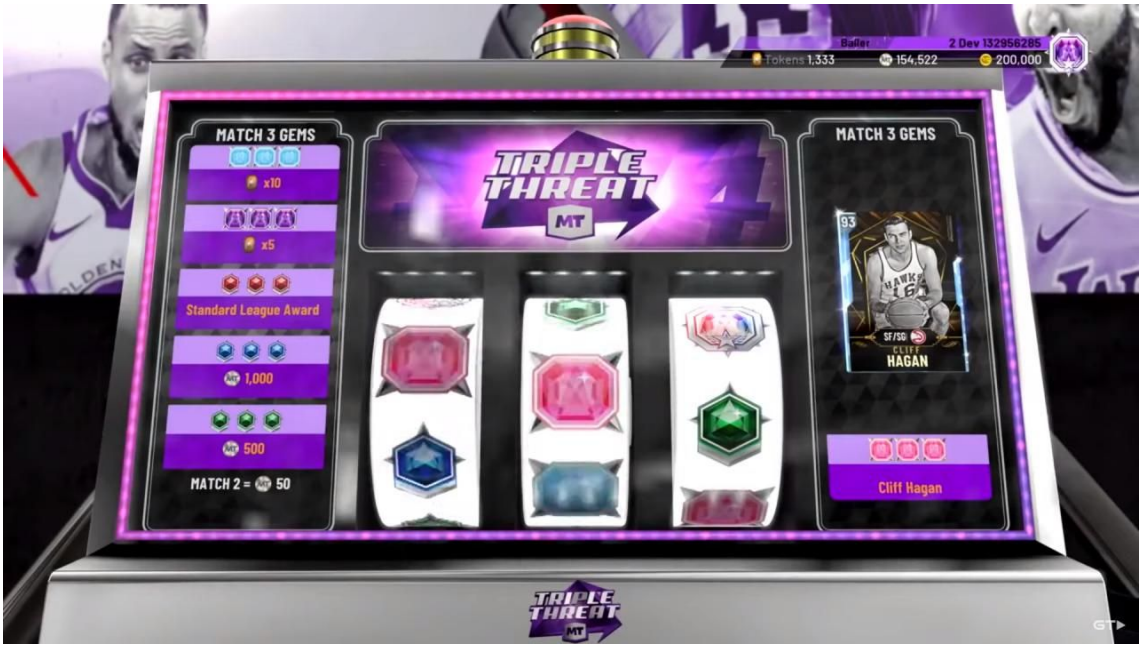


Figura 6. Trailer NBA 2K20 MyTEAM en el que se publicitan máquina tragaperras

Otro de los factores clave para entender la escena de la creación de videojuegos en los últimos años es la entrada de los videojuegos hechos por empresas indie. Si bien hace apenas 10 años la mayoría de los videojuegos eran creados por medianas o grandes empresas, ahora es común encontrar juegos realizados por un pequeño número de

personas o incluso una sola. Los factores clave para este auge de la escena indie son: la descarga digital , la difusión por internet y los motores de videojuegos con entrada gratuita.

La descarga digital viene dada de forma natural junto al aumento de las velocidades de descarga de internet. Si se puede descargar un juego en 15 minutos desde casa, ya no es necesario el disco del juego. Lo que reduce los costes de postproducción. Es decir, una compañía ya no tiene que estimar los beneficios y realizar un gran desembolso inicial para la distribución de las copias de su videojuego.

Plataformas como youtube, twitch o twitter, así como las tiendas online de Apple, Android, PlayStation, Xbox o Steam han facilitado el marketing a pequeños estudios que no podrían permitirse rivalizar con campañas de marketing realizadas por empresas millonarias.

Además, motores como Unreal Engine 4 o Unity, los cuales no es necesario comprar, sino que reciben un porcentaje de los beneficios de un videojuego, han facilitado la entrada a una nueva oleada de desarrolladores.

En el caso de Unity, un claro ejemplo de estudio indie que triunfó es Team Cherry, un pequeño estudio australiano formado por 3 personas que lograron alcanzar, según los propios desarrolladores, 1.8 millones de unidades vendidas de su primer videojuego: Hollow Knight. Estas ventas se alcanzaron en formato digital, a través de las tiendas mencionadas previamente y se promocionó en youtube, twitch y redes sociales.

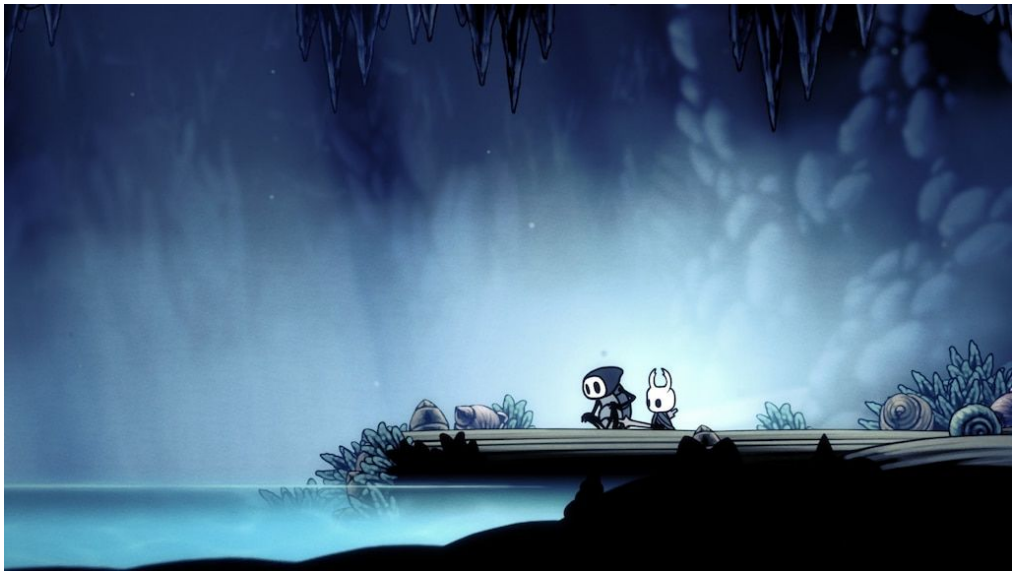


Figura 7. Imagen del videojuego Hollow Knight.

Todo lo mencionado anteriormente se aplica, en su medida, también a España, el cuarto mercado europeo y noveno mundial. Pero como sentencia el Libro Blanco del Desarrollo Español de Videojuegos 2018: “la industria española de desarrollo y

producción de videojuegos está todavía lejos de ocupar el lugar que le corresponde en el ranking internacional por el tamaño de su mercado.“

España es una industria en crecimiento (En 2017 facturó 713 millones de euros, un 15,6% más que 2016), formada por empresas jóvenes (El 81 % de las empresas activas se creó en los últimos 10 años) y pequeñas o *indie* (el 74 % emplea a menos de 10 personas). Además, la mayoría de las empresas autofinancian sus propios videojuegos, lo que lleva a que la mayoría de dichas empresas tengan una baja financiación. Todos estos datos están recogidos en el Libro Blanco del Desarrollo Español de Videojuegos 2018.

2.2. Motores

En la última década ha habido un auge del número de motores gráficos disponibles para los desarrolladores. Además, no es necesario hacer un desembolso inicial para utilizar la mayoría de dichos motores. Para retratar una imagen general de este ámbito del sector, analizaré tres de los motores más relevantes actualmente y sus pros y contras para utilizarlos en este proyecto.

2.2.1. Unreal Engine 4

El motor desarrollado por Epic Games lanzado en el año 2015 destaca por su gran potencia en el renderizado de entornos 3D que le permite desarrollar juegos de estilo foto realista, lo que le hace atractivo incluso para juegos de grandes compañías como son Jedi: Fallen order y Days Gone, desarrollados por estudios de Electronic Arts y Sony respectivamente (Figura 4).

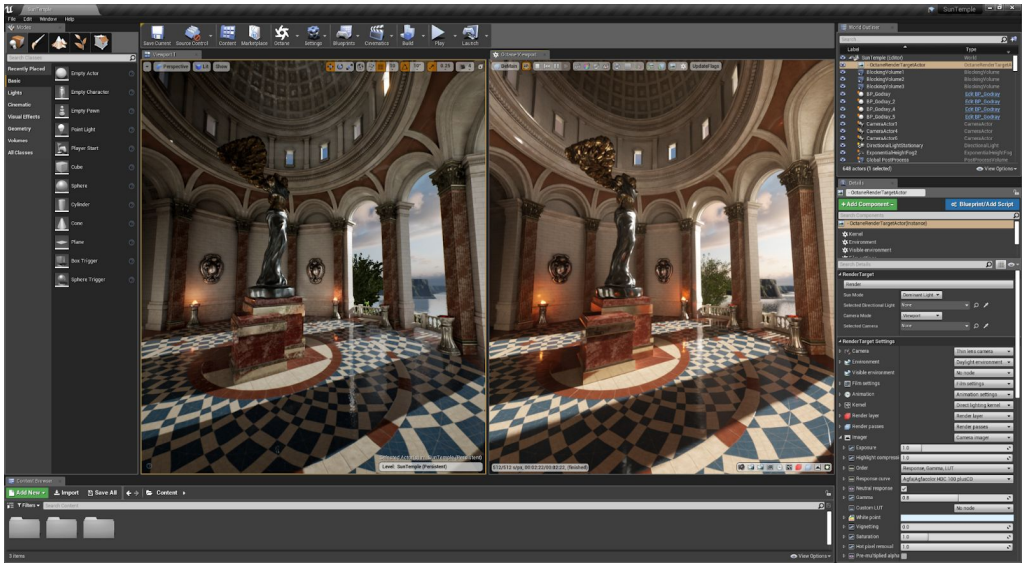


Figura 8. Motor de videojuegos Unreal Engine 4.

Al mismo tiempo, es capaz de recrear estilos más *cartoon* con gran fluidez como es el caso del famoso juego Fortnite, desarrollado por la propia

Epic Games. El cual es posible jugar en ordenador y consolas, pero también en dispositivos móviles.

Además, el sistema de código formado por módulos llamados Blueprints facilita la creación de código para gente del sector que no sea ingeniero software pero tenga unos conocimientos básicos de programación como, por ejemplo, los diseñadores. En caso de que se deseara, puede ser desarrollado en C++, al fin y al cabo, el sistema de Blueprints es solo una abstracción de este lenguaje.

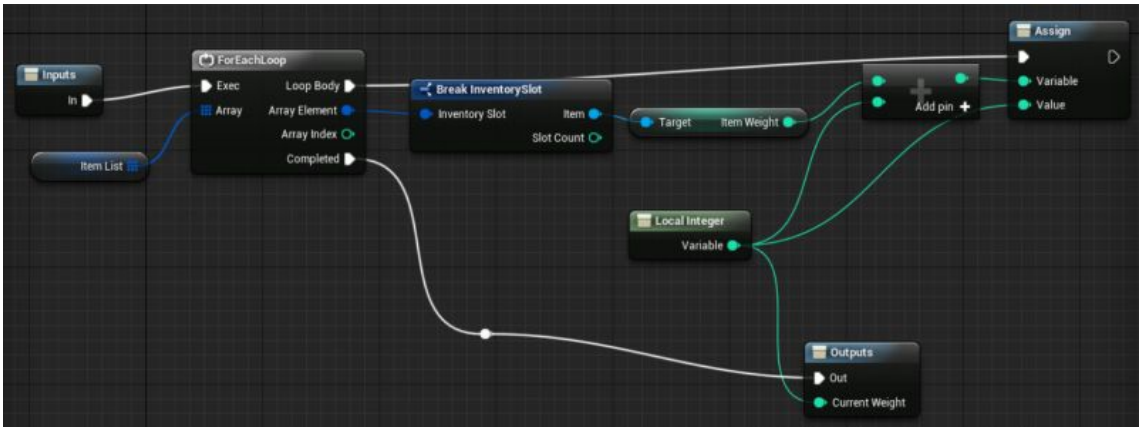


Figura 9. Blueprints.

Sin embargo, a pesar de que es posible desarrollar videojuegos 2D en Unreal, es más sencillo en otros motores como son Unity o GameMaker Studio 2. Además, como veremos más adelante, en Unity es más rápida la creación de 'builds' para dispositivos móviles.

2.2.2. Unity

Unity fue lanzado en 2015 por Unity Technologies, está programado en C++ y C, pero utiliza C# como lenguaje de desarrollo y soporta tanto 3D como 2D.

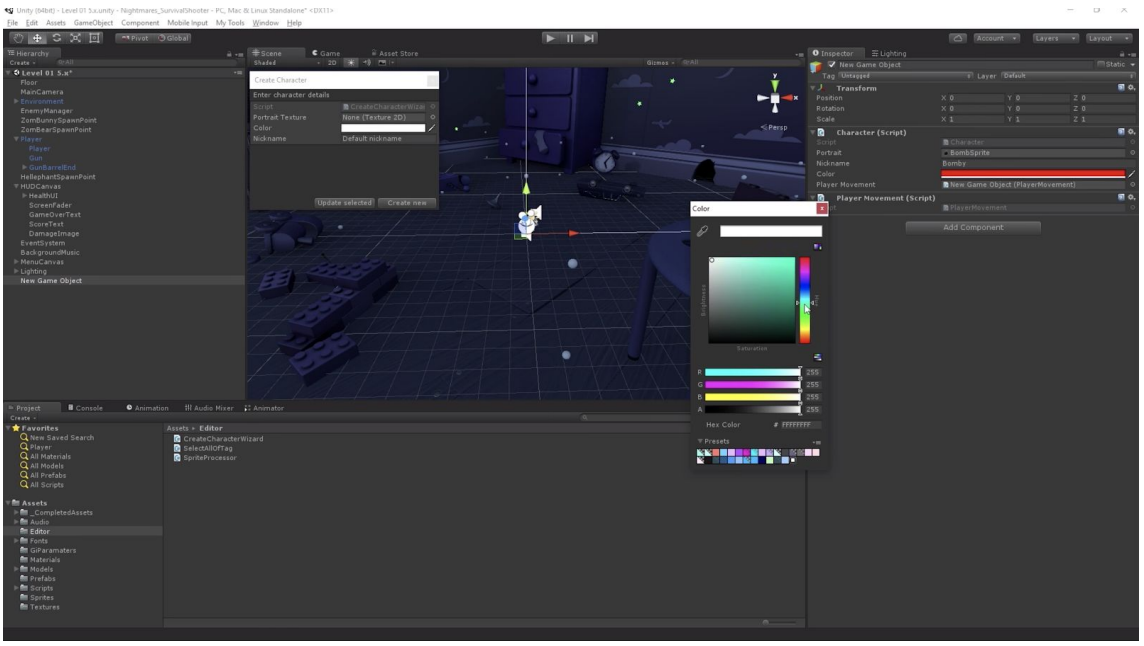


Figura 10. El motor de videojuegos Unity.

Una de las principales bazas de Unity fue desde sus inicios su accesibilidad, tanto a nivel económico (Unreal no fue gratuito hasta el año 2015) como de conocimientos en comparación a otros motores de gráficos 3D. Gracias a esta accesibilidad la comunidad fue creciendo y en consecuencia el motor.

Si bien al principio era conocido por ser utilizado por pequeñas compañías que realizaban juegos simples y con bastantes fallos, poco a poco dichas compañías se convirtieron en el sector indie, fruto de innovación y reconocimiento tanto por crítica como por público y Unity se convirtió en el abanderado de dichas compañías. Juego tan exitosos como Hollow Knight (mencionado previamente), Cuphead, Firewatch o Gris son ejemplos del impacto de este sector.

Figura 11. Imagen del videojuego Gris.

Algunos de los motivos por lo que es tan utilizado Unity son: su accesibilidad, la facilidad para la implementación en diferentes plataformas, gran variedad de plugins que amplían la funcionalidad del motor y la posibilidad de combinar tanto 2D como 3D.

El motivo principal para escogerlo como motor para desarrollar AR Catapult es, por un lado, que es el motor utilizado por el estudio por lo que se perdería parte de la motivación al aprender un motor que no va a ser utilizado fuera de este desarrollo y, por otro lado, la facilidad para hacer 'builds' en dispositivos móviles, con un solo botón se pueden probar sistemas como el de la Cámara, GPS o giroscopio que no pueden ser 'testeados' desde el ordenador.

2.2.3. GameMaker Studio 2

Yoyo Games lanzó en el año 2017 la segunda versión de su aclamado motor de desarrollo 2D. Esta nueva iteración traía una nueva interfaz, mejor implementación para consolas y mejor renderizado de las imágenes.

Dado que el motor está pensado para el desarrollo exclusivo de videojuegos 2D es mucho más sencillo de usar que motores como Unity o Unreal. Es por su simpleza que ha sido utilizado para desarrollar grandes juegos como son Nuclear Throne, Undertale o Hyper Light Drifter.

Figura 12. Imagen del videojuego Hyper Light Drifter.

En su contra tiene, que no puede ser utilizado para desarrollar videojuegos 3D, cosa que, si bien no es necesaria para AR Catapult, si lo es para el estudio en general, la poca versatilidad de plugins que aumentan las funcionalidades del motor, menor facilidad para hacer 'builds' que Unity y el utilizar un lenguaje de desarrollo propio, en contraposición a otros motores que pueden ser programados en lenguajes como C# o C++. Además, si bien tiene una versión de prueba que nos permite utilizar la gran mayoría de sus funciones, no tiene una entrada 100% gratuita como si tienen Unity o Unreal Engine 4.

2.3. Referencias

Ya sea tanto para la creación de una obra de arte, como para la invención de un producto, siempre hay referentes que te guían durante el desarrollo. En el caso de AR Catapult, al ser un videojuego diseñado para un cliente, estas influencias han venido marcadas en mayor medida por este último que por el equipo de desarrollo. Las dos principales referencias que han determinado en gran medida como es AR Catapult son el videojuego Pokémon Go y los Filtros de reconocimiento facial.

2.3.1. Pokémon Go

El juego desarrollado por Niantic para The Pokémon Company hizo su aparición en verano de 2016 y se convirtió rápidamente en un fenómeno de masas que llenó las cabeceras de los telediarios. Según estimaciones de Sensor Tower [11] superó los 3.000 millones de ingresos en el año 2019.

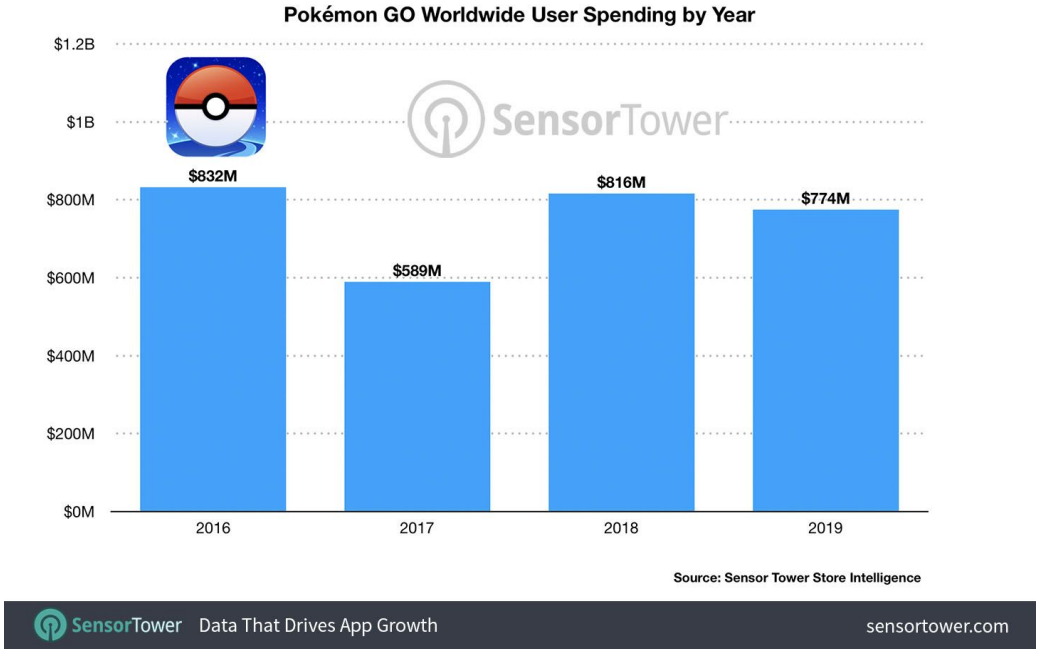


Figura 13. Estimación beneficios Pokémon Go por año.

En cuanto a jugabilidad, Pokémon Go se divide en dos fases: encontrar a los pokémon y capturarlos. En la primera fase, encontrar a los pokémon, el jugador tiene una posición asignada, utilizando la localización GPS del móvil en un mapa. Dicho jugador puede observar los pokémon que se encuentran a su alrededor y debe desplazarse si quiere capturar dichos pokémon saber que pokémon se encuentran en otras zonas del mapa. En la segunda fase, capturarlos, el jugador debe lanzar una pokeball al pokémon, un símil a esta mecánica podría ser lanzar un balón a una canasta .

Figura 14. Fases de juego Pokémon Go.

Si bien es cierto que Pokémon Go no fue el primer juego en utilizar la localización GPS como parte del sistema de juego (El juego Ingress de la propia Niantic ya lo hacía años atrás), fue el juego que popularizó dicha mecánica.

2.3.2. Filtros de reconocimiento facial

La tecnología de reconocimiento facial tiene un enorme potencial y una gran variedad de usos, pero dentro del sector del entretenimiento y las redes sociales hay uno que destaca sobre el resto, los llamados Filtros.

Los Filtros son un conjunto de imágenes y efectos visuales que, valiéndose de la tecnología de reconocimiento facial, caricaturizan a la persona a la que están retratando. Dichos Filtros son muy populares en redes sociales donde, principalmente adolescentes, comparten dichas caricaturas.

Dado el presupuesto del proyecto, no se va a utilizar reconocimiento facial, pero son fuente de inspiración para la realización de caricaturas del jugador utilizando otro tipo de técnicas y tecnología.

Figura 15. Filtros de reconocimiento facial.

3. Marco Teórico

Durante el desarrollo se utilizarán diferentes recursos teóricos como: patrones de diseño, programación OO, metodologías ágiles, diseño de interfaces de usuario y conocimientos sobre el motor Unity y el lenguaje de programación C#.

3.1. Unity

El motor diseñado por Unity Technologies, debido a su amplio número de funcionalidades y a pesar de su relativa accesibilidad, es complejo de manejar. Es por esto que, con el objetivo de facilitar el entendimiento y mejorar los conocimientos de quien lea esta memoria, se va a explicar el funcionamiento, en mayor medida, de este motor.

Es importante aclarar que cualquier funcionalidad descrita a continuación es relativa a la versión 2018.1.3f1 de dicho motor.

3.1.1. Editor

Lo primero que se observa al crear un nuevo proyecto de Unity es el editor. Con el objetivo de familiarizar al lector con las diferentes partes de este y así, al más tarde mostrar imágenes, poder explicar con mayor facilidad, se va a analizar las diferentes partes del editor.

El editor de Unity está compuesto por Ventanas. Las ventanas que se muestran en la disposición por defecto son : Scene, Game , Hierarchy, Inspector, Project, y Asset Store. Scene y Game muestran la escena y el juego en ejecución respectivamente. Hierarchy muestra los elementos que se encuentran en una escena e Inspector muestra la información de uno de esos elementos. Project muestra todos los archivos y carpetas que tenemos en el proyecto y la Asset Store nos permite descargar archivos de la tienda oficial de Unity. Las ventanas de editor pueden ser desencajadas y encajadas en otras ventanas. También se puede cambiar su posición y tamaño para trabajar de la forma que más cómoda le resulte al desarrollador.

Además de las ventanas hay una serie de botones que nos son de gran utilidad: los encargados de controlar un elemento dentro de la escena (A la izquierda y enmarcados de azul en la siguiente figura) y los que nos permiten darle a play, pausa y siguiente paso en el juego (En el centro y marcados de rojo en la siguiente figura).

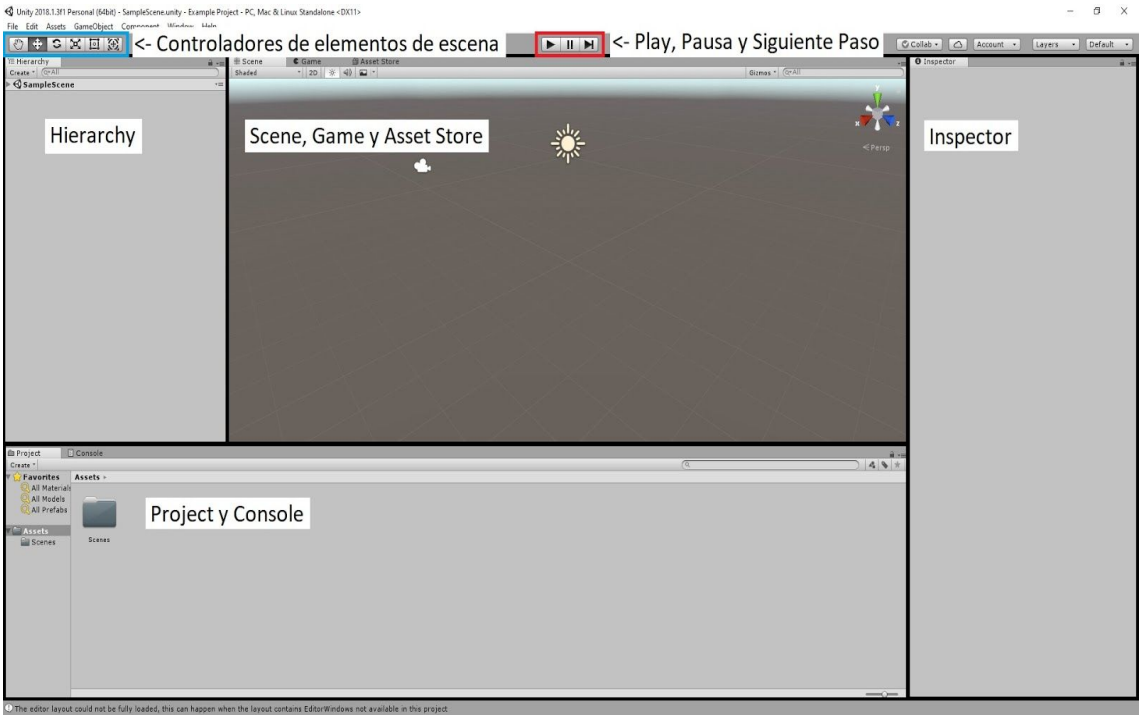


Figura 16. Editor.

3.1.2. GameObject

Un GameObject es una entidad con un Transform y una serie de componentes. Un Transform representa la posición, la rotación y la escala de un GameObject en una escena. Tanto la posición, como la rotación, como la escala son un Vector3. Un Vector3 es un conjunto de 3 variables float que representa cada uno de los ejes de coordenadas. Todo elemento de una escena es un GameObject. Un GameObject puede tener un hijo, que será otro GameObject. Más adelante se analizarán los componentes con más profundidad.

Figura 17. GameObjects en la ventana Hierarchy.

Dada la flexibilidad que nos aporta el sistema de componentes, una GameObject puede ser desde un panel de interfaz, hasta un personaje o enemigo, pasando por un manager que regule el sistema de input.

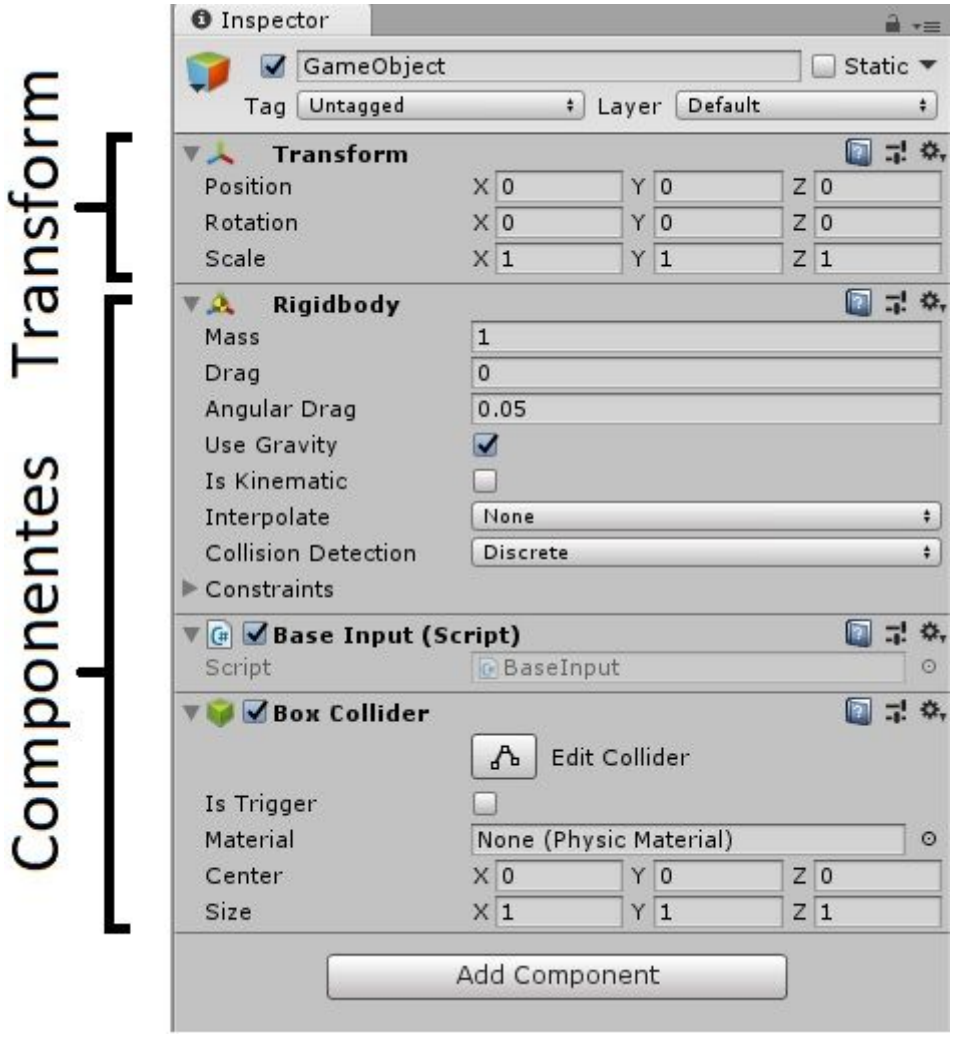


Figura 18. Representación de un GameObject en Inspector

3.1.3. Prefab

Todo GameObject que esté en una escena es persistente dentro de esa escena, pero si queremos que un GameObject persista independientemente de esa escena debemos crear un Prefab de ese GameObject.

Un ejemplo sencillo sería el personaje de un juego. Imaginemos que queremos que el personaje pueda estar en varias escenas y que esas escenas pueden ser ejecutadas en cualquier orden. Si el personaje no tuviera un Prefab, deberíamos tener GameObjects con los mismos componentes duplicados en todas las escenas, y, si el personaje precisara ser modificado, se debería modificar una a una, recorriendo cada escena, cada una de las copias. En cambio, si tenemos el GameObject del personaje almacenado en un Prefab, solo

necesitamos una copia, de la cual crearemos instancias en cada una de las escenas que lo requiera, y en caso de ser modificado solo se modificará una vez, reduciendo esfuerzo y errores.

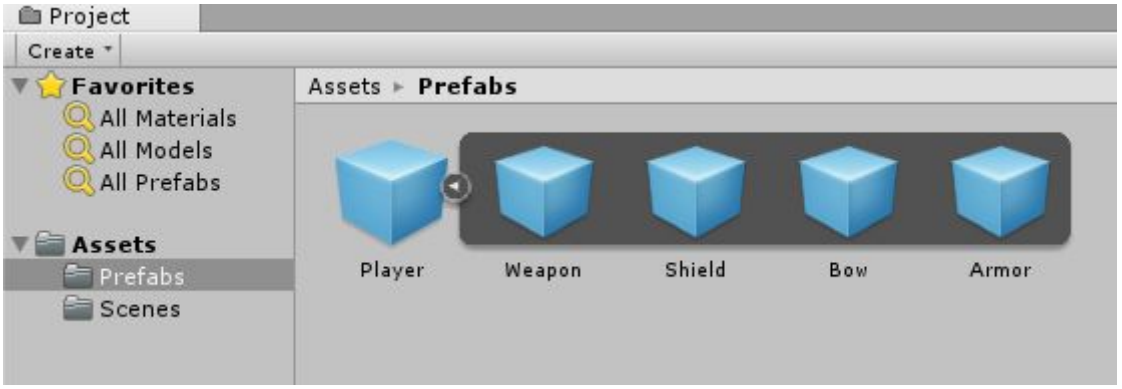


Figura 19. Prefab.

Para crear una instancia de un prefab puedes hacerlo de forma manual, arrastrando el prefab a la escena para que el editor cree el GameObject, o se puede hacer por código. Un uso de la instanciación por código lo encontramos en las balas de un arma de un juego de disparos.

No es posible crear manualmente tantas balas como vaya a usar el jugador porque pueden ser infinitas. Así que cada vez que el jugador pulse el botón de disparar instanciamos una bala. Esto puede ser mejorado creando una pool de balas que sea tan grande como el número de balas que son necesarias concurrentemente y volviendo al estado inicial dichas balas cuando acaben su función en lugar de eliminarlas. La pool no puede ser creada manualmente porque el tamaño de la pool cambiará en función de diferentes variables como: el tamaño del cargador del arma, la velocidad de disparo o el tiempo de vida de las balas.

```
private void Shoot()
{
    Instantiate(bullet, Vector3.zero, Quaternion.identity);
}
```

Figura 20. Código para crear una instancia de un GameObject.

3.1.4. Escena

La escena es el lugar donde sucede la acción en un videojuego. Un videojuego tiene como mínimo una escena y es posible crear tantas escenas como desee el desarrollador.

Es recomendable, en la medida de lo posible y poniendo siempre la jugabilidad por delante, separar las diferentes partes del juego en escenas diferentes para desacoplar grupos de información que no es necesario que estén

juntos. Identificar las diferentes partes de un juego con buen criterio es parte de la tarea de un desarrollador. Para ayudar a identificar dos partes de un juego se van a retratar una serie de ejemplos y situaciones.

Dos partes de un juego que están claramente divididas son: el menú de inicio y cualquier parte del juego propiamente dicho. Con esto conseguimos que el panel de interfaz del menú principal, que solo se va a utilizar de forma esporádica, con todos sus GameObjects y respectivos componentes, no esté cargado durante todo el juego ni en memoria ni en la jerarquía, facilitándonos el trabajo al tener menos ruido visual, evitando confusiones y mejorando el rendimiento. No confundir la interfaz del menú de inicio, con paneles usados más frecuentemente durante el transcurso del juego como pueden ser: el menú de opciones, el panel del inventario o el mapa.

Figura 21. Menú (izquierda) y Nivel (derecha) Super Mario World.

Otros motivos para separar dos partes de un juego pueden venir dados por la jugabilidad. Si hay dos mecánicas muy diferentes, que no se realizan de forma simultánea, deben separarse para evitar fallos y facilitar el trabajo a los programadores. Por lo que si un juego tiene una jugabilidad claramente separada en diferentes mecánicas, se crean diferentes escenas para cada mecánica. Un ejemplo teórico sería Pokemon Rojo Fuego que tiene separadas la exploración del combate. Por lo que se podría separar cada fase en una escena.

Figura 22. Exploración (izquierda) y Combate (derecha) Pokémon Rojo Fuego.

También pueden separarse dos partes de un juego por limitaciones técnicas. Si, por ejemplo, el interior de una casa, tiene un gran número de elementos y el jugador puede navegar tanto por dentro como por fuera, quizás es necesario separar el interior en otra escena para tener un mayor rendimiento cuando el jugador esté en el exterior y viceversa. También puede optimizarse una zona que va a repetirse diversas veces separándole en una escena, para no tener los elementos duplicados.

Figura 23. Exterior (izquierda) e Interior (derecha) Resident Evil 2.

Entre dos escenas, sobre todo si el tamaño de la carga tiene un volumen considerable, se puede intercalar una escena de pantalla de carga. Esta escena puede utilizarse para unir cualquier par de escenas que necesiten de un tiempo para ser cargadas en memoria o ser renderizados por la GPU. La escena de pantalla de carga suele tener, pero depende de los requerimientos del juego, un clase que gestiona que paquetes de archivos necesita cada escena del juego y los carga y descarga en función de dichas necesidades.

En el caso de que haya un GameObject que se quiera destruir al pasar de una escena a otra, ya sea un personaje, un panel de interfaz o una clase que se dedique a monitorizar la persistencia del juego, se puede utilizar el método DontDestroyOnLoad y pasarle una referencia de dicho objeto.

3.1.5. Componentes

Los componentes definen un GameObject. Para tener una imagen general de las funcionalidades de los componentes se van a describir algunos de los componentes más utilizados en Unity. Se dejará para más adelante los componentes relacionados con scripts y la interfaz gráfica de usuario ya que se definirán en otra sección.

3.1.5.1. Colliders

Los colliders se encargan de detectar las colisiones. Hay diferentes tipos de colliders dependiendo de si son 2D o 3D y por su forma: cuadrados, circulares, ovalados, etc.

Podemos modificar el centro y la escala de un collider y asignarle un material que modificará su fricción y elasticidad. Pero probablemente la variable más significativa es IsTrigger.

IsTrigger es un bool, si está activado el collider será un sensor. Si el collider es un sensor detectará las colisiones pero no aplicará las propiedades físicas sobre el GameObject.

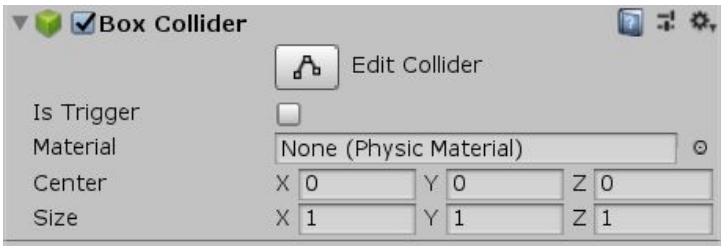


Figura 24. Box Collider en ventana Inspector.

Un collider puede ser utilizado sin un Rigidbody, pero no detectará ningún GameObject que no tenga un rigidbody.

3.1.5.2. Rigidbody

Un rigidbody encapsula el comportamiento físico de un GameObject. Para tener la simulación de físicas deseada podemos modificar la masa, el rozamiento y si le afecta la gravedad al objeto. También podemos hacer que sea Kinematic, esto es, que afecte físicamente a otros GameObjects pero él no se vea afectado por el resto. Esto es muy útil en plataformas que se mueven, queremos que desplacen al jugador pero no queremos que la plataforma se desvíe por el movimiento del personaje.

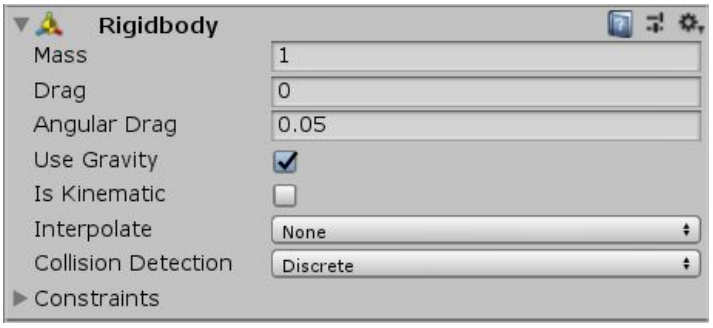


Figura 25. Rigidbody en ventana Inspector.

3.1.5.3. SpriteRenderer y MeshRenderer

El Sprite y Mesh renderers se encargan de reproducir una imagen 2D y un modelo 3D respectivamente. Ambos tienen propiedades para regular como les afecta la luz o el color y otro tipo de detalles estéticos.

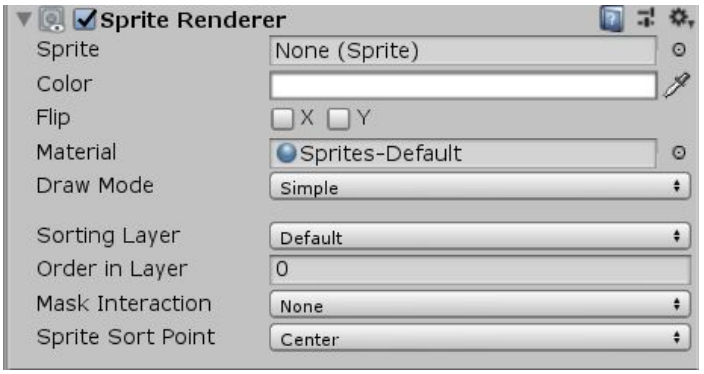


Figura 26. Sprite Renderer en ventana Inspector.

No puede haber un SpriteRenderer y MeshRenderer en un mismo GameObject. Porque cada GameObject tiene su funcionalidad, por lo que no es posible ser a la vez una imagen y un objeto 3D. Este tipo de incompatibilidades se dan con más componentes, pero siempre es por la misma razón: no puede haber dos funcionalidades dispares en el mismo GameObject.

3.1.5.4. Animator

El Animator se encarga de regular las animaciones que tenga un GameObject. El Animator es una máquina de estados finita que divide cada una de estas animaciones o conjunto de animaciones en estados y une dichos estados por transiciones.

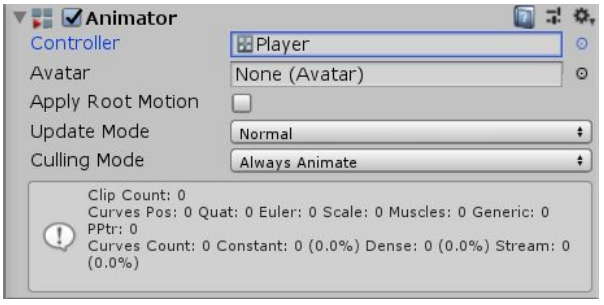


Figura 27. Animator en ventana Inspector.

Para poder visualizar dicha máquina de estados, existe una ventana en el editor de Unity que nos permite crear, modificar y eliminar transiciones y estados manualmente.

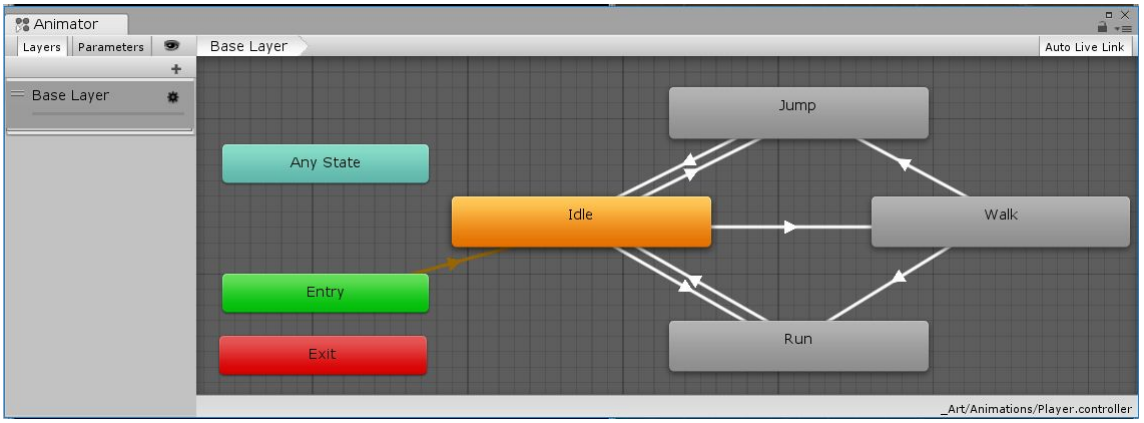


Figura 28. Ventana de Animator.

3.1.5.5. Camera

No puede haber un videojuego sin una cámara. En el caso de Unity la cámara es la misma tanto para 2D como para 3D. Esto nos permite, en caso de que lo deseemos, diseñar juegos que utilicen ambas dimensiones.

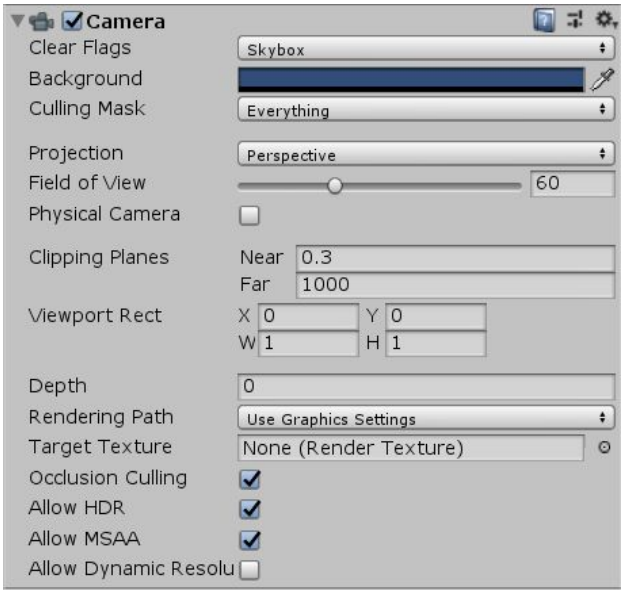


Figura 29. Camera en ventana Inspector.

3.1.6. Script

La creación de código para videojuegos en Unity, al realizarse con el lenguaje C#, es muy similar a el diseño de cualquier otra aplicación realizada a partir de lenguajes orientados a objetos, pero, como es natural, tiene sus propias peculiaridades. Muchas de estas peculiaridades vienen dadas por las propiedades de los GameObjects y el uso de los scripts como componentes de estos.

Monobehaviour es, sin duda, una de las clases más importantes en Unity. Dicha clase dota de la información necesaria al script para que pueda ser un componente ligado a un GameObject. Para esto, cualquier clase que herede de Monobehaviour puede acceder a otros componentes con el método GetComponent, puede acceder al GameObject, y a los métodos ligados a él, y puede implementar eventos como OnCollisionEnter o OnTriggerEnter dados por los colliders previamente analizados.

Funciones muy importantes que vienen dadas por la clase monobehaviour son: Start y Update. Start se ejecuta al iniciar un GameObject, mientras que Update es un bucle que se ejecutará mientras el GameObject este activo.

Dado que un script puede ser un componente de un GameObject, quizás necesitemos acceder a sus variables desde el editor. Para esto, podemos añadirle un encabezado a una variable llamado SerializeField. También es posible hacer la variable publica para que aparezca en el editor, pero si por la funcionalidad del código queremos que sea privada, pero necesitamos que aparezca en editor, es mucho más recomendable usar el encabezado SerializeField y evitar accesos desde otras clases que pueden dar pie a errores.

```
1   using ...
4
5   public class Example : MonoBehaviour
6   {
7
8       private Rigidbody _rigidbody;
9
10      [SerializeField]
11      private float _speed;
12
13      // Use this for initialization
14      void Start ()
15      {
16          _rigidbody = GetComponent<Rigidbody>();
17      }
18
19      // Update is called once per frame
20      void Update ()
21      {
22          _rigidbody.velocity = Vector3.forward * _speed;
23      }
24  }
25
```

Figura 30. Código relativo a la clase MonoBehaviour.

Es posible que se requiera que un método de un script se ejecute al iniciar una escena, pero no se quiera heredar de Monobehaviour porque no se va a utilizar ninguna de las funcionalidades que aporta más allá de poder poner el script como componente de un GameObject y el GameObject instanciarse al inicio de una escena. Para ello nos sirve el encabezado

RuntimeInitializeOnLoadMethod, que nos permite ejecutar un método al inicio de una escena.

3.1.7. ScriptableObject

Los scriptables objects son una forma de script, que existe independientemente de un GameObject y son utilizados principalmente para almacenar información.

Es una buena práctica tener los datos de las variables de un enemigo o personaje en un scriptable object en lugar de guardados en el propio script que contenga el comportamiento del personaje, porque así están desacoplados del GameObject del personaje, lo que hace que sea más fácil de ajustar dichos datos por parte de los desarrolladores. Además, estos datos solo se escribirán una vez en memoria, en lugar de duplicarse cada vez que se cree una instancia de ese enemigo. Esta práctica se puede extrapolar a todo tipo de scripts, desde los encargados de gestionar el sistema de misiones hasta los controladores del mando del juego. Es importante separar la funcionalidad de la persistencia.

3.1.8. Interfaz gráfica de usuario

La interfaz gráfica de usuario, como cualquier elemento de una escena, está construida a partir de GameObjects. Cada uno de estos GameObjects tiene componente relativos a la interfaz que permiten visualizar imágenes o crear botones. Es importante saber que un GameObject que se utilice en una interfaz no tiene un transform sino un rect transform. El rectransform funciona de forma similar a un transform pero tiene variables propias que nos permiten ajustar el tamaño y la posición a la pantalla.

Figura 31. Rect Transform en ventana Inspector.

El componente base de toda interfaz en Unity es el Canvas. Sobre el Canvas se construye la interfaz, si él no es posible visualizar imágenes o barras de progreso o cualquier componente que sea propio de la interfaz. Esto se debe

a que el Canvas es elemento sobre el que se ajustará la posición del resto de componentes, ya sea del mismo gameobject o de sus hijos. Es decir, todo gameobject utilizado en una interfaz tiene una posición, dada por un rect transform, relativa al Canvas que lo contiene.

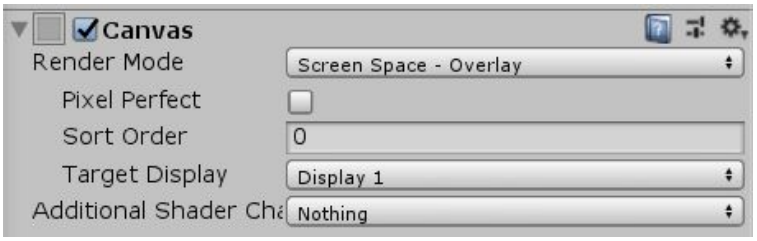


Figura 32. Canvas en ventana Inspector.

Además del Canvas encontramos componentes tales como: Image, Panel, Horizontal, Vertical y Grid layout, Button, Event Trigger (similar a Button pero con más opciones) , Text y ScrollView entre otros. Todos estos componentes tienen una funcionalidad similar a otros entornos y aplicaciones como pueden ser Windows Forms o JavaFX.

Como último apunte decir que es recomendable utilizar el plugin TextMeshPro en lugar del Text por defecto, porque tiene una mayor variedad de opciones y funcionalidades que facilitan la creación de interfaces.

3.1.9. Time

En Unity podemos discernir dos tipos de tiempo diferentes que se ejecutan de manera simultánea. Por un lado, tenemos el tiempo real, que viene dado por el frame rate (la velocidad que hay entre cada fotograma). Este tiempo es con el que funciona la función Update, previamente vista. Por otro lado, tenemos el fixed step time, que es con el que funciona el FixedUpdate. Este tiempo es fijo, se puede cambiar su velocidad en el editor de Unity y se utiliza para calcular las físicas (Lo usan componentes como los colliders y el rigidbody). El motivo por el que las físicas de unity utilizan un valor fijo no dependiente del frame rate es que es más seguro hacer cálculos complejos como son los requeridos por un sistema de físicas.

3.2. Patrones de Diseño

Para la mejora de la calidad del software y la resolución de problemas técnicos se han empleado una serie de patrones de diseño. A continuación se describirán brevemente y se definirá su uso en el proyecto.

3.2.1. Comman

El patrón command es, según el libro Design Patterns [12], "una remplazo orientada a objetos de las callbacks". Y además, uno de los patrones más utilizados en la programación de videojuegos.

En videojuegos es un diseño básico para la creación del sistema de input, ya que, el botón X del mando puede servir tanto para saltar, como para darle a aceptar en una interfaz, un jugador puede cambiar el botón X por cualquier otro en la configuración para cualquiera de las funciones anteriores y el juego puede jugarse tanto con mando como con teclado y ratón. A este comportamiento se le llama 'bindeo' y 'rebindeo' del input.

En AR Catapult se ha utilizado este patrón de diseño para 'rebindear' el input del plugin Online Maps (que permite hacer zoom y desplazarse por el mapa) del ratón y el teclado a la pantalla táctil de los dispositivos móviles.

3.2.2. Pool

Una pool es la mejora del rendimiento y el uso de memoria gracias a la reutilización de un número de objetos fijo, en lugar de crearlos, destruirlos y modificarlos individualmente.

En el caso de Unity la pool es de GameObjects. Y en el caso concreto de AR Catapult, la pool creada es de obstáculos y recompensas. Al inicio de la escena se instancian los objetos necesarios. Cuando uno de los GameObjects mencionados requiere ser utilizado se activa (los GameObjects pueden activarse y desactivarse) y hace su función. Al terminar se lo devuelve a su estado inicial y se desactiva.

3.2.3. Inyección de dependencia

La inyección de dependencias ayuda a separar al consumidor de un servicio de su proveedor, no requiriendo así que el consumidor requiera saber de dónde debe obtenerlo, sino que es un gestor externo (el inyector) quien se lo provee. De este modo para el programador de la clase consumidora es irrelevante de dónde venga y puede asumir que tendrá ese servicio provisto.

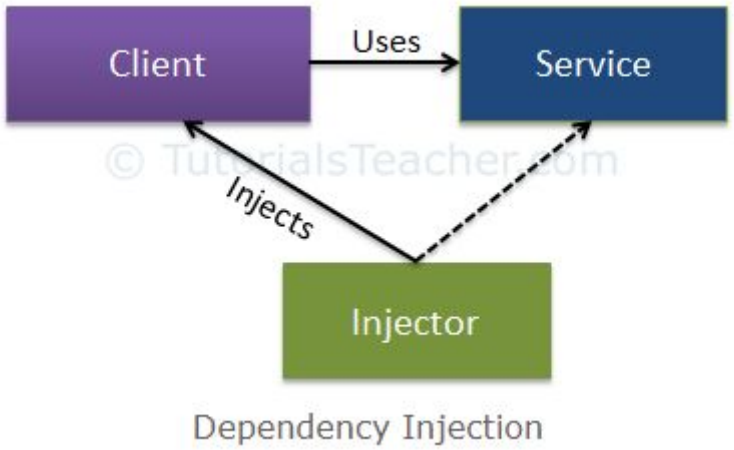


Figura 33. Representación Inyección de dependencia (Imagen de tutorialsteacher.com)

En el caso de Unity y el plugin Zenject el inyector es la clase BindingInstaler. En el caso de AR Catapult, tanto las dependencias entre el FlowManager y los managers de cada escena como las de los diferentes managers con las clases encargadas de funcionalidades específicas están resueltas con inyección de dependencia.

4. Desarrollo

La producción de un videojuego se divide en tres fases: preproducción, producción y posproducción. Como puede llevar a confusión el hecho de que la producción sea a la vez una fase y el conjunto de todas estas, dentro de este apartado nos referiremos a la fase como producción y al conjunto como elaboración.

Previo a la elaboración hay un propuesta inicial donde se plantea la idea del videojuego, del cual se hacen varias iteraciones hasta satisfacer a todas las partes implicadas. En el caso de AR Catapult se partió de la idea del cliente. Su intención era crear un juego que fuera: "como Pokémon Go pero con portales dimensionales", sin duda una idea atractiva, pero muy lejos de la realidad del presupuesto. Tras varias iteraciones se alcanzó un acuerdo con el cliente, en el que, con el presupuesto dado, se realizaría las fases de preproducción y producción. Durante estas fases se emplearían el GPS del móvil de la misma forma que lo hace Pokémon Go y se utilizarían la cámara trasera y delantera para crear y lanzar tu avatar, pero se aplazaría el uso de la realidad aumentada con la cámara trasera a la fase de postproducción. La cual quedaría fuera de presupuesto y, por tanto, de esta memoria.

En la fase de preproducción se realiza el prototipado. En proyectos de gran envergadura se realizan varios prototipos de forma simultánea: prototipos de jugabilidad o de una parte de la jugabilidad, prototipos de arte y prototipos técnicos. En cada uno de estos prototipos se prueba si es posible realizar la idea inicial, y se realizan varias iteraciones hasta tener el resultado esperado para cada una de las partes. En el caso de AR Catapult, al ser un proyecto de pequeñas dimensiones, se realizará un único prototipo donde se pondrán a prueba la jugabilidad, el arte y el software conjuntamente.

En la fase de producción se desarrolla el videojuego como tal. Si bien es cierto que parte del código y del arte pasan del prototipo a la fase de producción la gran mayoría es abandonado, sobre todo en el caso del código, debido a que en la fase de preproducción se ha trabajado con la intención de crear más en cantidad que en calidad. Por lo que, con la intención de establecer unas bases sólidas, se crea un nuevo proyecto y se inicia el código desde cero. En el caso de AR Catapult, al ser un juego relativamente sencillo y tener un equipo con experiencia previa, parte del código pasó del prototipo al nuevo proyecto. Pero partes como la integración con el GPS, algo que no se había realizado antes en el estudio, o el sistema de input fueron rehechos, ahora sí, con los conocimientos necesarios para hacerlo correctamente y de forma estructurada.

En la fase de postproducción se le pone "azúcar" al videojuego. Se le añaden efectos de sonido y visuales, música, cinemáticas, la traducción de los textos, voces y se arreglan bugs encontrados por el equipo de QA (Quality Assurance). En el caso de AR Catapult la postproducción queda fuera de presupuesto y, por tanto, de la planificación. En cualquier caso, si se tuviera que realizar la postproducción, las tareas a realizar serían: diseñar trajes para el personaje que se comprarían en la tienda, implementar la funcionalidad de realidad aumentada, añadir textos en diferentes idiomas y arreglar bugs.

4.1. Planificación

Segmentaremos el desarrollo de AR Catapult en los siguientes milestones:

1. Prototipado: Implementación de una primera versión simplificada de los sistemas software y de su integración con los dispositivos móviles en la que se probarán los sistemas de input y GPS. No se requiere ni de interfaz ni de arte. Tiempo estimado: 2 Semanas

2. Esperar respuesta del cliente. Tiempo estimado: 1 Semana.

3. Versión Alpha: Implementación del sistema software en una fase cercana a la final, integración con dispositivos móviles, construcción de interfaces en una versión simplificada e instauración de arte esencial. Tiempo estimado: 3 Semanas

4. Esperar respuesta del cliente. Tiempo estimado: 1 Semana.

5. Finalización: Pulido del software, acabado de interfaces e integración del arte no esencial. Tiempo estimado: 2 Semanas.

Tiempo desarrollo: 7 semanas * 40h/semana = 280 horas. Tiempo total: 9 semanas. Figura 34.

El milestone de prototipado se encuentra dentro de la fase de preproducción. Los milestones de la versión alpha y de finalización son parte de la fase de producción.

4.2. Análisis y diseño Orientado a Objetos

Con el fin de tener un sistema lo más desacoplado posible que nos otorgue la flexibilidad necesaria para soportar los posibles cambios en el diseño de la jugabilidad, se ha diseñado un sistema basado en escenas, previamente explicadas, e inyección de dependencia donde la única relación entre escenas viene dada por la clase FlowManager.

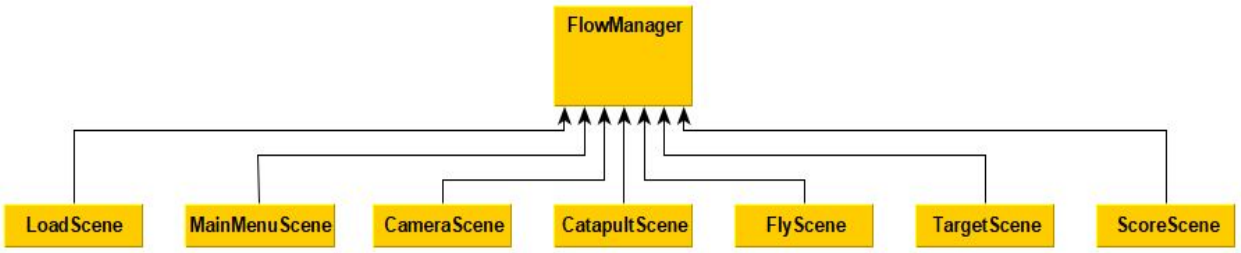


Figura 35. Distribución por escenas.

Cada escena tiene su propio manager que se encarga de los aspectos generales como son: el control del inicio y fin de la escena, la comunicación con el FlowManager y gestionar los diferentes subsistemas.

Finalmente, tenemos clases encargadas de funcionalidades más específicas como son: el comportamiento del personaje en una escena, tomar una foto o el movimiento de obstáculos, etc.

4.3. Diseño de Interfaz gráfica de usuario

La interfaz gráfica de usuario se diseñó a partir de una serie de mockups. Cada mockup representa cada una de las fases del juego y sirve tanto como base para la posterior implementación, como para ilustrar todas esas ideas abstractas que se han hablado con el cliente.

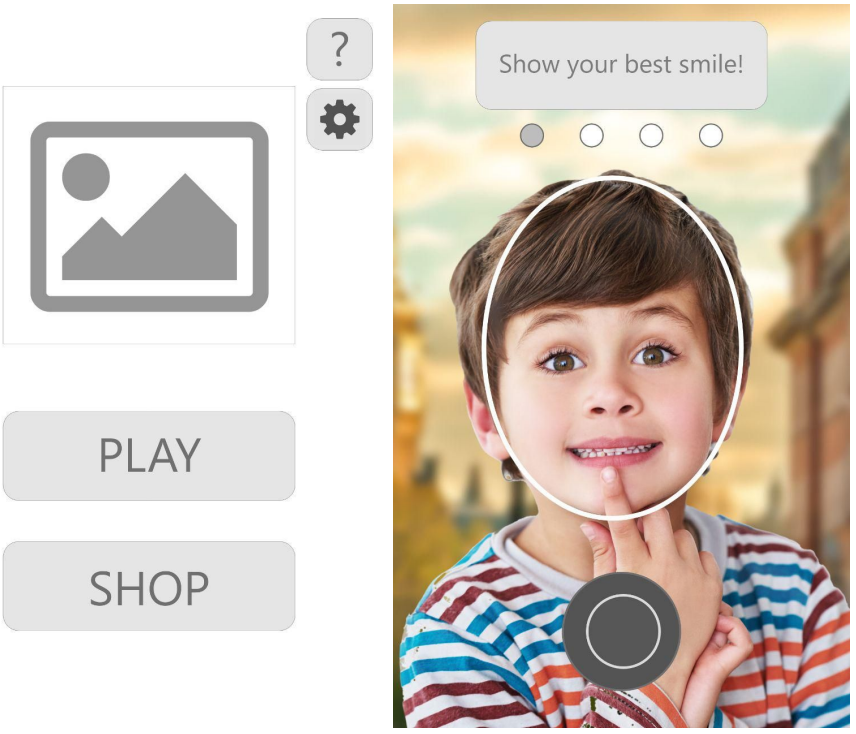


Figura 36. Mockup menú de inicio (izquierda) y escena de foto (derecha) .

El menú de inicio es bastante sencillo, tiene dos botones situados en el centro para comenzar a jugar o ir a la tienda y en la parte superior derecha otros dos botones para ver las opciones e información sobre el juego respectivamente. La imagen que se encuentra encima de los botones centrales representa el espacio que ocupará el logo del juego.

Si apretamos el botón de Play iremos a la escena de foto, donde el jugador podrá hacerse unas fotos a él mismo o a otra persona. La idea es que el jugador haga 4 fotos a la misma persona con expresiones diferentes, y estas expresiones cambien en

función de lo que está ocurriendo en el juego. En la parte superior hay un panel que nos indica que número de fotos se han realizado y qué expresión debe hacerse.

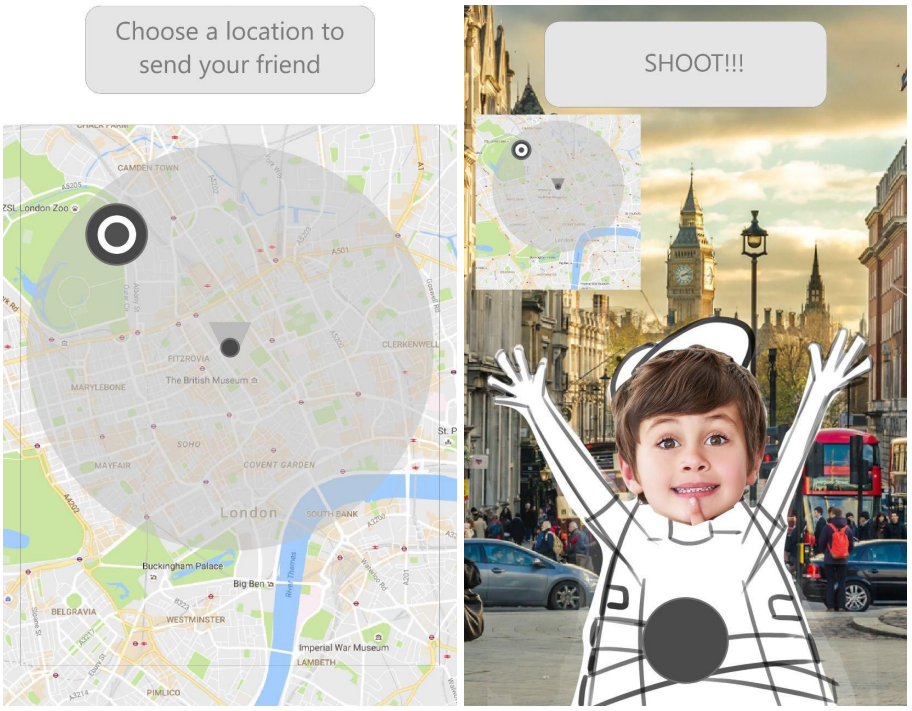


Figura 37. Mockup escena mapa (izquierda) y lanzamiento (derecha) .

Tras realizar las fotos, se localiza la posición del jugador. Una vez se ha localizado su posición el jugador puede seleccionar un lugar dentro de su radio de lanzamiento. En esta escena existen dos paneles: el de la parte superior que nos indica que hacer y el que está integrado en el mapa que muestra la posición del jugador y permite seleccionar una localización dentro del radio que nos muestra.

Con una posición ya precisada, el jugador pasa a la siguiente escena donde se muestra en primer plano el personaje con las fotos previamente realizadas y en el fondo lo que muestre la cámara del móvil. Volvemos a tener un panel en la parte superior que nos guía y una versión más pequeñas del mapa anterior. También tenemos como parte de la interfaz un botón que nos permite lanzar el personaje.

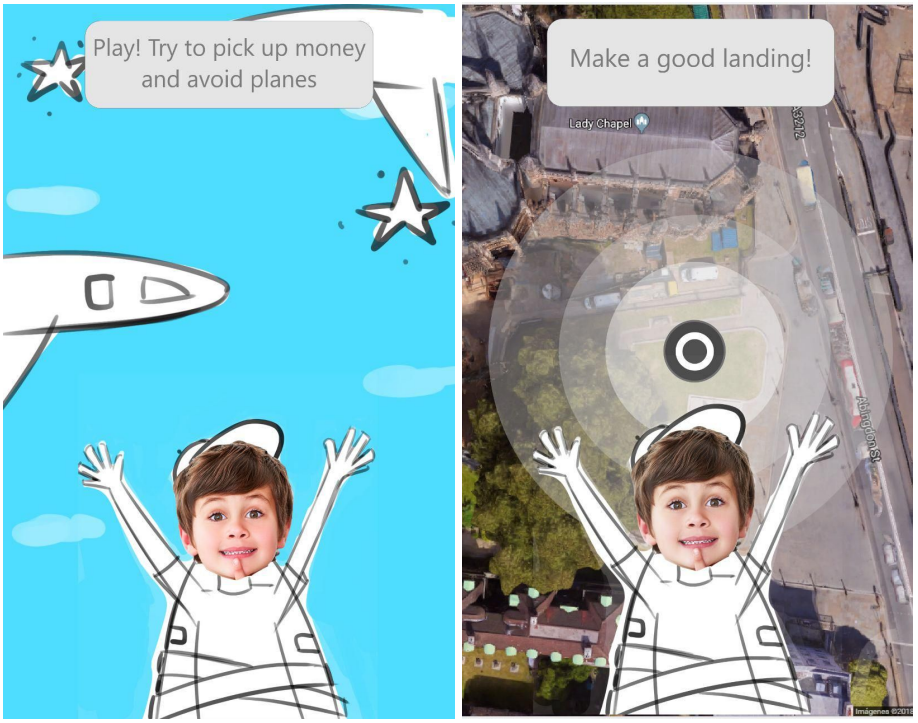


Figura 38. Mockup fase vuelo (izquierda) y fase aterrizaje (derecha)

En el momento en el que el personaje es lanzado se pasa a la siguiente escena, donde aparece el personaje volando y una serie de obstáculos (aviones) y recompensas (estrellas). El jugador debe usar el giroscopio del móvil para mover el personaje.

Si el jugador consigue esquivar los obstáculos y termina el recorrido llegará a fase final del viaje, donde deberá caer en el centro de la diana para multiplicar su puntuación. En la dos últimas escenas también se observan paneles en la parte superior que ayudan al jugador con un poco de información.

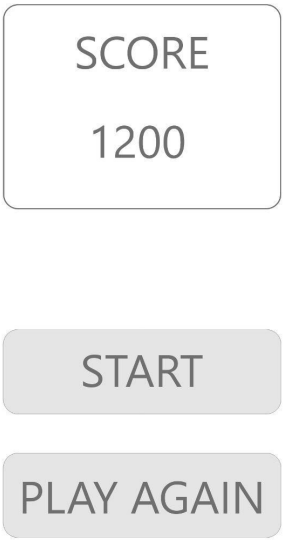


Figura 39. Mockup pantalla de puntuación

Finalmente tenemos el menú de puntuación que es muy similar al de inicio, solo que quitando los botones de la parte superior derecha y cambiando el logo del juego por la puntuación.

4.4. Implementación

A continuación se irán analizando el trabajo realizado durante los diferentes milestones. Empezando por prototipado, continuando en la versión alpha y concluyendo con la finalización.

La prioridad en el milestone de prototipado era poner a prueba la jugabilidad. Debido a esto, tras generar las diferentes escenas y el FlowManager que nos sirviera de base, se crearon los sistemas que se identificaron como clave para dicho propósito.

Los sistemas elegidos fueron: el encargado de hacer una foto, el lanzamiento del personaje, un mapa básico donde poder seleccionar el destino, los obstáculos y recompensas de la fase de vuelo y el control del personaje tanto en la fase de vuelo como en la fase de caída.

Para la integración con la cámara del dispositivo móvil se utilizó la clase WebCamTexture integrada en la API de Unity. Esta clase te devuelve lo que ve la lente del dispositivo en forma de textura, la cual se aplica a un panel. Este panel se pone más tarde en la posición de la escena que desees, en el caso de este proyecto delante de la cámara de la escena ocupando toda la pantalla. Sobre lo que proyectaba la cámara del dispositivo se puso un marco para que el jugador pusiera el rostro en la zona central restante.

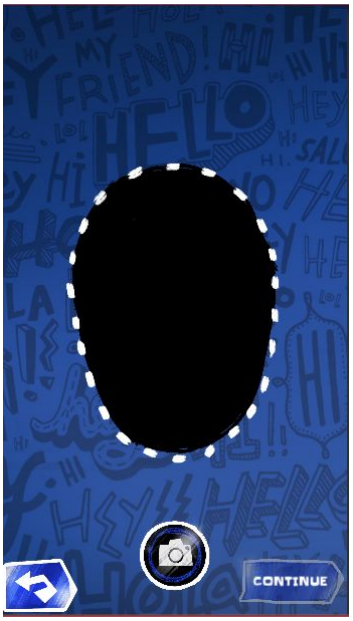


Figura 40. Escena de foto

Como se ha dicho, el panel con la textura, a pesar de que no se aprecie en la figura 40 porque tiene un marco, ocupa toda la pantalla. Pero solo se desea tomar la foto de la parte central, para ello se va a utilizar una máscara. Una máscara es un componente que, a partir de una primera imagen, modifica una segunda. En este caso, se creó una primera imagen con la forma del hueco central y se recortó la imagen con el rostro del jugador.

También podemos observar una interfaz muy básica con un botón para hacer las fotos y un botón para continuar a la siguiente fase del juego. Por último decir que, como no se quería llenar el dispositivo móvil de jugador de fotografías, el botón de hacer foto no realiza una foto propiamente dicha, sino que crea una copia de la textura.

La siguiente escena a realizar es la de selección del lugar de lanzamiento en un mapa. Para este propósito se utilizó el plugin OnlineMaps. Dicho plugin está formado por una serie de componentes que, a partir de mapas de terceros como google maps, nos permiten formar un mapa interactivo. Durante la fase de prototipado se creó un mapa funcional, pero no se añadieron las funciones de localización del GPS del dispositivo móvil.

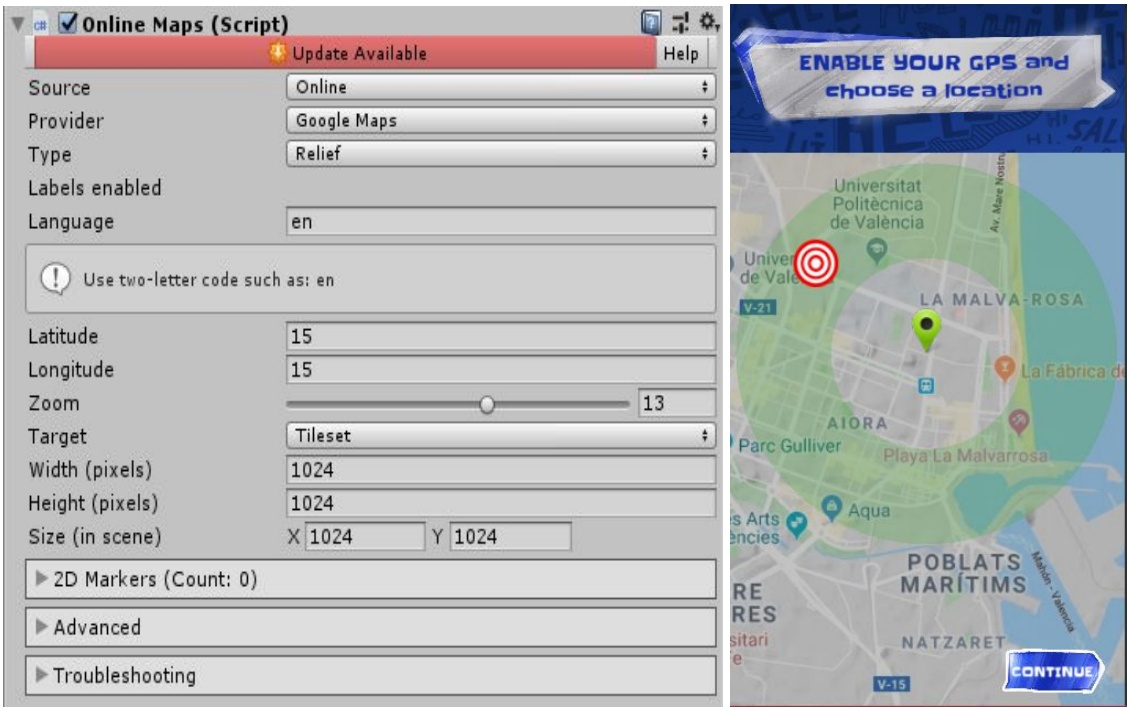
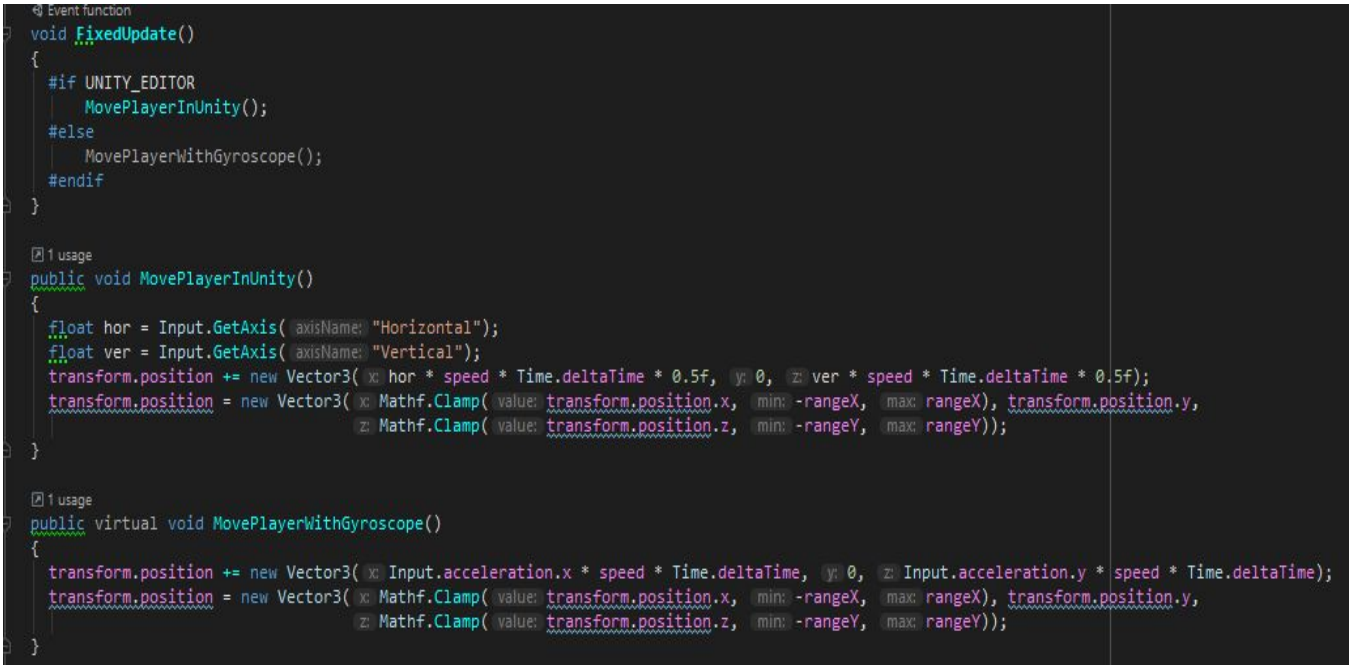


Figura 41. Componente de online maps (izquierda) y escena mapa (derecha)

A continuación se realizó la escena del lanzamiento del personaje. Utilizando lo aprendido en la escena de fotos se situó la cámara del dispositivo de fondo, el personaje en primer plano y un botón que, al pulsarlo, lo lanzaba.

La fase de vuelo tiene dos sistemas principales: el encargado del comportamiento del personaje y el sistema de objetos (tanto los enemigos como las estrellas).

El personaje tiene que, por un lado, responder al input del jugador y, por otro, reaccionar a los obstáculos. Como el sistema de input va a ser igual tanto en la fase de vuelo como en la fase de caída, que veremos más adelante, se ha creado una clase genérica llamada PlayerController. Dado que en la fase de vuelo hay obstáculos a los que reaccionar, pero en la fase de caída no, se ha construido la clase PlayerControllerInFlight que hereda de PlayerController y añade la funcionalidad de interactuar con los obstáculos y las recompensas.



```
Event function
void FixedUpdate()
{
  #if UNITY_EDITOR
      MovePlayerInUnity();
  #else
      MovePlayerWithGyroscope();
  #endif
}

1 usage
public void MovePlayerInUnity()
{
  float hor = Input.GetAxis( axisName: "Horizontal");
  float ver = Input.GetAxis( axisName: "Vertical");
  transform.position += new Vector3( x: hor * speed * Time.deltaTime * 0.5f, y: 0, z: ver * speed * Time.deltaTime * 0.5f);
  transform.position = new Vector3( x: Mathf.Clamp( value: transform.position.x, min: -rangeX, max: rangeX), transform.position.y,
                                    z: Mathf.Clamp( value: transform.position.z, min: -rangeY, max: rangeY));
}

1 usage
public virtual void MovePlayerWithGyroscope()
{
  transform.position += new Vector3( x: Input.acceleration.x * speed * Time.deltaTime, y: 0, z: Input.acceleration.y * speed * Time.deltaTime);
  transform.position = new Vector3( x: Mathf.Clamp( value: transform.position.x, min: -rangeX, max: rangeX), transform.position.y,
                                    z: Mathf.Clamp( value: transform.position.z, min: -rangeY, max: rangeY));
}
```

Figura 42. Código clase PlayerController.

Como se puede observar en la figura 42, en la clase PlayerController hay una función llamada FixedUpdate (un bucle que se llama con el fixed step time visto previamente) y dos métodos encargados del movimiento: MovePlayerInUnity y MovePlayerWithGyroscope. El motivo de la existencia de estos dos métodos viene dado por una facilidad a la hora de desarrollar el juego. Como un ordenador no tiene giroscopio, y es muy caro (en términos de tiempo) probar cada cambio en la escena directamente en el móvil, tenemos un método (MovePlayerInUnity) que nos permite mover el personaje con las flechas del teclado. Para que este método no se use fuera del desarrollo está encapsulado dentro de la función #if UNITY_EDITOR, que, como su nombre indica, solo se llama cuando el juego se ejecuta desde el editor.

```
Scripting component
public class PlayerControllerInFlight : PlayerController {

    [Inject]
    FlightScreenManager _flightScreenManager;

    Event function
    private void OnTriggerEnter2D(Collider2D other)
    {
        if (other.gameObject.CompareTag("Enemy"))
        {
            _flightScreenManager.LoseGame();
        }
        else if (other.gameObject.CompareTag("Pickup"))
        {
            _flightScreenManager.ReturnItemToPool(other.gameObject);
            UpdatePoints();
        }
    }

    1 usage
    public void UpdatePoints()
    {
        _flightScreenManager.pointsValue++;
        _flightScreenManager.pointsText.text = _flightScreenManager.pointsValue.ToString();
    }
```

Figura 43. Código clase PlayerControllerInFlight.

La función principal en la clase PlayerControllerInFlight es OnTriggerEnter2D. Dicho método nos permite, a través del sistema de colliders de Unity, detectar los objetos. Para diferenciar entre los distintos tipos de objetos, cada uno de ellos tiene un tag diferente. Además destacar el uso de la inyección de dependencia de la clase FlightScreenManager.

El sistema de objetos también puede dividirse en dos subsistemas. Por un lado, el sistema encargado gestionar la aparición de objetos que se encarga de decidir cuántos objetos aparecen en total, si estos objetos son obstáculos o recompensas y que intervalo de aparición hay entre cada objeto. Y por otro lado, el movimiento de dichos objetos.

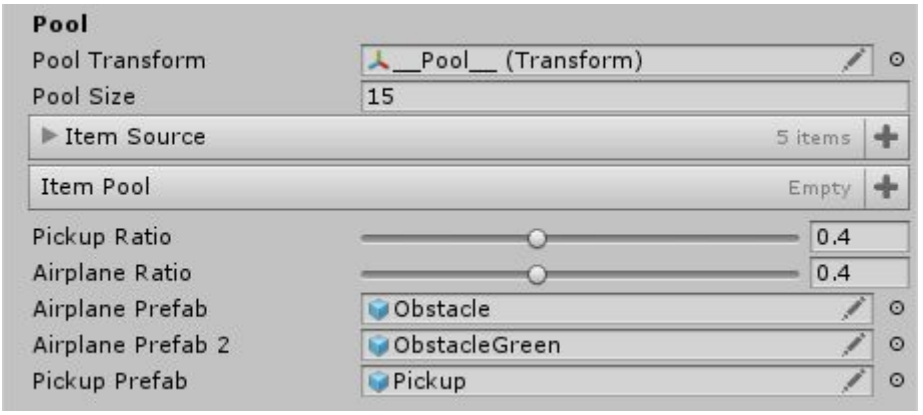


Figura 44. Parte del componente FlightScreenManager dedicada a los obstáculos.

La gestión de los objetos es parte de la funcionalidad del FlightScreenManager. Dicha clase genera una pool al inicio de la escena y elige, en un intervalo determinado, un elemento de esa pool con un random pesado. Como vemos en la figura 44, hay un

40% de probabilidades de que aparezca un “PickUp”, las recompensas, y otro 40% de que aparezca un “Airplane”, un obstaculo, que más tarde se cambiaría por un pájaro. El 20% restante es totalmente aleatorio.

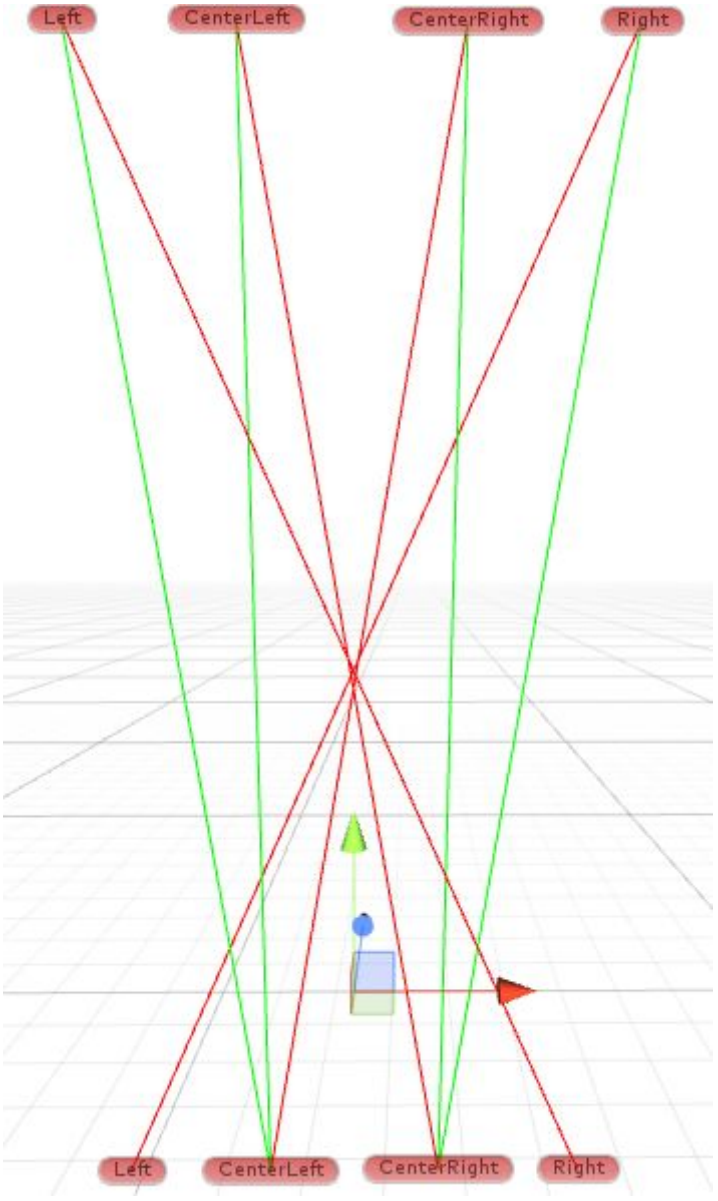


Figura 45. Trayectorias obstáculos (rojo) y recompensas (verde)

Cada uno de los obstáculos es generado en un SpawnPoint de forma aleatoria. Cada SpawnPoint tiene dos DestinationPoint: uno para las recompensas y otro para los obstáculos. En la figura 45, la trayectoria desde un SpawnPoint a un DestinationPoint de una recompensa está representada en verde, y en rojo la de un obstáculo. Como se puede observar, los obstáculos siempre pasan por el centro, por lo que si el jugador está en el centro de la pantalla siempre le van a golpear. Pero, las recompensas siempre van al centro, por lo que si quiere conseguir puntos deberá centrar al personaje. Con esto se consigue que el jugador siempre esté activo, ya sea moviéndose de esquina a esquina

para esquivar los obstáculos, como desplazándose de una esquina al centro para recoger una estrella y volviendo a una esquina para evitar un obstáculo.

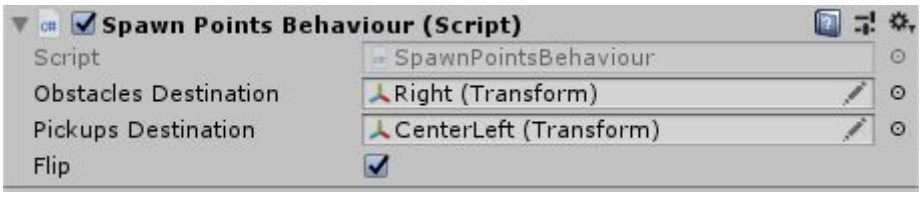


Figura 46. Componente SpawnPointsBehaviour.

Si el diseñador quisiera añadir nuevos SpawnPoint podría hacerlo en cuestión de segundos, dado que toda la funcionalidad está encapsulada en un componente. Por lo que, para crear un nuevo SpawnPoint solo hay que crear un nuevo GameObject en el editor y añadirle el componente. De la misma forma, para añadir o modificar las trayectorias solo hay que arrastrar el DestinationPoint al componente. Además, en caso de que el objeto se desplace de derecha a izquierda (en lugar de izquierda a derecha), con clicar en la checkbox de la variable 'Flip' del componente se invertirá la imagen del GameObject, por lo tanto no es necesario crear dos imágenes y animaciones por cada obstáculo o recompensa.

Finalmente, en la escena de caída se reutiliza tanto el mapa como el sistema de input de personaje. Respecto al personaje, el único trabajo a realizar es, detectar en la clase PlayerControllerInLand el momento en el que el personaje llegue a la posición 0 en el eje Y, poner el personaje, modificando su transform en el editor, a la altura deseada y en su Rigidbody darle al checkbox que hace que se le aplique gravedad para que de ese efecto de caída que se desea. Para finalizar la escena, se añade una diana que, basándose en la distancia del centro en el momento de tocar el suelo, de una puntuación al jugador.

Tras el visto bueno del cliente, se pasó al desarrollo de la versión alpha. Lo primero a implementar fue la escena de pantalla de carga, que si bien ya estaba creada aún no tenía la funcionalidad implementada. Durante esta escena se cargan en memoria todos los archivos que se van a utilizar durante el juego. Si el juego fuera de mayor tamaño, solo se cargarían parte de los archivos que son necesarios y se intercalarían más pantallas de carga entre escenas. También se añadió el menú de inicio junto al arte final y se mejoró la escena de foto con un mayor número de indicaciones, efectos e iconos.

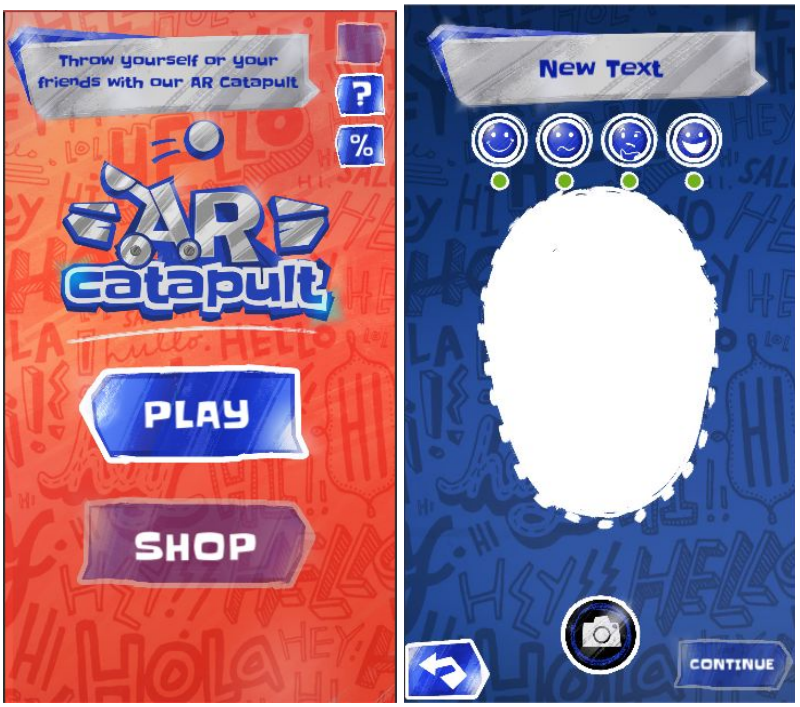


Figura 47. Menú de inicio (izquierda) y escena de fotos (derecha).

La integración del sistema GPS fue la siguiente tarea a realizar. Utilizando el plugin OnlineMaps, mencionado previamente, se utilizó la posición del jugador para, en la escena del mapa, determinar una zona de lanzamiento basado en la posición en el mundo real del jugador, lo que aporta, a la experiencia del jugador, un mayor grado de inmersión, lo que mejora su experiencia de juego y, por tanto, invierta más tiempo en el juego.

Además, se mejoró la interfaz de las fases de vuelo y caída, para que, como ya hacían previamente otras fases, mostrar en paneles, en la parte superior, información que ayuda al jugador, y en la fase de vuelo, se añadió, un indicador de la distancia recorrida y de la velocidad a la que se está desplazándose el personaje

Respecto a la jugabilidad, se mejoró la escena de lanzamiento. Se cambió el pulsar botón por mantener presionado y soltar. Al mantener presionado una barra crece y decrece, si es soltado el botón cuando la barra está crecida se recibe una recompensa. Además, se integró en el sistema de input los controles por pantalla táctil como alternativa al giroscopio.

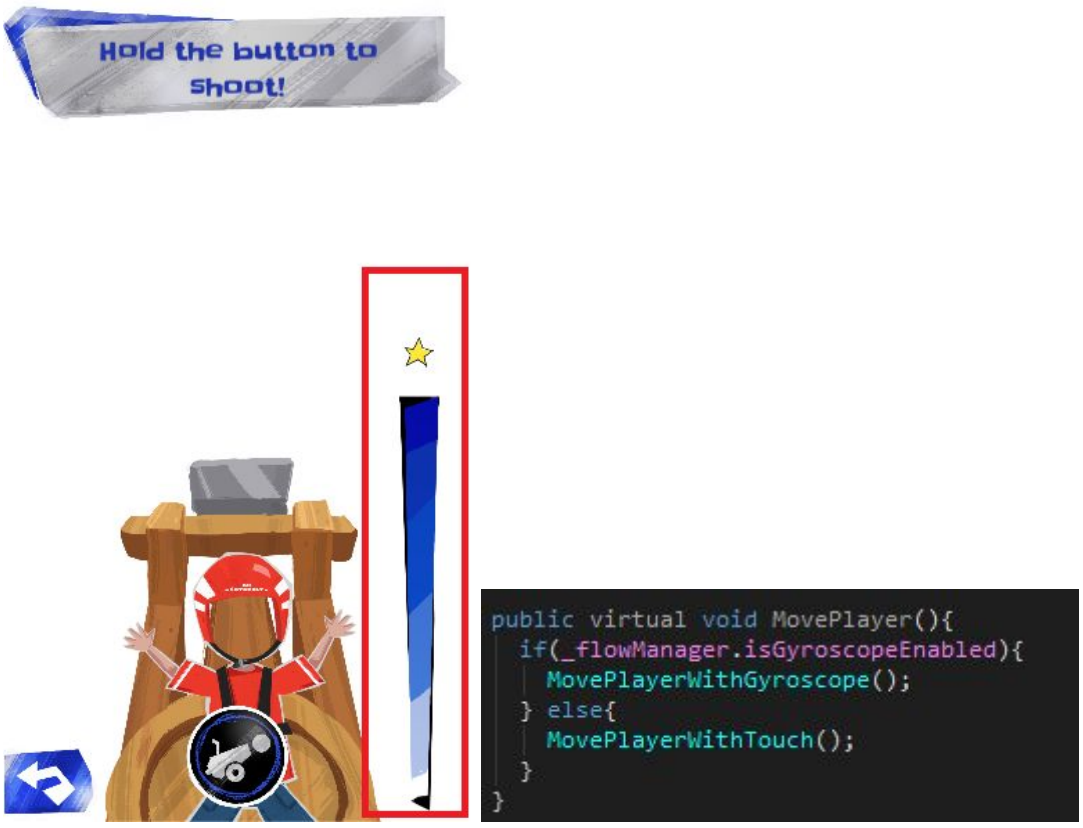


Figura 48. Escena lanzamiento (izquierda) y método movimiento del personaje (derecha)

Asimismo, se implementó la escena de puntuación, donde se muestra los puntos alcanzados y se permite volver a jugar o regresar al menú de inicio. La interfaz de esta escena está basada en el menú de inicio, lo que reduce la complejidad y el coste de la implementación.

Figura 49. Pantalla de puntuación.

Finalmente, además de solucionar una serie de errores propios del rápido desarrollo en la fase de prototipado y mejorar la flow general de información, se integraron el arte, a falta de animaciones, del juego. Por una parte, se añadió todo el arte de la interfaz gráfica de usuario (tanto imágenes como efectos). Por otra parte, se implementó el arte que forma el juego: diseño del personaje, los obstáculos, las recompensas, etc.

Figura 50. Pantalla de vuelo.

De nuevo, tras esperar la respuesta favorable del cliente, se pasó a la finalización del proyecto. Se añadieron animaciones tanto al personaje como a los obstáculos y recompensas. Se mejoraron interfaces dado el feedback del cliente. Se añadió la opción de hacerse un selfie en la escena de fotos. Se integró la hora del dispositivo móvil para que en la escena de vuelo se hiciera el cielo más oscuro. Y, en definitiva, se mejoró el producto tanto en el apartado estético como en el funcional para dar al jugador y al cliente la mejor experiencia posible.

Además, se implementó el movimiento en tiempo real del personaje por el mapa en la escena de vuelo (Parte inferior izquierda figura 50). Para ello se integró una segunda cámara que apuntaba a el mapa, se renderizó la imagen de la cámara en una textura, la textura se integró en una imagen de la interfaz gráfica de usuario y, para que la imagen tuviera las esquinas acorde con la estética del juego, se recortó la imagen aplicando una máscara con la forma deseada.

5. Conclusión

De entre todos los motores gráficos disponibles, Unity fue la elección acertada tanto por su facilidad de uso como por su rápida portabilidad a dispositivos móviles. Además, los plugins, disponibles en la AssetStore, OnlineMaps y Zenject fueron imprescindibles para la finalización del proyecto en el tiempo estimado. También resultó de gran utilidad la flexibilidad aportada por el conjunto formado por: el sistema de componentes de Unity y los GameObjects.

Dividir las diferentes partes del juego en escenas y conectarlas mediante el FlowManager permitió un rápido prototipado y redujo en gran medida los errores de programación. La realización de pequeñas clases y métodos con funcionalidades específicas redujo la duplicidad de código y, por tanto, el tiempo requerido para el desarrollo del videojuego tanto en la implementación inicial como durante el periodo de iteraciones y mejoras.

La planificación se realizó adecuadamente y todos los milestone se entregaron con las calidades y funcionalidades esperadas, con margen para la iteración, y sin ser necesario la realización de horas extras. Como todo se entregó en los plazos estimados, podemos concluir que el proyecto fue un éxito al cumplir los objetivos deseados y ratificando las motivaciones iniciales tanto a nivel económico como a nivel de aprendizaje.

6. Bibliografía

[1] Stephen E. Siwek. Video Games in the 21st Century: The 2017 Report. ESA. Disponible en https://www.theesa.com/wp-content/uploads/2019/03/2017-EIR-National-Report.pdf

[2] Libro Blanco del Desarrollo Español de Videojuegos 2018. 28 de enero de 2019. Disponible en http://www.dev.org.es/images/stories/docs/Libro%20Blanco%20DEV%202018.pdf

[3] The Companies Making The Most From Video Games. 25 junio 2019. Disponible en https://www.statista.com/chart/8870/tencent-is-top-of-the-game-revenues-leaderboard/

[4] Tencent dominates China's mobile gaming market. 25 de julio 2018,. Disponible en https://www.statista.com/chart/14846/tencent-dominates-chinas-mobile-gaming-market/

[5] Number of registered users of Fortnite worldwide from August 2017 to March 2019. Marzo 2019. Disponible en https://www.statista.com/statistics/746230/fortnite-players/

[6] SuperData. Digital Games and Interactive Media Year in Review 2018. Disponible en https://www.superdataresearch.com/market-data/market-brief-year-in-review/

[7] New GTA V release tipped to rake in £1bn in sales. 8 de septiembre 2013. Disponible en https://www.scotsman.com/lifestyle/gadgets-gaming/new-gta-v-release-tipped-to-rake-in-1bn-in-sales-1-3081943

[8] This violent video game has made more money than any movie ever. 9 de abril de 2018. Disponible en https://www.marketwatch.com/story/this-violent-videogame-has-made-more-money-than-any-movie-ever-2018-04-06

[9] Wives of Rockstar San Diego employees have collected themselves. 1 de julio de 2010. Disponible en https://www.gamasutra.com/blogs/RockstarSpouse/20100107/86315/Wives_of_Rockstar_San_Diego_employees_have_collected_themselves.php

[10] Activision Blizzard Announces Fourth-Quarter and 2018 Financial Results. 12 de febrero de 2019. Disponible en https://investor.activision.com/news-releases/news-release-details/activision-blizzard-announces-fourth-quarter-and-2018-financial

[11] Pokémon GO Catches $3 Billion in Lifetime Gross Revenue. 29 de octubre de 2019. Disponible en https://sensortower.com/blog/pokemon-go-catches-3-billion-usd-in-lifetime-revenue

[12] Erich Gamma & Richard Helm & Ralph Johnson & John Vlissides. Design patterns : elements of reusable object-oriented software. 31 de octubre de 1994, Addison-Wesley, United States.

[13] Robert Nystrom. Game Programming Patterns. 2 de noviembre de 2014, Genever Benning.