Escola Tècnica Superior d'Enginyeria Informàtica
Universitat Politècnica de València

Desarrollo de un videojuego de plataformas en C# sobre el motor Unity

Trabajo Fin de Grado

Grado en Ingeniería Informática

Autor: Jesús Martínez Abril

Tutor: Ramón Pascual Mollá Vayá

2015/2016

Resumen

En este proyecto se desarrollará un videojuego del género plataformas desde cero. Para ello, se empleará el motor Unity3D y el código será en C#. La funcionalidad del juego contará entre otras cosas con modo solitario y multijugador. Para este último modo se utilizará el framework Node.js, la librería Socket.io y su correspondiente plugin en Unity para la comunicación cliente-servidor, que permitirá a usuarios de plataformas diferentes jugar de manera conjunta. Además, este servidor contará con una base de datos realizada en SQLite donde se guardará la puntuación que obtengan los jugadores al final de la partida.

Por otro lado, el juego contará con diseños y animaciones del héroe hechos de modo clásico con Photoshop, imagen a imagen, mientras que los enemigos utilizarán una técnica más novedosa llamada rigging, que emplea esqueletos hechos con Spine. Tanto el héroe como los enemigos, así como el entorno, contarán en bastantes ocasiones con máquinas de estado que irán realizando transiciones dependiendo de la situación y de las decisiones del usuario. El héroe contará con una gran cantidad de movimientos asignados a los controles del jugador: estarán personalizados tanto para plataformas de sobremesa como para dispositivos móviles. En el primer caso, se utilizará la librería XInputDotNet para dotar de más funciones a aquellos usuarios que decidan jugar con un joystick.

Palabras clave: videojuego, Unity, C#, plataformas, multijugador, máquina de estados, físicas, plugin

Abstract

A platform video game will be developed from scratch in this project. To develop it, I will use the Unity3D engine tool and I will program it in C # language. The functionality of the game will include solo and multiplayer modes. In the multiplayer mode, it will be used the Node.js framework, the Socket.io library and its corresponding plugin for client-server communication in Unity which it will allow users of different platforms to play together. Moreover, this server will have a database coded in SQLite, where players score won will be saved at the end of the game.

On the other hand, the game will have designs and animations of the main character made by Photoshop in a classic mode, image to image, while the enemies will be made with a new design technique called rigging that use skeletons made with Spine. The main character and the enemies, as well as the environment, will have state machines in several times, which will do transitions depending on the situation and the decisions of the user. The main character will have a great number of movements connected to the player controls. These movements will be customized for desktop platforms and mobile devices. In the first case, the XInputDotNet library will be used to provide more features to users who choose to play with a joystick.

Keywords: video game, Unity, C #, platforms, multiplayer, state machine, physics, plugin

Agradecimientos

A mi tutor Ramón Pascual Mollá Vayá por su paciencia, consejos y comprensión a lo largo del proyecto.

A mis compañeros Pablo Broseta por sus ideas y diseños en el aspecto gráfico del proyecto y a Jorge Marchán por su constante transmisión de conocimiento en lo relativo a la parte servidor.

A mi familia por su apoyo incondicional en toda la carrera.

Tabla de contenidos

1. Introducción 10
 1.1 Motivación 10
 1.2 Objetivos 11
 1.3 Metodología 12
 1.4 Estructura de la obra 12
 1.5 Convenciones 13
2. Estado del arte 14
 2.1 Crítica al estado del arte 16
 2.2 Propuesta 17
3. Análisis del problema 18
 3.1 Análisis de requisitos 18
  3.1.1 Alcance 18
  3.1.2 Personal involucrado 18
  3.1.3 Referencias 18
  3.1.4 Funcionalidad del producto 18
  3.1.5 Características de los usuarios 19
  3.1.6 Restricciones 19
  3.1.7 Suposiciones y dependencias 19
  3.1.8 Evolución previsible del sistema 19
  3.1.9 Requisitos funcionales 20
  3.1.10 Requisitos no funcionales 21
 3.2 Casos de uso 22
  3.2.1 Usuario no autenticado 22
  3.2.2 Usuario autenticado 25
  3.2.3 Administrador del servidor 30
 3.3 Análisis de las soluciones 32
  3.3.1 Motores 32
  3.3.2 Servidores 37
  3.3.3 Bases de datos 39
 3.4 Solución propuesta 40

3.5 Presupuesto 41
3.6 Colaboración 43
4. Diseño de la solución 43
4.1 Análisis de las herramientas 43
4.1.1 Unity3D 43
4.1.2 NodeJS,SocketIO 51
4.1.3 SQLite 51
4.1.4 XInputDotNet 52
4.1.5 Photoshop CS5 y Spine 53
4.2 Arquitectura software 53
4.2.1 Lógica de la aplicación 53
4.2.2 Capa de persistencia 55
5. Implementación 55
5.1 Héroe 55
5.2 Enemigo 63
5.3 Consumibles y *powerups* 64
6. Pruebas 66
7. Conclusiones 67
7.1 Técnicas 67
7.2 Personales 67
8. Trabajos futuros 68
9. Bibliografía 69

Índice de ilustraciones

Ilustración 1. Consumo de videojuegos en EEUU en el año 2015 según la ESA 10
Ilustración 2. Videojuego Tennis for two 14
Ilustración 3. Máquina arcade con videojuego Bubble Bobble 15
Ilustración 4. Videojuego Abe’s Oddysee de PSX 15
Ilustración 5. Evolución de las consolas y videojuegos más representativos 16
Ilustración 6. Caso de uso de usuario no autenticado 22
Ilustración 7. Caso de uso de usuario no autenticado 25
Ilustración 8. Caso de uso de administrador del servidor 30
Ilustración 9. Editor del motor Unity3D 33
Ilustración 10. Editor del motor Unreal, con las blueprints 35
Ilustración 11. Editor del motor CryEngine 36
Ilustración 12. Diagrama de Gantt 42
Ilustración 13. Editor de Unity, principales ventanas 43
Ilustración 14. Pestaña de Assets desplegada 45
Ilustración 15. Pestaña build, scenes añadidas al proyecto 45
Ilustración 16. Pestaña de GameObject desplegada 46
Ilustración 17. Prefab desplegado, en este caso del héroe 46
Ilustración 18. Propiedades del componente Box Collider 2D 47
Ilustración 19. Propiedades del componente AudioSource 48
Ilustración 20. Propiedades del componente Rigidbody 2D 48
Ilustración 21. Ciclo de vida de MonoBehaviour, por Richard Fine 50
Ilustración 22. Ejemplo de conexión entre clientes y servidor 51
Ilustración 23. Consulta de tabla mediante SQLite 52
Ilustración 24. Joystick Microsoft Xbox 360 Controller for Windows 52
Ilustración 25. Diagrama de flujo del videojuego 54
Ilustración 26. Base de datos del servidor 55
Ilustración 27. Máquina de estados del héroe, visto desde la ventana Animator 55
Ilustración 28. Árbol de estados del héroe y condicione de transición 56
Ilustración 29. Animación del héroe “Static” 56
Ilustración 30. Animación del héroe “Walk” 56
Ilustración 31. Animación del héroe “Run” 57
Ilustración 32. Animación del héroe “Crouch” 57
Ilustración 33. Animación del héroe “JumpAndFall” 57
Ilustración 34. Animación del héroe “BasicAttack” 57
Ilustración 35. Animación del héroe “Throw” 57
Ilustración 36. Animación del héroe “Block” 57
Ilustración 37. Animación del héroe “Push” 57
Ilustración 38. Animación del héroe “Teleport” 58
Ilustración 39. BlendTree de JumpAndFall 58
Ilustración 40. Bucle Update escuchando Input de usuario para cada plataforma 59
Ilustración 41. Bucle FixedUpdate actualizando valores del Animator 59
Ilustración 42. Comportamiento de Colliders en el héroe 60
Ilustración 43. Controles de joystick que corresponden a Unity 60
Ilustración 44. Input, entrada de eje X 61
Ilustración 45. Input, entrada de botón X 61

Ilustración 46. Implementación de vibración con XInputDotNet ... 61
Ilustración 47. Evento para multijugador, disparador ... 62
Ilustración 48. Método para multijugador, emisión al servidor ... 62
Ilustración 49. Recepción del servidor y emisión a clientes ... 62
Ilustración 50. Recepción de datos en el cliente y actualización de instancia de otro jugador ... 62
Ilustración 51. Inserción en base de datos del servidor y posterior envío ... 63
Ilustración 52. Estructura del esqueleto del enemigo ... 63
Ilustración 53. Ejemplo de Corrutina de animaciones del enemigo ... 63
Ilustración 54. Comportamiento de Colliders en ataque cuerpo a cuerpo del enemigo ... 64
Ilustración 55. Comportamiento de Colliders en ataque a distancia del enemigo ... 64
Ilustración 56. Ejemplo de rango de visión del enemigo ... 64
Ilustración 57. Máquina de estados de consumible ... 65
Ilustración 58. Ejemplo de Collider en consumible ... 65
Ilustración 59. Diseño inicial del héroe ... 77
Ilustración 60. Segundo diseño del héroe ... 77
Ilustración 61. Diseño final del héroe ... 78
Ilustración 62. Montaje de la evolución del héroe ... 78
Ilustración 63. Diseño de enemigos ... 79
Ilustración 64. Diseño de NPC's ... 80
Ilustración 65. Diseño de las posibles armas de los enemigos ... 80
Ilustración 66. Diseño del powerup "Invulnerable" ... 81
Ilustración 67. Resultado de recoger el powerup "Invulnerable" ... 81
Ilustración 68. Diseño del powerup "SpeedUp!" ... 82
Ilustración 69. Diseño del powerup "Brute" ... 82
Ilustración 70. Diseño de consumibles ... 83
Ilustración 71. Diseño de objetos ... 83
Ilustración 72. Primer diseño del mapa ... 84
Ilustración 73. Boceto de la interfaz ... 84
Ilustración 74. Primer diseño de interfaz de usuario ... 85
Ilustración 75. Primer diseño de diálogos ... 85

1. Introducción

En este proyecto se describe el videojuego que se ha realizado, llamado The last hero. Se trata de un videojuego clásico de plataformas con toques de géneros como los *shoot'em up* realizado en el motor Unity, al que se le añade ciertas características tales como la posibilidad de jugar tanto en modo de un jugador como multijugador (cooperativo, en red local o en distintas redes), desde distintas plataformas (Windows, Android), cada una con su manera de controlarlo.

Se ha desarrollado con fines tanto académicos como de autorrealización, aunque una meta a largo plazo podría ser competir contra otros juegos del mismo género, pues aporta aire fresco al actual mundo de los videojuegos. Las herramientas escogidas para su realización se decidieron debido al soporte que tienen, a los conocimientos previos con ellas y mirando hacia el futuro.

1.1 Motivación

Desde hace algún tiempo, el sector de los videojuegos está experimentando un gran auge. Cada año se baten cifras récords relacionadas con las ventas, que siguen creciendo año tras año pese a la crisis mundial que se ha vivido hace poco y sigue estando a la orden del día. Al mismo tiempo, se ha popularizado el uso de *smartphones* y tabletas como dispositivos orientados al ocio relacionado con videojuegos, ya que muchos están casi al mismo nivel que muchos ordenadores o consolas convencionales en cuanto a rendimiento. Las principales *stores* como Playstore o iTunes están infestadas de juegos que han sido desarrollados tanto por desarrolladores grandes como por estudios independientes. Algunos datos del año pasado se muestran en la figura 1.

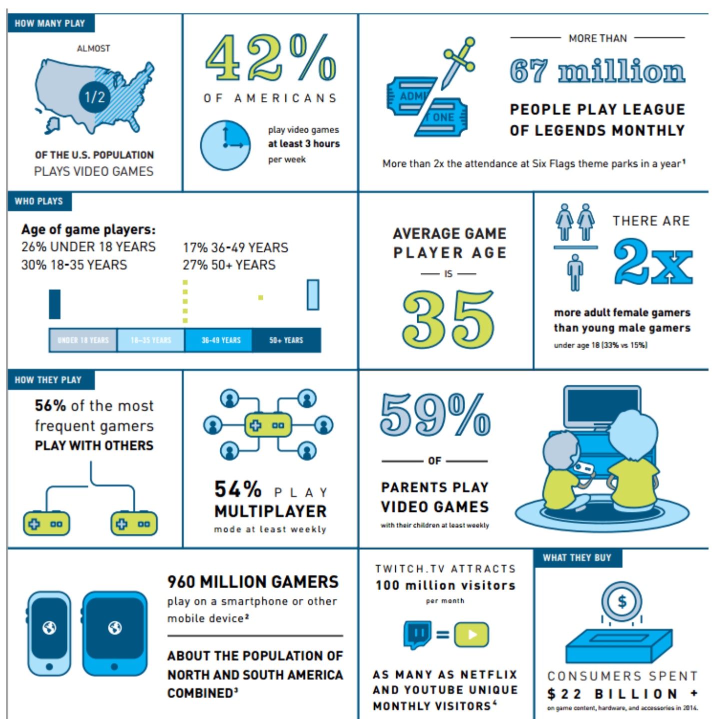


Ilustración 1. Consumo de videojuegos en EEUU en el año 2015 según la ESA

La fase de desarrollo de un videojuego siempre ha sido muy costosa de hacer y se ha necesitado mucho personal para su realización; no obstante, actualmente existe la cultura de la empresa independiente o como se autodenominan, empresas *indie*. Existen muchas herramientas que hoy en día facilitan la creación de videojuegos a equipos pequeños y medianos, entre ellas, el motor

que se utiliza en este proyecto, Unity, que permite reducir muchísimo los tiempos de desarrollo y, por lo tanto, aumentar la oferta de este tipo de productos en el mercado. Muchas de las empresas anteriormente citadas se nutren de estas herramientas. Incluso personas autodidactas, como es mi caso.

La motivación personal que ha dado paso a la realización de este proyecto ha sido el hecho de trabajar actualmente en el sector de los videojuegos, tanto para plataformas móviles como de escritorio. A esto, hay que añadir el interés que siempre me ha suscitado el género de plataformas. Con éste, se intenta realizar una propuesta diferente a los juegos que se suelen ver por las tiendas de las principales plataformas de videojuegos y volver a revivir un género al que le faltan ideas. El principal aliciente para su construcción es hacer realidad la posibilidad de jugar multijugador desde diferentes plataformas (PC, dispositivos móviles) con un gran grado de usabilidad adecuado.

1.2 Objetivos

El objetivo de este trabajo es el desarrollo de un videojuego de plataformas con la ayuda del motor de videojuegos Unity3D, cuya principal característica es la posibilidad de realizar un mismo proyecto para distintas plataforma. Entre ellas, Windows y Android, que son dos de las plataformas más importantes en el mercado. Además, se pretende el desarrollo de ciertos módulos:

- Entender los controles de juego multiplataforma.
- Entender el funcionamiento del modo multijugador y las interacciones cliente-servidor.
- Entender las interacciones entre personajes y entorno (físicas y máquinas de estado).

No se pretende realizar el videojuego entero, sino una demo para mostrar las posibilidades de las que se dispone. Una vez hecho esto, se planteará la posible ampliación del proyecto y posibles mejoras, así como la viabilidad de todas las herramientas externas que se han utilizado.

Para la realización de éste, se ha redactado un documento de diseño (GDD) que será la piedra angular del proyecto. Aquí se describe el diseño de interfaz, personajes y escenarios, así como la funcionalidad de los mismos. Consta como anexo al proyecto.

Por otro lado, también es de vital importancia para mí el proceso de aprendizaje para la realización del proyecto, ya que toda experiencia adquirida relacionada con éste ha sido de forma autodidacta y usada en mi puesto de trabajo.

1.3 Metodología

Se realizará el siguiente esquema de trabajo:

- Diseño del videojuego mediante GDD
- Creación del diagrama de Gantt para planificar el proyecto.
- Búsqueda de recursos audiovisuales.
    - Boceto del personaje principal.
    - Dibujar personaje y animaciones.
    - Obtención de recursos mediante el Asset Store.
    - Creación de escenarios.
- Programación del videojuego
    - Programación de menús e interfaces.
        - Pantalla de introducción.
        - Pantalla de inicio de sesión.
        - Interfaz del juego.
    - Programación de personaje principal y animaciones.
    - Programación de entorno e integración con los diferentes personajes y objetos.
    - Programación de controles multiplataforma.
        - Controles para plataforma Windows con teclado y ratón.
        - Controles para plataforma Windows con mando y *force feedback* mediante XinputDotNet.
        - Controles para plataforma Android con controles en pantalla.
    - Programación de servidor para modo multijugador.
        - Programación de la parte servidor con Node.js.
        - Programación de la parte cliente con Socket IO
    - Programación de sistema de base de datos y récords.
        - Programación de base de datos local con SQLite.
        - Programación de base de datos en servidor con SQLite.
- Testear y ajustar la dificultad del videojuego.
- Testeo en varios dispositivos con diferentes sistemas operativos
    - Testeo en máquina Windows, tanto con teclado como con mando.
    - Testeo en varios dispositivos Android
    - Testeo del modo multijugador con diferentes plataformas al mismo tiempo.
- Creación de la documentación

1.4 Estructura de la obra

Este TFG se divide en seis grandes bloques. El primer punto es una introducción al proyecto, que contiene la motivación del propio TFG, un resumen de la actualidad del mundo de los videojuegos y los objetivos que se desean conseguir con el proyecto, así como la metodología usada para su consecución.

En el segundo bloque se recorre la historia de la evolución tanto de los videojuegos como los dispositivos en los que se ejecutan. Una vez expuesto, se abordan la situación actual y las lagunas que existe en los temas relacionados, y se pretende dar una solución que rellene ese hueco de conocimiento, o al menos que complemente lo ya existente.

En el tercer bloque se hace un análisis de las diferentes soluciones que hay para abordar los problemas citados en el bloque anterior y elaborar la propuesta. Se determina cual es finalmente la elegida, ya que no existe una solución única, y se explica la justificación de su elección, haciendo comparativas con las demás. Por supuesto, ésta tiene que cumplir ciertos requisitos

especificados con anterioridad que deben ser preferiblemente satisfechos al acabar el proyecto. Los casos de uso se plantearán desde estos requisitos. También se realiza un presupuesto donde se muestra el esfuerzo realizado valorado de manera cuantitativa, junto a un diagrama de Gantt en el que apoyarse.

Es en el cuarto bloque donde se entra realmente en materia se desarrolla el plan de trabajo. El lector deberá entender los conceptos asociados a las herramientas seleccionadas, en este caso: **Unity3D** como motor de videojuegos, **Spine2D** y **Photoshop** para temas relacionados de diseño, **XinputDotNet** y **Unity-socket-io** como *plugins* para el correcto funcionamiento y **Node.js** y las librerías **Socket.io** y **sqlite** como servidor.

El quinto bloque contiene detalles sobre la implementación del proyecto, apoyándose en código de ejemplo y capturas de la aplicación.

En el sexto bloque se explican detalladamente todo el proceso de las pruebas realizadas y sus resultados.

El siguiente bloque, explica las conclusiones tanto a nivel técnico como personal.

Ya en el último bloque, se plantea posibles ampliaciones del proyecto o mejoras.

1.5 Convenciones

El texto de la memoria incluirá las siguientes normativas de marcado:

- Las palabras extranjeras de remarcarán en cursiva, excluyendo nombres propios.
- Se entrecomillarán las citas textuales externas a la obra.
- Se remarcarán ciertas palabras importantes del proyecto en negrita.

2. Estado del arte

Durante estas últimas décadas, el desarrollo y éxito de los videojuegos ha crecido prácticamente de forma exponencial. Tienen su origen sobre el año 1950, donde comenzaron a aparecer las primeras computadoras electrónicas y la idea de ocio con las mismas. Algunos de estos videojuegos fueron el Tennis for Two (ver **ilustración 2**) o el Spacewar: juegos sencillos que estaban dirigido a público muy selecto debido a las limitaciones tecnológicas de la época.

Ilustración 2. Videojuego Tennis for two

Acto seguido, en la época de 1970 comenzaron a descender los costes de fabricación y a salir a la luz los primeros videojuegos dirigidos a las grandes masas. Al mismo tiempo, se empezaron a desarrollar diferentes plataformas en la que se ejecutarían los mismos: las consolas. Un ejemplo de la unión de los dos conceptos podría ser el Pong de Atari (1972).

En aquel preciso instante, surgieron las primeras máquinas *arcade* y los videojuegos de plataformas, en su mayoría en 2D. Este género se caracteriza por tener que correr, caminar o saltar sobre plataformas mientras avanzas a través de las legiones de enemigos distribuidos por el mapa. Muchos de ellos también incorporan ataques del personaje, modo cooperativo y algún tipo de puntuación del jugador, para dar algo de sentido competitivo al juego. Probablemente, lo más característico de estos es la cámara que utilizan: se suele desplazar de izquierda a derecha de la pantalla, con algún movimiento mínimo en el eje vertical. Además, la existencia de gravedad para realizar las diferentes acciones como saltar fue lo que marcó la diferencia con sus antecesores.

Fueron grandes empresas de desarrollo de videojuegos las que se encargaron de estandarizar todo esto, como Nintendo o SEGA. Poseían grandes equipos de desarrollo que fueron la clave para que, en géneros como el de plataformas 2D junto a consolas como la NES (Nintendo), vivieran sus mejores momentos con juegos de la talla del Bubble Booble (ver **ilustración 3**), Metroid o Castlevania. Por supuesto, el género fue evolucionando, haciendo estilos más orientados a la exploración de escenarios que al juego lineal y rápido como el Super Mario Bros o los típicos de las máquinas de *arcade*.

Ilustración 3. Máquina arcade con videojuego Bubble Bobble

Sin duda alguna, fue en la década de 1990 cuando el PC terminó de dejar atrás la mayoría computadoras diferentes que había, lo que también significó un progreso tecnológico importante. Al mismo tiempo, ya con consolas como la Sega Mega Drive y SNES, pese a la cantidad descomunal de videojuegos del género de plataformas, se consiguió innovar algo mediante la evolución de las consolas de 8 a 16 bits. Además, se experimentó con esta nueva generación el terreno 2.5D y 3D, además de utilizar algunas técnicas de *pre-renderizado* de gráficos. El mayor pelotazo de la época fue el Super Mario 64, que presentó escenarios en tres dimensiones donde el jugador podía moverse con total libertad.

Fue en esa misma época cuando el concepto de inteligencia artificial dio un salto de calidad con uno de los títulos más conocidos de la historia, el Abe's Oddysee (ver **ilustración 4**), cuya temática estaba relacionada en la mayor parte del juego con la interacción entre enemigos y aliados, que percibían su entorno y llevaban a cabo ciertas acciones dependiendo del estado de éste último.

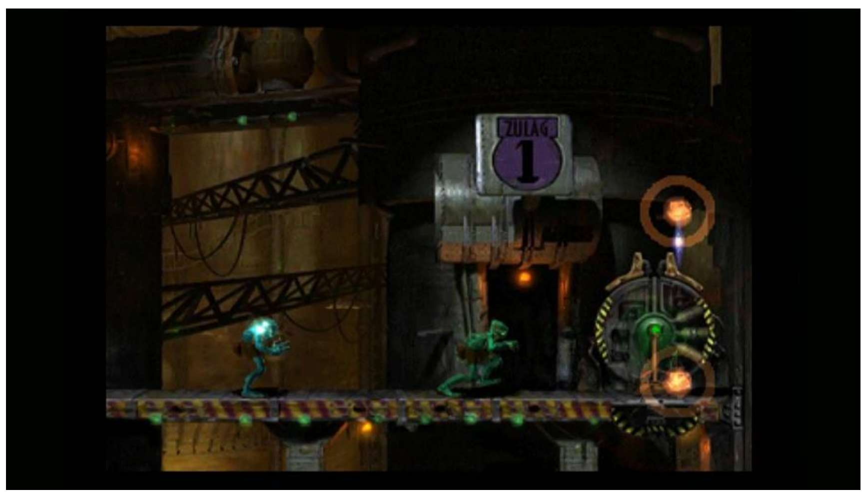


Ilustración 4. Videojuego Abe's Oddysee de PSX

Situándonos en la década de los años 2000, sigue la misma tónica de los últimos años: evolución de los *PC's* y consolas, con mejoras tecnológicas (32 y 64 bits) sustanciales con alguna que otra técnica innovadora y apostando generalmente más por el 3D. Pese a esto,

encontramos joyas como el Sonic Adventure 2 para la consola Dreamcast, que aun siendo en 2D, es considerado uno de los mejores del género en toda su historia.

Hasta el año 2010, se comienza a fraguar el denominado boom de los dispositivos portátiles. Algunas consolas como la Game Boy y luego Nintendo DS son claros ejemplos de ello. A su vez, comienzan a salir a la venta los primeros *smartphones* en el mercado, aunque fue realmente en 2007 cuando se desató la locura por ellos con la salida al mercado del iPhone de Apple, con el sistema operativo iOS. Esto dio paso a que surgieran competidores inmediatos como Google con su sistema operativo Android, o a hacer innovar a aquellos que ya estaban como Windows Phone de Microsoft o Blackberry.

A raíz de esto, el género de plataformas 2D volvió a resurgir, convirtiendo a los juegos de plataformas 3D en un género minoritario. Muchas de las franquicias que habían apostado por el 3D vuelven a la senda anterior debido a varias circunstancias:

- El coste de desarrollo es menor.
- Se puede ofrecer un apartado gráfico mejor elaborado.
- Las cámaras 2D son mucho más amigables para los usuarios.
- Por la naturaleza adictiva del género, ofrece un gran resultado en dichas plataformas.

Con la evolución de los dispositivos móviles y plataformas de distribución digital como Steam, sumado al ya citado aumento de desarrolladores independientes, el mercado se ha copado de videojuegos de todo tipo.

Ilustración 5. Evolución de las consolas y videojuegos más representativos

2.1 Crítica al estado del arte

Pese al notable incremento de la demanda de videojuegos tanto para dispositivos móviles como de escritorio, la dudosa calidad de muchos de ellos lleva a una profunda reflexión sobre el futuro de la industria de los videojuegos de hoy en día.

Muchos títulos hoy en día, incluso aquellos denominados AAA, presentan retrasos en sus lanzamientos, gran cantidad de fallos y modos de juego incompletos. No llegan a ser adictivos por las mecánicas.

Además, el modo multijugador de muchos videojuegos está limitados por plataforma. Actualmente, si un juego es lanzado en múltiples consolas y tiene también versión de *PC*, hacer

que un juego no sea *cross-platform* se debe probablemente a una decisión de diseño debido a razones de contrato. Cualquiera de los juegos de hoy en día se lanza en distintas plataformas y pese a que utilizan los mismos servidores para jugar, los usuarios simplemente pueden jugar contra otros de la misma plataforma. Muchas veces, esto es debido a que utilizan la *API* correspondiente para realizar el sistema de comunicación que llevan por detrás. Otras veces, como ante se ha citado, se deben a cumplimiento de contrato por las empresas o a las limitaciones de hardware que tienen ciertos dispositivos con algunos sistemas operativos concretos.

Otra de las razones por las cuales no se permite jugar entre plataformas distintas es la ventaja que tienen algunas sobre otras en lo que a controles se refiere. Generalmente, un usuario de *PC* tendrá ventaja sobre uno de móvil, por la comodidad del hardware que se utiliza en los primeros. No es el caso de usuarios de consolas como Xbox y PS4, que generalmente no pueden competir entre ellos sin razón aparente alguna. Sí que la hay, y no es otra que el interés que tienen las empresas de hoy en día de vender su producto.

"¿Quieres jugar con un amigo? Cómprale una de nuestras consolas o dispositivo."

2.2 Propuesta

El proyecto presenta la creación de un videojuego desde cero. Su principal objetivo es captar la esencia que tenían los juegos de plataformas de la década de 1980 con mejoras técnicas que permiten las tecnologías de hoy en día.

Empezando por el terreno del sistema de juego, emulará a los clásicos controles de los videojuegos de plataforma. Cosas simples como andar, correr, agacharse y atacar serán algunas de los movimientos que el usuario podrá realizar.

En el aspecto gráfico, se plantearán varias alternativas. Por una parte, el héroe estará realizado de una manera clásica, esto es: cada *frame* de las animaciones del mismo estarán hechas a mano por el diseñador, imagen a imagen. De diferente modo, los enemigos del juego estarán hechos con un programa basado en huesos que entre otras cosas permiten la realización del personaje por piezas. A su vez, estas piezas individuales se animarán por separado para crear animaciones de forma conjunta y así permitir el ahorro de la gran cantidad de tiempo que se utilizaría si se realizara de la forma clásica. Esto es una de las técnicas innovadoras que se están estandarizando en todo juego de plataformas que se precie hoy en día. Dota al juego de una gran sensación de fluidez.

Otra de los puntos a favor será la utilización de físicas en la escena. Esto se traduce en un entorno dinámico en el que el usuario podrá interactuar con el mismo.

Por último, el aspecto más importante del proyecto es el modo multijugador. Utilizará un sistema novedoso capaz de ser escalable. Será desarrollado de tal modo que será independiente de la plataforma en la que se ejecute tanto el juego como el servidor que atienda las peticiones de los usuarios conectados. Esto permitirá acabar con el problema citado en el apartado anterior, donde los jugadores solo podrán competir contra otros que posean el juego en la misma plataforma. Además, este modo será acompañado de una base de datos donde se harán uso de estadísticas de los usuarios.

Se ofrecerán también distintos tipos de control para cada plataforma, de forma que prime un gran grado de usabilidad.

3. Análisis del problema

3.1 Análisis de requisitos

La aplicación resultante tiene que satisfacer ciertos requisitos concretos al final de la misma. Estos determinarán la solución finalmente elegida para la implementación del proyecto y de ellos también dependerá las conclusiones y resultados que confirmen hasta donde ha llegado el trabajo.

3.1.1 Alcance

Los requisitos especificados están dirigidos a usuarios de la aplicación The last hero. Tiene como objetivo ser un juego de plataformas con modos de un jugador y multijugador, además de multiplataforma. El objetivo es recrear los juegos clásicos de este género y resaltar su sistema cooperativo independiente de la plataforma en la que se juegue.

3.1.2 Personal involucrado

A continuación se citan las personas que están involucradas en el proyecto: en primer lugar, el alumno que lo realiza y en segundo lugar el profesor que ha supervisado su desarrollo.

Nombre	Jesús Martínez Abril
Rol	Desarrollador software
Categoría profesional	Estudiante de Grado en Ingeniería Informática
Responsabilidades	Diseño, programación y análisis de la aplicación
Información de contacto	jesmarab@inf.upv.es

Nombre	Ramón Pascual Mollá Vayá
Rol	Director del proyecto
Categoría profesional	Profesor titular de la Universitat Politècnica de València
Responsabilidades	Supervisión y asesoramiento al desarrollador en su proyecto
Información de contacto	rmolla@dsic.upv.es

3.1.3 Referencias

Referencia	Título	Ruta	Fecha	Autor
1	Desarrollo de un videojuego de plataformas en C# sobre el motor Unity	Este documento	04/09/2015	Jesús Martínez Abril

3.1.4 Funcionalidad del producto

La aplicación pretende ser un videojuego que permita jugar tanto en modo un jugador como multijugador, con un sistema competitivo basado en puntos, obtenidos a lo largo del mismo.

La aplicación gestionará todos los movimientos del jugador o jugadores, y el servidor se encargará de, si es el caso, gestionar las comunicaciones necesarias para actualizar los valores de los demás jugadores. Será ese mismo servidor el encargado de guardar en la base de datos los récords hechos por los usuarios, y será a la que habrá que consultar para verlos.

3.1.5 Características de los usuarios

Tipo de usuario	Usuario no autenticado
Formación	Perfil diverso
Actividades	Ver pantalla de inicio de sesión, seleccionar las diferentes opciones del juego, iniciar sesión

Tipo de usuario	Usuario autenticado
Formación	Perfil diverso
Actividades	Jugar, salir del juego

Tipo de usuario	Administrador del servidor
Formación	Perfil diverso
Actividades	Iniciar y parar servidor, administrar base de datos, administrar servidor

3.1.6 Restricciones

- Interfaz y cámara que escalen con todo tipo de dispositivos.
- Lenguaje que usará la aplicación: C#.
- El servidor será capaz de atender varias peticiones de jugadores de forma concurrente.
- Controles personalizados para dispositivos móviles y dispositivos de escritorio.

3.1.7 Suposiciones y dependencias

Se supone el caso ideal donde los equipos en los que se va a ejecutar el videojuego tendrán una conexión a internet de banda ancha o similar, además de un sistema operativo con potencia suficiente para ejecutar la aplicación. El sistema operativo deberá estar actualizado para evitar problemas de compatibilidad.

El servidor, si se opta por el modo multijugador externo, estará alojado en un servidor OVH con una base de datos montada en SQLite. El sistema operativo será Linux. Si se desea trabajar en local, la plataforma es independiente.

Aunque el videojuego está pensado para funcionar en todo tipo de dispositivos móviles, no se asegura la funcionalidad completa en aquellos con sistema operativo Windows Phone.

3.1.8 Evolución previsible del sistema

El proyecto se desarrolla como un trabajo final de grado: su desarrollo terminará en el momento de su defensa. Por otra parte, existe la posibilidad de seguir desarrollándolo para más tarde monetizarlo o usarlo como ejemplo en las aulas.

3.1.9 Requisitos funcionales

En esta parte de la memoria se habla sobre distintas funciones tanto de la aplicación como de la parte servidor. Estos requisitos establecen cómo se comporta el sistema para cada uno de ellos.

Identificación del requisito	RF01
Nombre del requisito	Configurar dificultad
Características	Los usuarios podrán modificar la dificultad del juego
Descripción del requisito	El sistema ofrecerá a los usuarios distintas dificultades de nivel
Prioridad del requisito	Alta

Identificación del requisito	RF02
Nombre del requisito	Configurar controles
Características	Los usuarios podrán modificar el tipo de controlador que servirá como entrada de la aplicación
Descripción del requisito	Podrán elegir entre varios tipos de controles, dependiendo de la plataforma
Prioridad del requisito	Alta

Identificación del requisito	RF03
Nombre del requisito	Elegir modo de juego
Características	Los usuarios podrán elegir la modalidad en la que desean jugar
Descripción del requisito	El sistema ofrecerá a los usuarios distintas modalidades de juego para jugar en solitario o con otros jugadores
Prioridad del requisito	Alta

Identificación del requisito	RF04
Nombre del requisito	Elegir nombre
Características	Los usuarios deberán insertar un nombre para su personaje
Descripción del requisito	Deberán insertar un nombre correcto para iniciar el juego
Prioridad del requisito	Alta

Identificación del requisito	RF05
Nombre del requisito	Manejar héroe
Características	Los usuarios podrán realizar acciones con el héroe del juego
Descripción del requisito	Podrán realizar distintos movimientos
Prioridad del requisito	Alta

3.1.10 Requisitos no funcionales

A continuación se detallarán aquellos requisitos que pueden usarse para calificar operaciones del sistema, esto es: no describen información a guardar, ni funciones a realizar, sino características de funcionamiento.

Requisitos de rendimiento

El sistema debe de ser capaz de responder con una respuesta inferior a 2 segundos.

Requisitos de seguridad

La seguridad del usuario no se debe comprometer en ningún momento. La aplicación no violará la privacidad del usuario, y la información entre aplicación y servidor estará encriptada.

Requisitos de fiabilidad

Las operaciones que componen el sistema deben desarrollarse sin incidente alguno.

Requisitos de mantenibilidad

La tarea de administrar el sistema deben ser lo más sencillas posibles, para no cargar de trabajo al encargado de realizar esta actividad. Por otro lado, el sistema tiene que ser escalable por si fuera necesario añadir más funcionalidad.

Requisitos de disponibilidad

La disponibilidad del sistema debe ser a tiempo completo y en caso de fallo, reactivar el servicio se debe realizar en el menor tiempo posible.

Requisitos de portabilidad

El sistema debe de poder migrarse en el menor tiempo posible si surge esa posibilidad. Tanto aplicación como servidor.

3.2 Casos de uso

En este apartado se describen los casos de uso de la aplicación, que consta de tres actores: usuario autenticado, usuario no autenticado y administrador del servidor.

3.2.1 Usuario no autenticado

Es aquel usuario que inicia la aplicación.

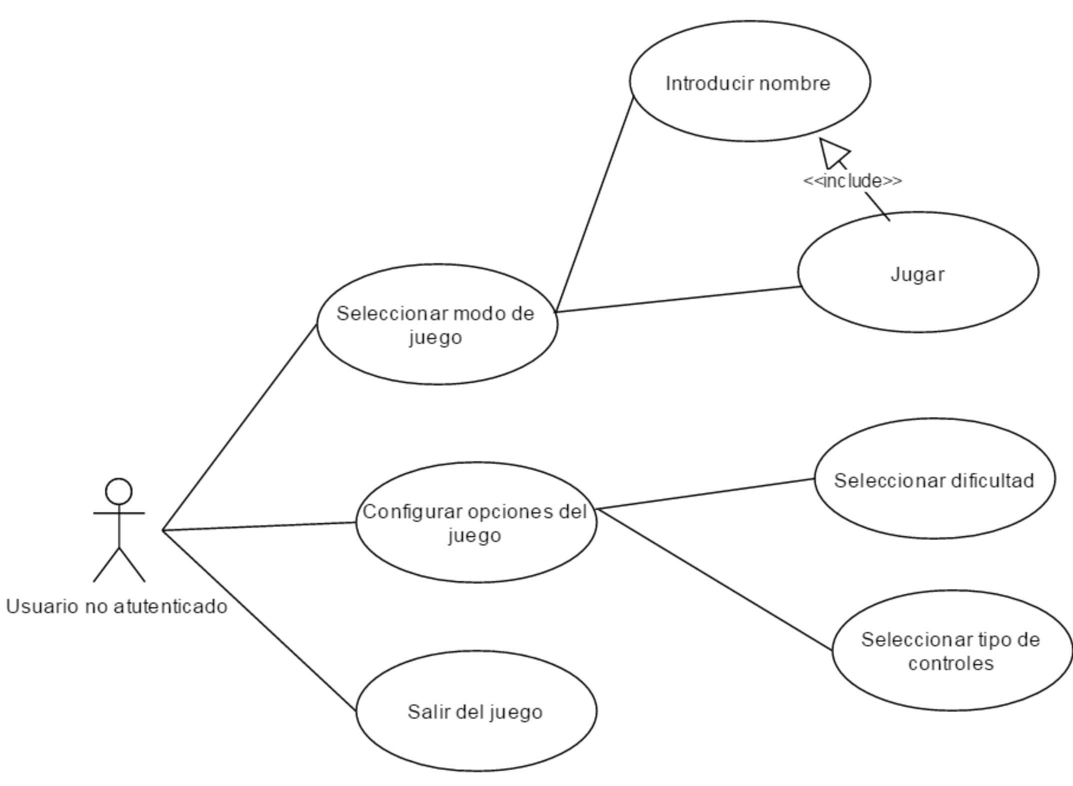


Ilustración 6. Caso de uso de usuario no autenticado

Nombre	Seleccionar modo de juego	
Descripción	El usuario puede elegir jugar en solitario o con otros jugadores	
Precondición	Ninguna	
Secuencia principal	**Paso**	**Acción**
	1.	Usuario inicia aplicación
	2.	Usuario selecciona “un jugador” o “multijugador”
Errores/Alternativas	**Error/Alternativa**	**Acción**
	No	-
Postcondición	Modo de juego seleccionado	
Notas	-	

Nombre	Configurar opciones del juego
Descripción	El usuario puede cambiar ciertas opciones del juego
Precondición	Ninguna

Secuencia principal	Paso	Acción
	1.	Usuario inicia aplicación
	2.	Usuario selecciona “opciones”
Errores/Alternativas	**Error/Alternativa**	**Acción**
	No	-
Postcondición	-	
Notas	-	

Nombre	Salir del juego	
Descripción	El usuario puede parar la ejecución de la aplicación	
Precondición	Ninguna	
Secuencia principal	**Paso**	**Acción**
	1.	Usuario inicia aplicación
	2.	Usuario selecciona salir del juego
Errores/Alternativas	**Error/Alternativa**	**Acción**
	No	-
Postcondición	-	
Notas	-	

Nombre	Introducir nombre	
Descripción	El usuario introduce un nombre para autenticarse y que el sistema lo identifique	
Precondición	Ninguna	
Secuencia principal	**Paso**	**Acción**
	1.	Usuario inicia aplicación
	2.	Usuario selecciona “un jugador” o “multijugador”
	3.	Usuario introduce nombre
Errores/Alternativas	**Error/Alternativa**	**Acción**
	No	-
Postcondición	Nombre introducido	
Notas	-	

Nombre	Jugar	
Descripción	El usuario inicia el juego	
Precondición	Nombre válido	
Secuencia principal	**Paso**	**Acción**
	1.	Usuario inicia aplicación
	2.	Usuario selecciona “un jugador” o “multijugador”
	3.	Usuario selecciona “jugar”
Errores/Alternativas	**Error/Alternativa**	**Acción**
	Usuario no ha introducido nombre válido	Se avisa de error de inicio de sesión

Postcondición	-
Notas	-

Nombre	Seleccionar dificultad	
Descripción	El usuario puede elegir el grado de dificultad	
Precondición	Ninguna	
Secuencia principal	**Paso**	**Acción**
	1.	Usuario inicia aplicación
	2.	Usuario selecciona "opciones"
	3.	Usuario selecciona "dificultad"
Errores/Alternativas	**Error/Alternativa**	**Acción**
	No	-
Postcondición	Dificultad seleccionada	
Notas	-	

Nombre	Seleccionar tipo de controles	
Descripción	El usuario puede elegir con qué tipo de control jugará	
Precondición	Ninguna	
Secuencia principal	**Paso**	**Acción**
	1.	Usuario inicia aplicación
	2.	Usuario selecciona "opciones"
	3.	Usuario elige "controles"
Errores/Alternativas	**Error/Alternativa**	**Acción**
	No	-
Postcondición	Modo de juego seleccionado	
Notas	-	

3.2.2 Usuario autenticado

Se trata del usuario que ya ha iniciado el juego.

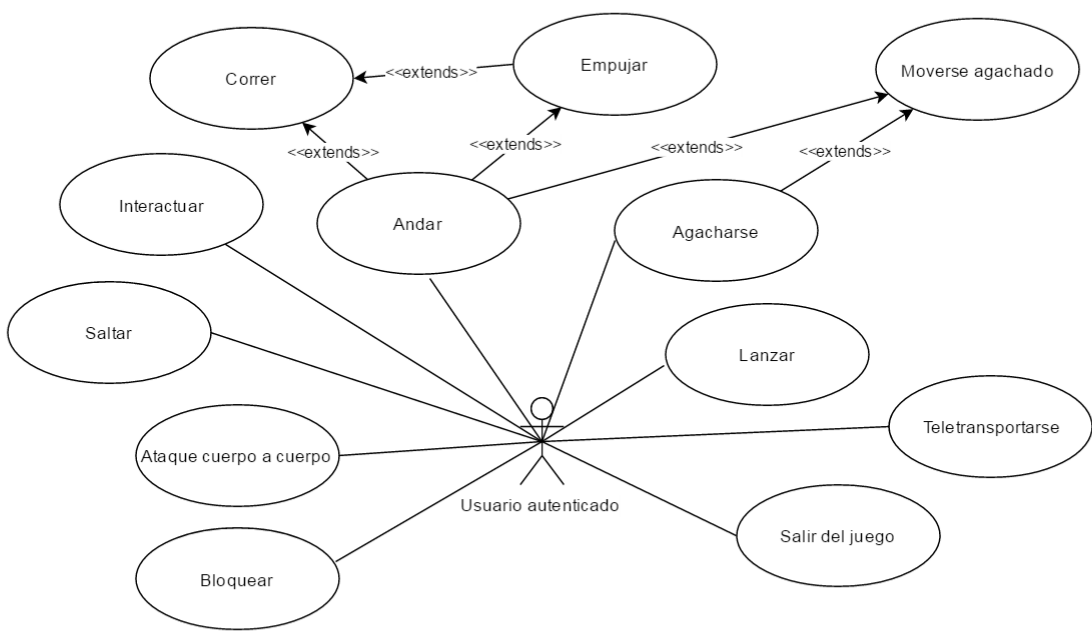


Ilustración 7. Caso de uso de usuario no autenticado

Nombre	Bloquear	
Descripción	El usuario realiza la acción de “bloquear” con el héroe	
Precondición	Usuario autenticado	
Secuencia principal	**Paso**	**Acción**
	1.	Usuario pulsa control asignado a acción “bloquear”
Errores/Alternativas	**Error/Alternativa**	**Acción**
	No	-
Postcondición	Actualiza máquina de estados del héroe	
Notas	-	

Nombre	Ataque cuerpo a cuerpo	
Descripción	El usuario realiza la acción de "ataque cuerpo a cuerpo" con el héroe	
Precondición	Usuario autenticado	
Secuencia principal	**Paso**	**Acción**
	1.	Usuario pulsa control asignado a acción "ataque cuerpo a cuerpo"
Errores/Alternativas	**Error/Alternativa**	**Acción**
	No	-
Postcondición	Actualiza máquina de estados del héroe	
Notas	-	

Nombre	Saltar	
Descripción	El usuario realiza la acción de "saltar" con el héroe	
Precondición	Usuario autenticado	
Secuencia principal	**Paso**	**Acción**
	1.	Usuario pulsa control asignado a acción "salar"
Errores/Alternativas	**Error/Alternativa**	**Acción**
	No	-
Postcondición	Actualiza máquina de estados del héroe	
Notas	-	

Nombre	Lanzar	
Descripción	El usuario realiza la acción de "lanzar" con el héroe	
Precondición	Usuario autenticado	
Secuencia principal	**Paso**	**Acción**
	1.	Usuario pulsa control asignado a acción "lanzar"
Errores/Alternativas	**Error/Alternativa**	**Acción**
	No	-
Postcondición	Actualiza máquina de estados del héroe	
Notas	-	

<table>
<tr><td>Nombre</td><td colspan="3">Teletransportarse</td></tr>
<tr><td>Descripción</td><td colspan="3">El usuario realiza la acción de “teletransportarse” con el héroe</td></tr>
<tr><td>Precondición</td><td colspan="3">Usuario autenticado</td></tr>
<tr><td rowspan="4">Secuencia principal</td><td>Paso</td><td colspan="2">Acción</td></tr>
<tr><td>1.</td><td colspan="2">Usuario pulsa control asignado a acción “teletransportarse”</td></tr>
<tr><td></td><td colspan="2"></td></tr>
<tr><td></td><td colspan="2"></td></tr>
<tr><td rowspan="2">Errores/Alternativas</td><td colspan="2">Error/Alternativa</td><td>Acción</td></tr>
<tr><td colspan="2">No</td><td>-</td></tr>
<tr><td>Postcondición</td><td colspan="3">Actualiza máquina de estados del héroe</td></tr>
<tr><td>Notas</td><td colspan="3">-</td></tr>
</table>

<table>
<tr><td>Nombre</td><td colspan="3">Salir del juego</td></tr>
<tr><td>Descripción</td><td colspan="3">El usuario termina la ejecución del juego</td></tr>
<tr><td>Precondición</td><td colspan="3">Usuario autenticado</td></tr>
<tr><td rowspan="4">Secuencia principal</td><td>Paso</td><td colspan="2">Acción</td></tr>
<tr><td>1.</td><td colspan="2">Usuario pulsa control asignado a acción “salir del juego”</td></tr>
<tr><td></td><td colspan="2"></td></tr>
<tr><td></td><td colspan="2"></td></tr>
<tr><td rowspan="2">Errores/Alternativas</td><td colspan="2">Error/Alternativa</td><td>Acción</td></tr>
<tr><td colspan="2">No</td><td>-</td></tr>
<tr><td>Postcondición</td><td colspan="3">-</td></tr>
<tr><td>Notas</td><td colspan="3">-</td></tr>
</table>

<table>
<tr><td>Nombre</td><td colspan="3">Andar</td></tr>
<tr><td>Descripción</td><td colspan="3">El usuario realiza la acción de “andar” con el héroe</td></tr>
<tr><td>Precondición</td><td colspan="3">Usuario autenticado</td></tr>
<tr><td rowspan="4">Secuencia principal</td><td>Paso</td><td colspan="2">Acción</td></tr>
<tr><td>1.</td><td colspan="2">Usuario pulsa control asignado a acción “andar”</td></tr>
<tr><td></td><td colspan="2"></td></tr>
<tr><td></td><td colspan="2"></td></tr>
<tr><td rowspan="2">Errores/Alternativas</td><td colspan="2">Error/Alternativa</td><td>Acción</td></tr>
<tr><td colspan="2">No</td><td>-</td></tr>
<tr><td>Postcondición</td><td colspan="3">Actualiza máquina de estados del héroe</td></tr>
<tr><td>Notas</td><td colspan="3">-</td></tr>
</table>

Nombre	Agacharse	
Descripción	El usuario realiza la acción de "agacharse" con el héroe	
Precondición	Usuario autenticado	
Secuencia principal	**Paso**	**Acción**
	1.	Usuario pulsa control asignado a acción "agacharse"
Errores/Alternativas	**Error/Alternativa**	**Acción**
	No	-
Postcondición	Actualiza máquina de estados del héroe	
Notas	-	

Nombre	Correr	
Descripción	El usuario realiza la acción de "andar" con el héroe	
Precondición	Usuario ha usado previamente la acción de "andar" con el héroe	
Secuencia principal	**Paso**	**Acción**
	1.	Usuario pulsa control asignado a acción "andar"
	2.	Usuario mantiene el control asignado a "andar" durante un tiempo
Errores/Alternativas	**Error/Alternativa**	**Acción**
	No	-
Postcondición	Actualiza máquina de estados del héroe	
Notas	-	

Nombre	Empujar	
Descripción	El usuario realiza la acción de "empujar" con el héroe	
Precondición	Usuario ha usado previamente la acción de "andar" o "correr con el héroe	
Secuencia principal	**Paso**	**Acción**
	1.	Usuario pulsa control asignado a acción "andar" o
	2.	Usuario mantiene el control asignado a "andar" durante un tiempo o "correr"
	3.	Usuario interactúa con objeto
Errores/Alternativas	**Error/Alternativa**	**Acción**
	No	-
Postcondición	Actualiza máquina de estados del héroe	
Notas	-	

<table>
<tr><td>Nombre</td><td colspan="3">Moverse agachado</td></tr>
<tr><td>Descripción</td><td colspan="3">El usuario realiza la acción de “moverse agachado” con el héroe</td></tr>
<tr><td>Precondición</td><td colspan="3">Usuario ha usado previamente la acción de “agacharse” y a continuación “andar”</td></tr>
<tr><td rowspan="4">Secuencia principal</td><td>Paso</td><td colspan="2">Acción</td></tr>
<tr><td>1.</td><td colspan="2">Usuario pulsa control asignado a acción “agacharse”</td></tr>
<tr><td>2.</td><td colspan="2">Usuario pulsa control asignado a acción “andar”</td></tr>
<tr><td></td><td colspan="2"></td></tr>
<tr><td rowspan="2">Errores/Alternativas</td><td colspan="2">Error/Alternativa</td><td>Acción</td></tr>
<tr><td colspan="2">No</td><td>-</td></tr>
<tr><td>Postcondición</td><td colspan="3">Actualiza máquina de estados del héroe</td></tr>
<tr><td>Notas</td><td colspan="3">-</td></tr>
</table>

<table>
<tr><td>Nombre</td><td colspan="3">Interactuar</td></tr>
<tr><td>Descripción</td><td colspan="3">El usuario realiza la acción de “interactuar” con el héroe</td></tr>
<tr><td>Precondición</td><td colspan="3">Usuario autenticado</td></tr>
<tr><td rowspan="4">Secuencia principal</td><td>Paso</td><td colspan="2">Acción</td></tr>
<tr><td>1.</td><td colspan="2">Usuario pulsa control asignado a acción “agacharse”</td></tr>
<tr><td>2.</td><td colspan="2">Usuario pulsa control asignado a acción “andar”</td></tr>
<tr><td></td><td colspan="2"></td></tr>
<tr><td rowspan="2">Errores/Alternativas</td><td colspan="2">Error/Alternativa</td><td>Acción</td></tr>
<tr><td colspan="2">No</td><td>-</td></tr>
<tr><td>Postcondición</td><td colspan="3">Actualiza máquina de estados del héroe</td></tr>
<tr><td>Notas</td><td colspan="3">-</td></tr>
</table>

3.2.3 Administrador del servidor

Es el encargado de realizar las tareas de mantenimiento en el servidor.

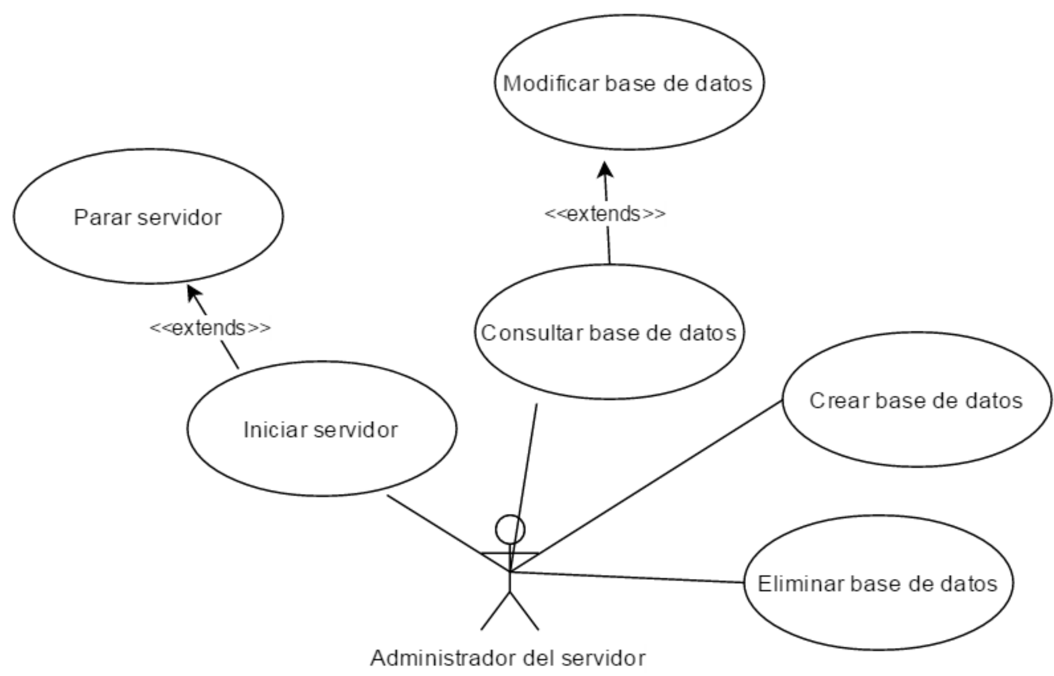


Ilustración 8. Caso de uso de administrador del servidor

<table>
<tr><td>Nombre</td><td colspan="3">Iniciar servidor</td></tr>
<tr><td>Descripción</td><td colspan="3">El administrador pone en marcha el servidor</td></tr>
<tr><td>Precondición</td><td colspan="3">Ninguna</td></tr>
<tr><td rowspan="4">Secuencia principal</td><td>Paso</td><td colspan="2">Acción</td></tr>
<tr><td>1.</td><td colspan="2">Administrador ejecuta el servidor en la máquina remota o local</td></tr>
<tr><td></td><td colspan="2"></td></tr>
<tr><td></td><td colspan="2"></td></tr>
<tr><td rowspan="2">Errores/Alternativas</td><td colspan="2">Error/Alternativa</td><td>Acción</td></tr>
<tr><td colspan="2">No</td><td>-</td></tr>
<tr><td>Postcondición</td><td colspan="3">Servidor en estado activo</td></tr>
<tr><td>Notas</td><td colspan="3">-</td></tr>
</table>

Nombre	Parar servidor	
Descripción	El administrador para la ejecución del servidor	
Precondición	Servidor en estado activo	
Secuencia principal	**Paso**	**Acción**
	1.	Administrador ejecuta el servidor en la máquina remota o local
	2.	Administrador para la ejecución del servidor
Errores/Alternativas	**Error/Alternativa**	**Acción**
	No	-
Postcondición	-	
Notas	-	

Nombre	Consultar base de datos	
Descripción	El administrador consulta los valores de la base de datos	
Precondición	Ninguna	
Secuencia principal	**Paso**	**Acción**
	1.	Administrador consulta base de datos
Errores/Alternativas	**Error/Alternativa**	**Acción**
	No	-
Postcondición	Base de datos visible	
Notas	-	

Nombre	Modificar base de datos	
Descripción	El administrador cambia valores de la base de datos	
Precondición	Base de datos visible	
Secuencia principal	**Paso**	**Acción**
	1.	Administrador consulta base de datos
	2.	Administrador modifica valores
Errores/Alternativas	**Error/Alternativa**	**Acción**
	No	-
Postcondición	Datos actualizados	
Notas	-	

Nombre	Crear base de datos	
Descripción	El administrador crea nueva base de datos	
Precondición	Ninguna	
Secuencia principal	**Paso**	**Acción**
	1.	Administrador crea nueva tabla
Errores/Alternativas	**Error/Alternativa**	**Acción**
	No	-
Postcondición	-	
Notas	-	

Nombre	Eliminar base de datos	
Descripción	El administrador elimina base de datos	
Precondición	Ninguna	
Secuencia principal	**Paso**	**Acción**
	1.	Administrador elimina tabla o archivo de base de datos
Errores/Alternativas	**Error/Alternativa**	**Acción**
	No	-
Postcondición	-	
Notas	-	

3.3 Análisis de las soluciones

En los últimos años, con la aparición de los dispositivos móviles como es el caso de los *smartphones* o las tabletas, han abierto un gran camino en el mercado de los videojuegos: aplicaciones para estos dispositivos. Al mismo tiempo, las herramientas como los motores de videojuego que se han ido aproximando a los desarrolladores han permitido satisfacer esta demanda con un incremento descomunal en el número y variedad de aplicaciones en las *stores*.

A continuación, se comparan algunas de las soluciones posibles para el proyecto. Las tres principales herramientas que se utilizarán para su desarrollo son: motor, servidor y base de datos.

3.3.1 Motores

Muchos de los que se nutren de estos motores son equipos pequeños a los que no les vale la pena desarrollar su propio motor, y solo les interesa desarrollar. Algunas de las ventajas de utilizarlos son, en líneas generales:

- Ahorro de código, puesto que en muchos casos te dan prácticamente hecho ciertas partes del mismo.
- El desarrollador se despreocupa de temas como el manejo de memoria, carga de recursos, iluminación y *renderización* de la escena, etc. Todo esto ha sido ya diseñado y testeado por la compañía que hizo el motor.

- Generalmente, los proyectos se pueden compilar para varias plataformas (*cross platform*), así que ahorran tener que hacer *ports*.

Por supuesto, también tienen sus contras:

- Para cada motor, hay que familiarizarse con la forma de desarrollar código, ya que probablemente no utilicen el mismo lenguaje.
- Si hay un bug en el motor, a no ser que sea código abierto, no se podrá arreglar.
- Aunque hay motores más enfocados hacia un tipo de juego u otro, es posible que sea menos eficiente utilizar el motor que realizar el código propio desde cero. Si se trata de un videojuego pequeño, por ejemplo.
- Generalmente, no suelen ser gratis o suelen tener limitaciones que se subsanan pagando.

Algunos de los motores más importantes en la actualidad son: Unity3D, Unreal Engine y CryEngine. A continuación se hace una breve explicación sobre cada uno de ellos.

Unity3D es un potente motor *cross-platform*, para realizar proyectos tanto 2D como 3D con un entorno de desarrollo muy ameno. Fácil de manejar para los principiantes y suficientemente potente para los expertos, permite crear fácilmente juegos y aplicaciones para multitud de plataformas.

Ilustración 9. Editor del motor Unity3D

Ventajas:

- Soporta oficialmente tres tipos de lenguaje: C#, UnityScript (básicamente Javascript) y Boo. El más generalizado en la comunidad es C#.
- Facilidad para realizar juegos tanto 2D como 3D.
- Curva de aprendizaje sencilla. Posee el Asset Store, que es una gran comunidad de creadores de *plugins* y recursos, así como muchos manuales oficiales para aprender rápido.
- Las herramientas de edición gráfica son muy buenas y pueden ser mejoradas con *plugins*.
- Soporta multitud de formatos de recursos y los convierte de manera automática a los formatos más óptimos dependiendo de la plataforma objetivo.
- Despliegue a múltiples plataformas de manera sencilla, tanto móviles como de escritorio y consola.
- Desde Unity 5.0, la licencia gratuita cubre prácticamente todas las necesidades del desarrollador, y algunas de las licencias de pago son bastante asequibles.

Inconvenientes:

- Hacer proyectos colaborativos es difícil y tedioso si no se usa su *asset server*, que es de pago. Se pueden montar alternativas para el uso de GIT, pero pueden llegar a causar problemas al actualizar recursos del proyecto.
- El rendimiento no es el mejor, ya que por la naturaleza del mismo motor, que hasta hace poco era totalmente de hilo único, no aprovecha al máximo los núcleos extra de la mayoría de dispositivos.
- El código fuente del motor no está disponible para los desarrolladores: ni siquiera en las versiones de pago. Esto significa que si existe algún tipo de bug en el mismo, la empresa deberá subsanar este fallo. Además, esto limita las posibilidades que hay de personalización del motor.

Por otro lado, **Unreal Engine** es un motor de videojuegos para PC y plataformas de consola especializado en 3D, que nació con el juego Unreal. Es sin duda alguna el más extendido en el mundo profesional del videojuego.

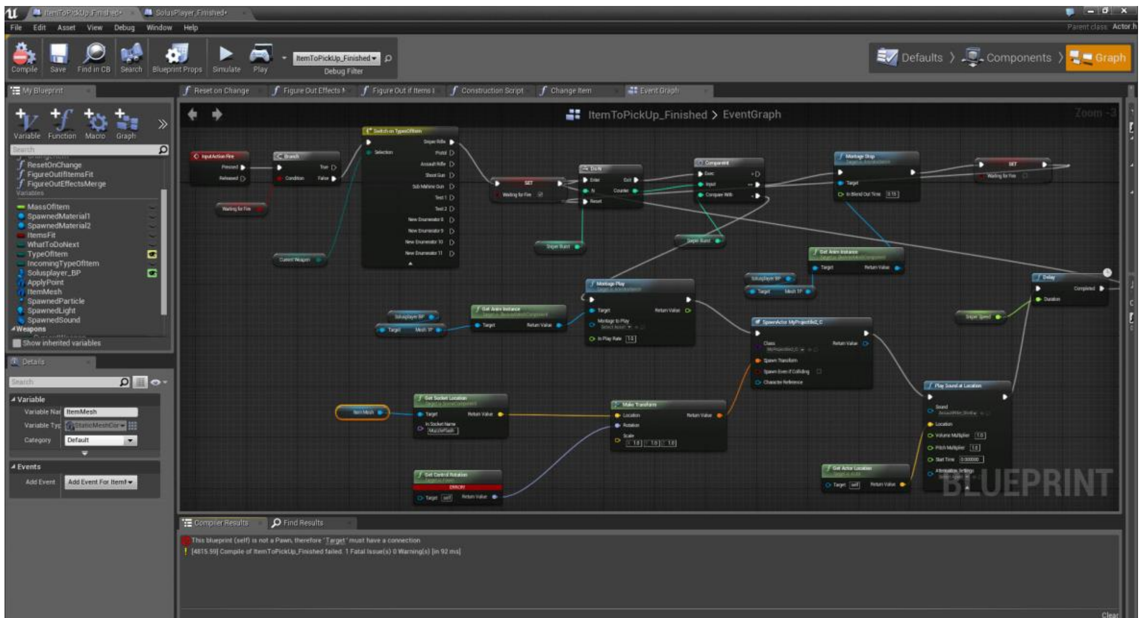


Ilustración 10. Editor del motor Unreal, con las blueprints

Su desarrollo está a cargo de la compañía Epic Games, que desde el año 1998, año del nacimiento del motor, han ido actualizando el mismo con técnicas novedosas hasta llegar a la versión actual: Unreal Engine 4. Algunos de los juegos más famosos hechos con este motor, además del Unreal, son la franquicia Gears of War y Mass Effect.

Ventajas:

- Está escrito en C++, aunque existe la posibilidad de utilizar las llamadas Blueprints, que permiten desarrollar de manera visual sin utilizar código propio alguno.
- Gran rendimiento. Es el mejor motor en cuanto a gráficos realistas se refiere por su iluminación, sobre todo en escenas con interiores.
- Control total sobre el código fuente del motor, que permite personalización total y reparación de fallos del mismo, si se dieran.
- Precios asequibles si se quiere monetizar aplicaciones.

Inconvenientes:

- Se ha de desarrollar en C++ si se pretende algo serio, y no es un lenguaje especialmente amigable para los principiantes.
- Comunidad pequeña con pocos tutoriales y recursos.

- No es posible publicar un juego si no se ha pagado al menos una mensualidad.

CRYENGINE®

CryEngine es uno de los motores más potentes que existen, superando en ciertos aspectos incluso a Unreal Engine. Aunque es conocido, nunca se ha llegado a extender tanto como otros motores como pueden ser Unity o Unreal, pero en los pocos juegos se han realizado con él se ha ganado la gran fama que tiene con creces.

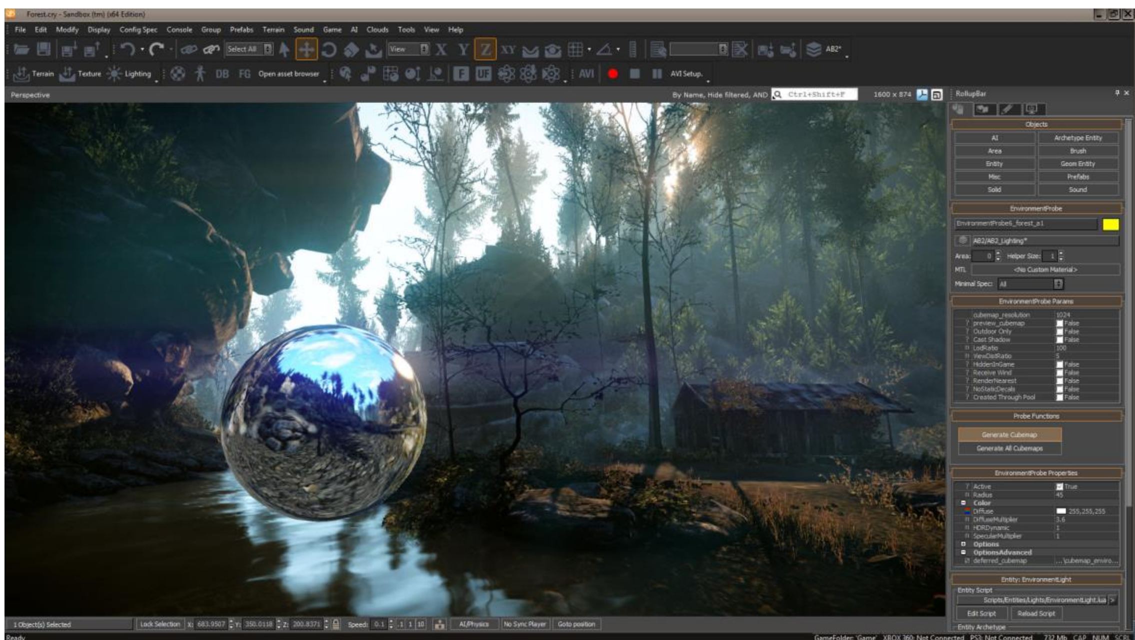


Ilustración 11. Editor del motor CryEngine

El juego más importante desarrollado con CryEngine es sin duda el Crysis, que marcó un antes y un después en lo que se refiere al apartado gráfico de un videojuego. Otros juegos Como los Sims 3 o cualquiera de la famosa franquicia Far Cry también están desarrollados con este motor.

Ventajas:

- Gran documentación.
- Control total sobre el código fuente del motor.
- Editor visual muy completo.
- Implementado completamente en puro C++.

Inconvenientes:

- Es difícil conseguir juegos que hagan bajo consumo de memoria del sistema.
- Comunidad bastante pequeña.
- Se necesita pagar licencia para hacer uso del motor, aunque sea para uso no comercial.
- La documentación está muy enfocada a juegos FPS (*First Person Shooter*).

3.3.2 Servidores

El servidor es la parte encargada de que el modo multijugador que se propone en el proyecto funciona correctamente. A continuación, se muestran algunas opciones para realizar dicha función.

Node.js es un entorno Javascript que trabaja en el lado de servidor y está basado en eventos.

Ventajas:

- Uso de un solo lenguaje de programación. Usa Javascript para el desarrollo del lado servidor, que por otro lado es lo que se suele usar en aplicaciones cliente: esto supone que no haya que hacer aprendizaje de ningún lenguaje extra.
- Fácil de escalar, tanto vertical como horizontalmente. De manera horizontal, existe la posibilidad de añadir más nodos al sistema existente, mientras se escala de manera vertical dotando a dichos nodos de más recursos.
- Gran rendimiento, debido a que se ejecuta en el motor V8 desarrollado por Google. El motor compila el código desarrollado en Javascript en código máquina de forma directa, por lo que el código es ejecutado de una manera más eficiente. Además, el entorno favorece las operaciones no bloqueantes de entrada y salida.
- Independiente de la plataforma.
- Código abierto, por lo que favorece que los desarrolladores extiendan la funcionalidad mediante la creación de librerías.

Inconvenientes:

- Ejecución en un solo hilo. Se ahorra problemas con sincronización entre hilos, sin embargo, también significa que los desarrolladores tienen que inventar soluciones para combatir los problemas relacionados con la concurrencia.
- Falta de madurez. Aunque el entorno es estable en líneas generales, es difícil calificar algunos módulos concretos del mismo.
- No es bueno en tareas con alto nivel computacional. Cuando se ejecutan tareas de larga duración, encolará las peticiones provenientes y esperarán hasta su ejecución, debido al bucle de eventos de Javascript, que es de único hilo.

SmartFoxServer es un SDK para la creación rápida de juegos y aplicaciones multijugador, famoso por su versatilidad y rendimiento, desarrollado en Java.

Ventajas:

- Gran documentación para empezar a configurar.
- Diseño de arquitectura de alto nivel.
- Herramienta de administración sencilla.
- Sesiones basadas en conexión.
- Hasta 100 jugadores disponibles con la licencia gratuita.

Inconvenientes:

- Pérdida de ancho de banda debido al diseño de eventos, comandos y nombres de parámetro.
- No soporte de UDP para eventos básicos.
- Hay que pagar licencia si se desean tener muchos jugadores concurrentes.

Photon es un *framework* que posee servidor y servicios en la nube. Desarrollado en C#.

Ventajas:

- Fantástico para juegos pequeños, utilizando la tecnología Photon Cloud, que permite ahorrar costes de tiempo al implementar el servidor.
- RUDP y UDP disponible para utilizar.
- Gran cantidad de casos prácticos y demos.
- Integrado completamente con Unity.
- Funciona en las principales plataformas: PC, dispositivos móviles, consolas, navegadores y móviles.

Inconvenientes:

- Mala documentación, ni siquiera para la lógica del servidor.
- Falta de herramienta de administración. El *dashboard* disponible no es suficiente.

3.3.3 Bases de datos

La base de datos sirve para guardar, consultar y modificar la puntuación de los usuarios del juego, que conjuntamente formarán un ranking que se traduce en una tabla en la base de datos.

SQLite es una base de datos relacional compatible con ACID, y famosa por operar con un solo fichero por base de datos.

Ventajas:

- La base de datos consta de un solo archivo en la unidad de almacenamiento.
- Extremadamente portable, debido a lo descrito en el apartado anterior.
- Aunque parezca que utiliza una implementación simple, SQLite utiliza caso todas las características de SQL.
- Buena para desarrollar y testear. Durante la fase de desarrollo de la mayoría de aplicaciones, se necesita una base de datos que escale, con la simplicidad de trabajar con un solo archivo.
- Software libre.

Inconvenientes:

- No permite la administración de usuarios, por lo que no existirá tampoco la posibilidad de dar privilegios a unos u otros tanto en las bases de datos como en las tablas del a misma.
- Por su diseño, no es posible mejorar el rendimiento de SQLite. La librería es simple de utilizar y configurar, por lo que no hay muchas posibilidades.
- No soporta concurrencia de usuarios con escritura.

MongoDB es un sistema de base de datos NoSQL. En vez de usar tablas y filas como las bases de datos relacionales, se basa en una arquitectura de colecciones y documentos que emularán su comportamiento.

Ventajas:

- Software libre.
- Realiza consultas mediante Javascript, así que son enviadas y ejecutadas de forma directa a la base de datos
- El usuario dispone de una *shard* o clave
- Configuración automática.
- Fácil replicación.

Inconvenientes:

- Bloquea la base de datos si se realiza una escritura, lo que reduce la concurrencia.
- Tiene problemas de rendimiento cuando el tamaño de los datos supera los 100GB.
- Poco fiable: si algo falla durante actualización de tabla, se pierde toda la información, y la reparación tarda mucho tiempo. Mejorable con réplicas.
- Indexación requiere mucha memoria RAM.
- No garantiza ACID.

3.4 Solución propuesta

La solución que se ha elegido en este proyecto teniendo en cuenta las herramientas mostradas en el anterior apartado gira en torno al motor **Unity**. Desde este, se creará un videojuego desde cero persiguiendo los objetivos anteriormente propuestos (ver apartado requisitos).

Los diseños de los personajes principales del juego se realizarán con las herramientas **Photoshop** y **Spine** de forma conjunta. Mientras el primero planteará la forma clásica de hacer animaciones, Spine se regirá por un sistema de huesos, también llamado *rigging*, que dotará de mayor fluidez a la animación.

A su vez, los controles del juego también suponen un reto. Se hace uso de la clase Input de Unity para crear controles específicos para cada plataforma. Además, para las plataformas donde se pueda disponer de un mando para jugar, se dispone de efectos de *force feedback* gracias a la implementación del *plugin* **XInputDotNet**.

En lo relativo a la parte multijugador, la parte servidor está compuesta por el conjunto de **Node.js** con **Socket.io**, que permite atender a las peticiones de los clientes mediante lanzamiento de eventos. En la parte cliente, se utiliza el *plugin* **unity-socket-io** para la implementación.

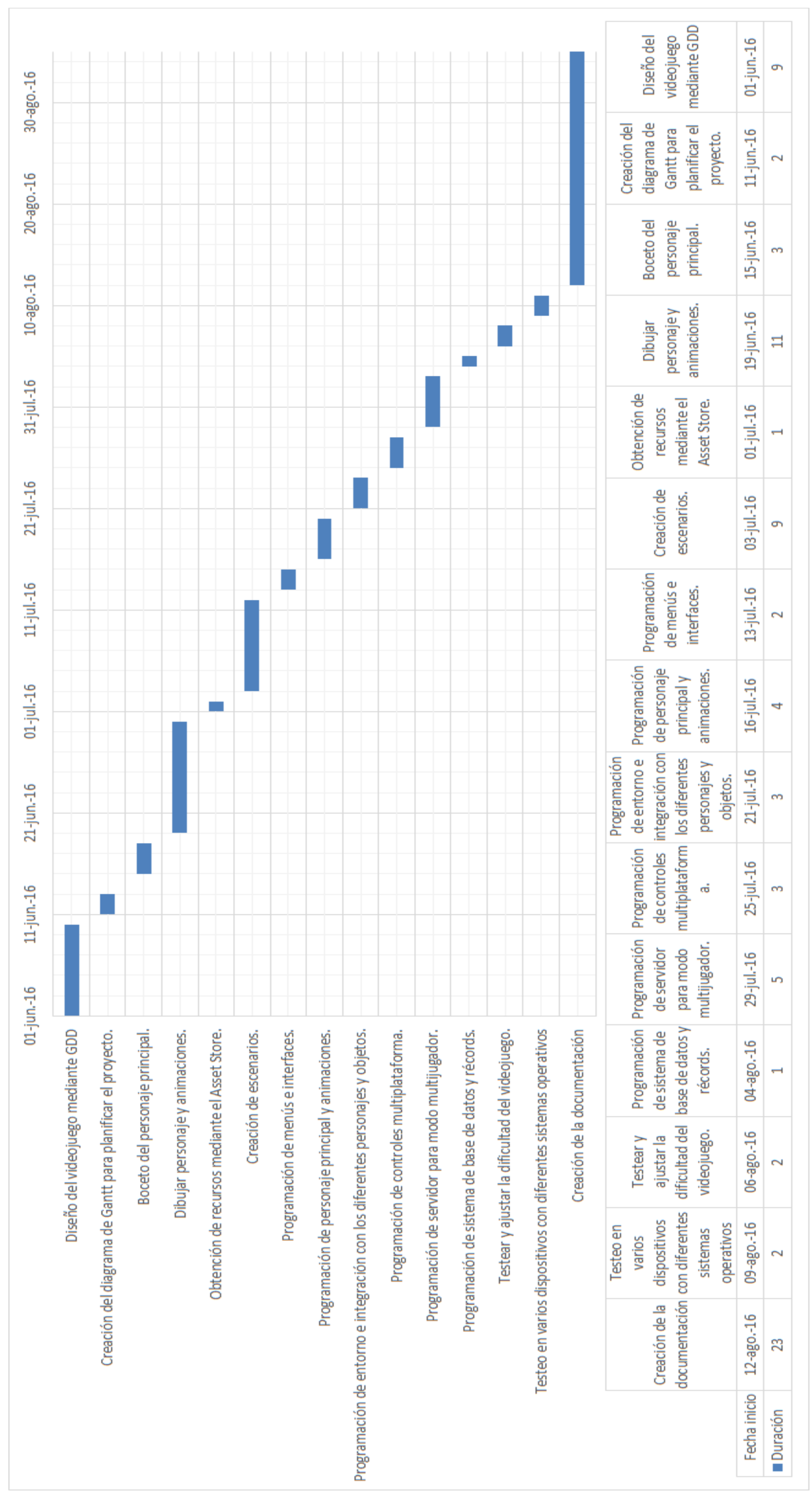
La base de datos de la que se dispone en el servidor será la sencilla y portable **SQLite**, que se implementará mediante la librería *npm* **sqlite**.

3.5 Presupuesto

A continuación se muestra el presupuesto del proyecto. Se ha tenido en cuenta tanto el coste del programador, como de las herramientas que se han utilizado en el proyecto, incluyendo tanto dispositivos para desarrollar el mismo como para realizar los test. Además, también se ha tenido en cuenta el trabajo del diseñador que colabora en el personaje principal del juego.

Presupuesto del proyecto	**(tiempo en horas, coste en euros)**
Coste de la mano de obra	
Coste hora diseñador	6
Número de horas de diseño	12
Coste hora ingeniero	6
Número de horas ingeniero	300
Total coste mano de obra	1872
21%IVA	391.12
Total coste mano de obra con IVA	2263.12
Coste del material informático	
Coste de los equipos	
Ordenador personal sobremesa	981
Dispositivos móviles	250
Total coste equipo informático	1231
Coste de material audiovisual	62
Coste del software propietario	0
Coste total del proyecto	3556.12

Estas horas son distribuidas a lo largo del tiempo que dure el trabajo. Para ello, se ha realizado una planificación aproximada del tiempo que va a conllevar el proyecto mediante un diagrama de Gantt (ver **ilustración 12**), con el que después se podrá comparar con el tiempo real que se ha tardado al final del mismo.



	Fecha inicio	Duración
Diseño del videojuego mediante GDD	01-jun.-16	9
Creación del diagrama de Gantt para planificar el proyecto.	11-jun.-16	2
Boceto del personaje principal.	15-jun.-16	3
Dibujar personaje y animaciones.	19-jun.-16	11
Obtención de recursos mediante el Asset Store.	01-jul.-16	1
Creación de escenarios.	03-jul.-16	9
Programación de menús e interfaces.	13-jul.-16	2
Programación de personaje principal y animaciones.	16-jul.-16	4
Programación de entorno e integración con los diferentes personajes y objetos.	21-jul.-16	3
Programación de controles multiplataforma.	25-jul.-16	3
Programación de servidor para modo multijugador.	29-jul.-16	5
Programación de sistema de base de datos y récords.	04-ago.-16	1
Testear y ajustar la dificultad del videojuego.	06-ago.-16	2
Testeo en varios dispositivos con diferentes sistemas operativos	09-ago.-16	2
Creación de la documentación	12-ago.-16	23

Ilustración 12. Diagrama de Gantt

3.6 Colaboración

Este proyecto ha contado con la colaboración de varios exalumnos de la Universitat Politécnica de València.

Por una parte, **Pablo Broseta**, actualmente ilustrador y licenciado en Bellas Artes, es de gran ayuda en la parte de diseño gráfico. Se encarga de dar vida al personaje principal del videojuego.

Por último, **Jorge Marchán**, actualmente desarrollador y licenciado en Grado en Ingeniería Informática, ayuda a materializar la idea de una solución diferente en el apartado relacionado con el modo multijugador del videojuego.

4. Diseño de la solución

4.1 Análisis de las herramientas

4.1.1 Unity3D

Interfaz

Unity cuenta con una interfaz muy intuitiva para usuarios principiantes. Las ventanas integradas en ella pueden amoldarse al gusto del usuario, y entre ellas encontramos las de proyecto, jerarquía, escena, juego, animador, animación e inspector. Estas son a grandes rasgos las más importantes. Se puede ver en la siguiente imagen.

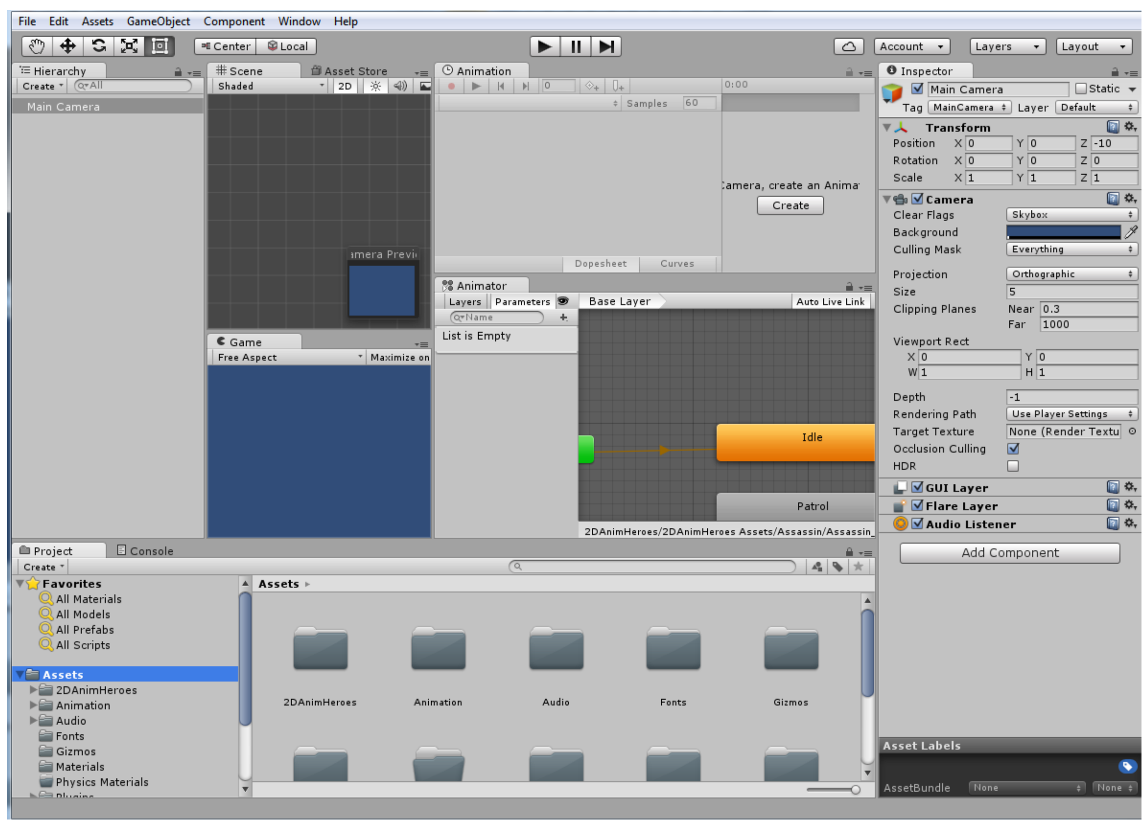


Ilustración 13. Editor de Unity, principales ventanas

La **ventana del proyecto** consta de un explorador de archivos en el que se puede añadir, organizar, modificar o eliminar elementos, llamados *assets*, del proyecto seleccionado. El directorio raíz de todos estos elementos es llamado Assets, y se sincroniza a tiempo real con la carpeta del sistema: si se realiza un cambio en la carpeta del mismo, Unity actualizará el proyecto.

Existe la posibilidad de que el usuario, por algún motivo, desee añadir elementos a esta carpeta para usarlos en un futuro, aumentando así el tamaño del proyecto. A la hora de compilar, Unity solo tendrá en cuenta los *assets* que se usen en las escenas del proyecto, dejando así de lado aquellos a los que no se les de uso alguno.

La **ventana de escena** es la que contiene la representación de aquellos elementos que se hayan instanciado en el videojuego, pudiéndolo configurar en vista 2D o 3D. Da la posibilidad de editar, escalar y mover los elementos. Además, se le puede añadir funcionalidad como la posibilidad de dibujar elementos para una mejor organización de la escena o líneas que tracen una trayectoria. Estos últimos hacen referencia al elemento *Gizmos*.

El **inspector** es probablemente uno de los elementos que hacen de Unity uno de los motores más fáciles de manejar. Permite ver que propiedades posee el elemento seleccionado y configurar muchos de ellos desde ahí mismo. La mayor ventaja es la posibilidad de ver variables públicas o serializadas de los scripts y arrastrar elementos a ellas desde el editor.

La **ventana de juego** muestra el resultado que se obtendrá al iniciar la escena actual. Esta acción se realiza desde el editor, y permite ajustar el juego a distintas resoluciones, pausarlo o ejecutar *frame* a *frame* para mayor detalle de las acciones que se realizan durante el tiempo.

La **ventana de animación**, donde se podrán realizar clips de animación desde cero y modificar frame a frame las propiedades deseadas de elementos e incluso se podrá modificar la amplitud de la curva que traza la animación durante el tiempo, mientras cambia de valor a valor.

La **ventana del animator**, donde se puede crear máquinas de estado y asignarles animaciones a cada una de las transiciones entre los mismos. Puede ser pensado como un diagrama de flujo, o un lenguaje visual de programación dentro de Unity.

Además de todas estas ventanas, también posee algunas importantes como la de la consola, que muestra todos los mensajes del sistema: errores, mensajes, etc. También podemos encontrar alguna personalizada de *plugins*.

Assets

Es uno de los componentes esenciales de todos los proyectos de Unity. Comprende multitud de elementos de distintos formatos: desde modelos 3D que sirven para formar mallas, texturas en forma de imagen hasta archivos de sonido que se utilizan como efectos en el videojuego.

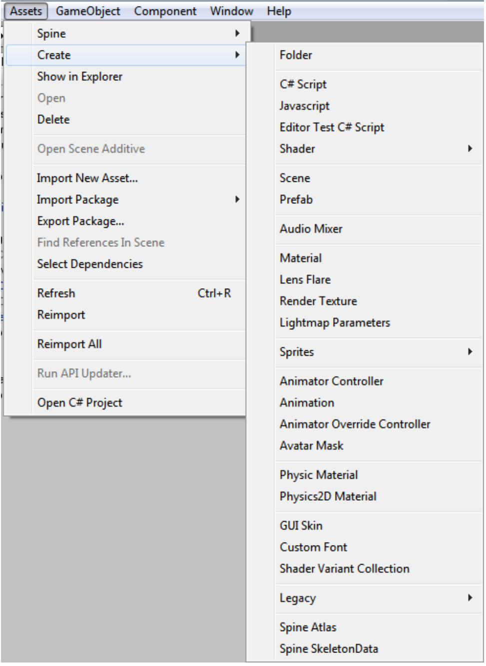


Ilustración 14. Pestaña de Assets desplegada

Todos estos se almacenan bajo un mismo directorio y constituyen todos los archivos que en un principio van a ser usados para la correcta ejecución de la aplicación. El directorio Assets está ubicado en la zona de la ventana del proyecto de la interfaz citada en el apartado anterior.

Scenes

El concepto de Scene de Unity es aquella parte del juego que contiene los objetos del mismo. Por decirlo de algún modo, es lo que permitirá a los juegos dividirlos entre varios niveles, no siendo siempre éste el caso, aunque si el mayoritario.

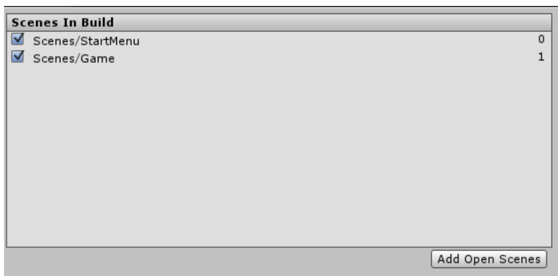


Ilustración 15. Pestaña build, scenes añadidas al proyecto

La elaboración de una aplicación como la de este presente proyecto se compone de varias escenas, en las que se hacen transiciones a través del código de la misma.

Estas escenas se manejan de manera individual y no sé puede interactuar con más de una al mismo tiempo. Se implementan en forma de jerarquía, y hay ciertos elementos que si que pueden permanecer entre ellas.

GameObjects

Los elementos u objetos del juego que componen la escena se denominan GameObjects. Algunos *assets* que se encuentren bajo el directorio raíz pueden hacer dicha función. Estos elementos deben contener al menos un componente para trabajar con ellos, y el más básico suele ser el componente Transform, que sirve para modificar la posición, rotación y escala del objeto.

De manera más concreta, un GameObject es un conjunto de componentes que vive en la escena y pueden tener otros GameObjects distintos como hijos, ya que se ordenan de forma jerárquica.

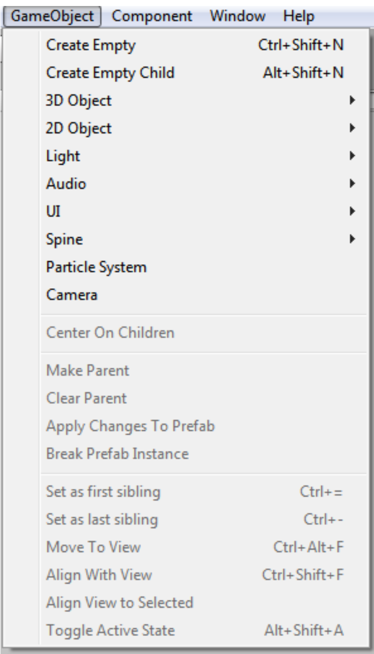


Ilustración 16. Pestaña de GameObject desplegada

Prefabs

Un prefab es una copia de un GameObject y convertida en un recursos reusable. Se muestran en el directorio del proyecto y está serializado en el disco como un archivo.

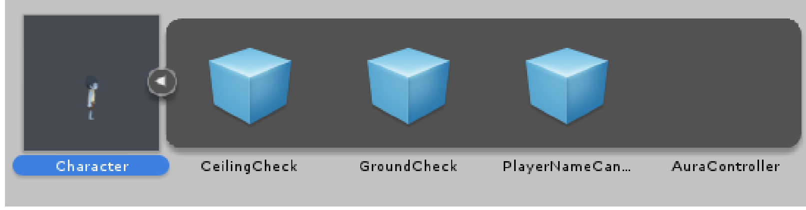


Ilustración 17. Prefab desplegado, en este caso del héroe

De forma simple, se puede crear un GameObject directamente, sin embargo, también se puede crear uno nuevo instanciado un prefab como un GameObject, arrastrándolo desde la ventana de proyecto a la vista de la escena.

La característica más notable de los prefabs es la facilidad que existe para realizar cualquier cambio en ellos. Aquellos que se realicen pueden aplicarse a todas las instancias que existan en ese momento. Del mismo modo, si se añade o quita un componente de él, esa conexión entre prefabs quedará rota.

En este proyecto se hace gran uso de *prefabs* para instanciar héroes de los diferentes jugadores, enemigos y objetos del entorno.

Components

S aquellos elementos que están adheridos a los GameObjects y que puede influenciar en su comportamiento o en el de otros de diferentes maneras. Algunos de los componentes más básicos son el Transform, cuya función es modificar la rotación, posición o escala del objeto, y el Camera, que es el que permite ver de una manera u otra el juego en sí, tanto desde la ventana de juego como en la propia aplicación.

Además de estos últimos, algunos de los más importantes en este proyecto serán los siguientes:

- **Colliders**. Define en los objetos un área en la que colisionarán con otros objetos que posean el mismo elemento. Podrán chocar con otros objetos que posean este elemento, o bien si hacemos que actúe como un disparador, simplemente detectara las colisiones con otros objetos, sin necesidad de chocar. En este trabajo se usan los tipos primitivos BoxCollider2D y CircleCollider2D en la mayoría de los casos. Sirven tanto para calcular las colisiones con el héroe como para los enemigos y el entorno.

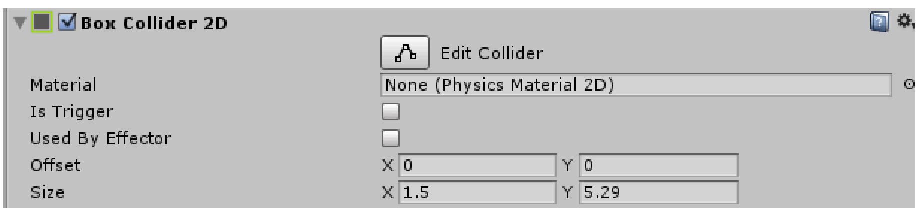


Ilustración 18. Propiedades del componente Box Collider 2D

- **AudioSource**. Se encarga de reproducir los AudioClip en la escena. Puede ser configurado para que reproduzca estos sonidos en modo 2D, 3D o una mezcla de los dos. El audio puede ser esparcido entre los altavoces (*spread*) o transformado entre 3D y 2D (*spatialBlend*) mediante las curvas de difuminación que posee. Los personajes principales del juego poseen uno, mientras que en el menú y en el juego usan otros propios para la música de fondo.

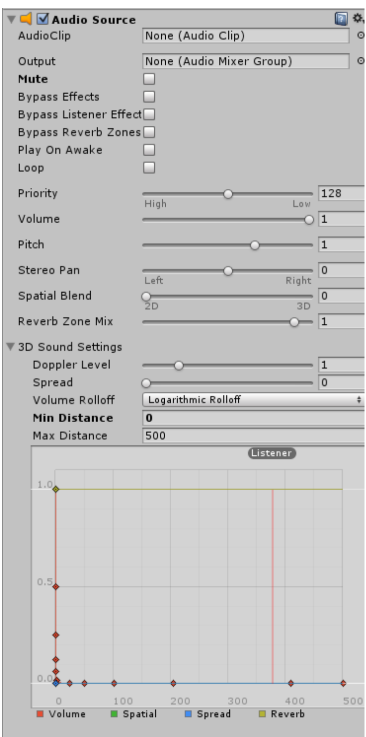


Ilustración 19. Propiedades del componente AudioSource

- **Rigidbody**. Es el componente que dota de físicas al GameObject. Algunos de sus parámetros modificables son la masa del objeto, el arrastre tanto normal como el angular, congelar el eje del objeto, la posibilidad de cambiar el modo de detección de colisiones o la posibilidad de cambiar la gravedad a la que se expone. La mayoría de elementos dinámicos del juego poseen este componente.

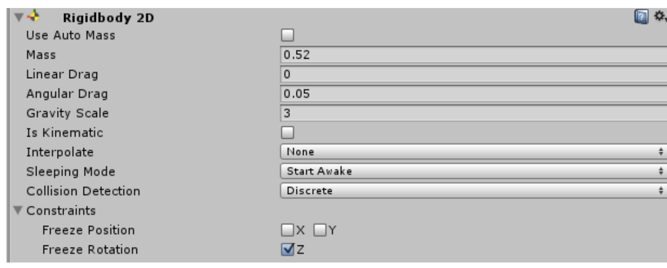


Ilustración 20. Propiedades del componente Rigidbody 2D

Scripts

Los scripts o secuencias de comando son un componente esencial en el funcionamiento del juego. Es lo que dota al juego de la mayoría de la funcionalidad modificando propiedades y valores de cualquier objeto.

Para la escritura de dichas secuencias, se hace uso del editor que trae Unity por defecto, llamado Monodevelop, aunque es posible sustituirlo por otro como Visual Studio. Si se guarda o modifican dichas secuencias en el editor de *scripts*, se actualizará de manera inmediata en el editor de Unity.

Para realizar *scripting* no es necesario conocer el funcionamiento interno del motor, pero si es muy importante la clase MonoBehaviour, que es la que traen por defecto los scripts. A continuación se muestra su ciclo de vida.

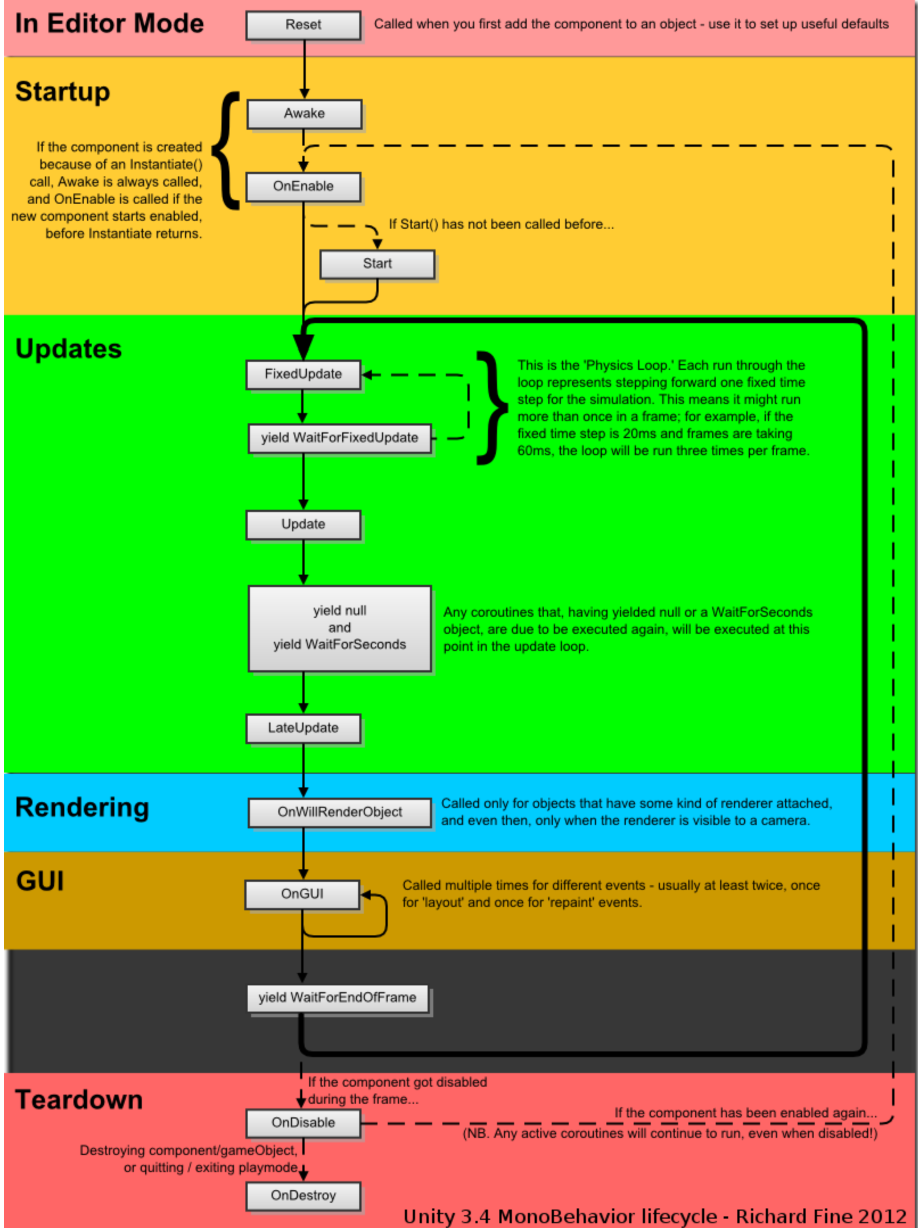


Ilustración 21. Ciclo de vida de MonoBehaviour, por Richard Fine

Cabe destacar algunos métodos importantes que tienen un gran peso dentro de este proyecto:

- **Start**. Es llamado al inicio de la creación del GameObject que lo posee. Generalmente se usa para asignar valores por defecto al script
- **Awake**. Se ejecuta antes que el método Start. Este tipo de iniciación en dos tramos de tiempo distintos puede resultar muy útil a algunos scripts si para su propia inicialización necesitan que otros scripts ya tengan algunos valores asignados.
- **Update**. Este método se ejecuta una vez por *frame*. Es útil para actualizar los *sprites* de la animación de un objeto o realizar bucles. El tiempo que tarda en ejecutarse puede variar dependiendo de las imágenes por segundo en ese preciso instante.
- **FixedUpdate**. Es invocado en cada iteración del bucle de físicas. A diferencia del Update, este sí que tiene un tiempo fijo entre llamada, por lo que es muy útil para consultar o modificar las físicas de los objetos. Sin embargo, también se debe tener en

cuenta que la sobrecarga de este método puede repercutir muy negativamente en el rendimiento del juego.

4.1.2 NodeJS,SocketIO

Node.js es un *framework* de Javascript de lado de servidor para implementar operaciones de entrada y de salida. Utiliza un modelo asíncrono basado en eventos y *streams*, y está construido sobre el motor Javascript V8, que es con el que funciona el Javascript de Google Chrome. Este último es actualizado de forma constante y es de los intérpretes más rápido que existe en la actualidad en lo que se refiere a lenguaje dinámico. Soporta protocolos TCP, DNS y HTTP.

Al contrario que el Javascript al que están acostumbrados muchos programadores, NodeJS se programa del lado del servidor, indicación de que el desarrollo de software se realiza de forma diferente al Javascript del lado del cliente. Por tanto, cuando trabajamos con éste, solamente existe la preocupación de que el código que se escriba se ejecute correctamente en el servidor.

Node es especialmente bueno en situaciones donde se requiere:

- Tiempos de respuesta bajos y alta concurrencia.
- Carga grande de usuarios.
- Solución de código abierto.
- Se optará por utilizar librerías que hacen usuarios.

A la vez que utilizamos Node.js como servidor, el encargado de manejar los eventos que lleguen al servidor es Socket.io. Se trata de una librería que permite manejar eventos en tiempo real y de manera bidireccional mediante una conexión TCP (ver **ilustración 22**). Está basado en Websockets y trabaja sobre un único socket TCP.

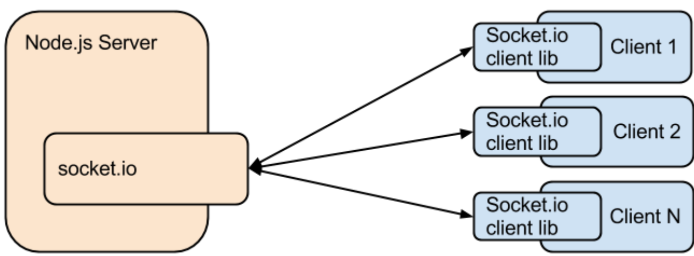


Ilustración 22. Ejemplo de conexión entre clientes y servidor

En la parte servidor, utilizaremos la librería Socket.io para manejar eventos, mientras que con Unity, en la parte cliente, se hará uso de la librería **unity-socket.io**, de código abierto para la comunicación.

4.1.3 SQLite

Se trata de un Sistema de gestión de base de datos relacional escrita en lenguaje C, que implementa un manejador de base de datos SQL embebido. Su código fuente es de dominio público con licencia GPL y es compatible con ACID.

La principal característica es el soporte completo para tablas e índices en un archivo por base de datos. Además, los tipos se asignan a valores individuales. Una columna puede albergar filas valores de diferentes tipos.

La base de datos puede ser consultada concurrentemente por varios procesos o hilos, pero el acceso de escritura solo puede ser servido si no existe ningún otro acceso de manera concurrente. En caso de fallo, se puede configurar bien para que devuelva un código de error o lo vuelva a reintentar de manera automática durante cierto tiempo.

Mediante el programa **sqlite**, que es multiplataforma, se puede consultar y modificar las tablas de manera manual. A continuación, se muestra un ejemplo.

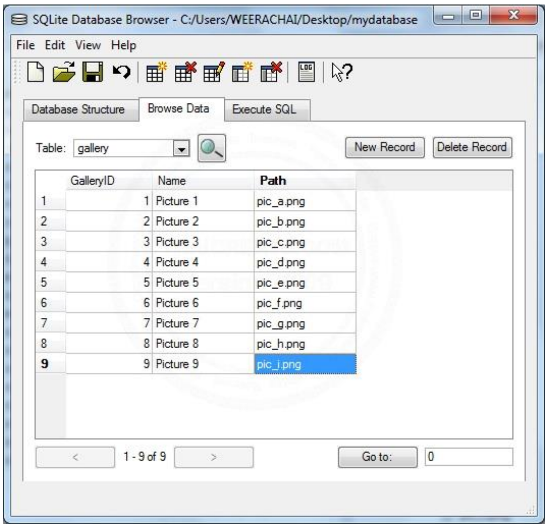


Ilustración 23. Consulta de tabla mediante SQLite

En este trabajo se hace uso de la base de datos para guardar los récords de puntuación que los diferentes jugadores realizan al acabar la partida. Se utiliza la librería hecha en Javascript **sqlite3**.

4.1.4 XInputDotNet

Cuando el usuario de ciertas plataformas como PC desee utilizar como tipo de controlador un joystick, además del componente Input de Unity, hará falta acceder a ciertas propiedades del mando como el *force feedback* o vibración.

Para realizar esto, se utiliza la librería XInputDotNet, *wrapper* que gira en torno a la API XInput de C y sirve para Unity. Es una API de DirectX para administrar los controladores conectados al PC, en este caso, un controlador del tipo Xbox 360/PC como el que muestra la siguiente figura.

Ilustración 24. Joystick Microsoft Xbox 360 Controller for Windows

La librería ahorra programar a bajo nivel con la que debería ser la clase XInput de C y permite consultar y modificar valores de las propiedades del dispositivo de forma rápida y sencilla.

4.1.5 Photoshop CS5 y Spine

Photoshop y Spine son las herramientas de edición utilizadas para realizar el diseño de los personajes.

En este proyecto, se propone la creación de un héroe de forma clásica, realizado a mano, frente a unos enemigos hechos mediante huesos, también denominado *rigging*. En el primer casi, se usa la herramienta Photoshop para crear las animaciones, haciendo un conjunto de *sprites* sobre un mismo elemento que se van intercambiando para darle vida al personaje. Esta es la manera clásica, más costosa y menos preparada ante los posibles cambios. Para cambiar una animación se necesita cambiar prácticamente todos los *sprites* de la misma, teniendo que realizar de nuevo el proceso de dibujo.

Por otro lado, con Spine se conseguirán solventar estos problemas. El diseño de los personajes se realiza mediante la división del *sprite* que deseemos en piezas, que se animarán como si fuera una marioneta. Esto, añadido a las características de este programa, que permiten deformar malla y modificar parámetros de manera rápida, hace que se consiga una fluidez impresionante en las animaciones finales. Además, ante cualquier cambio, no hará falta dibujar de nuevo al personaje, puesto que solo habrá que modificar la animación de las piezas deseadas.

Para importar personajes hechos con Spine, en este trabajo se utiliza el plugin **spine-runtimes**, que añade funcionalidad para cargar, manipular y *renderizar* animaciones de tipo esqueleto usando Unity.

4.2 Arquitectura software

4.2.1 Lógica de la aplicación

A continuación se explica el comportamiento de la aplicación mediante un diagrama de flujo (ver **ilustración 25**), mostrando las acciones que se pueden realizar en la misma. La leyenda de las figuras que se utilizan en el diagrama son las siguientes:

	Este icono representa las distintas **vistas** de la aplicación.
	Determina las **acciones** a realizar en la vista.
	Representa situaciones donde el usuario, al realizar una acción, es evaluado por el sistema y determina el **camino** a seguir.
	Mediante este icono se representa el **inicio** o **fin** de la ejecución de la aplicación.

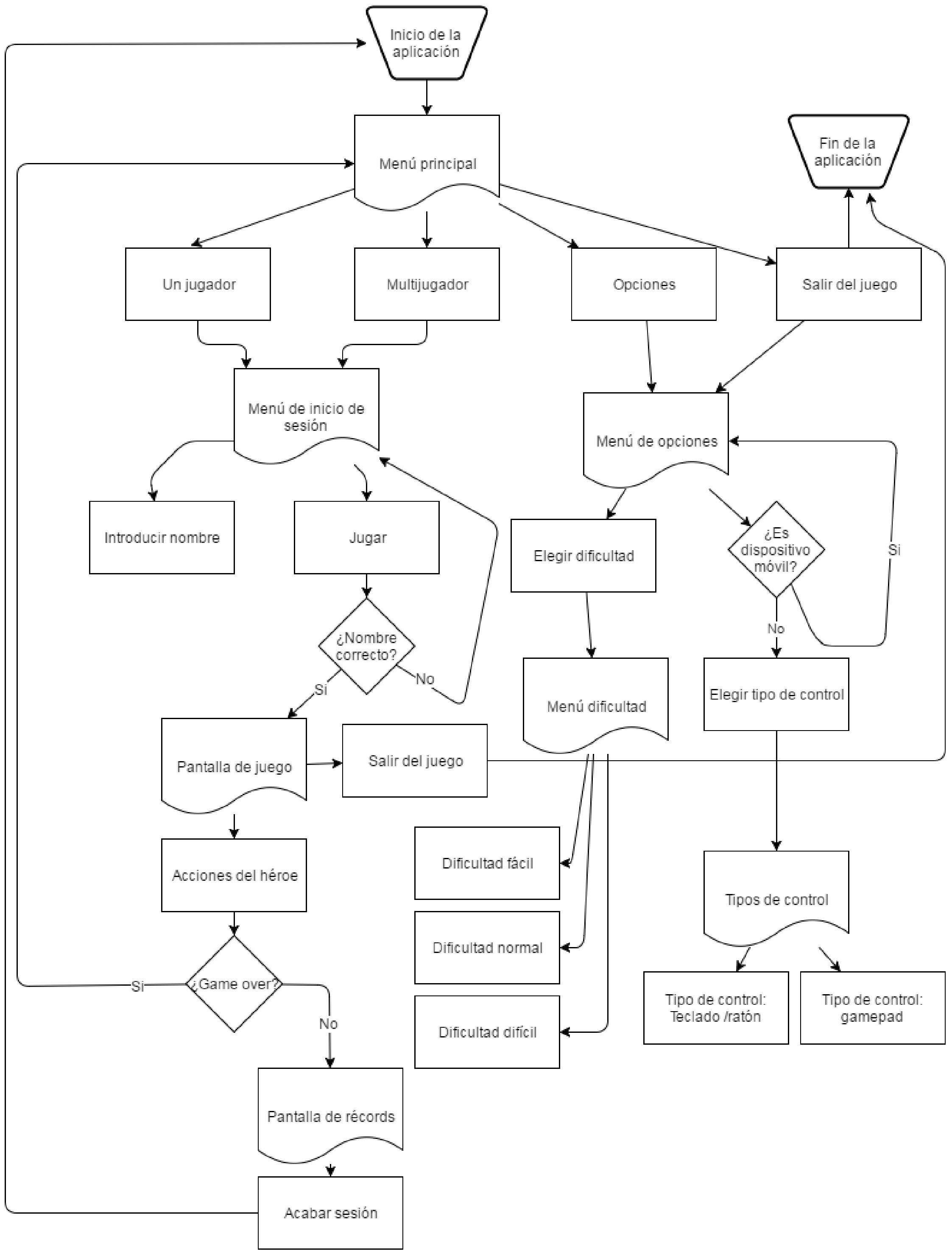


Ilustración 25. Diagrama de flujo del videojuego

Se puede observar que la aplicación es bastante básica y la mayoría de los elementos giran en torno al menú principal. También

4.2.2 Capa de persistencia

La base de datos que se utiliza en el servidor será muy sencilla, debido a las necesidades mínimas del sistema. Simplemente necesitará guardar récords de los usuarios de la aplicación junto a su nombre. La tabla se muestra en la siguiente ilustración.

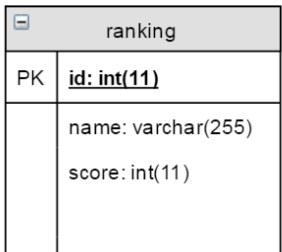


Ilustración 26. Base de datos del servidor

5. Implementación

En este apartado se observa la máquina de estado de objetos como el héroe, el enemigo o los consumibles del juego. Se explican las técnicas que se utilizan para variar esos valores mediante pequeñas porciones de código de ejemplo. Además, se explica la interacción entre cliente servidor.

5.1 Héroe

Para la implementación de las acciones que puede tomar el héroe, se ha optado por realizar una máquina de estados que es dependiente de la entrada que obtenga en la clase Input de Unity. El árbol de estados resultante se muestra en la figura de abajo, que muestra la ventana animator de Unity.

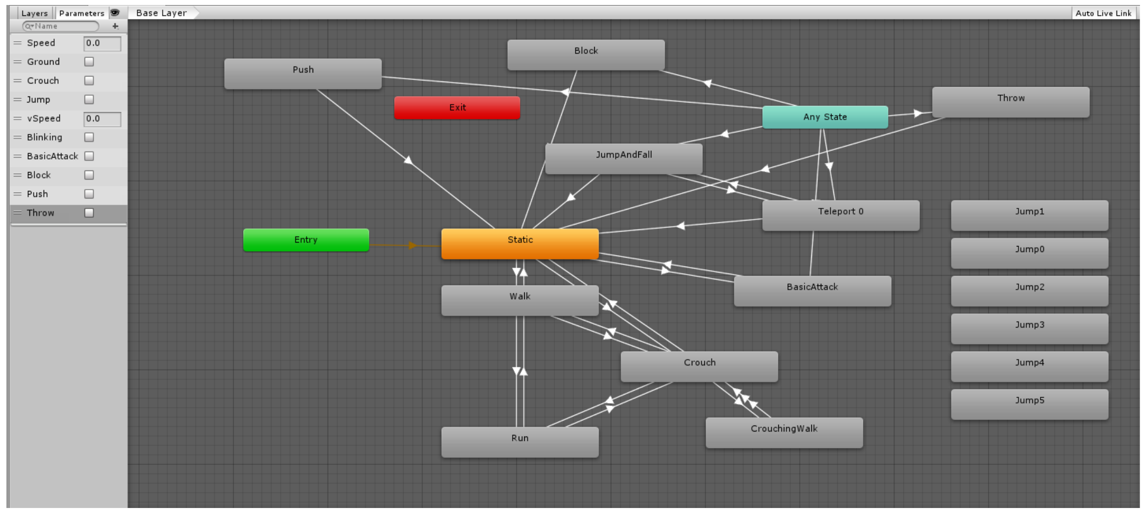


Ilustración 27. Máquina de estados del héroe, visto desde la ventana Animator

Cada uno de esos estados tiene asignado un clip de animación. Para llegar a cada uno de ellos, se debe pasar por una o más transiciones que deben reunir ciertas condiciones para hacerlo de manera exitosa. Esas condiciones aparecen en el la pestaña Parameters de la anterior figura, que

se irán actualizando a lo largo de la ejecución del juego. En la siguiente ilustración se mostrará la máquina de estados con las condiciones de cada transición.

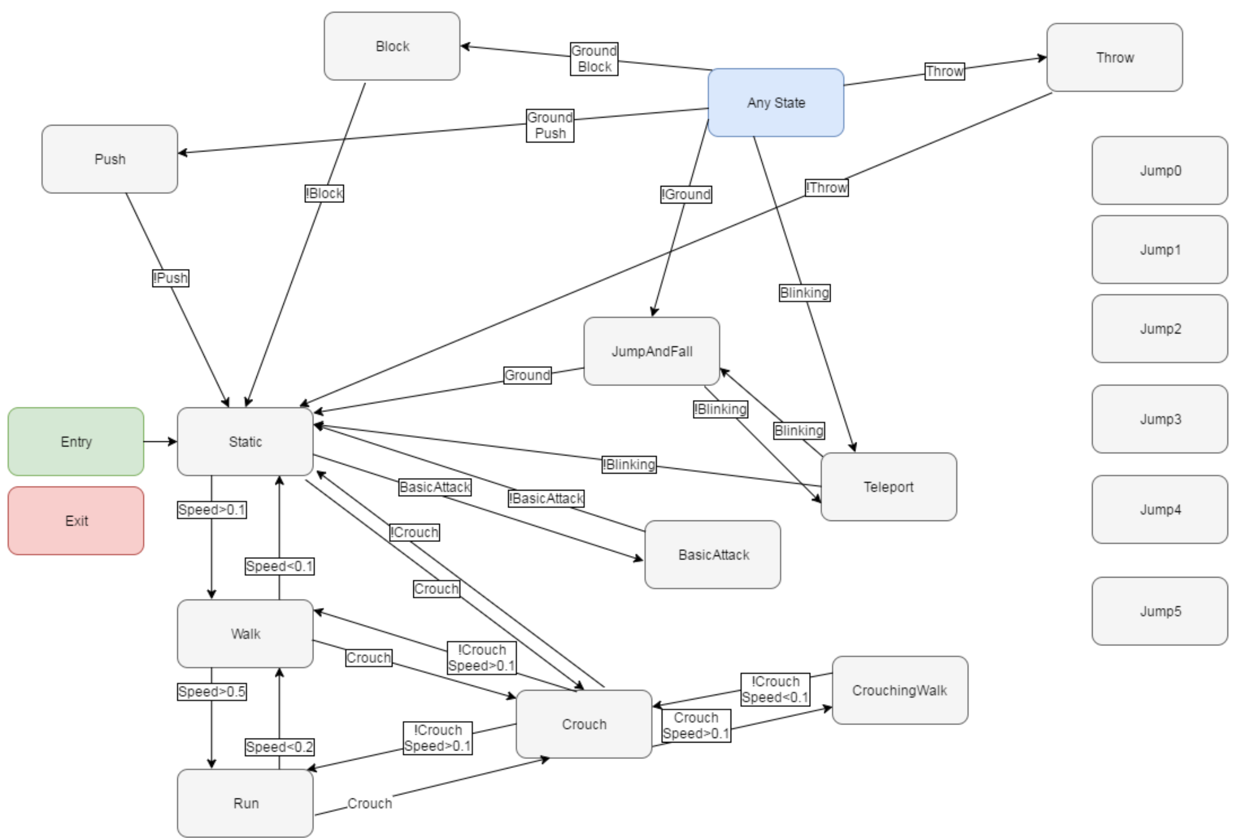


Ilustración 28. Árbol de estados del héroe y condicione de transición

El estado “Entry” representa el inicio de la máquina de estados, que tiene como estado por defecto “Static”. El estado “Any State” representa que desde cualquiera de los otros estados, se podrá tomar cualquiera de los que están ligados a él. El estado “Exit” representa el último de los estados.

Cada uno de los otros estados, como bien se ha afirmado antes, tendrán un clip asociado, que no es más que una animación. En las siguientes ilustraciones se muestran todas ellas.

Ilustración 29. Animación del héroe “Static”

Ilustración 30. Animación del héroe “Walk”

Ilustración 31. Animación del héroe "Run"

Ilustración 32. Animación del héroe "Crouch"

Ilustración 33. Animación del héroe "JumpAndFall"

Ilustración 34. Animación del héroe "BasicAttack"

Ilustración 35. Animación del héroe "Throw"

Ilustración 36. Animación del héroe "Block"

Ilustración 37. Animación del héroe "Push"

Ilustración 38. Animación del héroe “Teleport”

El estado “JumpAndFall” se considera especial, debido a su condición de BlendTree.

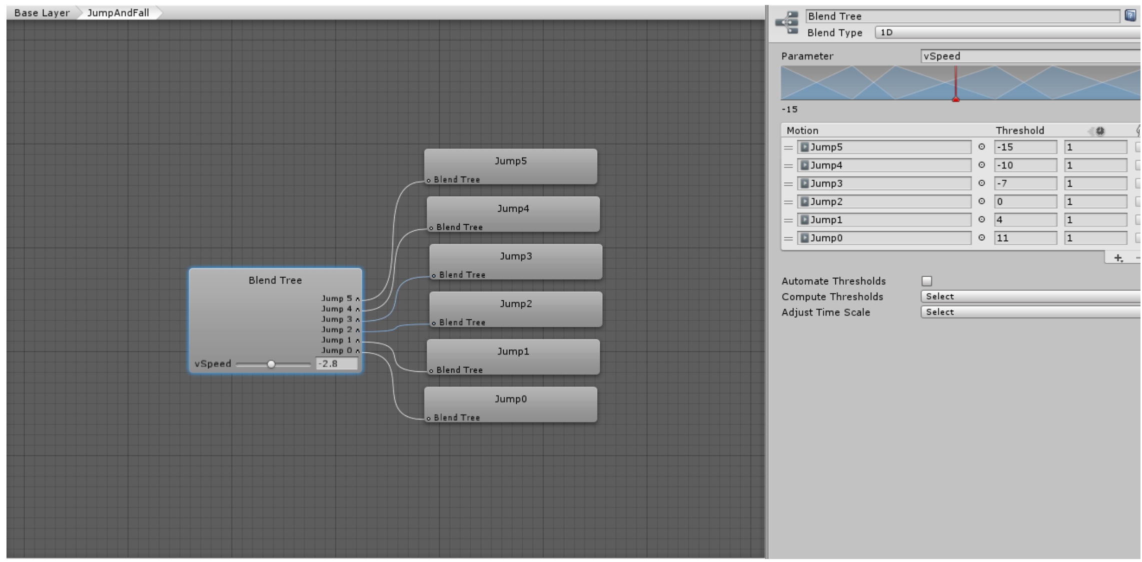


Ilustración 39. BlendTree de JumpAndFall

La peculiaridad de este elemento es que consigue mezclar varios movimientos similares mediante transiciones suaves. En este caso, depende del parámetro “vSpeed”, y hace transiciones entre los estados “Jump0”,”Jump1”,”Jump2”,”Jump3,”Jump4” y “Jump5” cuando llega a los límites establecidos.

La manera en que se actualizan todos estos parámetros de los que dependen las transiciones entre estados se realiza mediante *scripting*. Existen variables de distinto tipo, como *floats* o *bools* que se actualizarán con el bucle Update de MonoBehaviour. Son principalmente aquellas dependientes de la entrada del usuario: básicamente, los controles. A continuación se muestra un código de ejemplo.

```
public override void Update () {
    base.Update ();
    //Check input
    Vector3 WorldPos = Camera.main.ScreenToWorldPoint(new Vector3(Input.mousePosition.x,Input.mousePosition.y,10));

    if (gamepad) {
        jump = Input.GetButtonDown ("GamepadButtonA");
        crouch = Input.GetButtonDown ("GamepadButtonY");
        blinking = Input.GetButtonDown ("GamepadButtonB") && WorldPos.y > -5.6f;
        basicAttack = Input.GetButtonDown ("GamepadButtonX");
        throwAttack = Input.GetButtonDown ("GamepadButtonRight");
    } else if (keyboard) {
        jump = Input.GetKeyDown (KeyCode.Space);
        crouch = Input.GetKey (KeyCode.LeftControl);
        blinking = Input.GetKeyDown (KeyCode.F) && WorldPos.y > -5.6f;
        basicAttack = Input.GetMouseButtonDown (0);
        block = Input.GetMouseButton (1);
        throwAttack = Input.GetKeyDown (KeyCode.T);
    } else {
        if (leftPressed || rightPressed) {
            if (leftPressed) {
                move = -1;
            } else {
                move = 1;
            }
        } else {
            move = 0;
        }
    }
```

Ilustración 40. Bucle Update escuchando Input de usuario para cada plataforma

Estos valores se actualizarán en los respectivos Animators de la ilustración anterior con el bucle FixedUpdate, como muestra la ilustración siguiente.

```
public virtual void FixedUpdate(){

    characterAnimator.SetBool ("Push", push);
    characterAnimator.SetBool ("Ground", grounded);
    characterAnimator.SetBool ("Blinking", blinking);
    characterAnimator.SetFloat ("vSpeed", characterRigidBody.velocity.y);
    characterAnimator.SetFloat ("Speed", Mathf.Abs (move));
    characterAnimator.SetBool("Crouch",crouch);
    characterAnimator.SetBool ("BasicAttack", basicAttack);
    characterAnimator.SetBool ("Block", block);
    characterAnimator.SetBool ("Throw", throwAttack);
    //Check crouching
    if (crouch) {
        if(characterRigidBody.velocity.x!=0){
            characterRigidBody.AddForce(new Vector2(move*maxSpeed*-1,characterRigidBody.velocity.y));
        }
        else {
        }
    } else {
        characterRigidBody.velocity = (new Vector2(move*maxSpeed*2,characterRigidBody.velocity.y));
    }
    if (block) {
        characterRigidBody.velocity = new Vector2 (0, 0);
    }
```

Ilustración 41. Bucle FixedUpdate actualizando valores del Animator

Por otro lado, el héroe también cuenta con una serie de Colliders: uno para detectar si está en el suelo, otros dos para la caja o *hitbox* del personaje y por último otro en el arma.

Ilustración 42. Comportamiento de Colliders en el héroe

Todos ellos servirán para detectar colisiones con otros objetos, como pueden ser los enemigos u objetos del entorno, o para activar disparadores o *triggers* al entrar en cierta zona del escenario.

Para ciertas plataformas como PC o dispositivo móvil, se ha diseñado unos controles especiales para poder jugar: el dispositivo móvil cuenta con una interfaz adicional en pantalla simulando botones, mientras que el usuario de PC tiene la opción de jugar con un joystick. Para realizar la implementación de estos últimos controles, se recurre a la clase Input de Unity, al igual que con los controles clásicos de teclado y ratón. Se ha consultado en la documentación oficial del joystick los controles que corresponderían a cada tecla en Unity, como muestra la siguiente ilustración.

Ilustración 43. Controles de joystick que corresponden a Unity

Como se puede observar, esos valores son los que deben ir en la clase Input de Unity. A continuación se muestra un ejemplo para entradas de eje y botón.

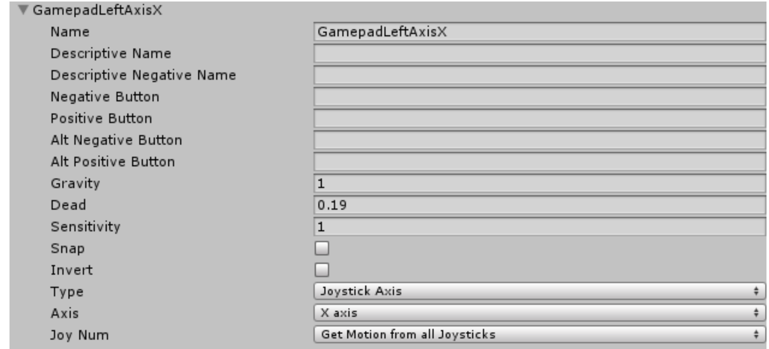


Ilustración 44. Input, entrada de eje X

GamepadButtonA	
Name	GamepadButtonA
Descriptive Name	
Descriptive Negative Name	
Negative Button	
Positive Button	joystick button 0
Alt Negative Button	
Alt Positive Button	
Gravity	1
Dead	0.001
Sensitivity	1
Snap	
Invert	
Type	Key or Mouse Button
Axis	X axis
Joy Num	Get Motion from all Joysticks

Ilustración 45. Input, entrada de botón X

Además de esto, también se ha utilizado el *plugin* XInputDotNet para añadirle efectos de *force feedback* al mando, al igual que se puede hacer en la mayoría de dispositivos con el método Handheld de la API de Unity.

```
XInputDotNetPure.GamePad gamepad = new GamePad ();
XInputDotNetPure.GamePad.SetVibration (PlayerIndex.One, 0.1f,0.5f);
```

Ilustración 46. Implementación de vibración con XInputDotNet

Con esto se conseguirá modificar la vibración de cada uno de los motores que lleva el joystick conectado al ordenador.

Respecto al modo multijugador, la implementación en el lado cliente se ha hecho mediante el *plugin* unity-socket-io y el *framework* Node.js junto a la librería Socket.io en la parte servidor.

Esta implementación trata de comunicar a los otros clientes conectados al servidor parámetros del héroe propio. Simplemente, se deberán lanzar los valores al servidor y este se encargará de comunicarlo al cliente respectivo, que lo actualizarán en las instancias que crearán en su aplicación de los demás clientes en los bucles Update y FixedUpdate.

A continuación, se muestra un ejemplo de aquellos valores que tienen relación con las físicas del héroe. Para empezar, se disparará al evento apropiado en el que se pasarán los valores del personaje local.

```
if (OnCommandPlayerPhysics != null) {
    OnCommandPlayerPhysics (move, grounded, velocity, scale,characterPosition);
}
```

Ilustración 47. Evento para multijugador, disparador

Este evento llamará al método que se encargará de encapsular toda esta información en un diccionario de clave/valor de tipo *string*, y lo emitirá por el socket hacia el servidor usando la instancia creada de Socket.io, que se encontrará escuchando eventos del tipo “PHYSICS” en este caso.

```
void OnCommandPlayerPhysics(float move,bool grounded,Vector2 velocity, Vector3 scale, Vector3 position){

    Dictionary<string,string> data = new Dictionary<string,string>();
    data ["move"] = move.ToString();
    data ["grounded"] = grounded.ToString();
    data ["velocity"] = velocity.x+","+velocity.y;
    data ["scale"] = scale.x + "," + scale.y + "," + scale.z;
    data ["position"] = position.x+","+position.y+","+position.z;

    socketIO.Emit ("PHYSICS", new JSONObject(data));
}
```

Ilustración 48. Método para multijugador, emisión al servidor

Cuando esta información llegue al servidor, este se encargará de transmitirlo a los demás haciendo un *broadcast*, y guardándose una copia en el mismo.

```
socket.on('PHYSICS', function(data) {
  currentUser.velocity = data.velocity;
  currentUser.move = data.move;
  currentUser.grounded = data.grounded;
  currentUser.scale = data.scale;
  currentUser.position = data.position;
  socket.broadcast.emit('PLAYER_PHYSICS', {
    id: socket.id,
    move: data.move,
    grounded: data.grounded,
    scale: data.scale,
    velocity: data.velocity,
    position: data.position
  });
  update(currentUser);
  console.log(Date()+'>>Event: PHYSICS');
});
```

Ilustración 49. Recepción del servidor y emisión a clientes

Una vez enviado, los clientes se encontrarán escuchando al evento con nombre “PLAYER_PHYSICS” en este caso, y actualizarán el GameObject que corresponda al del usuario que lo ha mandado. Esto se hará mediante un identificador.

```
void OnPlayerPhysics(SocketIOEvent obj){

    string id=JsonToString(obj.data.GetField("id").ToString(),"\"");
    bool grounded= JsonToBool(obj.data.GetField("grounded").ToString(),"\"");
    Vector3 scale = JsonToVector3(JsonToString(obj.data.GetField("scale").ToString(),"\""));
    float move = JsonToFloat(obj.data.GetField("move").ToString(),"\"");
    Vector2 velocity= JsonToVector2(JsonToString(obj.data.GetField("velocity").ToString(),"\""));

    OtherPlayerControls controls = ClientsControls [id];

    controls.move = move;
    controls.grounded = grounded;
    controls.velocity = velocity;
    controls.scale = scale;
}
```

Ilustración 50. Recepción de datos en el cliente y actualización de instancia de otro jugador

Respecto a la base de datos del servidor, cuando la partida acabe el jugador enviará la puntuación obtenida para que se guarda el récord. Se hará del mismo modo que lo anterior, como muestra la siguiente ilustración.

```
socket.on('FINALSCORE',function(data){
    var stmt = db.prepare("INSERT INTO ranking VALUES (?)");
    stmt.run(data.points);
    stmt.finalize();
    var scores;
    db.each("SELECT rowid AS id, info FROM ranking",function(err,row){
    scores+=row.info+",";
    });
    socket.emit('SCORES',scores);
});
```

Ilustración 51. Inserción en base de datos del servidor y posterior envío

5.2 Enemigo

Los enemigos tendrán una estructura diferente al héroe. Han sido realizados con un sistema de hueso que le proporciona un esqueleto, haciendo todas las animaciones mucho más fluidas. La siguiente ilustración muestra su estructura.

```
▼ Spine GameObject (Assassin)
  ▼ SkeletonUtility-Root
    ▼ root
      ▼ Character_midpoint
        ▼ torso
          ▼ arm_upper_far
            ▼ arm_lower_far
              ▼ hand_far
                ▼ item_far
                    item_part_far
          ▼ arm_upper_near
            ▼ arm_lower_near
              ▼ hand_near
                ▼ item_near
                    item_part_near
          ▼ neck
              head
          ▼ waist
            ▼ leg_upper_far
              ▼ leg_lower_far
                  foot_far
            ▼ leg_upper_near
              ▼ leg_lower_near
                  foot_near
```

Ilustración 52. Estructura del esqueleto del enemigo

La diferencia con las animaciones también es notable. En este caso, se activan por *scripting*, sin pasar por árbol de estados alguno.

```
private IEnumerator State(){
    for (int i = 0; i <= 5; i++) {
        m_Animation.state.SetAnimation (0, "idle", true);
        yield return new WaitForSeconds (2);
        m_Animation.state.SetAnimation (0, "reset", false);
        m_Animation.state.AddAnimation (0, "meleeSwing1", false, 0);
        m_Animation.state.AddAnimation (0, "meleeSwing1", false, 0);
        m_Animation.state.AddAnimation (0, "meleeSwing1", false, 0);
        m_Animation.state.AddAnimation (0, "meleeSwing2", false, 0);
        m_Animation.state.AddAnimation (0, "meleeSwing3", false, 0);
        m_Animation.state.AddAnimation (0, "idle", true, 0);
        yield return new WaitForSeconds (2);
    }
}
```

Ilustración 53. Ejemplo de Corrutina de animaciones del enemigo

En el caso del método "SetAnimation", encargado de comenzar la animación, el primer parámetro corresponde a la capa, el segundo a la animación deseada, el tercero si debe permanecer en bucle esta animación. Por otro lado, el método "AddAnimation", encargado de encolar otras animaciones, añade otro valor que corresponde con el tiempo de retraso de esa animación respecto a la anterior.

Los *colliders* son la parte que tienen en común tanto el héroe como los enemigos. Estos últimos también los usarán como método para detectar colisiones, en este caso, con el héroe.

Ilustración 54. Comportamiento de Colliders en ataque cuerpo a cuerpo del enemigo

Además del básico de los ataques cuerpo a cuerpo, también es posible encontrar alguno con ataque a distancia, que tendrá *colliders* extra en los proyectiles.

Ilustración 55. Comportamiento de Colliders en ataque a distancia del enemigo

Además de esto, los enemigos también contarán con un sistema de visión para detectar al héroe. Estará formado por unas líneas que, al colisionar con algún objeto de la capa "Player", harán que cambie su estado.

```
void RayCasting(){
    //Draw line on scene to check line of sight
    Debug.DrawLine (SighStart.position, SightEnd.position, Color.green);
    spotted = Physics2D.Linecast (SighStart.position, SightEnd.position, 1 << LayerMask.NameToLayer ("Player"));
}
```

Ilustración 56. Ejemplo de rango de visión del enemigo

5.3 Consumibles y *powerups*

Los elementos de esta categoría se encargan de dar puntos al jugador en el caso de los consumibles y de dotar de ciertos poderes especiales al héroe en el caso de los *powerups*.

Un ejemplo de implementación es el de los consumibles, que en el juego se encontrarán a lo largo del nivel en un estado flotante, y que a continuación se convertirá en puntos una vez lo recojamos. Su máquina de estados es bastante simple, como se muestra a continuación.

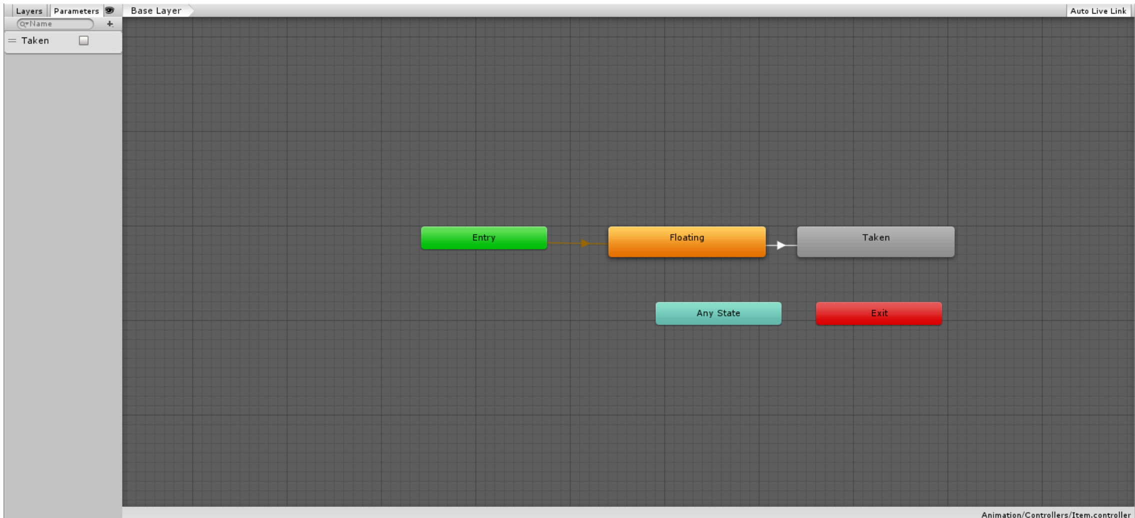


Ilustración 57. Máquina de estados de consumible

Por defecto, el objeto estará flotando, y solo cambiará si se pasa por encima del objeto. Para la detección de este tipo de evento, el objeto en sí tendrá asociado, como no podría ser de otro modo, un *collider*. En este caso, será de tipo disparador, por lo que no detectará ni se tendrán en cuenta colisiones, sino el *collider* que ha entrado en contacto con él

```
void OnTriggerEnter2D(Collider2D col){
    if (!taken) {
        if (col.GetType () == typeof(CircleCollider2D) && col.gameObject.tag == "Player") {
            taken = true;
            m_Animator.SetBool ("Taken", true);
        }
    }
}
```

Ilustración 58. Ejemplo de Collider en consumible

6. Pruebas

Para la comprobación del correcto funcionamiento de la aplicación, se han empleado tres dispositivos diferentes: un ordenador, un *smartphone* y una *tablet*.

Ordenador:

SO: Windows 7 Professional

Procesador: Inter Core i5-4690 3.5Ghz

GPU: Gigabyte GeForce GTX 970 Gaming G1 WindForce OC 4GB GDDR5

RAM: 8GB

Periféricos: Teclado y ratón, joystick

Dispositivo móvil: Samsung Galaxy S II (GT-i9100)

SO: Android 4.1.2

Procesador: ARM Cortex-A9 Dual Core 1.2GHz

GPU: ARM Mali 400 MP

RAM: 1GB

Tablet: Fnac Tablet 10”

SO: Android 4.1.2

Procesador: Cortex A9 Dual Core 1.6GHz

GPU: ARM Mali 400

RAM: 1GB

Primero se probó en el ordenador: se compiló el programa para la plataforma de PC y se construyó el ejecutable. Se comenzó probando el modo un jugador con teclado y ratón, siendo los resultados satisfactorios. A continuación, se insertó un joystick Microsoft Xbox360 Controller y se probó con los controles de mando. Lo siguiente fue probar el modo multijugador ejecutando el servidor en local, con dos clientes del juego ejecutándose en el mismo ordenador, y el resultado fue satisfactorio. Se inició más tarde en una máquina externa, en este caso un servidor OVH, y se volvió a probar la misma configuración de dos clientes en un mismo ordenador y el servidor remoto, con buenos resultados.

Por otro lado, se compiló el programa para Android, consiguiendo una *apk*. Esta se probó en los dos dispositivos en modo solitario, y el resultado fue satisfactorio. Pese a que no son dispositivos muy potentes, las imágenes por segundo del juego eran bastante altas y estables. A continuación, se probó a ejecutar el servidor en el ordenador, que se encontraba en la misma red

que los otros dos dispositivos, con un cliente en la *tablet*, otro cliente en el móvil y otro en el ordenador, con resultado satisfactorio. El mismo caso se probó ejecutando el servidor en la máquina OVH, obteniendo los mismos resultados. El servidor se llegó a probar con hasta 4 usuarios concurrentes, suficientes para una partida, sin problema alguno ni sobrecarga del mismo.

7. Conclusiones

Para acabar esta memoria, se exponen las dos caras de las conclusiones extraídas de la realización del mismo: en primer lugar, conclusiones técnicas y en segundo lugar, las personales. Ayudan a plantear las bases para el siguiente apartado, trabajos futuros.

7.1 Técnicas

Con este proyecto he aprendido a utilizar muchas de las características que aún no había usado de Unity. Pese a que llevo tiempo utilizando este motor, aún no había explorado mucho esta rama de conocimiento de los videojuegos, ya que la mayor parte de los proyectos que había realizado eran aplicaciones de escritorio. Sin duda alguna, el mercado laboral de hoy en día necesita de programadores de Unity con cierto nivel y estos conocimientos que he ido adquiriendo a lo largo del proyecto.

Por otro lado, también me ha servido como primer contacto con el mundo de los servidores en los juegos y sus posibilidades. El *framework* utilizado, en este caso Node.js, plantea un sinfín de posibilidades ya que está popularizándose por todo el mundo relacionado. Pese a haber hecho la especialización de redes en la carrera, no ha sido hasta este preciso instante la puesta en práctica de todos los conocimientos: un caso real. Me ha apasionado.

7.2 Personales

En el aspecto personal, este proyecto me ha dado una base para afrontar otros proyectos de igual o más envergadura en un futuro. Me ha ayudado mucho en el mundo laboral, ya que mi empleo está muy relacionado con el trabajo realizado; además, ha completado mi perfil profesional tanto con conocimientos de videojuegos como de redes.

También he sufrido el esfuerzo que supone realizar un proyecto y documentarlo, que en ocasiones parece que no tiene final. Por suerte, me ha encantado realizar este tipo de proyecto por sus características, y esto no hace más que aumentar mis ganas por seguir trabajando y formándome en cosas relacionadas con el mundo de los videojuegos y redes. Nunca lo había visto tan claro.

8. Trabajos futuros

El proyecto consta de los pilares fundamentales de un videojuego, pero debido a las limitaciones de tiempo, se han quedado en el tintero mejoras que podrían resultar interesantes. En su gran mayoría giran en torno al modo multijugador del videojuego.

La primera de ellas es un sistema de chat usando las tecnologías usadas en este proyecto Node.js y Socket.io. En cualquier videojuego en la que se coopere por red es básico esta característica para una mejor comunicación. Una alternativa sería un chat de voz. También se debe gestionar mejor el tema de las soluciones a los problemas que puedan surgir relacionados con la conexión del usuario.

El diseño de la aplicación es mejorable, ya que algunas de las animaciones de personajes están realizados con pocos *frames* y otros elementos del entorno están hechos a baja resolución. Debido al interés académico del proyecto no se ha considerado muy relevante centrarse en este aspecto y se ha preferido mostrar más interés por la parte de la implementación a nivel de programación.

Siguiendo con esto, aunque se deje claro al principio de la memoria, este videojuego pretende ser una demo para mostrar el gran poder que posee el motor, además de lo modular que puede llegar a ser. La idea es seguir desarrollando el videojuego, pero al mismo tiempo mostrar en el estado actual del proyecto todas sus posibilidades y potencial.

Por otro lado, la idea del proyecto comenzó con la idea de monetizar en un futuro la aplicación, junto con los colaboradores de ésta. Debido a las limitaciones económicas y de herramientas como algunos requisitos que ponen en las tiendas de las principales plataformas objetivo, se decidió esperar más adelante para adentrarse más en este tema.

El proyecto, pese a cumplir con la mayoría de requisitos que se habían definido al comienzo, es fácilmente escalable y no daría muchos problemas a la hora de desarrollar de realizar cambios por la naturaleza del motor y sus *plugins*.

9. Bibliografía

- Consumo de juegos en EEUU <http://www.gamerfocus.co/juegos/la-esa-comparte-datos-recientes-sobre-el-consumo-de-videojuegos-en-estados-unidos/>

- Why can't cross platform multiplayer games exist? <http://gamedev.stackexchange.com/questions/101040/why-cant-cross-platform-multiplayer-games-exist>

- Unity, Source2, UnrealEngine4 or CryEngine – Which game engine should i choose? <http://blog.digitaltutors.com/unity-udk-cryengine-game-engine-choose/>

- Unity server comparison <https://docs.google.com/spreadsheets/d/1xQMJPEi-SVUEHmdlv2FZwBEMyWLB5ucaAeDl1NPaQn4/edit?hl=en_US#gid=483412130>

- Documentación oficial de Unity <http://docs.unity3d.com/es/current/Manual/index.html>

- System properties comparison MongoDB vs. SQLite <http://db-engines.com/en/system/MongoDB%3BSQLite>

- Página oficial de Node.js <https://nodejs.org/en/>

- Página oficial de Socket.io <http://socket.io/>

- Repositorio de *plugin* para Socket.io en Unity <https://github.com/fpanettieri/unity-socket.io-DEPRECATED>

- Librería SQLite para Node.js <https://www.npmjs.com/package/sqlite3>

- Repositorio de XInputDotNet < https://github.com/speps/XInputDotNet>

Anexos

Anexo 1

Documento de diseño

The last hero

por Jesús Martínez Abril

ÍNDICE

1. Información general 74
   1.1 Concepto general 74
   1.2 Aliciente 74
   1.3 Sinopsis 74
   1.4 Título 74
   1.5 Año 74
   1.6 Género 74
   1.7 Plataforma 74
   1.8 Público objetivo 74
   1.9 Estilo visual 74
   1.10 Motor y editor 74
   1.11 Hardware necesario 74
2. Características del juego 75
   2.1 Jugabilidad 75
      2.1.1 Toma de decisiones y estrategia 75
      2.1.2 Gestión de recursos 75
      2.1.3 Rol del jugador 75
      2.1.4 Progresión 75
      2.1.5 Rejugabilidad 75
   2.2 Mecánica del juego 75
      2.2.1 Cámara 75
      2.2.2 Controles 75
      2.2.3 Puntuación 75
   2.3 Inteligencia artificial 76
      2.3.1 Comportamiento de los oponentes 76
      2.3.2 Calidad de la planificación de movimiento 76
   2.4 Niveles 76
3. Diseño conceptual del juego 77
   3.1 Diseño de los personajes 77
      3.1.1 Héroe 77
      3.1.2 Enemigos 79
      3.1.3 NPC's 80

3.2 Diseño de los objetos 80

3.2.1 Armas de los enemigos 80

3.2.2 *PowerUps* 81

3.2.3 Consumibles 83

3.2.4 Objetos del entorno 83

3.3 Diseño del mapa 84

3.4 Diseño de la interfaz 84

3.4.1 Menú de inicio 84

3.4.2 Juego 85

3.4.3 Diálogo 85

3.5 Recursos 85

3.5.1 Audios 85

3.5.2 Imágenes 85

1. Información general

1.1 Concepto general

Se basa en progresar durante el escenario a través de las líneas enemigas, consiguiendo puntos realizando ciertas acciones y eliminando otros personajes.

1.2 Aliciente

El jugador competirá por realizar la mayoría de puntos posibles con los recursos disponibles a su alcance para ser el mejor de la tabla de clasificaciones. Para ello, tendrá que progresar por el escenario de la manera más eficiente posible.

1.3 Sinopsis

Nos encontramos en el futuro. La civilización ha alcanzado un grado de tecnología importante, y poco a poco están agotando los recursos del planeta. Una misma organización criminal controla el mundo: nos encontramos con un nuevo orden mundial. La organización no tiene piedad alguna con los asentamientos que aún son libres, por lo que mandan sus soldados para arrasar con todo.

Pese a esto, aún hay esperanza. Los pueblos libres se unen para derrocar a los opresores con la ayuda de los guerreros de una pequeña aldea situada en un remoto rincón del planeta.

1.4 Título

Videojuego titulado "The last hero".

1.5 Año

Realizado en el año 2016.

1.6 Género

Plataformas.

1.7 Plataforma

Windows, iOS, Android

1.8 Público objetivo

Casual y competitivo. Además, se debe poder jugar en periodos de tiempo relativamente bajos.

1.9 Estilo visual

El juego tendrá un estilo 2D, con ciertas técnicas de animado novedosas.

1.10 Motor y editor

Se empleará Unity como motor y MonoDevelop como IDE.

1.11 Hardware necesario

PC/Mac o dispositivo móvil Android o iOS.

2. Características del juego

2.1 Jugabilidad

2.1.1 Toma de decisiones y estrategia

El jugador controlará al héroe, con el que tratará de sortear los diferentes obstáculos para intentar ayudar a derrotar a los forajidos. Las decisiones y la estrategia las decidirá el propio jugador: podrá optar por ir de forma sigilosa o agresiva. Esto tendrá repercusión:

- El estilo sigiloso permitirá enfrentarse a menos enemigos, pero a su vez, se obtendrán menos puntos y se tardará una cantidad de tiempo diferente en acabar.
- Del otro modo, el estilo agresivo tenderá a ganar más puntos al eliminar más enemigos, aunque también se deberá tener más cuidado con la vida y el tiempo que se emplee para acabar la partida.

2.1.2 Gestión de recursos

El héroe posee ciertas habilidades que puede usar en ciertas situaciones. Cada una de ellas tendrá un tiempo de reutilización concreto, y algunas solo estarán disponibles cuando se recojan cierto tipo de *power-up* del nivel.

Además, el jugador tendrá que vigilar la vida disponible. Cuando esta llegue a cero, el jugador morirá y deberá empezar desde el último *checkpoint*.

2.1.3 Rol del jugador

El jugador podrá influir en el transcurso de la historia. Dependiendo de las elecciones en ciertos diálogos del juego, el final será uno u otro.

2.1.4 Progresión

El jugador recorrerá el nivel, donde encontrará los denominados *checkpoints*. Estos se usan a modo de seguridad, de manera que si en algún momento el jugador muere, aparecerá aquí de nuevo. Obviamente, esto penalizará en la puntuación y tendrá cierto límite, el cual si se rebasa, no dará opción a volver a aparecer.

2.1.5 Rejugabilidad

Cuando el jugador decida jugar de nuevo, se empezará desde el principio, sin tener en cuenta *checkpoints* de anteriores partidas. Estos solo estarán disponibles en la misma sesión de juego, y estarán limitados a unos cuantos usos.

2.2 Mecánica del juego

2.2.1 Cámara

Se utilizará un tipo de cámara con proyección ortográfica que seguirá en todo momento al personaje en un universo 2D. Podrá variar en ciertas situaciones.

2.2.2 Controles

Los controles del juego se ubican en los periféricos que son habitualmente utilizados en equipos de sobremesa: teclado y ratón o *gamepad*. En el caso de los dispositivos móviles, una interfaz extra que se situará por encima del juego emulará el trabajo de éstos.

2.2.3 Puntuación

El sistema de puntuación se basará en un cálculo en el que intervendrá el tiempo que se haya tardado en acabar el nivel, los objetos de bonificación que hayamos recogido y los enemigos derrotados. A partir de estos datos, la información se guardará de manera local y, si hemos elegido modo multijugador, en la base de datos del servidor.

2.3 Inteligencia artificial

2.3.1 Comportamiento de los oponentes

Los enemigos poseen un rango de visión. Dependiendo de éste, pasarán por un estado u otro en función de la posición del héroe o la acción que se realice. A grandes rasgos, podemos describir cuatro grandes estados:

- Estático. El enemigo se encontrará parado, sin prestar atención a nada.
- Patrullar. Como su propio nombre indica, el enemigo estará haciendo guardia de un punto a otro.
- Alerta. El enemigo ha detectado algún movimiento extraño en su radio de acción y se encuentra en tensión, hasta que pase el siguiente evento.
- Hostil. El enemigo se encuentra en disposición de atacar al jugador mientras este se encuentre en su radio de acción.
- Derrotado. El enemigo es eliminado debido al ataque del héroe.

2.3.2 Calidad de la planificación de movimiento

Como se ha descrito en el apartado anterior, el enemigo dispondrá de varios tipos de rango:

- Rango de visión. Será sobre el eje horizontal, y funcionará de manera que si el héroe entra en él, el enemigo pasará a modo hostil de manera directa sin pasar por otro estado más.
- Radio. El enemigo podrá detectar si cerca de él está pasando algo, por lo que si el jugador entra en el mismo, se pondrá en modo alerta. Del mismo modo, podrá volver al estado anterior si salimos del radio de efecto.

2.4 Niveles

El juego se compone de un único nivel con dificultad ajustable.

3. Diseño conceptual del juego

3.1 Diseño de los personajes

3.1.1 Héroe

El héroe es el principal personaje del juego, controlado por el jugador. Solo habrá posibilidad de elegir un tipo de personaje. A continuación, se muestran las iteraciones hechas por el diseñador hasta obtener el resultado final.

Primera iteración

Ilustración 59. Diseño inicial del héroe

Segunda iteración

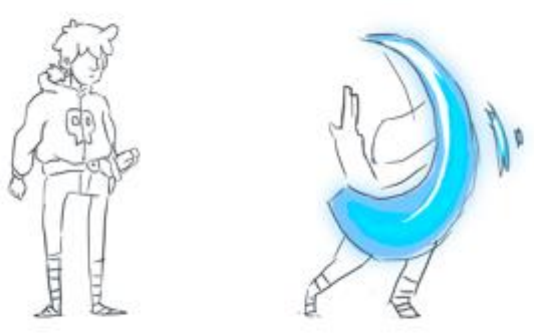


Ilustración 60. Segundo diseño del héroe

Tercera iteración

Ilustración 61. Diseño final del héroe

Evolución

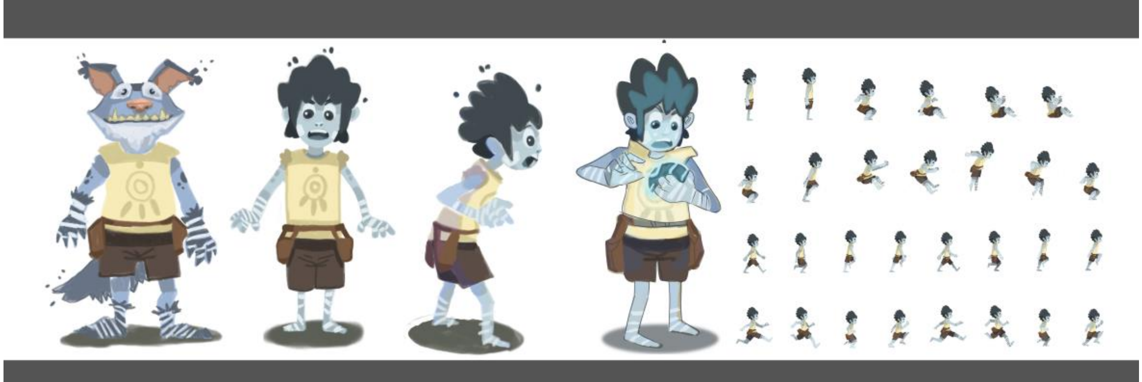


Ilustración 62. Montaje de la evolución del héroe

3.1.2 Enemigos

Los enemigos son variados en aspecto y atributos. Son los principales rivales del héroe y están distribuidos a lo largo del nivel. Su objetivo es tratar de eliminar al héroe. Algunos de ellos también hacen de jefes de nivel, y son algo más fuertes e incluso grandes que los enemigos normales. Los atributos de cada uno son configurables por dificultad.

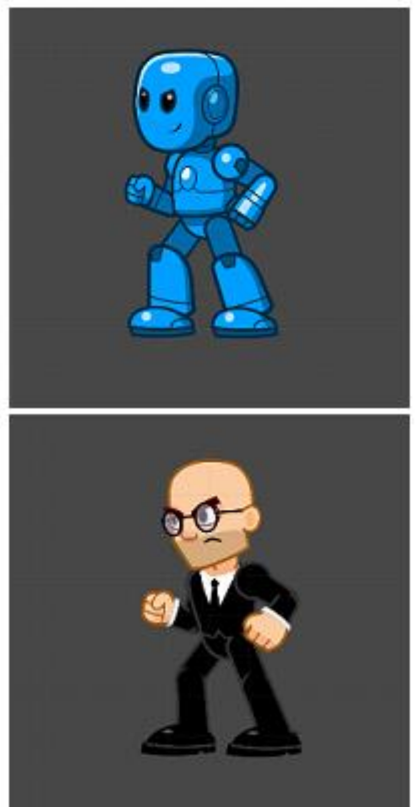


Ilustración 63. Diseño de enemigos

3.1.3 NPC's

Se tratan de personajes neutrales que ofrecerán objetos y algún que otro diálogo. El héroe podrá interactuar con éstos.

Ilustración 64. Diseño de NPC's

3.2 Diseño de los objetos

3.2.1 Armas de los enemigos

Los enemigos tendrán la capacidad de aparecer con distintas armas durante el juego, principalmente un arma cuerpo a cuerpo y un arma de largo alcance.

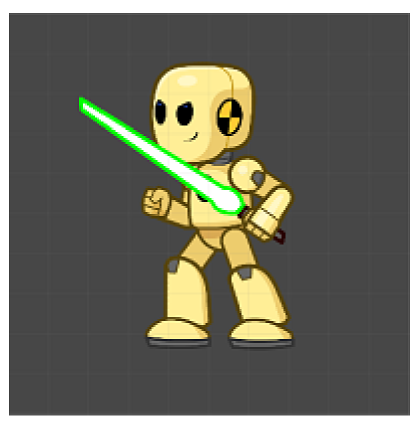


Ilustración 65. Diseño de las posibles armas de los enemigos

3.2.2 *PowerUps*

Son objetos con los que el héroe obtendrá diferentes poderes. Hay de tres tipos:

- Invulnerable: el héroe no sufrirá daño alguno durante la duración. Le rodeará un aura característica.

Ilustración 66. Diseño del powerup "Invulnerable"

Ilustración 67. Resultado de recoger el powerup "Invulnerable"

- *SpeedUp*!: modifica en gran medida la agilidad del héroe, lo que hace que aumente tanto su velocidad como su potencia de salto durante la duración.

Ilustración 68. Diseño del powerup "SpeedUp!"

- *Brute*: el héroe adquiere una fuerza sobrehumana, que se traduce en un aumento de fuerza y tamaño considerable durante la duración.

Ilustración 69. Diseño del powerup "Brute"

3.2.3 Consumibles

Se trata de objetos que proporcionan puntos, o bien proporcionan puntos de salud. La cantidad puede variar de uno a otro.

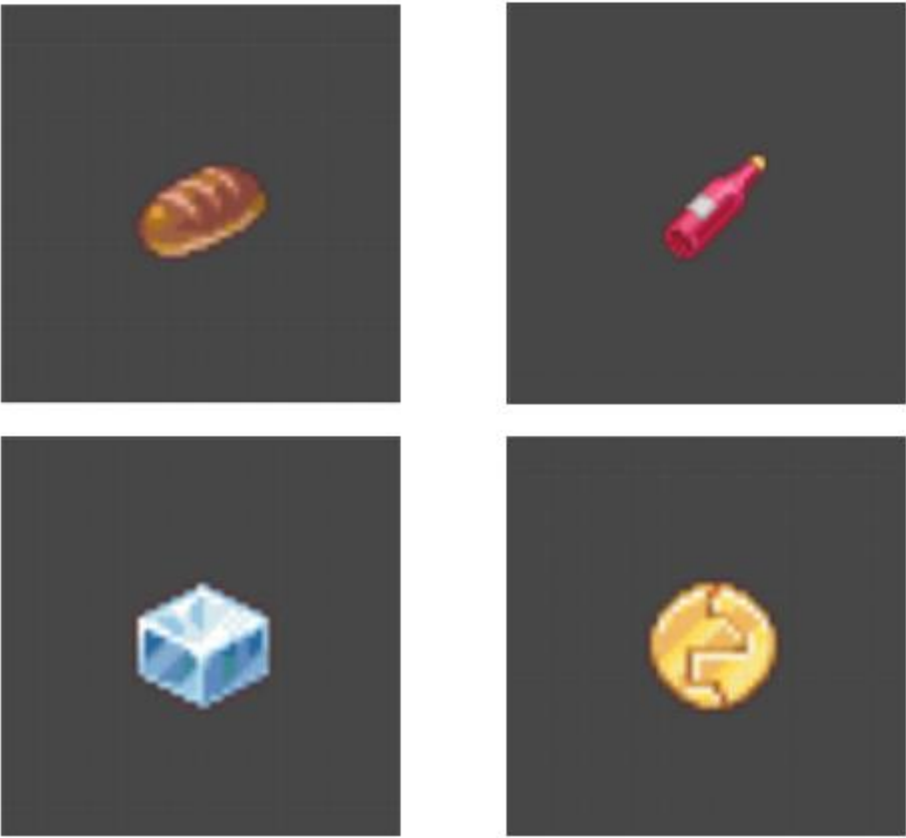


Ilustración 70. Diseño de consumibles

3.2.4 Objetos del entorno

En el entorno podemos encontrar objetos con los que el héroe puede interactuar mediante físicas, y será necesario para pasar ciertas zonas o sortear estos mismos objetos, que impedir el avance.

Ilustración 71. Diseño de objetos

3.3 Diseño del mapa

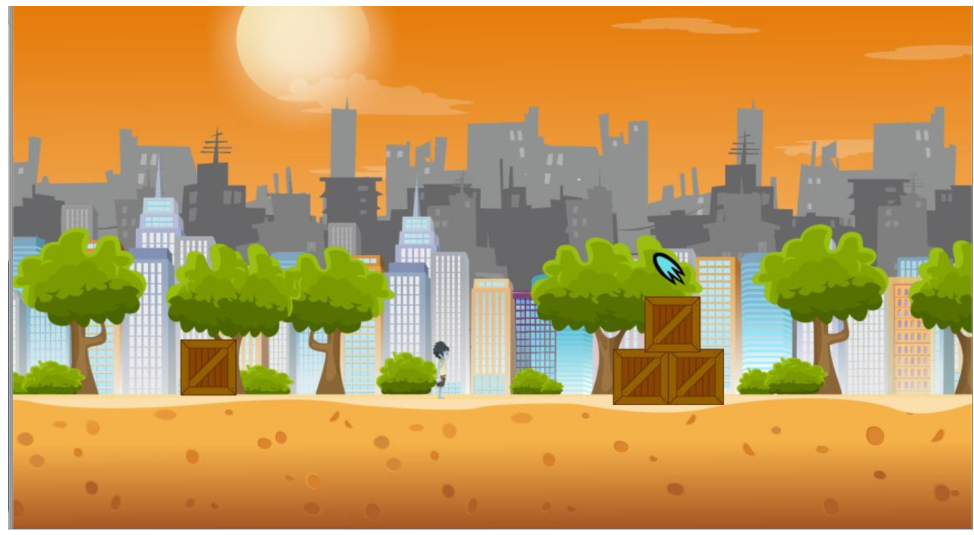


Ilustración 72. Primer diseño del mapa

3.4 Diseño de la interfaz

3.4.1 Menú de inicio

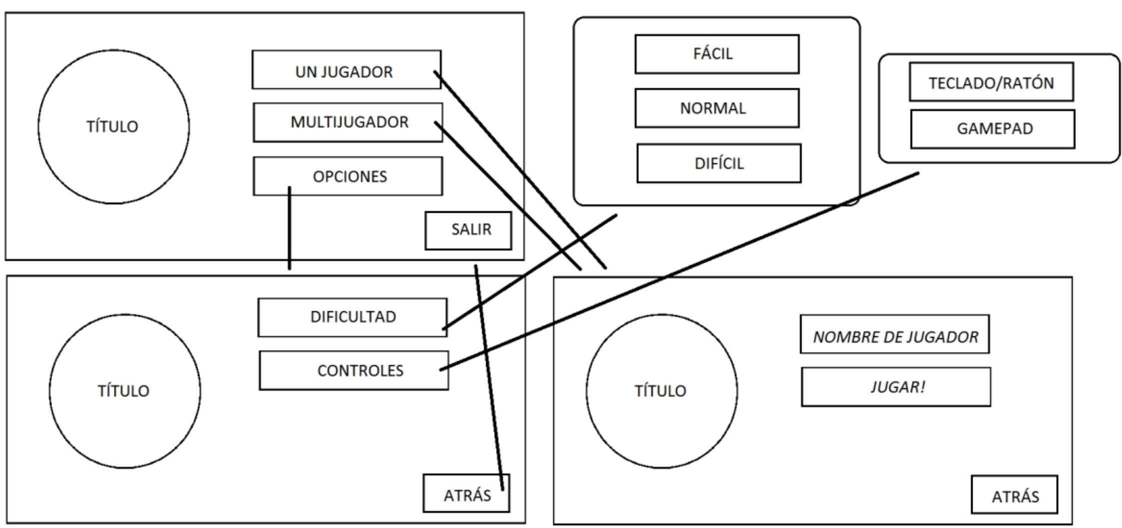


Ilustración 73. Boceto de la interfaz

3.4.2 Juego

Ilustración 74. Primer diseño de interfaz de usuario

3.4.3 Diálogo

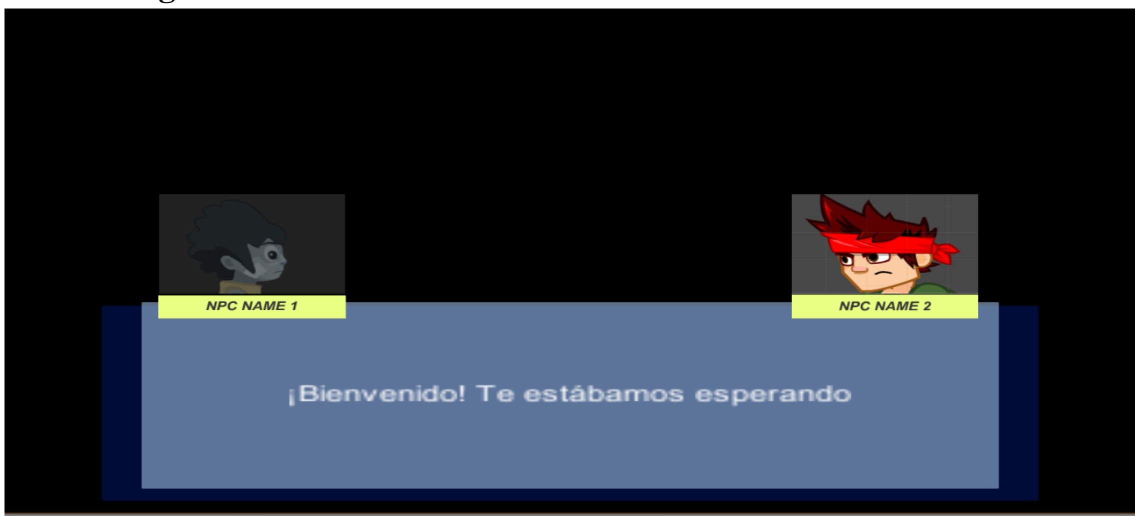


Ilustración 75. Primer diseño de diálogos

3.5 Recursos

3.5.1 Audios

- Menú: http://opengameart.org/
- Efectos de sonido: http://soundbible.com/

3.5.2 Imágenes

- Objetos y consumibles: http://opengameart.org/
- Enemigos: https://www.assetstore.unity3d.com/en/#!/content/41338
- Entorno: https://mobilegamegraphics.com/product/free-parallax-backgrounds/
- UI: http://www.spriters-resource.com/wii/mkwii/sheet/30506/
- Héroe hecho a mano. Autor: Pablo Broseta