
***Facultad***
***de***
***Ciencias***

# Desarrollo de un videojuego para enseñar programación a niños
## (Development of a videogame to teach programming to children)

**Trabajo de Fin de Grado**
**para acceder al**

**GRADO EN INGENIERÍA INFORMÁTICA**

**Autor: Pablo García de los Salmones Gómez**

**Director: Carlos Blanco Bueno**

**Co-Director: Juan Antonio Pila Fernández**

**Junio – 2017**

# 1 ÍNDICE DE CONTENIDO

# ÍNDICE DE ILUSTRACIONES

# ÍNDICE DE TABLAS

# ÍNDICE DE CÓDIGO

# AGRADECIMIENTOS

Llegado a este punto del camino, tengo que dar las gracias a todas las personas de mi vida que caminaron junto a mí.

Gracias a mi padre y a mi madre, porque sin su apoyo y su esfuerzo no tendría lo que tengo ni sería lo que soy. Ellos trajeron el primer ordenador a casa, compraron mi primera consola, se han encargado de que no me faltase de nada desde que he tenido necesidad de algo. Si he llegado hasta aquí es gracias a ellos.

Gracias también a mi hermana, otra parte esencial de mi familia y que siempre me ha ayudado a su manera; y a Fidel, que ha sido como un segundo padre, y también tiene mucha culpa de que hoy sea quien soy, junto a su interminable suministro de cine y teatro.

Tengo que acordarme, también, de aquellos que empezaron el camino conmigo, pero hoy faltan. Sé que mis abuelas y abuelos estarían orgullosos de ver hasta dónde he llegado.

Gracias a Carlos y a Juan, por ser mis tutores en este proyecto y transmitirme sus conocimientos y su entusiasmo, y al resto de mis profesores en la carrera por enseñarme todo lo que he aprendido durante estos cuatro años.

Gracias también a todos mis amigos. Gracias a Álvaro, Fernando, Íñigo, Javier, Joaquín, y Mario, por estar siempre conmigo, por ser una segunda familia, y hacer más divertidos los días al salir de clase; y gracias a Santos, Kat, Manuel, y el resto de mi comitiva en la facultad, por vuestra ayuda, por lo que he aprendido de vosotros, y por hacer más divertidos los días en clase.

Por último, muchas gracias de nuevo a mi padre y mi madre, porque son realmente la razón principal de que pueda estar hoy escribiendo esto, y todas las veces que se lo agradeciera serían pocas.

# RESUMEN

Junto con el avance de la tecnología y el desarrollo de la mente humana, vienen nuevas habilidades que permiten que evolucionemos hasta límites insospechados desde un punto de vista científico. La educación es una parte indispensable del desarrollo de nuestras vidas y debe tener en cuenta esa evolución. Una de las razones fundamentales para apostar por la programación a nivel formativo es la oportunidad que otorgamos a los alumnos de no ser meros espectadores, de tener la posibilidad de crear contenidos desde edades tempranas que transformen el mundo que los rodea.

El proyecto tiene como deseo y principal objetivo, no sólo el otorgar la oportunidad de adquirir conocimientos valiosos para los niños en un futuro a corto plazo, sino hacerles aprender valores como la organización o el trabajo metódico, tan indispensables en la programación. Todo ello mediante un juego continuo que les permita aprender con, y sentirse atraídos por, el desafío de pequeñas pruebas que aprovechen la tecnología de todas nuestras variadas plataformas y dispositivos. La estética será desenfadada y atractiva de manera que atraiga a nuestros hijos sin importar la edad de los mismos.

Para el desarrollo del proyecto se utilizará el motor Unity. Su gran capacidad de desarrollo multiplataforma permitirá adaptar el juego a un número mayor de sistemas, abarcando así todos aquellos que usan nuestros niños como pueden ser tabletas, ordenadores o consolas portátiles.

**Palabras Clave**: Juego serio, Unity, aprendizaje, programación, pensamiento computacional

# ABSTRACT

Along with the advance in technology and the development of the human mind, new abilities come that enable us to evolve to unsuspected limits, from a scientific point of view. Education is an essential part in the development of our lives and must keep that evolution in mind. One of the main reasons for betting on programming, at an educational level, is the opportunity we give to students of being more than mere spectators, having the chance of creating, from early on, content that could transform the world that surrounds them.

This project's desire and main objective is, not only giving children the chance of acquiring valuable knowledge for the short-term future, but helping them taking in skills such as organization or methodical work, so essential for programming. All that, through a continuous game that allows them for learning with, and feeling attracted by, the defiance of tiny challenges that make the most of all our varied platforms and devices. The appearance will be casual and appealing so it will attract children, no matter their age.

Unity engine will be used for the development of the project. Its great multiplatform development ability will allow for adapting the game to a greater number of systems, covering all those that children use, such as tablets, computers, or handheld consoles.

**Keywords**: Serious game, Unity, learning, programming, computational thinking

# 1 INTRODUCCIÓN

Desde que se originó la industria del videojuego, éste se ha entendido principalmente como un entretenimiento electrónico, un producto creado para el ocio del usuario. Pero paralela a esta corriente principal, siempre ha existido una rama de estudio que utilizó los videojuegos (y cualquier tipo de juego, en general) con un propósito distinto: el de la educación. Este tipo de juegos es conocido como *serious-games* (o juegos serios).

A medida que avanza el sector del videojuego, va incorporando los últimos avances tecnológicos a su metodología de trabajo. De la misma manera, en el campo de la educación también se intentan adoptar las nuevas tecnologías paulatinamente, tanto como herramientas educativas, como a modo de contenido, para que los alumnos se familiaricen con ellas, de forma que parece un paso lógico introducir a los estudiantes más jóvenes en el campo de la programación mediante juegos.

## 1.1 ESTADO DEL ARTE

Clark C. Abt publicó ya en 1970 un libro en el que utilizaba ese concepto para explicar cómo podían utilizarse juegos (en este caso, en el sentido más analógico del término) para mejorar la educación que recibían los alumnos en las aulas (Abt, 1970) (Djaouti, Alvarez, & Jessel, 2011).

Ya en 2002, con la industria del videojuego ya en auge, Ben Sawyer dio una definición más actualizada y precisa del término, que presentaba como el unir un propósito serio con una serie de tecnologías y conocimientos del sector del videojuego (Sawyer & Rejeski, 2002).

Desde que Abt explicase cómo potenciar el aprendizaje de los alumnos con juegos de mesa hasta hoy, son numerosas las ocasiones en que se han utilizado los conocimientos detrás del desarrollo de videojuegos y la teoría de juegos para crear productos que propiciaran la adquisición de ciertas aptitudes entre sus usuarios.

Desde juegos para concienciar a los ciudadanos europeos acerca de diferentes organismos de la UE, o que divulgasen la práctica de deportes regionales como el bolo palma, hasta sistemas que optimizan horarios de patrullas en la lucha contra el terrorismo (Nguyen, 2016), la lista de aplicaciones es inabarcable.

A propósito del inmenso volumen de tipos de juego que nos podemos encontrar categorizados como *serious-game*, Djaouti, Alvarez, y Jessel escribieron en 2011 “Classifying Serious Games: the G/P/S model”, un paper en el que establecieron una clasificación de juegos en base a su *gameplay* (“G”; la forma en que se juega, en un sentido amplio del término), su intención (“P”, por *purpose*), y su objetivo o alcance (“S”, por *scope*) (Djaouti, Alvarez, & Jessel, 2011). Esta clasificación, que podemos ver en la Ilustración 1, nos da una idea de los muchos propósitos a los que puede servir un juego de este tipo.

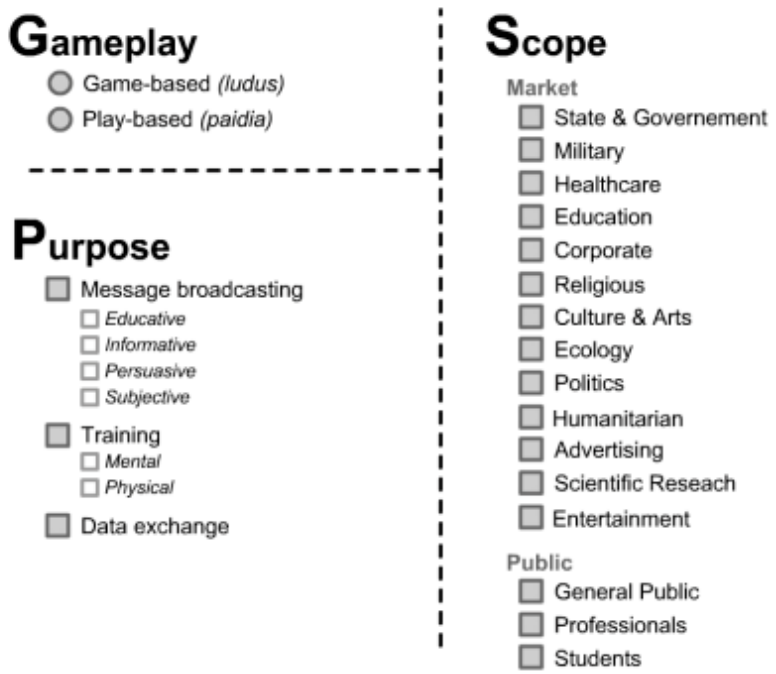


*Ilustración 1 - Clasificación G/P/S para serious games*

## 1.2 MOTIVACIÓN

La motivación detrás de este proyecto es crear un *serious-game* que ponga la tecnología actual de desarrollo de videojuegos al servicio del aprendizaje de la programación, una disciplina que hasta hace poco no estaba muy presente en las aulas de los colegios e institutos, pero que va cobrando protagonismo a medida que nos adentramos en una era eminentemente informática. La capacidad de programar es una habilidad extremadamente útil, no sólo en lo concerniente a comunicarse con un ordenador, sino porque el llamado razonamiento computacional otorga beneficios más allá de la programación, como nuevos enfoques a la hora de resolver problemas y plantear estrategias para cualquier ámbito de la vida (Wing, 2006).

No se puede decir que este proyecto sea pionero en enseñar programación con *serious-games*. En esta línea de trabajo hay ya varias iniciativas, siendo quizá la más conocida de todas Scratch, proyecto del Grupo Lifelong Kindergarten del MIT Media Lab (MIT Scratch Team, s.f.).

Sin embargo, aunque es innegable la repercusión que ha tenido y el avance que ha propiciado, se puede decir de Scratch que puede ser difícil de adoptar por usuarios que no tengan ninguna base de programación, pues peca de ser demasiado concreto en sus módulos, y tiende a dar muchas posibilidades para que el usuario las explore.

En su lugar, la iniciativa que más ha servido de inspiración durante la concepción del trabajo ha sido Code.org, proyecto en el que han colaborado multitud de empresas y celebridades, como Steve Ballmer y su fundación, Bill Gates, Mark Zuckerberg, Google, Disney, Rovio… (CODE.org, 2017).

Code.org está pensada como una plataforma para que los jóvenes practiquen desde sus hogares tanto como para que profesores puedan incorporar sus dinámicas a las aulas. Hay proyectos como "*Hour of code*", donde se pretende ir consiguiendo avances en sesiones de una hora, y

también existen diferentes cursos repartidos por niveles de dificultad y grupos de edad. Lo más atractivo que encuentro, a título personal, en esta iniciativa es que la línea de progreso está mucho mejor marcada que con Scratch, y es algo que me gustaría incorporar a mi proyecto.

## 1.3 OBJETIVO

Mi objetivo con este trabajo es ser capaz de desarrollar de principio a fin un sistema funcional, un juego serio, que sea entretenido y pueda utilizarse para introducir a los más jóvenes a un tipo de razonamiento más computacional, de una forma atractiva, pedagógica y divertida.

# 2 HERRAMIENTAS, TECNOLOGÍAS, Y MATERIALES UTILIZADOS

En este capítulo repasaré el conjunto de herramientas, tecnologías y materiales que he utilizado durante la realización del proyecto.

Parte de la explicación detalla elementos que podrían ser necesarios para entender ciertas profundizaciones en el apartado de Diseño e implementación.

## 2.1 UNITY

Unity es un motor de videojuegos (es decir, un *framework* diseñado específicamente para la creación y desarrollo de videojuegos) propiedad de Unity Technologies, que lo desarrolla desde 2005.

Es extremadamente flexible: permite crear para hasta 27 plataformas distintas (desde Windows hasta Android TV, pasando por iOS, Linux, PlayStation4…) y soporta multitud de formatos de imagen, audio, texto, animaciones, modelos 3D…, además de modos de desarrollo tanto en 2D como en 3D (Unity Technologies, 2017). En la Ilustración 2 se puede ver la interfaz de usuario de Unity, con su *layout* predeterminado, en el modo de desarrollo para 2D.

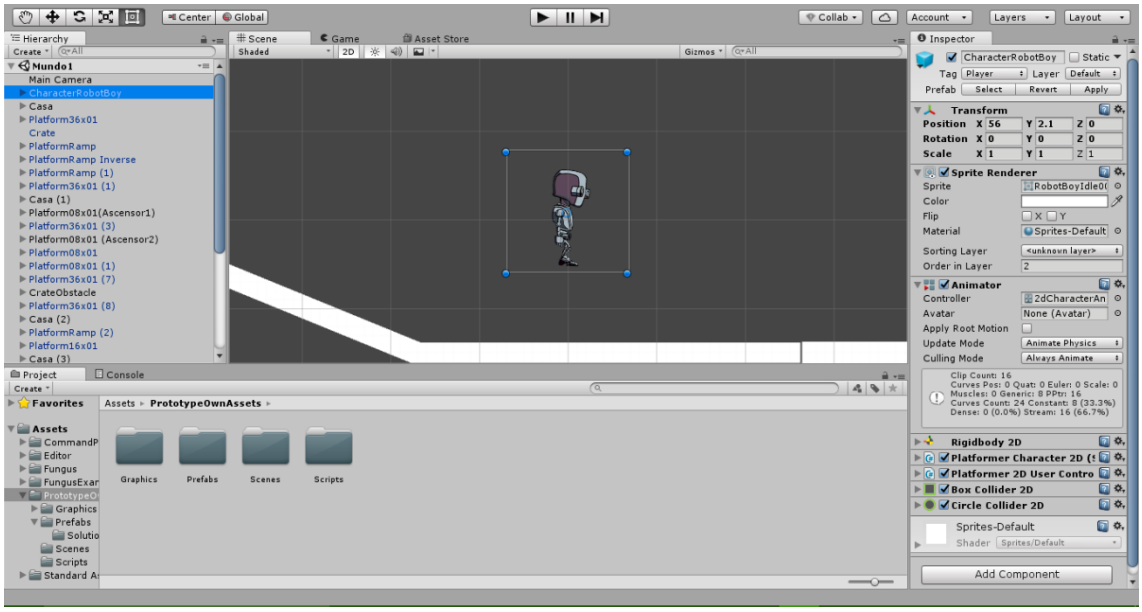


*Ilustración 2 - Interfaz de Unity, layout por defecto*

Hay varias licencias del motor disponibles, yendo desde la gratuita a las de pago mensual o personalizadas para empresas, que varían en servicios extra que Unity Technologies presta a sus usuarios y en diferentes límites impuestos como el máximo de ingresos mensuales de la entidad o el número de posibles jugadores simultáneos que se soportarían en un supuesto juego online.

Una de las mayores ventajas de la plataforma Unity es que su documentación es muy completa y accesible, y además cuenta con una inmensa base de usuarios, por lo que sus foros de ayuda y resolución de dudas tienen mucha actividad (haciéndolos, por tanto, potencialmente más útiles).

### 2.1.1 Las tripas de Unity

Con este apartado, mi intención es explicar brevemente el cómo es Unity en términos de funcionamiento; qué tiene y cómo se usa. Esto puede ser útil para comprender algunas de las explicaciones que daré más adelante, en el apartado de Diseño e implementación.

La base de todo el funcionamiento de Unity es el *GameObject*. Es la clase básica que representa a cualquier entidad en el motor. Por sí solos no cumplen una gran función, sino que su potencial reside en los llamados Componentes que pueden contener. Un *gameobject* siempre contiene un componente *Transform*, que indica su posición, rotación y escala absolutos en el entorno del juego; y se le pueden añadir un número de componentes distintos que lo dotan de su funcionalidad real (Unity Technologies, 2017).

Componentes son todos los "módulos" que se pueden añadir a un *gameobject* para proporcionarle nuevas capacidades. Hay multitud de componentes distintos, y en su uso y combinación reside el secreto para crear todos los diferentes elementos que puede contener un juego. Los scripts C# o JavaScript que definen el comportamiento de los objetos también se añaden a esos objetos como nuevos componentes. Existen componentes Luz para crear luces, Cámara para crear cámaras, *Collider* para crear cajas de colisión, Audio para producir o escuchar sonidos… Todos los componentes tienen una serie de propiedades, que dependen de su tipo (intensidad de la luz, distancia focal de la cámara, si se debe simular la gravedad para un cuerpo o no…), cuyos valores se muestran en el editor y se pueden modificar, para personalizar al máximo cada objeto (Unity Technologies, 2017).

Las escenas son estructuras que se utilizan para dividir las partes de un juego. El concepto detrás de ellas es que contienen todos los objetos de un mismo nivel. Pueden ser utilizadas para crear cualquier tipo de nivel, bien sea un menú, una escena cinemática, un nivel de juego… Se pueden cargar o "descargar" de forma dinámica durante la ejecución para gestionar los recursos del sistema que consumen, y existe una clase en la API de Unity dedicada a la gestión de escenas, incluyendo eventos para diversas situaciones que las atañen (Unity Technologies, 2017).

Una de las características más útiles de Unity son los llamados *prefabs* (entiéndase como "objetos prefabricados"). Al crear un juego en el editor, utilizamos un *gameobject* y uno o varios componentes para crear útiles que utilizaremos en las escenas. En muchas ocasiones, crearemos objetos que querríamos usar más de una vez, como un árbol, un personaje no controlable, o una farola. En esos casos, Unity nos permite utilizar un objeto para crear un *prefab* a partir de él, y utilizarlo para tener indefinidas copias de ese objeto que podemos modificar de forma independiente, con la ventaja de que podemos hacer cambios en el *prefab* y que se repliquen en todas sus copias. Es una forma sencilla y ágil de reutilizar objetos y poder personalizarlos (Unity Technologies, 2017).

*MonoBehaviour* es la clase base para todos los scripts que se escriben para Unity, que tienen que derivar de ella, para poder acceder a una serie de funciones, propiedades, y eventos, extremadamente útiles cuando programamos videojuegos (Unity Technologies, 2017). Por ejemplo, tenemos acceso a funciones como *Awake*() y *Start*(), que se llaman al crearse un objeto (con sus diferencias), o a *Update*(), que es llamada en cada *frame* del juego – algo que en muchas situaciones sería una mala práctica de programación, constituye un patrón esencial en programación de videojuegos, llamado Game Loop (Nystrom, Game Programming Patterns: Game Loop, 2014)–.

Por último, es importante mencionar una característica muy conveniente cuando escribimos clases C# para Unity, y es que cuando declaramos variables como públicas en un script que hereda de *MonoBehaviour*, hacemos que esas variables queden expuestas en el editor al asignar el script como un componente a un *gameobject* (Unity Technologies, 2017). Esto tiene dos funciones principales: – la primera es que podemos ejecutar pruebas en las que los comportamientos dependan de valores del script simplemente jugando en el editor, y cambiando en el componente los valores de las variables, ahorrándonos así el paso de volver al IDE para ir haciendo cambios; y – la segunda es que si esas variables pueden ser referencias a objetos de la escena o *assets* del proyecto, podemos asignar esos valores arrastrando el *asset* directamente en el editor, en vez de consumir recursos en buscar el activo en tiempo de ejecución dentro de la escena o el proyecto.

## 2.2 UNITY REMOTE

Unity Remote, que está actualmente en su versión 5, es una app móvil desarrollada para Android e iOS que está ideada para ayudar, precisamente, con el desarrollo en Unity orientado a esas plataformas.

La aplicación toma el input del dispositivo en lugar del editor del motor, y manda a dicho dispositivo el output gráfico una vez Unity lo ha procesado. Es reseñable que no se realiza la ejecución en el propio dispositivo móvil.

Para ese input, la aplicación es capaz de recabar del dispositivo Android o iOS los datos de acelerómetro y giroscopio, la entrada táctil y multitáctil, el posicionamiento GPS y la brújula, e incluso datos desde la cámara del dispositivo (Unity Technologies, 2016).

## 2.3 MONODEVELOP

MonoDevelop es un IDE multiplataforma (Linux, MacOS y Windows) con soporte para cualquier tipo de lenguaje, aunque orientado especialmente a C#, C++, y Visual Basic. Posee características como autocompletado del código, *layout* personalizable, o un *debugger* integrado.

Aunque es software libre, y puede ser obtenido directamente desde la página web del desarrollador por quien lo desee, Unity proporciona MonoDevelop en sus descargas porque lo utiliza como IDE por defecto al abrir los scripts escritos para el motor.

## 2.4 PHOTOSHOP

Adobe Photoshop es el software de creación y edición de gráficos más conocido y utilizado de su sector. Es desarrollado por Adobe Systems Incorporated. Tiene infinidad de herramientas de edición y soporta multitud de formatos, entre los cuales está el .PNG con capa de transparencia, que es el motivo por el que lo utilicé para generar algunos *sprites* muy básicos con los que probar el funcionamiento del juego.

Las licencias de Photoshop pueden ser adquiridas por diferentes precios según el plan dentro del cual se compren, aunque también puede obtenerse una versión de prueba del programa, gratuita y de duración limitada.

## 2.5 C#

C# (C Sharp) es un lenguaje de programación desarrollado por Microsoft como parte de su *framework* .NET. Es un lenguaje orientado a objetos, basado en clases, y de tipado fuerte. Actualmente está en su versión 7.0, lanzada este mismo año 2017.

C# es uno de los lenguajes que proporciona Unity de forma nativa para la programación de los juegos, junto con JavaScript. También se admitía Boo, pero por falta de usuarios de ese lenguaje, se le dejó de dar soporte en 2014.

## 2.6 MATERIALES

Los materiales utilizados en este proyecto (entendiendo aquí como materiales los activos, o *assets*, utilizados en el juego) provienen de 3 fuentes diferenciadas.

Antes de detallarlas, conviene destacar que el apartado de los materiales no es un propósito principal de este proyecto, sino un medio de mostrar de forma visualmente agradable su trasfondo tecnológico. Los materiales y sus fuentes aparecen mencionados a título meramente informativo.

Parte de los activos forman parte del paquete de activos estándar de Unity, que permiten importar por defecto en cada proyecto, y que se distribuyen con licencia para darles cualquier uso personal o comercial, alterado o sin alterar.

En segundo lugar, una segunda parte de estos activos – más concretamente, iconos para la interfaz y los tinteros que aparecen en los puzles y el panel de progreso – son obra de Kenney Assets, que los publica en su web libres de copyright, para uso personal o comercial sin necesidad de requerir permiso explícito ni atribución (Kenney Assets, 2010-2017).

Por último, otra parte del material gráfico que aparece en el juego es obra de la empresa Creative Rainbow, cuyo director colabora en el proyecto como co-tutor, u obra de mí mismo, utilizando Adobe Photoshop para crear *sprites* de prueba durante el desarrollo.

# 3 METODOLOGÍA

La metodología utilizada durante la realización del proyecto ha sido una metodología iterativa incremental. Este tipo de metodologías se basan en desarrollar el sistema realizando iteraciones

del ciclo “Análisis – Diseño – Implementación – Pruebas”, e ir añadiendo características nuevas en cada iteración (Sommerville, 2005) (Pantaleo & Rinaudo, 2014). En la Ilustración 3 se muestra un esquema representativo de esta metodología.

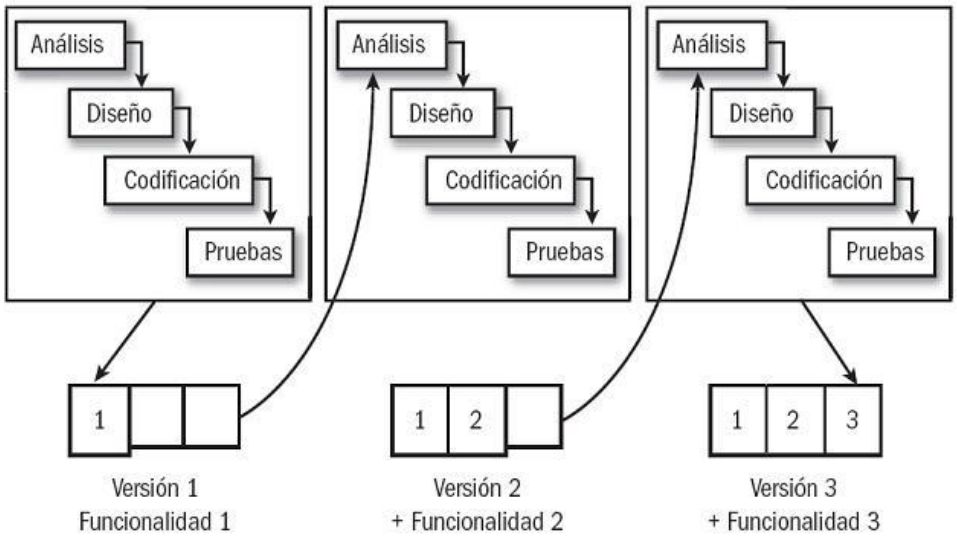


*Ilustración 3 - Desarrollo iterativo incremental*

Esta forma de trabajar permite ejecutar pruebas sobre módulos individuales del juego sin necesidad de implementarlo completamente, y detectar fallos en cualquier fase anterior lo antes posible.

Las iteraciones se pueden definir a intervalos de tiempo (por ejemplo, completando un ciclo cada dos semanas) o mediante hitos (trabajando en un módulo o ciertas características del sistema y finalizar la iteración cuando quedan listos). En mi caso, al tener a la empresa Creative Rainbow haciendo las veces de cliente del proyecto, decidí basar mis iteraciones en ciclos de dos semanas que acabasen con una “entrega” o sesión de validación por parte de la empresa.

## 3.1 PLANIFICACIÓN

El proyecto contaba con unos plazos bien definidos, que se han utilizado para dar forma a las etapas de desarrollo y las fases de las mismas. En la Ilustración 4 se puede observar un diagrama de Gantt que ilustra el avance del proyecto.

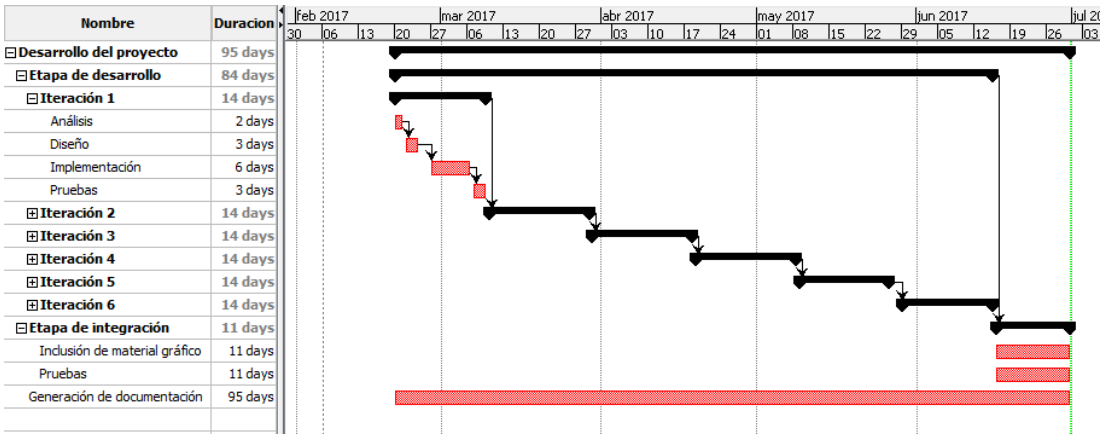


*Ilustración 4 - Diagrama de Gantt*

Podría distinguir dos etapas diferenciadas:

### 3.1.1 Etapa de desarrollo

La etapa de desarrollo es la más larga y determinante del proyecto. Se divide en las fases más típicas de cualquier desarrollo software: análisis, diseño, implementación y pruebas.

- **Análisis:** Durante las fases de análisis se determinaban los hitos que era necesario alcanzar, y los medios con los que alcanzarlos, y se concretaban en requisitos que debería satisfacer la posterior implementación. Evidentemente, la primera etapa tuvo la fase de análisis más larga, porque hubo que determinar los requisitos que se quería que cumpliera el sistema final. El resto de iteraciones traían pequeñas adiciones o modificaciones, en su mayor parte.
- **Diseño:** Las fases de diseño estaban orientadas a las decisiones de cómo se cumplirían los requisitos definidos en la fase de análisis. Hablamos tanto de diseño a nivel abstracto, con lo que es el diseño de software propiamente dicho (patrones de programación, estructura de las clases…); como en términos visuales, dando forma a los diferentes escenarios o partes de la interfaz de usuario (cosas para las que el editor de Unity se presta extremadamente útil).
- **Implementación:** Las fases de implementación se basaban en crear la lógica del juego, en programar todos los comportamientos que se habían venido diseñando de forma que satisficieran los requisitos fijados.
- **Pruebas:** En estas fases se comprobaba que el funcionamiento de todo lo que se había creado durante la iteración fuera el correcto. Más adelante, en el apartado 6, se detallan todos los tipos de prueba que se han llevado a cabo durante la realización del proyecto.

### 3.1.2 Etapa de integración

En la última etapa, los cometidos eran: incluir los materiales gráficos finales, para dar una apariencia más cuidada y agradable al juego; comprobar que todo en el sistema funcionase correctamente, a modo de prueba íntegra final; y, generar la documentación restante del proyecto, como esta memoria, y comprobar su exactitud.

# 4 ANÁLISIS DE REQUISITOS

A grandes rasgos, el proyecto es un juego que permita a los usuarios familiarizarse con el tipo de razonamiento más cercano a la programación a través de puzles que necesiten de la aplicación de ese tipo de razonamiento para ser resueltos.

En la fase de análisis del proyecto, se han recogido distintos requisitos que se deben cumplir para poder decir que el juego cumple su función plenamente. Estos requisitos han sido capturados a través de un análisis personal de las necesidades del sistema, así como de entrevistas con la otra parte del proyecto, la empresa co-tutora del mismo.

Además, como el software que está siendo desarrollado es un videojuego, hay que analizar el sistema también con un enfoque distinto, en lo que se conoce comúnmente como *Game Design*. El nombre *Game Design* no debe inducir a confusión: a pesar de incluir la palabra ‘diseño’, es una disciplina que se basa en el análisis del juego y su propósito para sustentar el trabajo que se deberá hacer en la fase de diseño. Se generarán documentos como el *concept pitch*, que resumen brevemente las características del juego.

## 4.1 GAME DESIGN

Cuando se desarrollan videojuegos, se añade una nueva dimensión al análisis y diseño de un proyecto software. Los videojuegos son programas, no sólo funcionales, sino con un componente de entretenimiento del usuario, que es un aspecto más a tener en cuenta. Trabajamos un paso más cerca de la psicología, y entra en juego lo que llamamos *Game Design*: un análisis de las necesidades del proyecto fundamentado en los puntos que tiene que cubrir un juego.

Debemos perseguir la diversión del usuario, lo cual se puede conseguir de distintas maneras: la competición, la exploración, la creación… (Telefónica Educación Digital, 2017). En nuestro caso, los elementos determinantes para esto serán el reto y la sensación de progreso.

Es de gran ayuda, para definir la idea de juego que tenemos en mente, elaborar lo que se llama un "documento de concepto" del juego (o "*Concept Pitch*"). En él reflejamos un resumen de la idea de juego, el apartado gráfico, las plataformas y públicos objetivo, los puntos fuertes (o USP, del inglés *Unique Selling Points*), y la jugabilidad y las mecánicas que la definen (Telefónica Educación Digital, 2017). El documento de concepto de este proyecto figura a continuación (hay que recordar que este documento se escribió con un proyecto de mayor alcance en mente para la empresa codirectora, y el proyecto que a nosotros nos ocupa resulta de una generalización y primer acercamiento al original):

***

1. RESUMEN

"Aventura" de puzles, orientada a enseñar programación a niños (jóvenes) a través de la resolución de pequeños ejercicios.

2. JUGABILIDAD

- Moverse, para poder avanzar por el escenario. No es necesaria una segunda velocidad de movimiento, no hay ninguna amenaza ni prisa para el jugador.

- Saltar, para superar obstáculos o activar ciertos mecanismos básicos.

- Interactuar, para hablar con NPCs (siglas en inglés para Non-Playable Character, o personajes no jugables) o activar ciertos mecanismos básicos.

- Seleccionar botones, que representan acciones.

- Desarrollo por mundos, con un tema de programación por mundo (secuencial, funciones, bucles, condicionales).

- [Alguien o algo] introduce al jugador a lo que vas a aprender en cada nivel.

- Fuera de cada nivel hay un apartado "Tutorial" por si el jugador necesitase refrescar lo aprendido (o terminar de entenderlo). Incluiría ejemplos resueltos.

3. APARTADO GRÁFICO

El desarrollo es en 2D, de avance lateral.

La estética es desenfadada, con atmósfera de diversión, inofensiva, y un aspecto colorido. Los personajes y los entornos no tienen apariencias estrictamente realistas.

Diseño muy inspirado en la saga Scribblenauts, en términos de desarrollo y diseño. Se muestra a continuación una captura de pantalla de Scribblenauts:

*Ilustración 5 - Pantalla de Scribblenauts*

La apariencia de las pantallas de resolución de los puzles se inspirará mayormente en la interfaz de Lightbot, que se muestra a continuación:

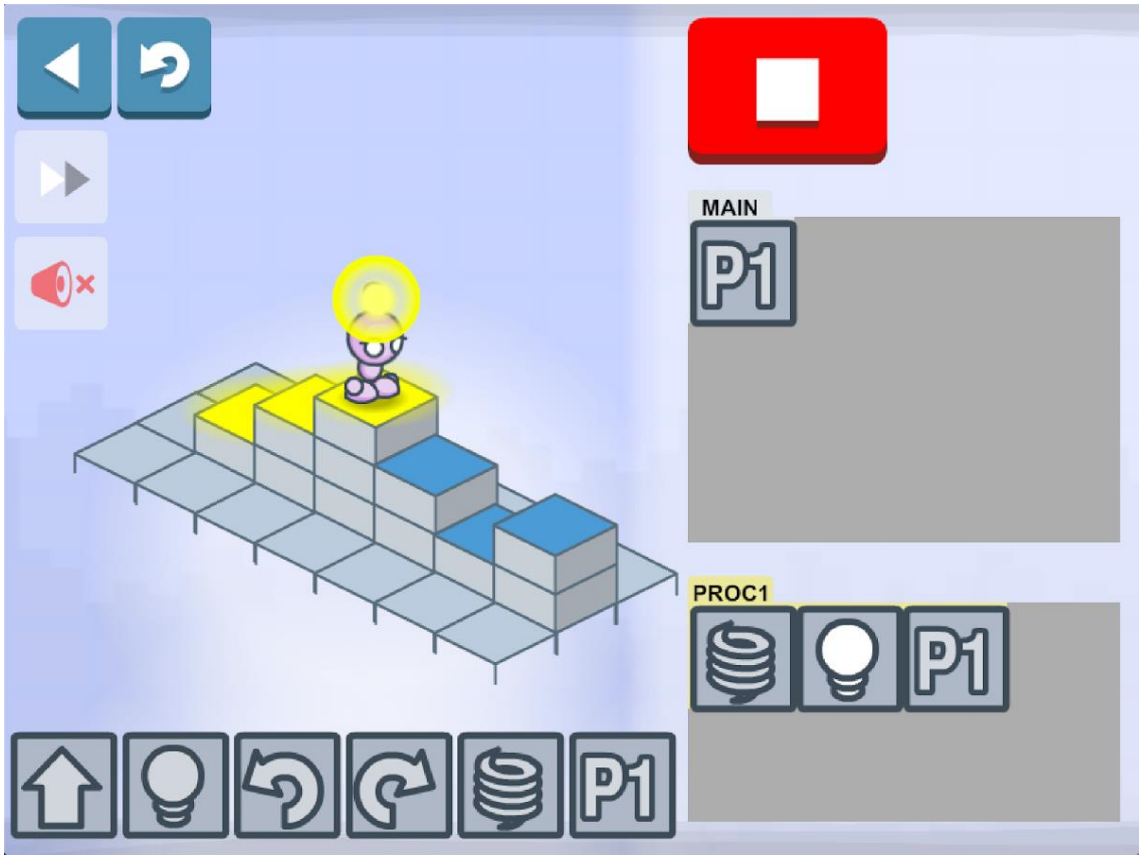


*Ilustración 6 - Pantalla de Lightbot*

4. PLATAFORMA(S) Y PÚBLICO OBJETIVO

PC, móviles y consolas portátiles, en ese orden de preferencia.

El público objetivo principal es los jóvenes, de 7 a 15 años aproximadamente; el género es indiferente; el mercado geográfico es indefinido, aunque es en Europa y Norteamérica donde más fuerza tiene la corriente educativa que trata de impulsar el aprendizaje de programación desde edades tempranas.

En segundo lugar, se tiene como público a cualquier usuario potencial que carezca de conocimientos de programación, por el carácter educativo del juego.

5. PUNTOS FUERTES

¿Qué es lo que hace que nuestro juego sea único y especial? ¿Qué lo diferencia de todos los demás?

- Muchos de los juegos para enseñar programación actualmente son adaptaciones del programa Scratch con apariencias más o menos atractivas para los niños. Este juego adopta la mecánica de soluciones como Lightbot, donde es más sencillo intuir qué se está haciendo sin tener que entender pseudocódigo como el de Scratch.

- El juego no frustrará con los puzles y atraerá la atención porque los ejercicios se enmarcan en un mundo de juego navegable. No se reduce a las pantallas para resolver los ejercicios.

- La distribución de contenido del juego es lógica y responde a modelos académicos reales (asignatura Introducción al Software).

***

Las mecánicas que hemos mencionado se definen como la parte más lógica del juego, lo que queda si eliminamos todos los componentes estéticos (visuales, narrativos…). Son las mecánicas lo que define cómo se juega a un juego (lo enmarca dentro de un género típico) y cómo el usuario interactúa con el sistema. Son mecánicas no sólo los controles, sino también reglas del juego, el espacio de juego (2D, 3D, o variantes mixtas), u objetos del entorno y su interacción con el jugador (Telefónica Educación Digital, 2017).


*Ilustración 7 - Modelo MDA*

Y todo esto tiene un objetivo, y es generar dinámicas en el juego. Son un concepto más difícil de definir porque son algo más abstracto. Son la conexión entre las acciones del jugador y sus objetivos, lo que pasa por la cabeza del usuario mientras utiliza el sistema. Es culpa de las dinámicas de juego que el usuario tenga una experiencia de juego determinada, y el papel del (conocido como) *game designer* es, teniendo en cuenta la experiencia que desea transmitir, pensar en unas dinámicas de juego que las transmitan, y generarlas creando mecánicas adecuadas. Después, el usuario recorrerá el camino en sentido inverso: las mecánicas que deberá ejecutar, generarán unas dinámicas que transmitirán las experiencias para la que fue creado un juego (Telefónica Educación Digital, 2017). Esto es lo que se conoce como modelo MDA (*mechanics, dynamics, aesthetics* o experiencias) y se muestra en la Ilustración 7.

## 4.2 REQUISITOS FUNCIONALES

Primero se detallan, en la Tabla 1, los requisitos funcionales, las características que debe cumplir el sistema.

| ID | DESCRIPCIÓN |
|---|---|
| **RF01** | El juego proveerá un entorno jugable por el que moverse entre los distintos puzles |
| **RF02** | Siempre que se navegue el entorno jugable, el sistema informará al usuario de su progreso actual, es decir, el número de puzles resueltos sobre el total |
| **RF03** | Todos los puzles del juego tienen una sola casilla inicial y un elemento objetivo unívocos |
| **RF04** | Los puzles siempre se finalizarán cuando el jugador recoja el elemento objetivo |
| **RF05** | El jugador solucionará los puzles seleccionando comandos para componer una posible solución |
| **RF06** | Los puzles tendrán un número máximo de comandos para crear una solución |
| **RF07** | Cuando un puzle es correctamente resuelto, debe otorgar un incremento de uno (1) al número máximo de comandos usables |
| **RF08** | Cuando un puzle es correctamente resuelto, el juego debe devolver el jugador a la zona jugable entre puzles |
| **RF09** | Cuando la solución de un puzle es incorrecta, el sistema informará al jugador y reiniciará el puzle |
| **RF10** | El usuario podrá en todo momento pausar el juego |
| **RF11** | El usuario podrá en todo momento salir del juego |
| **RF12** | Mientras no hay una solución en ejecución, el usuario podrá resetear una solución |
| **RF13** | Mientras no haya una solución en ejecución, el usuario podrá salir del puzle |

| | |
|---|---|
| **RF14** | El sistema debe mantener la información necesaria para que el progreso del jugador persista |
| **RF15** | En los puzles, el jugador podrá moverse a izquierda o derecha |
| **RF16** | En los puzles, el jugador podrá ascender o descender escaleras |
| **RF17** | En los puzles, el jugador podrá escalar o descolgarse de salientes a izquierda o a derecha |
| **RF18** | En los puzles, el jugador podrá coger objetos del suelo o soltarlos |

*Tabla 1 - Requisitos funcionales*

### 4.3 REQUISITOS NO FUNCIONALES

Detallo a continuación (Tabla 2) los requisitos no funcionales, que atañen al diseño y la implementación del sistema (ISO25000, 2017).

| ID | TIPO | DESCRIPCIÓN | IMPORTANCIA |
|---|---|---|---|
| **RNF01** | Portabilidad | El juego debe poder ejecutarse en escritorios de Windows, MacOS y Linux | Alta |
| **RNF02** | Usabilidad | El sistema debe ser accesible: contar con una interfaz clara y autoexplicativa | Alta |
| **RNF03** | Usabilidad | El juego debe ser apto para todo tipo de públicos por contenido y sencillez | Media |
| **RNF04** | Portabilidad | El juego debe ser compatible con tecnología WebGL para poder ser ejecutado en la mayoría de navegadores web actuales | Alta |
| **RNF05** | Rendimiento | El juego no debe bajar nunca de los 30 *frames* por segundo | Media |

*Tabla 2 - Requisitos no funcionales*

## 5 DISEÑO E IMPLEMENTACIÓN

En este capítulo hablaré primero del diseño arquitectónico del sistema, para después entrar en los detalles de diseño e implementación de cada una de las partes.

Recuerdo que en este apartado se mencionarán conceptos como *game object* (u objeto de juego), escena, o la "capacidad de exponer variables en el editor", que son propios de Unity y que he explicado en la sección 2.1.1.

Debo mencionar que debido al contrato de confidencialidad con Creative Rainbow, la empresa que codirige el proyecto, no puedo mostrar la totalidad de la implementación del juego, por lo que se mostrarán únicamente fragmentos del código de ciertas clases, que no comprometan el acuerdo.

## 5.1 ARQUITECTURA DEL SISTEMA

El modelo arquitectónico que se aplica a este juego es un patrón de arquitectura por capas. Es habitual encontrar este tipo de patrón dividido en capas de presentación, negocio y datos. En este caso, prefiero reducirlo a la capa de presentación y la de negocio, y dejar la capa de datos como algo tácito.

Esta distribución que hago tiene su explicación en la arquitectura y el comportamiento de Unity. Es fácil identificar la capa de presentación con la interfaz de usuario y la capa de negocio con la lógica detrás del juego, pero a la hora de identificar los datos que requieren persistencia en el sistema, nos encontramos dos casos especiales.

El primero de esos casos especiales es que los datos que realmente necesito almacenar para que el jugador pueda guardar y continuar sus partidas son únicamente un par de valores, y el guardado y cargado de los mismos se realiza mediante llamadas a la API de Unity, que se encarga de salvarlos en archivos del juego (dependientes de la plataforma). Esto significa que no es necesario crear un sistema distinto de almacenamiento de datos ni una interfaz para comunicarse con él.

El segundo caso especial sería considerar también las escenas (y su contenido, por descontado) como datos que deben persistir, puesto que, aunque parte del contenido son elementos visuales del juego (presentación), o puedan tener asignados comportamientos en relación con el jugador (negocio), también son información que debe ser consistente entre sesiones de juego. Es decir, durante el desarrollo se han diseñado el entorno y los puzles de una determinada manera, y es vital que permanezcan en ese estado. Es obvio que durante el desarrollo – como proyecto –, o una vez finalizado el mismo – como juego –, esos elementos deben ser almacenados. Pero, aunque así lo interpretáramos, de nuevo es Unity quien se encarga de almacenar toda esa información para el juego sin que dependa de nosotros el cómo o el dónde.

Con respecto al caso de las escenas, lo más natural será definirlas como parte de la capa de presentación, porque, obviando la interfaz, también son un elemento indispensable para la comunicación entre el juego y el usuario.

Con ese punto aclarado, los siguientes apartados ahondarán en el diseño y la implementación de las capas de presentación y negocio.

## 5.2 CAPA DE PRESENTACIÓN

La capa de presentación está formada por dos piezas: la interfaz de usuario y las escenas de juego.

La interfaz de usuario es algo presente en cualquier software con el que un usuario interactúe, por lo que su existencia aquí es natural. Sin embargo, las escenas pueden parecer algo más ajeno a la costumbre. Para entenderlo mejor, podemos entenderlas como algo análogo a las pantallas de navegación de una aplicación móvil, con el añadido de que, para movernos entre ellas, la interacción requerida consistirá, la mayor parte de las veces, en jugar para avanzar.

### 5.2.1 Interfaz de usuario

La interfaz de usuario del juego se puede dividir en dos elementos, esencialmente.

Por un lado, tenemos la parte más gráfica de la interfaz: el conjunto de botones, paneles, textos… que creé en el motor, con las herramientas de diseño de Unity, de acuerdo a diseños previos que había realizado a mano, como los que se pueden ver en las ilustraciones siguientes.

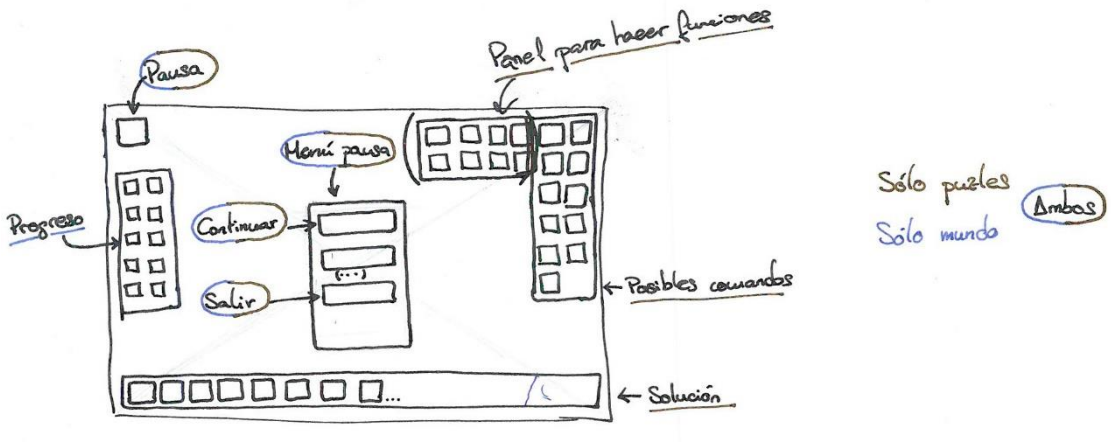


*Ilustración 8 - Diseño preliminar de la GUI*

En la Ilustración 8, vamos gran parte de los elementos que conforman la interfaz, y un código de colores para discernir si cada elemento debe aparecer en las fases de puzle, las de mundo explorable, o ambas. Se pueden identificar el botón de pausa, arriba a la izquierda; el panel que informa del progreso del usuario, justo debajo; el menú de pausa, centrado, con botones para continuar o salir del juego; y dos paneles exclusivos de las escenas de puzles: el panel de comandos utilizables y el de comandos seleccionados para la solución. También hay un panel que en primera instancia fue diseñado para crear funciones, pero en iteraciones posteriores se descartó para no introducir un elemento que incrementase tanto la curva de dificultad del juego.

En la Ilustración 9, vemos algunos elementos repetidos con respecto a la anterior, pero aparecen ya los botones de ejecución o reseteo de la solución, y valores que establecían tamaños y espacios entre los elementos. Se asumía un tamaño de pantalla de 1024x768 píxeles, de forma orientativa, pero con la utilización de anclas debidamente colocados durante el diseño de la interfaz, los tamaños y espacios entre los elementos son proporcionales y admiten cualquier tipo de resolución.

Todos estos elementos forman parte de la escena, y están contenidos en un objeto de juego que incluye un sistema de eventos para recoger las interacciones del usuario con los elementos de la interfaz.

Para responder a esos eventos, además de gestionar otros comportamientos de la interfaz, tenemos el segundo elemento clave de ésta: *InterfaceManager*. *InterfaceManager* es un script C#, escrito por mí, que se añade como componente al objeto de juego que contiene toda la interfaz.

*InterfaceManager* se compone de métodos de respuesta a los eventos de interacción con la interfaz, y métodos para modificarla según las acciones del jugador. Una de esas posibles acciones es la mecánica principal para resolver los puzles:

El usuario dispone, en la parte derecha, de un panel que contiene todos los posibles comandos que puede ejecutar en la solución. En la zona inferior de la pantalla, un segundo panel, vacío al

comienzo, se va rellenando con los comandos que el usuario va pulsando en el panel derecho. Así, existe un *feedback* gráfico y directo de la solución que el jugador va componiendo.

Junto al panel inferior hay dos botones. El primero permite dar por finalizada la composición de la solución y comenzar a ejecutarla. El segundo es útil en caso de que el usuario se equivoque al introducir algún comando. En vez de tener que ejecutar una solución con la que no está conforme para volver a empezar, puede pulsar ese segundo botón para limpiar el panel y resetear la solución, agilizando así el proceso de juego.

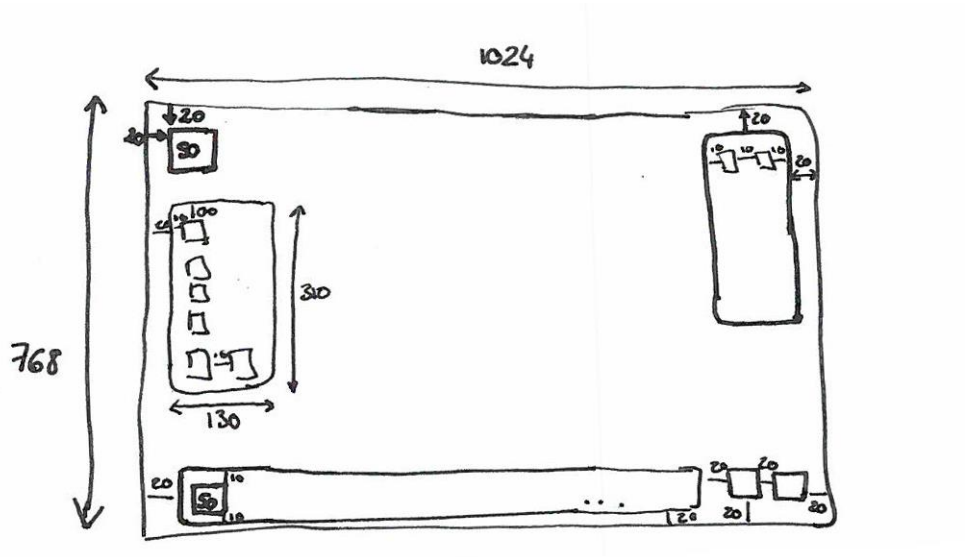


*Ilustración 9 - Diseño de GUI con relación de aspecto*

Respecto a la implementación de *InterfaceManager* cabe destacar la utilización del patrón *Observer*. Aunque este patrón de propagación de eventos no ha sido implementado por mí mismo, sino que Unity (C#) lo tiene incluido de forma nativa, merece la pena mencionarlo porque su uso permite conseguir un resultado con una cantidad mucho más resumida de trabajo (y porque implementar los delegados de forma que respondan al funcionamiento del evento también requiere un esfuerzo en términos de programación).

El evento con el que se usa el patrón es *activeSceneChanged*; es parte de la clase *SceneManager* (de la librería *SceneManagement* de Unity) y se activa cuando la escena actualmente activa en el juego cambia a otra escena. Los delegados que suscribimos a este evento tienen que ser referencias a métodos que tomen dos argumentos del tipo *Scene* (incluido en la librería antes mencionada) y con un valor de retorno nulo.

En *InterfaceManager*¸ el delegado ha sido un método que he llamado *AdaptUI*. Conociendo las escenas origen y destino, se encarga de activar o desactivar elementos de la interfaz para adatar ésta al contexto en el que nos encontremos. Esto es posible por la capacidad que nos da Unity de exponer variables del script en el editor, de forma que puedo darle a la clase referencias a todos los elementos de la interfaz que me interese conocer antes de comenzar la ejecución.

En la Ilustración 10 se muestra el diseño final de la interfaz en el editor de Unity.

En el lateral izquierdo se aprecia el panel de progreso (activo sólo en la escena de mundo), con las imágenes correspondientes a los diez puzles, aún sin rellenar.

En el lado opuesto se encuentra el panel de comandos, conteniendo los botones de los diez comandos disponibles, que sólo forma parte de las escenas de puzle.

En la zona inferior está el panel que contiene la solución, y los botones de ejecución y reseteo de la misma, también disponibles sólo durante las escenas de puzle.

En la esquina superior izquierda nos encontramos el botón de pausa, siempre activo, y el de regreso, que sólo lo está durante los puzles para permitir volver a la escena de mundo.

Por último, en la región central, nos encontramos tres paneles superpuestos. Uno es el menú de pausa, que aparece al pulsar el botón de pausa. Los otros dos son paneles informativos que aparecen cuando un usuario finaliza el puzle satisfactoriamente, uno, o cuando ejecuta una solución incorrecta, el otro.

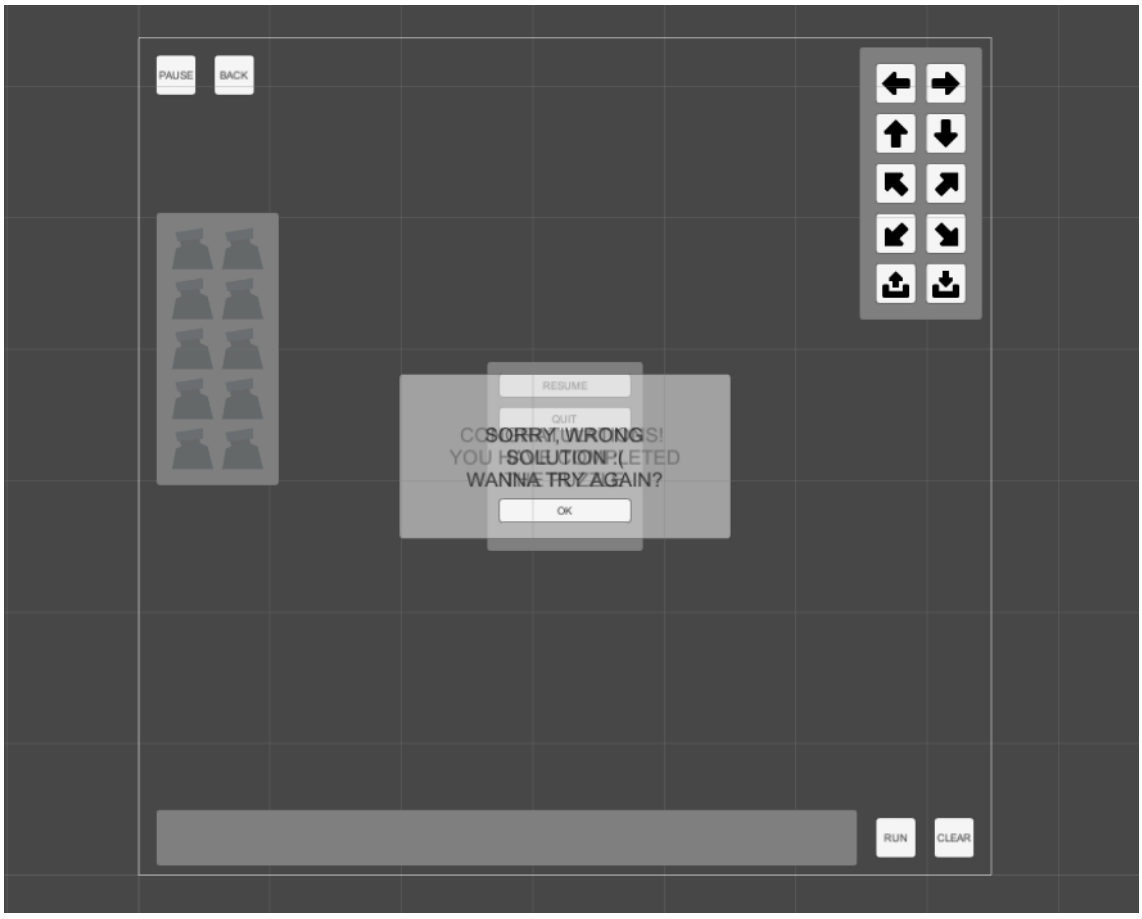


*Ilustración 10 - Diseño final de la GUI, en el editor*

### 5.2.2 Escenas

Podríamos decir que las escenas son la parte jugable de la interfaz. Representan gráficamente el entorno por el que el jugador se mueve durante el juego. En la Ilustración 11 se muestra un diagrama de navegación que ilustra cómo puede el usuario moverse en el juego por las diferentes escenas.

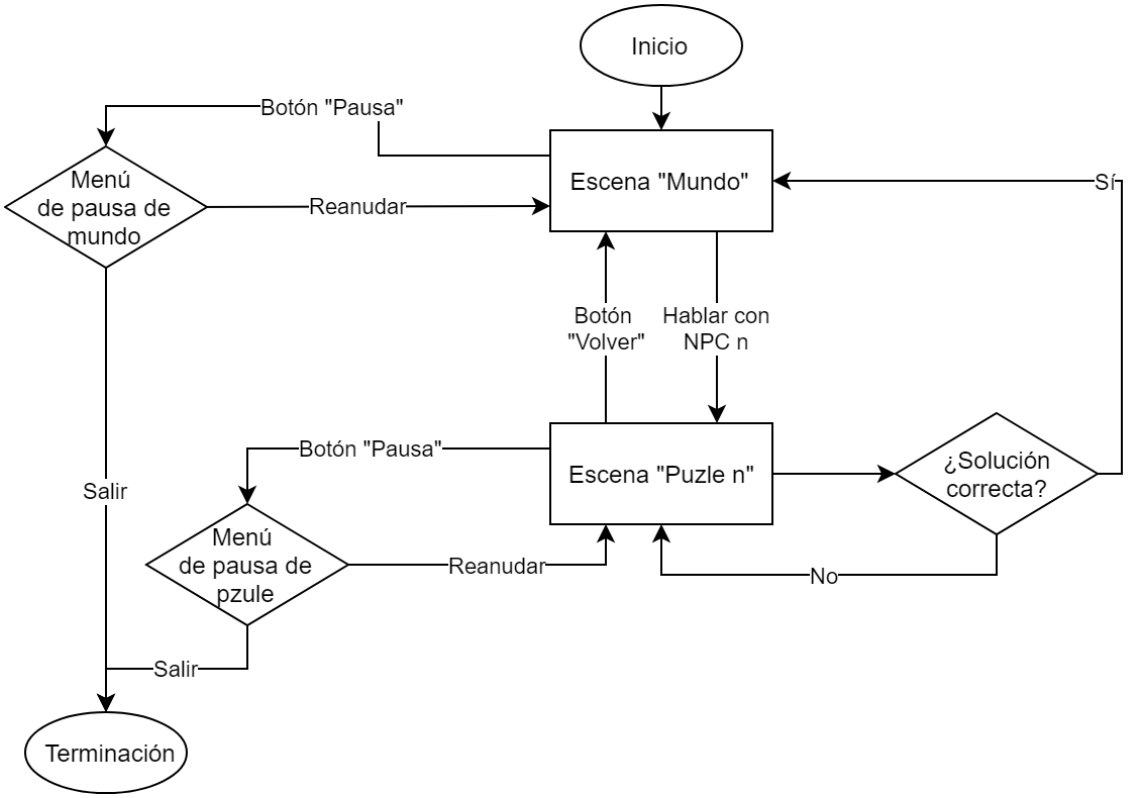


*Ilustración 11 - Diagrama de navegación de las escenas del juego*

Hay dos tipos de escenas en el juego: una gran escena por la que el jugador se mueve libremente, controlando al personaje e interactuando con elementos del entorno (a la que llamaré “escena del mundo”), y diez escenas más pequeñas que conforman los diez puzles del juego.

Durante la etapa de análisis descrita en el apartado 0, se diseñaron las escenas a mano, antes de crearlas en el motor de juego, teniendo en cuenta distintos aspectos.

Para la escena del mundo, quise crear un entorno que propiciara que el jugador fuera familiarizándose con los conceptos importantes del juego: los controles, la posibilidad de interactuar con ciertos elementos, como ascensores o NPCs, la importancia de la exploración…

En la Ilustración 12 se puede apreciar un primer diseño, de la escena de mundo, incluyendo anotaciones acerca de las dinámicas que generan diferentes elementos. Fue un documento útil desde el principio, al comenzar a prototipar el juego, y que, además, ayuda a tener claros todos los mecanismos antes de empezar a implementar los comportamientos del juego, y al que, debo añadir, me he mantenido bastante fiel en la versión final, descontando alteraciones debidas a la inclusión del apartado gráfico. Podría decirse que es el esqueleto del mapa final del juego.

En lo que respecta a las escenas de puzle, la complicación fue aún mayor. El objetivo de los puzles era que tuvieran que ser resueltos en un orden concreto, para que el jugador fuera sintiendo un progreso, puesto que la dificultad de los puzles era creciente. Había que conseguir que el usuario tuviera que solucionar los puzles en el orden previsto sin forzarlo a hacer una simple lista de puzles, porque eso haría que el avance del jugador fuera demasiado guiado.

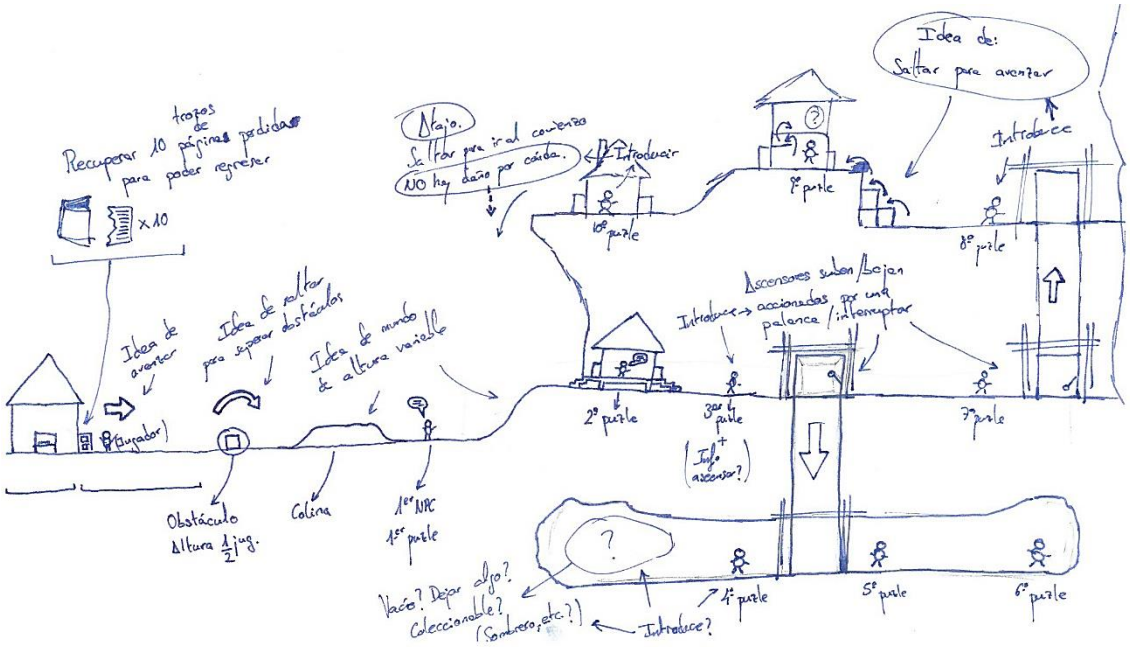


*Ilustración 12 - Concepto de la escena del mundo*

Para cumplir ese objetivo, los puzles se diseñaron con dos premisas: – el jugador tiene un máximo de movimientos para ejecutar, y – cada vez que el jugador resuelve un puzle, amplía su límite a un movimiento más. De esta manera, los puzles están creados para resolverse en un número determinado (y creciente) de movimientos. Se pueden observar algunos de los conceptos de diseño de los puzles en la Ilustración 13 y la Ilustración 14.

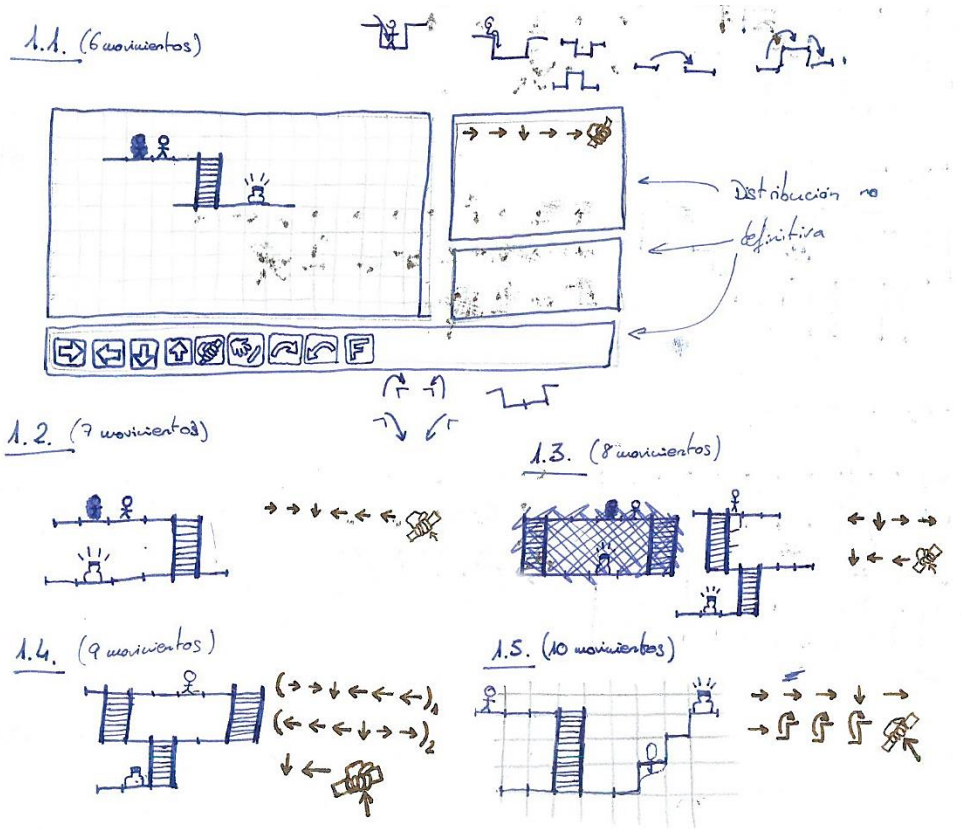


*Ilustración 13 - Concepto del diseño de los primeros 5 puzles*

Casos como el del puzle 1.4 que también aparece en la Ilustración 13, tienen como objetivo generar ideas en el usuario, en el proceso de desarrollo de su razonamiento computacional. Por ejemplo, en el caso de ese puzle, quiero hacer que el usuario comprenda que en ocasiones hay más de un camino correcto para llegar a una solución.

También ocurre que hay que prestar especial atención a cómo se podrían aplicar las reglas de juego en casos no esperados para que el usuario no se pueda saltar las limitaciones impuestas. Hablo no sólo de que se pueda encontrar una solución más corta que la que yo pensé al crear el puzle, sino también casos como el último puzle, en la parte inferior de la Ilustración 14, donde eliminé el promontorio que aparece tachado con una 'X' negra porque descubrí que, por cómo estaban definidas las reglas del juego, el jugador podía saltar del bloque al piso superior a pesar de que no aparecía un medio físico para ello en la escena. Convertir el bloque en suelo llano eliminaba el problema sin alterar el número de movimientos requeridos para alcanzar la solución.

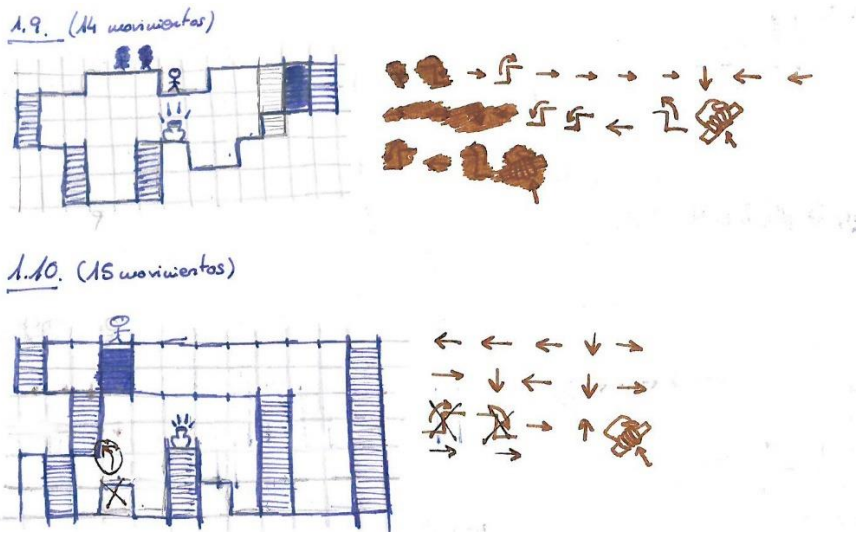


*Ilustración 14 - Concepto del diseño de los dos últimos puzles*

Al igual que el de la escena del mundo, el diseño de los puzles también se ha respetado, y aunque visualmente cambie, la estructura que subyace es la que aparece en las imágenes. Para representar las casillas en las que se dividen los puzles, utilicé un *prefab* que hice usando cajas de colisión y un *sprite* (una imagen) para dar apariencia.

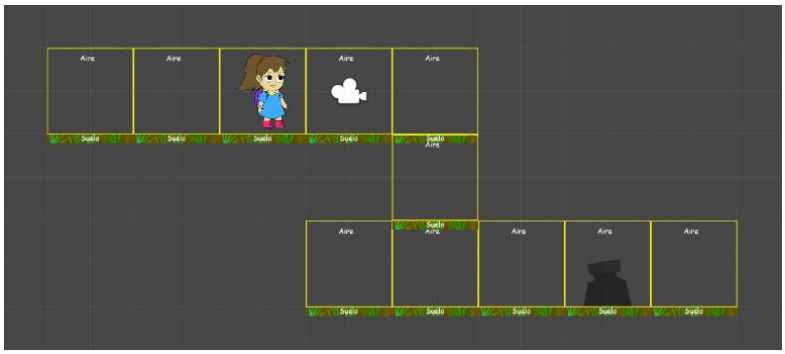


*Ilustración 15 - Puzle 1.1 en el editor, con la apariencia dada por el provisional*

Se muestra, en la Ilustración 15,**¡Error! No se encuentra el origen de la referencia.** la apariencia d el primer puzle en el editor durante el desarrollo, en comparación con lo visto en las imágenes conceptuales, una vez aplicado el *sprite* mencionado sobre la estructura del puzle.

## 5.3 CAPA DE NEGOCIO

En este apartado, hablaré de la lógica detrás del juego.

Aunque en todo el juego hay comportamientos programados, muchos no encarnan una gran complicación. En la parte del entorno jugable, la escena del mundo, existen clases que hacen funcionar elementos, como los personajes no jugables, scripts para controlar al personaje, o los ascensores (por ejemplo, el movimiento de los ascensores en la escena del mundo es una corrutina que va actualizando la posición a cada *frame,* interpolando entre la posición inicial y la final).

Aunque su programación no es trivial, sí que tiene un impacto menor en el resultado final del juego, y su importancia relativa frente a otros módulos es baja. Es por eso que no entraré a detallar elementos de esta sección del juego, sólo explicaré los puntos más reseñables de cómo está organizado, y los patrones que he utilizado en ellos.

De nuevo, en esta sección, deberé omitir cierta información debido al acuerdo de confidencialidad firmado con Creative Rainbow, la empresa que colabora en el proyecto.

Podemos dividir el juego en una parte de entorno jugable y otra de puzles. Por encima de ambas hay un elemento, una clase, llamada *Game Manager.* Es un gestor del juego, de sus comportamientos, y encargado de mantener la consistencia de todo lo que ocurre y unir los módulos cuando lo necesitan. Será el primer punto que detallaré (5.3.1).

Por último, en la parte que atañe a los puzles, hay que mencionar el sistema que hace que funcionen los comandos en el juego, piedra angular del proyecto, pues es el mecanismo para resolver los puzles y avanzar. Éste será el otro punto que entraré a describir (5.3.2).

Para suplir la falta de detalle en algunos puntos, debido al mencionado acuerdo de confidencialidad, y ante la dificultad de realizar un diagrama de clases sencillo por cómo funciona Unity en sí mismo y la forma en que se relacionan los elementos en el juego, quiero dar una idea del volumen de trabajo realizado mediante una relación de todos los scripts desarrollados, que muestro en la Tabla 3.

- *GameManager.cs*
- *InterfaceManager.cs*
- **Patrón Command**
  - *PuzzleInputHandler.cs*
  - *Command.cs*
  - **Comandos**
    - *Ascend.cs*
    - *Descend.cs*
    - *Grab.cs*
    - *Drop.cs*
    - *MoveRight.cs*
    - *MoveLeft.cs*
    - *ClimbDownLeft.cs*
    - *ClimbDownRight.cs*
    - *ClimbUpLeft.cs*
    - *ClimbUpRight.cs*
  - **Receivers**
    - *AscendReceiver.cs*
    - *DescendReceiver.cs*
    - *GrabReceiver.cs*
    - *DropReceiver.cs*
    - *MoveRightReceiver.cs*
    - *MoveLeftReceiver.cs*
    - *ClimbDownLeftReceiver.cs*
    - *ClimbDownRightReceiver.cs*
    - *ClimbUpLeftReceiver.cs*
    - *ClimbUpRightReceiver.cs*
- **Comportamiento del mundo**
  - *LiftInteraction.cs*
  - *NpcInteraction.cs*
- **Control y comportamiento de los puzles**
  - *CameraHandler.cs*
  - *GridSquare.cs*
  - *PuzzleCharacter.cs*

*Tabla 3 - Relación de scritps desarrollados*

### 5.3.1 Gestor del juego (Game Manager)

Los juegos suelen estar formados por varios módulos, varias partes, que realizan funciones diferentes y no tienen por qué conocer de su existencia recíproca. Pero no es raro que distintos de estos módulos necesiten coordinarse para que el juego funcione correctamente.

También es habitual encontrarse con situaciones en las que determinados comportamientos parecen estar demasiado por encima de cualquier entidad del sistema, como el hecho de pausar el juego, y se genera la duda de a quién responsabilizar con una función así.

Todos estos casos los cubre una clase que llamamos gestora del juego, *GameManager*. Y lo más destacable de esta clase es el patrón de diseño que se le aplica: Singleton.

El patrón Singleton se utiliza sobre una clase para asegurar que es creada una y sólo una instancia de dicha clase, y proporcionar un punto de acceso global a ella. Como se muestra en Código 1, se consigue el acceso global y único usando una referencia estática pública y otra privada: cuando el "exterior" intenta acceder a la pública, el método get comprueba que exista un valor no nulo en la privada. Si ese valor existe, se replica a la pública para que sea accedido, y si no existe, caso que se dará en el primer intento de acceso a la clase, se creará una única instancia que quedará almacenada en la referencia privada durante toda la ejecución.

```
using System;
using UnityEngine;
using UnityEngine.SceneManagement;

public class GameManager : MonoBehaviour {

    static private GameManager _instancia;

    static public GameManager instancia {
        // metodo get
        // se ejecuta al acceder por GameManager.instancia
        get {

            if (_instancia == null) {
                // creamos un nuevo objeto llamado "_MiGameManager"
                GameObject go = new GameObject("_MiGameManager");

                // anadimos el script "GameManager" al objeto
                go.AddComponent<GameManager>();

                // guardamos en la instancia el objeto creado
                // debemos guardar el componente ya que _instancia es del tipo GameManager
                _instancia = go.GetComponent<GameManager>();

                // hacemos que el objeto no se elimine al cambiar de escena
                DontDestroyOnLoad(go);
            }

            // si no existia, en este punto ya la habra creado
            return _instancia;
        }

        // metodo set
        // no implementado para no permitir modificar la instancia "GameManager.instancia = x;"
    }
```

*Código 1 - Patrón Singleton en fragmento de GameManager.cs*

Al ser de acceso público, la clase puede encargarse de aspectos muy altos del funcionamiento del juego, que podrán ser solicitados desde cualquier punto. Esto también es útil para almacenar en la clase información que afecta a gran parte del juego y que puede ser accedida por varios agentes. Por ejemplo, aquí guardaremos el número de puzles que ha resuelto el jugador, en vez de guardarlos en una clase para su personaje, porque es un dato accedido habitualmente por entidades que no tienen por qué tener conocimiento del propio personaje (y buscarlo en la escena con los métodos de la API de Unity es más costoso que acceder a la instancia pública del *GameManager*).

Y al garantizarse su unicidad, puede cumplir un doble papel en su gestión. Por un lado, organizará de forma óptima las situaciones en las que varias clases tienen que funcionar de una

manera coordinada para cumplir un propósito; y por otro, podrá contener procedimientos que, de ejecutarse en distintas instancias de la clase, podrían producir incongruencias al ocurrir de forma concurrente (por ejemplo, si accediéramos desde dos instancias a hacer cambios en un archivo a la vez).

La clase *GameManager* se encarga, entre otras cosas, de pausar y reanudar el juego, salir de la aplicación, gestionar las escenas, o cargar y guardar partidas.

### 5.3.2 Lógica para puzles

La parte de la capa de negocio más importante del juego es, sin duda alguna, la que se encarga del funcionamiento de los puzles. Esta parte está formada por un script que coordina lo que ocurre en las escenas de puzle, llamado *PuzzleInputHandler*, del cual podemos ver el fragmento inicial en Código 2; una jerarquía de clases para representar los comandos (que detallaré a continuación); una serie de clases *receiver* para esos comandos; y una clase llamada *GridSquare*, que es la representación de cada una de las celdas que conforman los escenarios de los puzles a resolver.

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using CommandPattern;

public class CommandLimitExceededException : System.Exception { public CommandLimitExceededException () : base () {} }

public class PuzzleInputHandler : MonoBehaviour {

    // public float moveDistance = 2f;

    private List<Command> commands;
    private GameObject objectToMove;
    private IEnumerator coroutine;

    public const float PLAY_LENGTH = 1.0f; //seconds

    void Start() {

        // Busco el personaje del jugador porque será el objeto a mover
        objectToMove = GameObject.FindGameObjectWithTag ("Player");
        // Le asigno como casilla la casilla inicial del puzle
        GameObject[] found = GameObject.FindGameObjectsWithTag ("GridSquare");
        foreach (GameObject go in found) {
            if (go.GetComponent<GridSquare> ().isStart) {
                objectToMove.GetComponent<PuzzleCharacter> ().currentTile = go.GetComponent<GridSquare> ();
            }
        }
        // Creo la lista de comandos
        commands = new List<Command>();
    }

    public void AddCommand (Command c) {
        if (commands.Count < GameManager.instancia.maxMovs) {
            commands.Add (c);
        } else {
            throw new CommandLimitExceededException ();
```

*Código 2 - Fragmento de PuzzleInputHandler.cs*

Exceptuando las clases *PuzzleInputHandler* y *GridSquare*, toda esta sección del programa, hecha para hacer funcionar los comandos, está implementada utilizando el patrón Command.

El patrón Command, que es el mayor cimiento del juego, es un diseño muy útil que permite encapsular diferentes comportamientos en objetos, permitiendo al solicitante y al propio objeto abstraerse del funcionamiento del objeto en concreto.

Para implementarlo, primero creé una clase *Command*. Se trata de una clase abstracta que declara un único método *Execute()*. Para cada comando del juego, una nueva clase heredará de *Command* y redefinirá ese método llamando a su *receiver* particular. Por último, existe una clase

*receiver* para cada comando, que es la que verdaderamente ejecuta las operaciones necesarias, por lo que es la que más complejidad entrama.

Por último, además de la clase abstracta, los comandos concretos, y los *receivers*, el patrón Command tiene un último componente: el *Invoker*. Se suele designar como *invoker* una clase que se destina a la creación y ordenación de comandos concretos, y a ordenar su ejecución. En nuestro caso, el papel de *invoker* es ejercido por la clase que administra las pantallas de puzle, *PuzzleInputHandler*, pues reunir los comandos que el usuario ejecute y pedir que se ejecuten ya es parte de su cometido.

A modo aclarativo, en la Ilustración 16 se muestra un diagrama de clases que representa al núcleo de este patrón Command, la jerarquía de comandos.

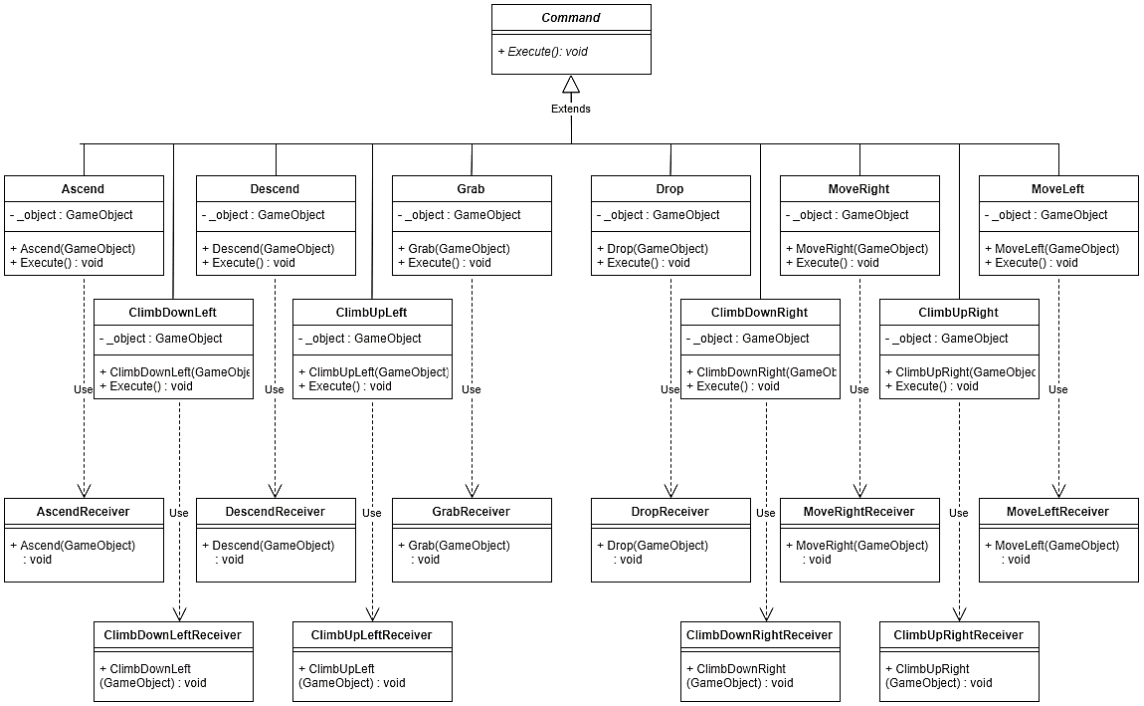


*Ilustración 16 - Diagrama de clases de la jerarquía de comandos*

Una posible implementación usando el patrón Command también suele pasar por utilizar una sola o muy pocas clases *receivers* que reúnan los comportamientos que se necesite ejecutar, pero yo decidí tener una clase para cada comando, porque, aunque esto genera un mayor número de clases, facilita la comprensión individual de cada una al leerlas o depurar su funcionamiento. En Código 3 y Código 4 muestro el comando concreto MoveRight y su *receiver* particular.

La clase que ejerce el papel de *Invoker*, nuestra *PuzzleInputHandler*, cumple su función manteniendo una lista de comandos que conformarán el intento de solución del puzle, y ofreciendo métodos para añadir todos los tipos de comando a dicha lista (cuando el usuario se va a sobrepasar del máximo de comandos permitido, la clase eleva una excepción propia que alerta de ello).

```
using UnityEngine;

namespace CommandPattern
{

    class MoveRight : Command {

        // Atributos necesarios para poder ejecutar el comando
        private MoveRightReceiver _receiver;
        private GameObject _object;

        // Constructor de una instancia del comando
        public MoveRight (GameObject mainObject) {
            this._receiver = new MoveRightReceiver();
            this._object = mainObject;
        }

        // Codigo para ejecutar este comando en concreto
        public override void Execute() {
            // Llamar al RECEIVER
            _receiver.MoveRight(_object);
        }
    }
}
```

*Código 3 - MoveRight.cs*

Cuando se quiere ejecutar la solución, la clase ejecuta la lista de comandos en orden mediante una corrutina, lo que permite utilizar órdenes que ceden la ejecución para detenerse durante cierto tiempo entre comando y comando. Ese tiempo lo define un valor que debe ser la longitud de las animaciones del personaje, para que se mueva entre comandos. Si no hiciéramos estas cesiones de ejecución, la solución se ejecutaría de golpe, y el usuario sólo alcanzaría a ver a su personaje en la posición final de la lista de comandos.

```
namespace CommandPattern
{
    class MoveRightReceiver {

        public void MoveRight(GameObject objectToMove) {

            PuzzleCharacter charScript = objectToMove.GetComponent<PuzzleCharacter> ();
            GridSquare currentPos = charScript.currentTile;
            Dictionary<GridSquare.Directions, GridSquare> neighbourhood = currentPos.getNeighbourhood ();

            GridSquare destination;
            neighbourhood.TryGetValue (GridSquare.Directions.Derecha, out destination);
            if (destination != null && destination.isFloor) {
```

*Código 4 - Fragmento de MoveRightReceiver.cs*

Por último, *GridSquare* es la última de las piezas clave del funcionamiento de los puzles. La clase tiene una serie de variables booleanas que cada instancia tiene marcadas como ciertas o falsas según el tipo de casilla que sea, y un enumerado de direcciones donde cada valor es una de las ocho posiciones adyacentes a la casilla, como en un vecindario Moore (Weisstein, s.f.). También tienen un diccionario donde cada clave es uno de los valores el enumerado, y cada valor la casilla en esa posición, si la hay, de forma que se modela el vecindario de la casilla para ser accedido fácil y rápidamente. Su implementación me recuerda en cierto modo a los problemas de planificación para inteligencias artificiales.

De hecho, los puzles en general recuerdan a situaciones de planificación, siendo condiciones los valores de cada *GridSquare*, operaciones los comandos, y teniendo tan claros los estados inicial (que viene dado) y final (la casilla que tiene el objeto que se debe recoger para solucionar el puzle).

# 6 EVALUACIÓN Y PRUEBAS

## 6.1 PRUEBAS UNITARIAS Y DE INTEGRACIÓN

Las pruebas unitarias son técnicas para comprobar que pequeñas unidades de código funcionan correctamente.

Durante el desarrollo, se ha hecho una aproximación a estas pruebas mediante funciones Log que escriben por consola los valores de variables, de forma que se podía ir comprobando cuándo un valor no era correcto y solucionarlo.

Pero tras la implementación imperan este tipo de pruebas para asegurar que no existen defectos que no hayamos detectado durante el desarrollo.

Para ello he utilizado la herramienta UnityTestTools de Unity. Esta herramienta se importa directamente al proyecto y ofrece una serie de componentes que realizan aserciones de valores en concreto.

Las pruebas con TestTools no son exactamente unitarias, aunque podamos querer considerarlas así, puesto que en sistemas como Unity es casi imposible aislar unidades de código para hacer pruebas unitarias sobre ellas. Por eso las mencionamos unidas a las pruebas de integración, entendiendo que el uso de estos componentes asertivos se encarga de unas pruebas de integración a un nivel más profundo de detalle, cercano a la unidad, pero sin llegar a ser pruebas unitarias.

Las pruebas de integración se entienden como pruebas para asegurarnos de que diferentes módulos creados se comportan de la manera correcta cuando son puestos a funcionar juntos, en una situación más cercana al juego real.

Por las características de Unity, se puede asumir que cada vez que pulsamos Play en el editor estamos ejecutando una prueba de integración, pues, por cómo está diseñado, permite la ejecución de escenas rápidamente para utilizarlas en una situación muy cercana a la de juego. De esta manera, se pueden probar todos los elementos desarrollados a medida que se van generando, y detectar cualquier error en ellos rápidamente, agilizando las iteraciones del desarrollo.

Tras la implementación, para realizar pruebas de integración definitivas, y verificar que todos los componentes interactúan de la forma apropiada entre sí, se ejecutó el juego en el editor del motor y se fue recorriendo escena a escena utilizando los mecanismos del juego. Cuando se comprobó que toda la navegación de podía hacer de forma satisfactoria, se dieron por aprobadas las pruebas de integración.

## 6.2 PRUEBAS DE SISTEMA

Para verificar el comportamiento del sistema en su conjunto, se realizaron las pruebas de sistema de diversos tipos.

### 6.2.1 Pruebas de usabilidad

Teníamos dos requisitos de usabilidad definidos.

El primero de ellos requería que el contenido del juego fuera apto y suficientemente sencillo para todos los públicos. Como esto es difícil de determinar objetivamente sin pasar por un órgano de calificación por edades para este tipo de productos, sometí el resultado de la prueba al juicio del codirector del proyecto, de la empresa Creative Rainbow, que había llevado a cabo con anterioridad una serie de estudios de mercado y públicos objetivos, y tenía conocimiento del tipo de contenido adecuado para ciertas franjas de edad. Por otro lado, como casi la totalidad del contenido gráfico era decidido y suministrado por la mencionada empresa codirectora del proyecto, puedo considerar que este requisito estuvo controlado desde el comienzo.

El segundo requisito consistía en que la interfaz gráfica de usuario fuera clara y fácil de entender. Aunque en primera instancia parecía satisfacer las condiciones, se sometió a un control encuestando a usuarios acerca de los posibles significados de ciertos botones. El resultado fue parcialmente satisfactorio. Se encontró que la mayoría de símbolos y botones eran autoexplicativos, pero dos botones en concreto (los utilizados para coger y soltar objetos en los puzles) inducían a error. En vez del binomio Coger/Soltar, se confundían con Saltar/Agacharse o Entrar/Salir.

Por tanto, se determinó cambiar los iconos por otro más esclarecedores.

### 6.2.2 Pruebas de portabilidad

Los requisitos no funcionales de portabilidad del sistema especificaban que el juego debía poder ser ejecutado tanto en versiones de escritorio de los sistemas operativos principales como en versión WebGL para navegadores web que soporten esa tecnología.

Para verificar el cumplimiento de estos requisitos, la prueba se ha reducido a generar entregas finales para las plataformas deseadas y comprobar que la ejecución era correcta.

### 6.2.3 Pruebas de rendimiento

Como requisito de rendimiento teníamos marcado que el juego no bajase de los 30 *frames* (imágenes) por segundo, o fps. Esta cantidad suele ser un estándar en el sector del videojuego; a pesar de que con menos imágenes por segundo también se puede crear sensación de movimiento para el ojo humano (el cine tradicional trabaja a 24 *fps*), subir hasta 30 proporciona una experiencia más natural y suave a la hora de jugar.

Hay que reseñar que no conviene limitarse a 30 *fps*. Las nuevas generaciones de juegos, ejecutados en máquinas capaces de ello, utilizan tasas de *frames* de 60 o superiores, y ello sólo conlleva mejoras a la experiencia del usuario que interactúa con el juego. Por eso, aunque es posible, no bloquearemos la tasa de *frames* cuando alcancemos 30, sino que dejaremos que se refresque a la máxima velocidad posible.

Para evaluar el rendimiento del juego, Unity proporciona una herramienta llamada *profiler*, que monitorea el uso de recursos de la máquina y otras medidas como la tasa de refresco de los *frames*, durante la ejecución de los juegos, para darnos una imagen del rendimiento del sistema.

Se realizó una ejecución del juego en su versión de escritorio de Windows con el *profiler* para obtener todas las medidas, que se muestran en la Ilustración 17.

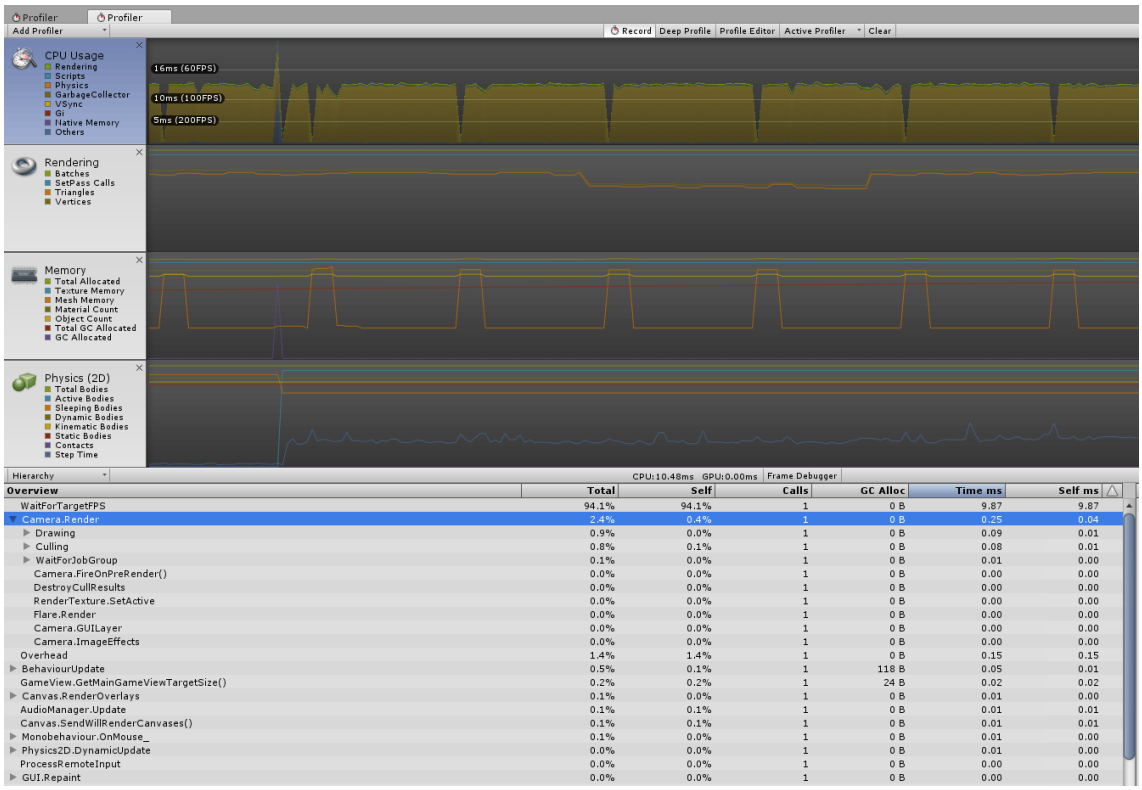


*Ilustración 17 - Captura con el profiler de Unity*

Lo primero que vemos en la captura del *profiler* es que nuestra tasa de refresco de pantalla se mantiene generalmente entre los 10 y los 16 milisegundos, lo que significa que nuestro *framerate* es de entre 100 y 60 FPS. Esto no solo cumple el requisito impuesto durante la fase de análisis, sino que garantiza una buena experiencia del usuario, al menos en lo que a pantalla se refiere.

También observamos en la captura, marcado el *profiling* de CPU y mirando en el detalle de la parte inferior de la pantalla, que se nos muestra qué procesos consumen más tiempo de CPU. Aunque el primero es significativamente mayor, debemos ignorarlo, pues WaitForTargetFPS se dedica a hacer esperar a la CPU cuando sobrepasa un límite de *frames* por segundo, para hacer coincidir las tasas de refresco del juego con las del monitor en las que se ejecute. Así, después de él, el proceso que más ocupa la CPU es el renderizado de la cámara (como es lógico). Concretamente, los dos procedimientos más pesados son el dibujado de las texturas (imágenes) y el *culling* de la geometría (el *culling* es un proceso del cálculo gráfico por el que se determina qué caras de un objeto están visibles a la cámara, para decidir si calcular lo necesario para renderizarlas o no).

El resto de métricas del *profiler* no son demasiado reseñables, porque están dentro de los valores normales: no hay nada que produzca problemas de rendimiento en el juego.

A destacar, solamente, en la gráfica de memoria, la línea morada de la que sólo vemos dos picos. Indica asignaciones al recolector de basura de C#. El hecho de que sean tan poco habituales en la gráfica de tiempo indica que el código hace una buena gestión de las referencias a objetos, sin desperdiciar mucha memoria.

### 6.2.4 Pruebas de regresión

En esencia, cada vez que se detectaba un defecto en el juego y se hacían los cambios pertinentes para solucionarlo, se ejecutaban una serie de pruebas de regresión, que consistían en volver a ejecutar pruebas de integración sobre las partes afectadas por el defecto, para constatar que la solución no hubiera estropeado alguna parte del resto del sistema.

## 6.3 PRUEBAS DE ACEPTACIÓN

Ya con el sistema completo y finalizado, se procedió a ejecutar las pruebas de aceptación. El objetivo no es ya descubrir defectos en el software, sino que el cliente validase el sistema.

Dada mi situación, realizando este proyecto bajo la codirección de la empresa Creative Rainbow, la aceptación de mi trabajo estaba sujeta a ellos.

Para realizarlas, primero realicé una primera validación pidiendo a un grupo de usuarios que jugase al juego en su estado final. Después de que jugasen, pedí sus opiniones y evalué su experiencia, por si alguien notificaba algún defecto que no se hubiera detectado hasta entonces. Al no detectar incidencias mayores, asumí que el juego estaba en un estado cercano al final.

Tras eso, le llevé al director de la empresa una entrega del juego para que pudiera ejecutarlo por sí mismo, y me mantuve presente durante la sesión de prueba para responder a cualquier duda o problema que pudiera surgir.

Una vez el cliente estuvo satisfecho con las pruebas realizadas y dio por concluida la sesión, se puede considerar que el sistema ha sido aceptado por el cliente y que ha finalizado la etapa de desarrollo.

# 7 CONCLUSIONES Y TRABAJOS FUTUROS

Llegado a este punto, donde doy por finalizado el trabajo que nos ocupa, me veo en la tesitura de recapacitar con respecto a él. Es momento de valorar aquello que he hecho, preguntarse qué quedó por hacer, y pensar en lo que podría llegar a hacerse.

## 7.1 CONCLUSIONES

Para hablar de conclusiones, debería, en primer lugar, recordar que mi objetivo principal con este proyecto era la creación de un *serious-game* para enseñar programación a niños, un juego con el que pudieran aprender los fundamentos del razonamiento computacional mientras se divertían.

A la vista del resultado final, puedo considerar cumplido el objetivo, satisfactoriamente. Si bien es cierto que el juego no tiene la profundidad necesaria para que alguien aprenda todos los aspectos de la programación, sí que tiene la capacidad de introducir a los jóvenes en los fundamentos del mencionado razonamiento computacional, de mostrarles la utilidad de la lógica y hacer ver que existen diferentes maneras de pensar y enfocar los problemas.

Finalizo este proyecto, además, con la esperanza de que sirva de base para un objetivo mayor, para que siga creciendo y acabe cumpliendo las expectativas más altas que se tienen para él.

Espero que mi aporte ayude a que se consiga un juego con mayor alcance del que yo podría conseguir, que acerque a los jóvenes la programación en su completitud.

Probablemente, la parte de la que más orgulloso me siento de todo el proyecto es su arquitectura y diseño. He conseguido crear un sistema muy modular y extensible, al que es fácil seguir añadiendo nuevos comandos, gracias a una eficaz implementación del patrón Command; y cuyos puzles tienen unas reglas muy determinadas (no sólo para resolverse, sino también para crearse) por lo que es igual de fácil generar nuevos puzles, o incluso entrenar a un ordenador para que los generase.

Que el sistema sea tan escalable es probablemente un punto a favor de que de él pueda surgir un proyecto mayor que se acabe convirtiendo en un gran juego para aprender a programar.

Tras la realización del trabajo, reconozco que me he dado cuenta del gran esfuerzo que tiene detrás un proyecto software, y de la importancia que tiene en ellos la figura del ingeniero de software. Una metodología de trabajo definida puede facilitar y agilizar mucho todo el proceso. Eso es algo que me quedo como un valor añadido.

Por último, en el ámbito personal, necesito expresar el enriquecimiento que me ha supuesto el proyecto. Mi aspiración es hacer carrera profesional de este tema, los videojuegos. Y mi sensación es que este trabajo me ha ayudado a aprender mucho al respecto. No sólo adquiriendo experiencia con herramientas de la industria del videojuego como Unity, además de mejorar mi habilidad con el lenguaje C#, sino también empapándome de otros elementos distintivos como flujos de trabajo, técnicas específicas, formatos habituales…

Igualmente, también siento que he crecido como programador en términos generales. Mi manera de enfocar ciertos escollos, mi técnica de depurado de errores, mi capacidad de diseño… han mejorado enormemente. Por ejemplo, considero muy valioso mi aprendizaje al respecto de patrones de diseño en programación, a los cuáles no era ajeno, pero de los que tampoco me sentía muy cercano.

En líneas generales, creo que el proyecto ha sido finalizado satisfactoriamente, cumpliendo con todos los objetivos y requisitos propuestos, y yo me he desarrollado como informático, por lo que mis conclusiones al respecto de este trabajo difícilmente podrían ser mejores.

## 7.2 TRABAJOS FUTUROS

A pesar de que las conclusiones en relación con el proyecto son positivas, cabe admitir que existen diversas líneas de mejora del juego en su estado actual.

La primera de las posibles mejoras es el desarrollo de una versión del juego para dispositivos móviles, Android e iOS (con preferencia sobre Android por ser un sistema con mayor cuota de mercado y más accesible). Las versiones para escritorio y web son una buena opción para una primera implantación, pero si se quiere conseguir la máxima difusión posible, es mandatorio dar el salto a otras plataformas, y eso pasa por el mercado de los smartphones, innegablemente masivo.

Otra de las líneas de trabajo abiertas, bastante obvia pero alejada de la programación en sí misma, es la mejora estética de la interfaz. Hacerla más atractiva ayudaría a la imagen general

del juego. Además, debería ser accesible a personas con defectos en la visión, como el daltonismo o la vista cansada.

Siguiendo el tema de la interfaz gráfica, considero que sería una buena idea añadir un módulo de breves explicaciones acerca del juego. Es cierto que el público al que se orienta el juego es parte de la generación que llamamos “nativa digital” – por buenas razones –, y que no es descabellado esperar que sean capaces de averiguar los controles de un juego o identificar su objetivo con relativa facilidad; pero sería un error conformarse con que esa base de usuarios entienda los propósitos del juego, mientras dificultamos su uso por parte de otros usuarios con perfiles distintos. Por eso, creo que sería de agradecer un apartado que explicase brevemente el objetivo de los puzles, los controles, o el significado de cada comando, por si los botones no fueran suficientemente ilustrativos.

Por último, el avance más ambicioso sería usar este juego como cimiento de un juego más grande, donde cada nivel sea equivalente al volumen de trabajo que yo he realizado, y al avanzar se fueran incorporando nuevas mecánicas. Por ejemplo, introducir el uso de funciones, o sentencias condicionales. El diseño del estado actual del juego se hizo teniendo en cuenta una posible escalabilidad de este tipo en el futuro.

Dentro de ese punto, uno de los aspectos más atractivos para mí sería implementar una sección para prácticas compuesta de niveles que se pudieran generar de forma procedural. La modularidad con la que están diseñados los puzles facilitaría la adaptación de éstos a un formato más automatizado. Y ese cambio serviría para que los usuarios pudieran resolver una cantidad de puzles mayor, diferente de los que conformarían el progreso principal en el juego.

# 8 REFERENCIAS


Abt, C. C. (1970). *Serious Games.* EE.UU.: Viking Press. Recuperado el Junio de 2017

Banks, K. (26 de 11 de 2016). *Unity3D: Panning and Zooming (pinch-to-zoom) Your Camera With Touch and Mouse Input*. Recuperado el Junio de 2017, de Kyle Banks: https://kylewbanks.com/blog/unity3d-panning-and-pinch-to-zoom-camera-with-touch-and-mouse-input

CODE.org. (2017). Recuperado el Junio de 2017, de Code.org: https://code.org/

Djaouti, D., Alvarez, J., & Jessel, J. P. (2011). Classifying Serious Games: the G/P/S Model. En P. Felicia (Ed.), *Handbook of research on improving learning and motivation through educational games: Multidisciplinary approaches* (Vol. I, págs. 118-136). Recuperado el Junio de 2017

Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (2003). *Patrones de diseño.* Addison-Wesley. Recuperado el Junio de 2017

ISO25000. (2017). *ISO/IEC 25010*. Recuperado el Junio de 2017, de ISO25000: http://iso25000.com/index.php/normas-iso-25000/iso-25010

Kenney Assets. (2010-2017). *Assets: Kenney*. Recuperado el Junio de 2017, de Kenney.nl: http://kenney.nl/assets

MIT Scratch Team. (s.f.). *Acerca de Scratch*. Recuperado el Junio de 2017, de https://scratch.mit.edu/about

Nguyen, T. (5 de Diciembre de 2016). Serious games. *Significance, XIII*(6), 14-19. Recuperado el Junio de 2017

Nystrom, R. (2014). *Game Programming Patterns.* EE. UU.: Genever Benning. Recuperado el Junio de 2017, de http://gameprogrammingpatterns.com/contents.html

Nystrom, R. (2014). *Game Programming Patterns: Game Loop*. Recuperado el Junio de 2017, de Game Programming Patterns: http://gameprogrammingpatterns.com/game-loop.html

Pantaleo, G., & Rinaudo, L. (2014). *Ingeniería de Software.* Alfaomega. Recuperado el Junio de 2017

Sawyer, B., & Rejeski, D. (2002). *Serious games: Improving public policy through game-based learning and simulation.* Recuperado el Junio de 2017

Sommerville, I. (2005). *Ingeniería del software.* Pearson Educación. Recuperado el Junio de 2017

Telefónica Educación Digital. (2017). Análisis y Tipología de la Diversión. *Introducción al Diseño de Videojuegos(03)*. España. Recuperado el Junio de 2017

Telefónica Educación Digital. (2017). Dinámicas de juego. *Introducción al Diseño de Videojuegos(06)*. España. Recuperado el Junio de 2017

Telefónica Educación Digital. (2017). El concepto y la experiencia del juego. *Introducción al Diseño de Videojuegos(4)*. España. Recuperado el Junio de 2017

Telefónica Educación Digital. (2017). Mecánicas de juego. *Introducción al Diseño de Videojuegos(5)*. España. Recuperado el Junio de 2017

Unity Technologies. (2016). *Unity Remote 4.* Recuperado el Junio de 2017, de Unity Documentation: https://docs.unity3d.com/es/current/Manual/UnityRemote4Android.html

Unity Technologies. (2017). *Editor.* Recuperado el Junio de 2017, de https://unity3d.com/es/unity/editor

Unity Technologies. (2017). *GameObject*. Recuperado el Junio de 2017, de Unity Manual: https://docs.unity3d.com/Manual/class-GameObject.html

Unity Technologies. (2017). *MonoBehaviour*. Recuperado el Junio de 2017, de Unity Scripting API: https://docs.unity3d.com/ScriptReference/MonoBehaviour.html

Unity Technologies. (2017). *Prefabs*. Recuperado el Junio de 2017, de Unity Manual: https://docs.unity3d.com/Manual/Prefabs.html

Unity Technologies. (2017). *Scenes*. Recuperado el Junio de 2017, de Unity Manual: https://docs.unity3d.com/Manual/CreatingScenes.html

Unity Technologies. (2017). *Using Components*. Recuperado el Junio de 2017, de Unity Manual: https://docs.unity3d.com/Manual/UsingComponents.html

Unity Technologies. (2017). *Variables and the Inspector*. Recuperado el Junio de 2017, de Unity Manual: https://docs.unity3d.com/Manual/VariablesAndTheInspector.html

Weisstein, E. W. (s.f.). *Moore Neighborhood*. Recuperado el Junio de 2017, de MathWorld - A Wolfram Web Resource: http://mathworld.wolfram.com/MooreNeighborhood.html

Wing, J. M. (March de 2006). Computational Thinking. *Communications of the ACM, XLIX*(3), 33-35. Recuperado el Junio de 2017