VIDEOJUEGOS PARA APRENDER A PROGRAMAR VIDEOJUEGOS

MEMORIA DEL PROYECTO

Laura María de Castro Saturio
Samuel García Segador
Mariano Hernández García

Dirigido por Guillermo Jiménez Díaz

Trabajo de fin de grado en Ingeniería Informática
Departamento de Ingeniería del Software e Inteligencia Artificial
Facultad de Informática
Universidad Complutense de Madrid

Junio 2015

VIDEOJUEGOS PARA APRENDER A PROGRAMAR VIDEOJUEGOS

Memoria del proyecto

06/15

Departamento de Ingeniería del Software e Inteligencia Artificial
Facultad de Informática
Universidad Complutense de Madrid

Junio 2015

Autorización

Nosotros, Laura María de Castro Saturio, Samuel García Segador y Mariano Hernández García, alumnos matriculados en la asignatura Trabajo de Final de Grado (TFG) en la Universidad Complutense de Madrid durante el curso 2014/2015; dirigidos por Guillermo Jiménez Díaz, autorizamos la difusión y utilización con fines académicos, no comerciales, y mencionando expresamente a sus autores del contenido de esta memoria, el código, la documentación adicional y el prototipo desarrollado.

Laura María de Castro Saturio

Samuel García Segador

Mariano Hernández García

Agradecimientos

En primer lugar, queremos agradecer a nuestro director de proyecto, Guillermo Jiménez Díaz, la oportunidad que nos ha dado para adentrarnos en la programación de videojuegos aceptando el proyecto que nosotros le propusimos.

No ha sido fácil avanzar en el desarrollo del videojuego, pero Guille siempre ha estado ahí proporcionándonos información y soluciones varias para salir de los aprietos. Nunca nos ha puesto la solución delante de nosotros, sino que nos ha guiado para que la encontráramos por nosotros mismos, cosa que nos ha gustado sobremanera. Nos hemos reunido con él siempre que lo hemos necesitado y siempre nos ha dado su punto de vista de forma sincera. Gracias Guille.

Han pasado cuatro/cinco años desde que entramos por esas puertas moradas que han cambiado totalmente nuestra manera de pensar. A lo largo de estos años hemos ido conociendo gente que, poco a poco, ha ido cambiando nuestra mentalidad y nuestra capacidad de expresarnos mejor con el resto de compañeros. Hemos tenido profesores buenos y otros no tan buenos pero, gracias a todos ellos, estamos hoy aquí. Nos han ayudado a madurar y nos han enseñado a buscarnos la vida por nuestra cuenta.

Durante este periodo hemos conocido a muchos compañeros. A algunos los conocimos el primer día al entrar en el aula, mientras que a otros después de varios años en la facultad. Queremos agradecer en especial a Eddy, Jorge, Manu, Omar, Stephanía y a todos aquellos que, por unas circunstancias u otras perdimos el contacto, todo lo que nos han enseñado, los momentos que hemos tenido con cada uno, los grandes días que pasamos juntos y los que nos quedan por pasar. Muchas gracias chicos. Ya fuera del ambiente de nuestra facultad, queremos agradecer a nuestros amigos y amigas todos los años de apoyo y diversión que nos han dado. Siempre han estado allí y siempre estaréis.

Por último, aunque no por ello menos importante, queremos agradecer a nuestras familias la oportunidad que nos han dado para estudiar lo que realmente nos gusta. Gracias a ellos estamos a punto de terminar un ciclo que nos permitirá dedicarnos a trabajar en lo que más amamos. No hay palabras en el mundo suficientes para agradecerles todo lo que han hecho, hacen y harán por nosotros.

Muchas gracias a todos. Laura, Samuel & Mariano.

Resumen

Desde siempre nos han gustado los videojuegos. Actualmente forman parte de la cultura de un país, como pueden ser el cine o el teatro y desde hace unos años es posible ganarse la vida desarrollando videojuegos.

El objetivo de este proyecto es desarrollar una herramienta que enseñe a programar un videojuego. La particularidad de esta herramienta es que será un videojuego, es decir, desarrollaremos un videojuego que enseñará las bases de la programación de videojuegos. En concreto, llevaremos a cabo esta tarea utilizando el *framework* de desarrollo de videojuegos llamado *Phaser*.

No sólo desarrollaremos dicho videojuego, sino que además tendremos que enfrentarnos a los problemas que plantea ejecutar en tiempo real el código que el jugador escriba dentro de nuestra herramienta. También necesitaremos verificar si el código escrito por el usuario es correcto, darle ayudas en caso de que sea erróneo y mostrarle la documentación del *framework* que hemos utilizado. Además, queremos validar con usuarios reales el prototipo que entreguemos.

Palabras clave: *framework, JavaScript, HTML5, Phaser,* lenguaje de programación.

Abstract

We have always been passionate about videogames. Nowadays they form part of the country's culture as the cinema or the theater and for some years ago it's possible to work in developing videogames.

The goal of this project is the development of a tool to teach how to program a videogame. The point of this tool is that it will be a videogame, that is to say, we will develop a videogame that will teach the bases of videogame programming. Concretely, we will use for this purpose the videogame developer *framework* called *Phaser*.

We do not only develop this game, but also we will have to deal with some problems as inserting user's code within the videogame. Also we will need to check if the code written by the user is correct, give him/her some tips in case the code will be wrong and show him/her the *frameworks'* documentation that we have used. Finally, we will want to validate with real users our prototype.

Keywords: *framework, JavaScript, HTML5, Phaser,* program language

Índice de contenidos

Índice de figuras

Índice de tablas

1. Motivación y objetivos

Desde siempre nos han apasionado los videojuegos. Pensamos que forman parte de la cultura de un país del mismo modo que el cine o el teatro. Forman parte de nuestra vida diaria y, además, son una forma muy agradable de enseñar nuevos contenidos. Desde hace un tiempo hemos estado pensando en realizar un videojuego, pero viendo la situación actual del mundo de los videojuegos y que últimamente es cada vez más normal que los videojuegos formen parte de nuestro aprendizaje en algunas materias ¿Por qué no intentar hacer un videojuego que enseñe programar a la gente? Por ello, hemos decidido enseñar la programación de un videojuego en *HTML5* [1] y *JavaScript* [2] utilizando como medio un videojuego en sí mismo.

Actualmente, los equipos de personas encargadas de desarrollar un videojuego no tienen que hacer todo ellos, sino que existen *frameworks*[1], motores gráficos, motores de físicas y otras herramientas que facilitan esta tarea. Como ejemplos de motores de desarrollo de videojuegos tenemos *Unity 3D* o *Unreal Engine*. Es por ello por lo que hemos decidido escoger entre los muchos *frameworks* de desarrollo de videojuegos que existen, uno que se adapte a las necesidades de nuestro trabajo.

Nuestro objetivo es desarrollar una herramienta que enseñe a programar videojuegos utilizando un *framework* de desarrollo de videojuegos en *HTML5*. La particularidad de esta herramienta es que enseñará a partir de un videojuego en el que el jugador deberá programar ciertos aspectos del mismo para poder completarlo.

Pasemos ahora a definir de una forma más detallada los objetivos del proyecto y cómo se ha organizado la memoria.

[1] Estructura conceptual y tecnológica que puede servir de base para la organización y desarrollo de software.

1.1 Objetivos

El objetivo general del proyecto es enseñar los conceptos básicos de la programación de videojuegos mediante el uso de un videojuego. Más concretamente, vamos a enseñar la programación de un videojuego en *HTML5* [3] y *JavaScript* [4] por medio del *framework Phaser* [5].

Para conseguir este propósito, presentamos al usuario un videojuego en el que él mismo deberá programar ciertos aspectos del juego, tales como: física del mundo, movimientos del jugador o comportamiento de los enemigos, entre otras mecánicas. En resumen, el trabajo ha consistido en realizar estas actividades:

- Buscar un *framework* de desarrollo de videojuegos que se adapte a nuestras necesidades.

- Aprender cuál es la mejor forma de enseñar a programar videojuegos.

- Entender la arquitectura de un videojuego, es decir, entender los distintos módulos que conforman este tipo de arquitecturas.

- Desarrollar el videojuego que vamos a utilizar como medio para enseñar dentro de nuestra herramienta. En primera instancia, desarrollar el juego completo y, más tarde, “trocearlo” en función de la dificultad de la programación para que el jugador programe por sí mismo los aspectos anteriormente mencionados.

- Desarrollar e implementar la interfaz de nuestra herramienta, que integrará el videojuego y la interacción con el jugador.

- Realizar evaluaciones sobre la experiencia de uso de la herramienta de enseñanza desarrollada.

1.2 Organización de la memoria

La memoria se halla organizada de la siguiente forma:

- En el primer capítulo se han expuesto la motivación y los objetivos del proyecto.

- En el segundo capítulo hablamos del estado del arte. Hemos considerado necesario realizar un estado del arte sobre los entornos de enseñanza en la programación porque necesitamos aprender la mejor forma de enseñar a programar. Toda nuestra vida hemos sido alumnos y, aunque en alguna ocasión hemos dado clase a compañeros, amigos o familiares; es cierto que esta es la primera vez que nos enfrentamos a algo así.

- El tercer capítulo versa sobre la estructura general de un videojuego y las arquitecturas más extendidas para el desarrollo de videojuegos. También hemos analizado algunos de los motores de desarrollo de videojuegos en *HTML5* y hemos elegido el que usaremos para desarrollar nuestro videojuego.

- En el cuarto capítulo hablamos sobre los módulos que componen nuestro prototipo y por qué los necesitamos. No nos hemos centrado en cuestiones técnicas, sino que nos hemos centrado en su diseño.

- En el quinto capítulo nos hemos centrado en la implementación de los módulos que componen nuestra aplicación: la interfaz de usuario, el sistema de diálogos, el editor de código, los test de unidad, la documentación y la implementación del videojuego.

- Desde un principio hemos querido evaluar el prototipo que hemos diseñado para comprobar que carezca de errores y fallos de diseño. En el sexto capítulo hablamos sobre la evaluación que hemos realizado, así como de los resultados y conclusiones que hemos obtenido de la misma.

- En el séptimo capítulo aunamos las conclusiones finales y miramos al futuro pensando qué podríamos hacer para mejorar nuestro proyecto.

- En el octavo capítulo contamos cómo nos hemos organizado para realizar el proyecto y cada uno de nosotros hemos contado nuestra contribución personal al mismo.

- En el noveno capítulo están recogidos los anexos.

- Por último, el décimo capitulo recoge la bibliografía que hemos utilizado.

Una vez concluidas la motivación, objetivos y organización de la memoria pasamos al Estado del arte.

1.3 Motivation and goals

We have always been passionate about videogames. We think that they are part of the country's culture as the cinema or the theater. They are part of our daily life and, besides, they are a nice way to learn new contents. For a long time we have been thinking about developing a game, but seeing the current situation in the world of videogames, and the fact that nowadays is more regular for games take part in our learning, Why not developing a game that teaches how to program? That is why we have decide to teach how to program a game using *HTML5* and *JavaScript* using a game itself for this purpose.

Nowadays, the teams of people attendant to develop a videogame don't have to do it all; there are frameworks, graphic engines, physics engines, and other tools that ease this task. As examples of game engines we have *Unity 3D* or *Unreal Engine*. That is why we have decided to choose between a lots of videogame developer *frameworks*, one that suit to our work.

Our goal is the development of a tool for teaching programming a videogame using a *HTML5* videogame developer *framework*. The point of this tool is that it will teach using a videogame in which the player must programming some aspects of the own game to complete it.

Next, we are going to explain in a more detailed way the goals of this project and how we have organized this final degree project.

1.3.1 Goals

The general goal of this project is teaching the basic concepts about programming a videogame using the own game. More precisely, we are going to teaching how to program a game in *HTML5* and *JavaScript* using the *Phaser* framework.

To get this purpose, we introduce to the user a videogame in which he/she must program some aspects of the game, as the physics world, player moves or enemies behaviors, among other mechanics. The project has consisted in the following activities:

- Finding a videogame developer framework that suits to our needs.
- Deciding the best form to teach how to program videogames.
- Understanding the videogame architecture what means understanding the different modules that shapes this kind of architectures.
- Developing the videogame that we will use as the way to teach in our tool. Firstly, developing the whole videogame, and later, divide it in some parts in function the difficulty of the programming. Then, the user could program by him/herself the aspects that we have mentioned.
- Developing and implementing the interface of our tool; it will include the game and the interaction with the user.
- Performing assessments about the experience of using the teaching tool developed.

1.3.2 Project organization

The project is organized by the following form:

- In the first chapter we explain the motivation and goals of the project.
- In the second chapter we talk about the state of the art. We have considered necessary do a state of the art about the teach environments in programming because we need to learn the best way to teach programming. All our lives we have been students and, although some occasions we've given class to classmates, friends or family, is true that this is the first time that we have to deal with something like that.
- In the third chapter we explain the videogame general structure and the architectures more extended to the videogame development. Also we have analyzed some videogame development engines in *HTML5* and we have chosen one for using it to develop our game.

- In the fourth chapter we talk about the modules that compose our prototype and why we need them. We haven't focused in technical issue; instead of that, we have focused in its design.

- In the fifth chapter we have focused in the implementation of the modules that compose our application: the user interface, the dialogs system, the code editor, the unit tests, the documentation and the videogame implementation.

- From the beginning we have wanted to evaluate the prototype that we have designed to check that if it has got some errors and design flaws. In the sixth chapter we talk about the evaluation that we have done, as the results and conclusions that we have obtain from it.

- In the seventh chapter, we add the final conclusions and we think about the future and what could do to improve our project.

- In the eighth chapter we speak about how we have organized to make the project and each of us have tell our contribution to it.

- In the ninth chapter are the annexes.

- Finally, the tenth chapter has the bibliography that we have use.

Once finished the motivation, goals, and the project organization, we go to the State of the art.

2. Estado del arte

En este apartado vamos a realizar un estado del arte sobre las herramientas que existen actualmente para enseñar a programar. La mayor parte de ellas están centradas en la programación de videojuegos, pero también hemos considerado acertado incluir alguna que enseñe a programar en general, como es el caso de *Scratch* o *KhanAcademy.*

Comenzaremos analizando las características principales de los entornos de enseñanza de programación que más nos han llamado la atención. Luego condensaremos los resultados en una tabla comparativa. Y, por último, obtendremos una serie de conclusiones. El resultado de este apartado nos dará una idea de los métodos que existen actualmente y nos ayudará a elegir el método que queremos aplicar nosotros en nuestro proyecto.

2.1 Entornos de enseñanza de programación

En primer lugar, vamos a repasar brevemente las características principales de alguno de los *frameworks* de desarrollo de videojuegos en *HTML5* de la actualidad. Nos hemos centrado especialmente en los dirigidos a los videojuegos, como *Alice 3D*, *CodeSpells*, *CodeAvengers*, *Light Bot* y *Code Combat*. Por otro lado, también hemos analizado *Scratch* dado que es muy conocido y su uso es muy extendido; y *KhanAcademy* porque enseña de una manera algo diferente a los demás, como veremos a continuación.

Nos hemos fijado principalmente en el lenguaje de programación que enseñan, si están o no orientados a la programación de videojuegos, el mecanismo que emplean para enseñar. También hablamos sobre los aspectos que más nos gustan, y los que menos nos gustan, de cada uno de los entornos.

2.1.1 Scratch

Scratch [6] es un entorno de aprendizaje de programación de escritorio dirigido a que personas inexpertas en la programación aprendan a programar de manera sencilla y sin necesidad de aprender la sintaxis de un lenguaje de programación específico.

Con *Scratch* se pueden programar historias interactivas, juegos, animaciones e incluso simulaciones y compartirlas con todo el mundo. Para crear proyectos contamos con un conjunto de instrucciones expresadas en lenguaje natural, clasificadas en distintas categorías (movimiento, apariencia, eventos, control, operadores…).

Como vemos en la Figura 2.1, estas instrucciones se representan con bloques que pueden ser arrastrados y concatenados unos con otros a modo de puzle para crear los programas. Esto es lo que se denomina *scratching.* Además de poder mover las cajas para dar funcionamiento al programa, se pueden modificar valores para adaptarlo a nuestro gusto, esto podremos hacerlo en las cajas que tienen huecos en blanco.

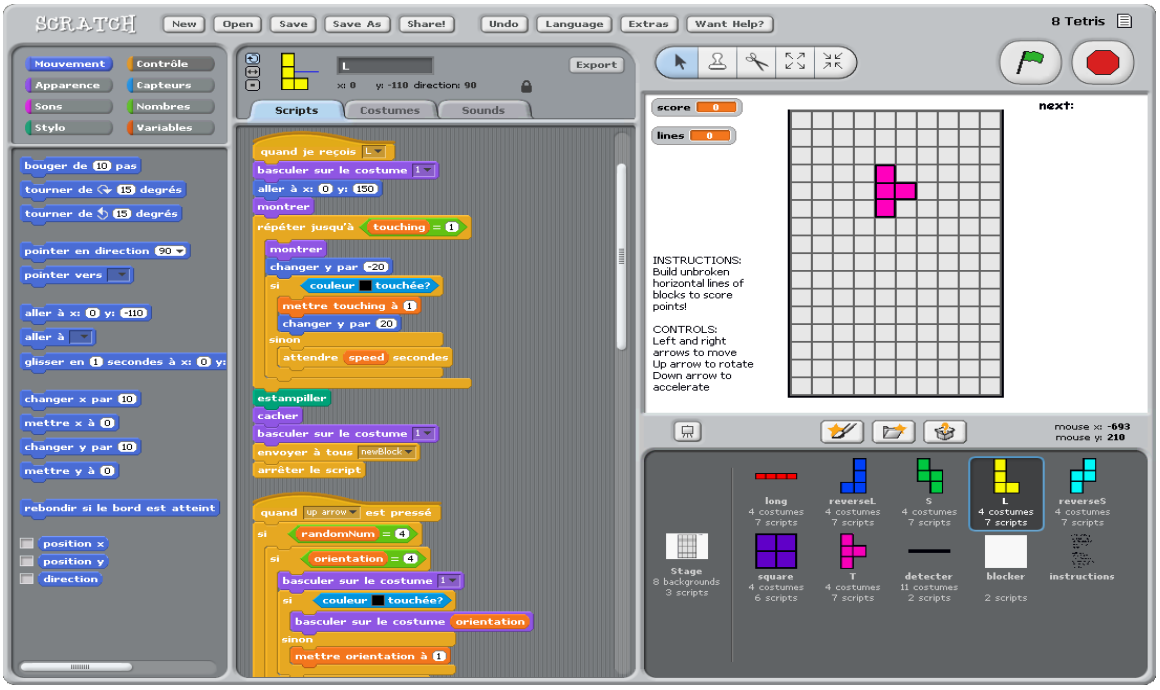


Figura 2.1: Captura de pantalla de Scratch

La manera en la que un usuario interactúa con *Scratch* es la siguiente:

- El usuario programa su código arrastrando las cajas para darle comportamiento.

- Cuando quiere ver visualmente qué es lo que hace el código, pulsará el botón ejecutar (en la imagen podemos ver una banderita verde).

- También puede parar lo que se está ejecutando pulsando sobre el botón parar (en la imagen lo vemos como una señal de tráfico roja)

Estos son los puntos que más nos han gustado de *Scratch*:

- La diversidad que tiene, ya que permite crear gran cantidad de proyectos diferentes.

- La capacidad de personalización que ofrece, permitiendo personalizar los proyectos con fotos, gráficos, música, grabaciones e incluso creando nuevos gráficos.

- Disponible para múltiples sistemas operativos.

Como hemos visto en la Figura 2.1, la interfaz es muy colorida, sencilla e intuitiva, cosa que nos ha gustado bastante. Sin embargo, *Scratch* no es el tipo de herramienta que buscamos por varios motivos:

- Queremos enfocarnos en un estilo de programación más cercano a la realidad, es decir, sin cajas ni pseudolenguaje, sino escribiendo el código.

- Queremos centrarnos en que el usuario aprenda a utilizar un lenguaje de programación real, en concreto *JavaScript*.

- Queremos centrarnos en usuarios con un conocimiento de programación básico. Por ello, consideramos que el método de enseñanza de *Scratch* es infantil y puede aburrir a nuestros usuarios.

Concluimos que *Scratch* es un buen entorno de programación para usuarios totalmente inexpertos. Sin embargo, para usuarios que saben algo de programación puede resultar demasiado lento y, por lo tanto, volverse aburrido. Aunque no nos guste que nuestros usuarios tengan que mover cajas para dar funcionamiento a su programa, nos gusta la idea de que puedan modificar los valores de algunos parámetros dados.

2.1.2 Alice 3D

Alice 3D [7] es un entorno de programación 3D para crear animaciones y juegos interactivos. Es una herramienta gratuita orientada a que estudiantes y programadores principiantes puedan aprender programación orientada a objetos de manera más intuitiva.

Para ello cuenta con una interfaz interactiva en la que puedes arrastrar distintas instrucciones para construir tu programa. Estas instrucciones no están en un lenguaje determinado sino que siguen un estándar común a lenguajes como *Java, C++ o C#.*

En Alice se ofrece al usuario una gran base de datos con funciones, parámetros, variables…para que pueda crear muchas animaciones y juegos distintos. Estas instrucciones se arrastran hacia el programa para crear diferentes secuencias.

La interacción con *Alice 3D* es parecida a la manera en la que lo hacemos con *Scratch*: primero construimos nuestro código y después lo ejecutamos para ver si es el comportamiento deseado. El resultado del código ejecutado se muestra en una ventana aparte a la ventana en la que escribimos el código. Todo esto podemos apreciarlo en la Figura 2.2:

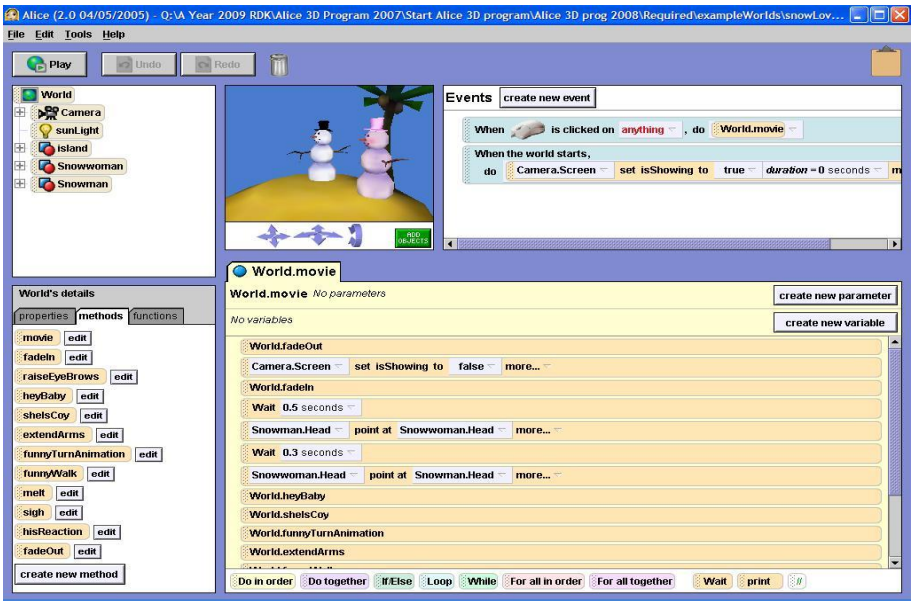


Figura 2.2: Captura de pantalla de Alice 3D

En *Alice 3D* el usuario cuenta con un mundo formado por varios objetos, cuya lógica puede ser programada por el jugador. Nos ha gustado que el lenguaje que se enseña en *Alice 3D* se parezca a un lenguaje de programación real, aunque no sea un lenguaje de los que existen en la actualidad.

Lo que más nos ha gustado es que permite al usuario crear múltiples animaciones y juegos distintos gracias a las numerosas combinaciones que podemos hacer con las variables y funciones que nos ofrece. Además, la interfaz es intuitiva y la curva de aprendizaje resulta sencilla.

Lo que menos nos ha gustado es que no enseña un lenguaje de programación en concreto, sino que utiliza estándares de lenguajes reales. Esto podría ser bueno para hacer que un usuario se familiarizase con las bases y fundamentos de los lenguajes de programación sin centrarse en uno en concreto. Sin embargo, de esta manera se pierden los matices particulares de cada lenguaje de programación.

2.1.3 KhanAcademy

A grandes rasgos, *KhanAcademy* [8] es una organización sin ánimo de lucro con el objetivo de llevar la enseñanza a todos los rincones del mundo a través de Internet, proporcionando enseñanza gratuita.

Entre sus muchos tutoriales, hemos encontrado uno que nos enseña a dibujar[2] figuras geométricas con muy pocas líneas de código.

La manera en la que nos enseña es un tanto peculiar: a través de una especie de video interactivo. El usuario ve el código en una ventana con botones de reproducción, pausa y *stop*. Cuando el video está en reproducción, el código se escribe solo y en la ventana de la derecha se ve el resultado. Sin embargo, si el usuario pulsa sobre el botón de pausa, el video se congela y el usuario puede modificar el código que aparece en él.

Para ayudarnos a realizar nuestras tareas, en cada tutorial que estemos haciendo se nos proporcionan una serie de funciones que podremos utilizar para desarrollar nuestro programa.

Como vemos en la Figura 2.3 en la parte superior de la ventana vemos el título e introducción del tutorial. En la parte inferior se puede ver la documentación con las cabeceras de las funciones. En la barra lateral izquierda se ve el progreso del tutorial que estemos haciendo. Y en la parte derecha tenemos la sección para escribir el código y la vista del resultado de ejecutarlo.

[2] https://es.khanacademy.org/computing/hour-of-code/hour-of-code-tutorial/p/intro-to-drawing

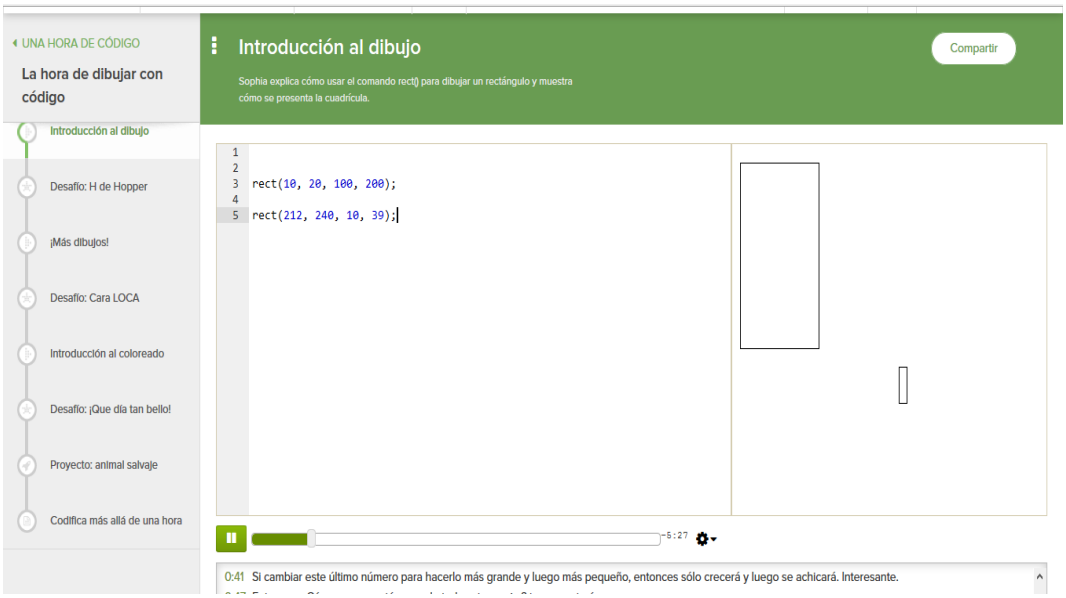


Figura 2.3: Captura de pantalla de KhanAcademy

El punto fuerte de *KhanAcademy* es la gran diversidad de temas con los que el usuario puede aprender, los cuales pueden ser: matemáticas, economía y finanzas, computación, etc. En la mayoría de ellos, el método de aprendizaje es a través del video interactivo que hemos comentado antes. A nosotros nos interesa solo el tema de computación, ya que tenemos un editor de texto en el que podremos ir probando y viendo los resultados en tiempo real sin darle a ningún sitio.

La idea de enseñar a través de una ventana en la que a veces parece que estamos viendo un vídeo pudiendo "pausar" su reproducción y añadir el código que queramos en tiempo real y sin necesidad de pulsar ningún otro botón nos ha parecido divertida, intuitiva y, sobre todo, muy original. Podríamos usar esta idea para realizar retos o puzles que el usuario debería resolver o para mostrarle lo que tiene que hacer a modo de tutorial guiado.

No destacamos nada negativo. En general nos ha gustado.

2.1.4 CodeSpells

Code Spells [9] es un videojuego en primera persona que te enseña a programar en *Java* a través de su *API*[3]. El protagonista es un mago que necesita aprender hechizos para poder ayudar a los habitantes de un pueblo. El jugador, deberá programar en *Java* estos hechizos.

La manera en la que se programan los hechizos es la siguiente: el jugador tiene un libro de hechizos que contiene los hechizos ya aprendidos por el mago, cuyo código puede examinarse para aprender la mecánica con la que están programados. Además de esto, el código de los hechizos viene acompañado de una explicación

[3] Conjunto de subrutinas, funciones y procedimientos que ofrece una biblioteca para ser utilizado por otro software.

sobre lo que hace cada instrucción. Alguno de los hechizos son: teletransporte, levitación, capacidad para prender fuego a los objetos, entre otros.

Para ejecutar un hechizo solo hay que escribir el código y arrastrarlo al objeto al que se lo queramos aplicar. Todo ello con una interfaz que nos recuerda a la de un videojuego y no a una interfaz de un programa orientado a enseñar a programar, como podemos ver en la Figura 2.4:

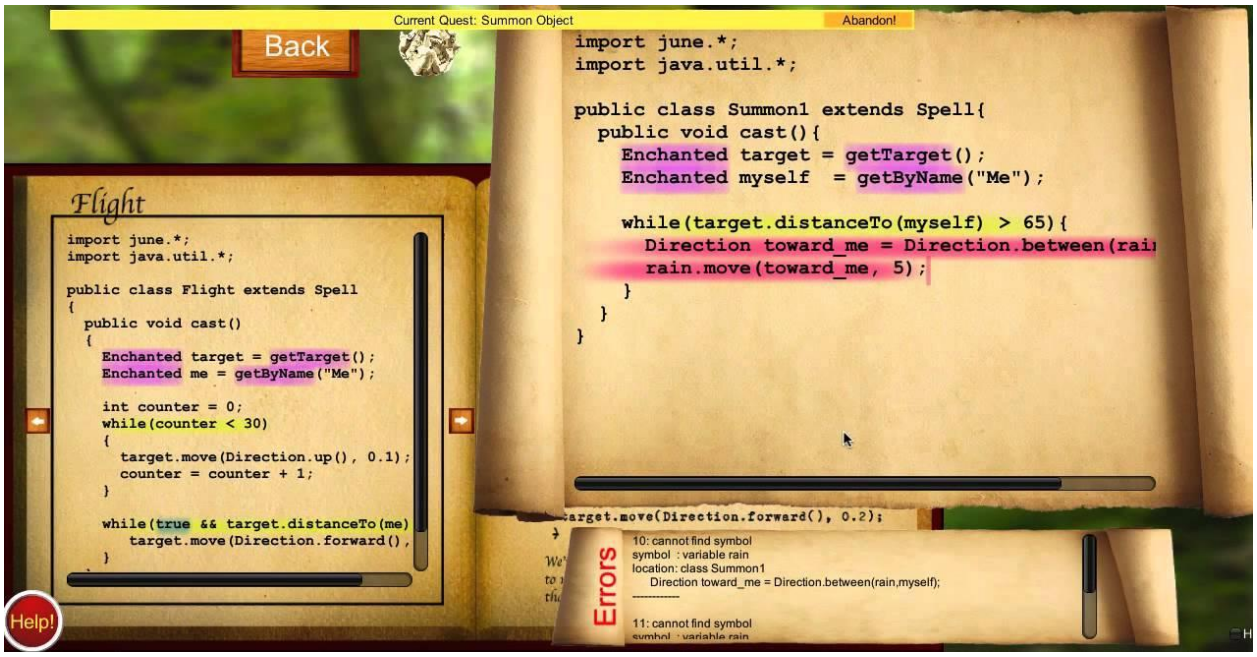


Figura 2.4: Captura de pantalla de Code Spells

Podemos contemplar que para jugar a *Code Spells* se necesitan conocimientos básicos de *Java*, si no la experiencia de juego podría volverse frustrante. Sin embargo, creemos que está bastante bien para usuarios que quieren mejorar sus conocimientos sobre este lenguaje.

Recientemente, el equipo de desarrolladores de *Code Spells* ha propuesto un proyecto en *Kickstarter*[4] que también nos enseñará a programar con un 3d más logrado. Exceptuando la explicación de la historia del juego, hemos visto que han cambiado su manera de enseñar a programar código, ahora utilizan el entorno de aprendizaje *Scratch.* La Figura 2.5 nos muestra el resultado de este nuevo proyecto:

[4] https://www.kickstarter.com/projects/thoughtstem/codespells-express-yourself-with-magic

Figura 2.5: Captura de pantalla de Code Spells Kickstarter

Para nuestro propósito, lo que más nos gusta es la manera de enseñar *Java* del *Code Spells* antiguo y los gráficos de *Code Spells Kickstarter*, ya que llamaría mucho más la atención y la gente lo querría probar.

Sin embargo, en términos de herramientas de aprendizaje, nos quedamos con la manera que usaba *Code Spells* de enseñar un lenguaje escribiendo el código y no moviendo cajas, aunque no descartamos la posibilidad de utilizarlo en algún momento.

2.1.5 Code Avengers

Code Avengers [10] ofrece cursos, algunos gratuitos y otros no, de cómo aprender a programar en lenguajes como *HTML/CSS3* y *JavaScript*. También ofrece la posibilidad de aprender a desarrollar videojuegos en *JavaScript.* Nosotros hemos centrado nuestro análisis en este último aspecto.

Enseña a programar videojuegos a base de tutoriales guiados con unas tareas u objetivos a realizar. Desde la propia web nos dicen que su objetivo es enseñar a programar escribiendo código, con la mínima lectura. Vamos a explicar su funcionamiento observando la Figura 2.6: en la parte izquierda de la pantalla se explica qué es lo que pasa al ejecutar ciertas funciones. Más abajo en esa misma zona nos muestra los retos que nos plantea en este nivel. En la parte central se muestra un editor de texto en el cual podremos escribir el código. Por último, en la parte derecha, muestra en tiempo real el resultado de ejecutar el código.

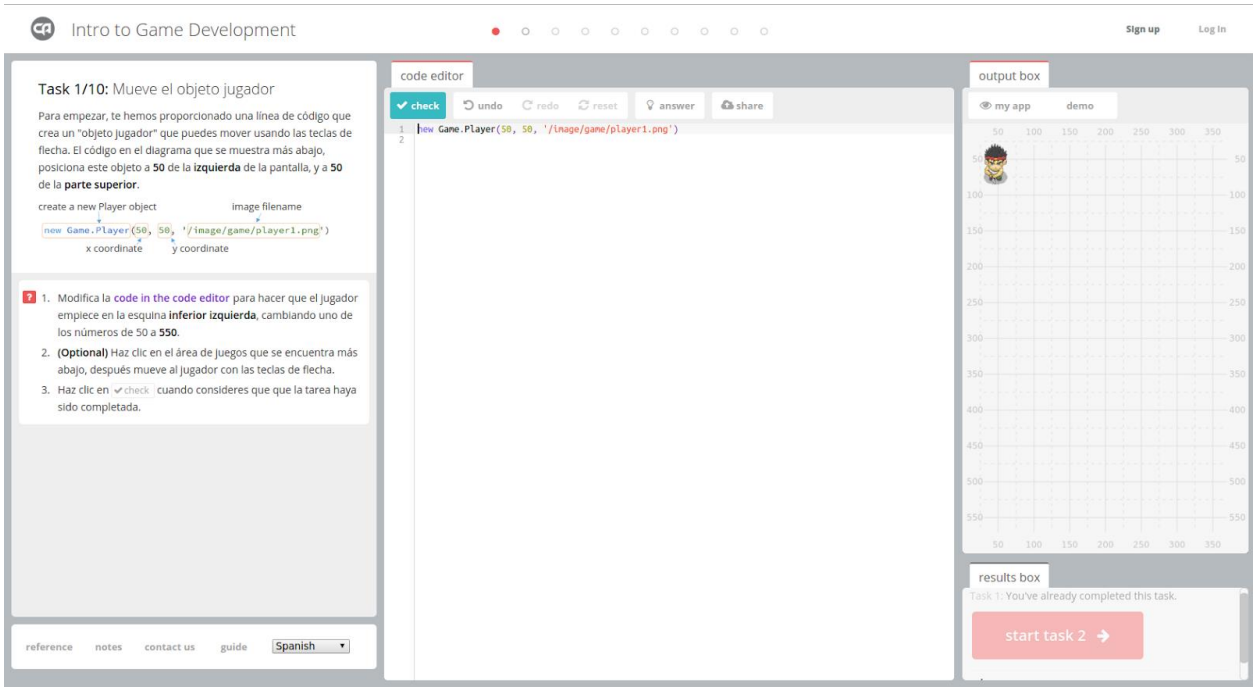


Figura 2.6: Captura de pantalla de Code Avengers

El juego cuenta con un sistema de avatares, logros y habilidades que se van desbloqueando a medida que se completan los retos que nos plantea. A medida que avanzamos los retos que nos proponen son cada vez más complejos, sin embargo, cuando el jugador comete 2 errores consecutivos el juego le muestra la solución.

Una de las cosas que más nos han gustado es el constante *feedback* que se le da al usuario. Cuando una función o variable es errónea, se informa al usuario del error inmediatamente. Otro punto que nos ha gustado es la interfaz. Ésta es bastante limpia y lo más importante, el editor de código, se muestra con un tamaño adecuado en la parte central de la ventana.

Por otro lado, pensamos que este estilo de enseñar a programar es un poco aburrido y monótono, ya que realmente te dicen todo lo que hay que hacer y solo tienes que leerte el tutorial y copiar y pegar trozos.

En conclusión, Code Avengers presenta algo muy cercano a lo que queremos realizar nosotros: enseñar a programar escribiendo el código, con una interfaz agradable y planteando un sistema de dificultad lineal. Sin embargo, pensamos que se les dan a los usuarios demasiadas facilidades para que completen los retos y esto es contrario a la idea que tenemos nosotros. A nosotros nos gustaría más bien dejar que sea el propio usuario el que se pelee con la documentación para saber cómo ha de completar el código para poder seguir avanzando. Nuestro videojuego va orientado a programadores que ya se han peleado alguna vez con algún lenguaje, así que no consideramos que esto es un punto negativo.

2.1.6 Light Bot

Light Bot [11] es un juego en el que el jugador deberá resolver una serie de puzles, consistentes en llevar a un robot a la casilla azul. Para ello, el jugador cuenta con un recuadro en el que aparecen las acciones que puede realizar el robot en ese nivel: avanzar, girar 90º en un sentido y en el otro, saltar y encender una bombilla, entre otros. Una vez comprendidas que hace cada acción, tendremos que rellenar nuestro programa *main* para poder avanzar hasta la casilla objetivo y encender la luz de nuestro amiguito. Para hacer más interesante el videojuego, no podremos utilizar todas las acciones que queramos, sino que tendremos que ajustarnos al hueco que nos proporcionan.

La salida del juego se mostrará cuando pulsemos el botón *play*. Una vez pulsado, el robot realizará las acciones que hemos programado. Si lo hemos hecho correctamente, pasaremos al siguiente nivel y, en caso contrario, deberemos repetir el nivel de nuevo.

Este juego no enseña un lenguaje de programación, sino que hace que el usuario, inconscientemente, especifique el algoritmo que permita al robot alcanzar la casilla de salida. Esto se podría asemejar a los métodos y funciones que habría que programar e invocar dentro de un videojuego real.

Una vez más, y como parece ser tendencia actualmente, la interfaz es sencilla, con colores planos y con el mínimo texto posible. El foco de atención recae en el robot y en los obstáculos que debe superar, que como vemos en la Figura 2.7, aparece con iluminación de fondo:

Figura 2.7: Captura de pantalla de Light Bot

Light Bot se nos queda muy corto para nuestro objetivo, ya que los jugadores pueden no tener intenciones de aprender a programar y pasarse el juego de la misma manera. Lo que sí podemos destacar son los puzles que debemos resolver para llevar al robot a la casilla azulada, idea que nos parece interesante ya que podemos utilizar puzles del mismo tipo para que nuestros jugadores programen ciertos aspectos que les permitan avanzar en el juego.

Hemos considerado acertado incluir *Light Bot* en este apartado de investigación porque consideramos que es una buena forma de adquirir ideas, pese a que no enseñe a programar videojuegos como tal. Lo que más nos ha gustado es la manera en la que se resuelven los puzles.

2.1.7 Code Combat

Code Combat [12] es un juego *RPG* [13] para navegadores que enseña a programar en *JavaScript*. La mecánica del juego consiste en abatir a los enemigos mediante código. El flujo de una partida normal de un nivel es el siguiente: el héroe

estará situado en un mapa específico y en una posición determinada. En el mapa puede o no haber ogros, dependiendo del objetivo de la misión. Los enemigos tienen un comportamiento predefinido, por lo que el jugador deberá programar las acciones del protagonista para realizar la misión con éxito. Para ello, el usuario dispondrá de un editor de texto en la parte derecha de la pantalla como la que aparece en la Figura 2.8. Además del código, también muestra una *API* con las cabeceras de las funciones que el usuario podría utilizar, como ponerse el escudo o encontrar el enemigo más cercano. El resultado del código no se ejecuta en tiempo real, sino que es necesario pulsar sobre un botón.

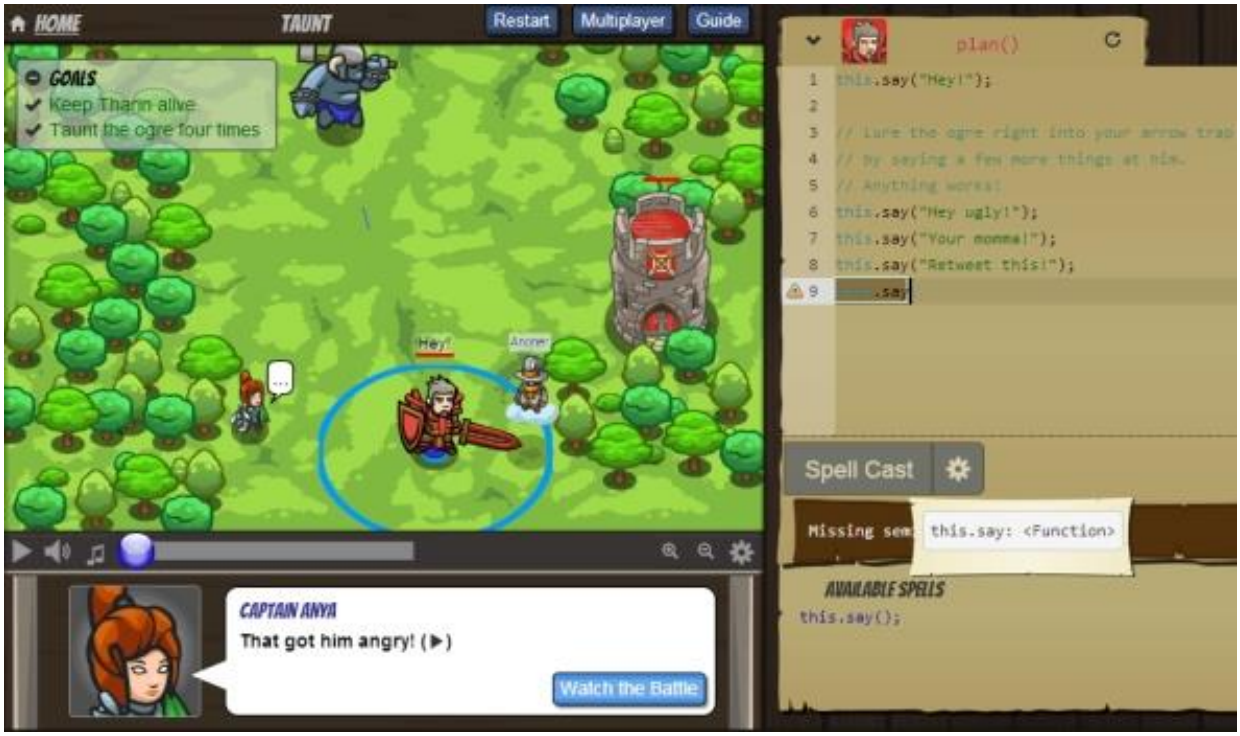


Figura 2.8: Captura de pantalla de Code Combat

Tenemos dos modos de juego distintos: una campaña en solitario, que enseña a programar, como la que vemos en la Figura 2.9, y el modo multijugador. El modo multijugador está pensado para jugadores que ya tienen soltura con el juego y ya tienen las bases para programar en *JavaScript*. Este modo permite enfrentar los códigos de usuarios diferentes en batallas, lo que puede ser muy entretenido y divertido. Además, hace más ameno el aprendizaje del lenguaje.

Nos parece bastante adecuada la manera de enseñar de *Code Combat*. Nos gusta que se le planteen retos al usuario para que mirando la *API* que el juego les proporciona puedan resolver los problemas que se plantean en cada una de las misiones. También nos ha gustado bastante que *Code Combat* notifique a los

usuarios de problemas de compilación del código, mostrándoles un mensaje de error para que arreglen el problema.

Figura 2.9: Captura de pantalla de Code Combat (II)

Nos parece realmente útil que en el editor de código se muestren los errores que tiene, ya que se asemeja al modo de programar *JavaScript* en la vida real.

Para concluir nuestro análisis de *Code Combat*, podemos decir que se asemeja bastante a lo que es programar en la vida real. Por otro lado, pensamos que necesita una documentación más extensa para ayudar al jugador a entender mejor el funcionamiento de *JavaScript*. En general, pensamos que es una herramienta bastante buena para aprender a programar *JavaScript* jugando a un videojuego.

2.2 Tabla comparativa

Hemos reunido las características de los entornos de enseñanza que hemos analizado en la Tabla 2.1. Aunque en primera instancia comentamos que nos íbamos a fijar en una serie de particularidades, analizándolos hemos obtenido algunas otras que hemos considerado interesantes. Por tanto, los campos que hemos incluido son:

- La facilidad de uso del entorno.
- Si es necesario tener conocimientos previos de programación para utilizarlo.
- Si enseñan un lenguaje conocido o uno inventado para asentar las bases de un paradigma de programación, y si este último se acerca a la programación real.
- La manera de enseñar: que puede ser escribiendo código o arrastrando bloques.
- Si el entorno funciona en tiempo real. Es decir, si es necesario pulsar un botón una vez que se ha escrito el código o si, a medida que se escribe el código, podemos ver los resultados.
- El nivel de libertad que se le da al usuario. Si para avanzar es necesario escribir exactamente lo que espera la aplicación o si da la libertad de escribir la solución de múltiples formas.
- Si gestiona los errores, dando *feedback* al usuario.
- Si cuenta o no con documentación.
- Si la interfaz es sencilla o por el contrario está sobrecargada.
- Por último, si se desarrolla con puzles o se trata de tutoriales guiados.

	Scratch	Alice 3D	Khan Academy	Code Spells	Code Avengers	Light Bot	Code Combat
Puzles o tutoriales guiados	**Puzles**	**Puzles**	**Guiado**	**Guiado**	**Guiado**	**Puzles**	**Guiado**
¿La interfaz es sencilla?	Sí	Sí	Sí	Sí	Sí	Sí	Sí
¿Está bien documentado?	Sí	No	Sí	Sí	Sí	No	Sí
¿Gestiona los errores?	Sí	Sí	Sí	Sí	Sí	No	Sí
Nivel de libertad que se le da al usuario (alto, medio o bajo)	Alto	Medio	Bajo	Alto	Bajo	Bajo	Medio
¿Es en tiempo real?	No	No	Sí	No	No	No	No
¿Se acerca el método de enseñanza a la programación real?	No	No	Sí	Sí/No*	Sí	No	Sí
¿Se arrastran bloques?	Sí	No	No	No	No	Sí	No
¿Se escribe código?	No	No	Sí	Sí	Sí	No	Sí
¿Enseña un lenguaje conocido?	No	No	Sí (varios)	Sí (Java)	Sí (Javascript, HTML y CSS3)	No	Sí (Javascript)
¿Se requieren conocimientos previos de programación?	No	No	No	Sí	Sí	No	No
¿Fácil de usar?	**Sí**	**No**	**Sí**	**No**	**Sí**	**Sí**	**Si**

Tabla 2.1: Características del estado del arte

*Sí en el método de enseñanza anterior y No en el método de *Kickstarter*

En la Tabla 2.2 hemos sintetizado los puntos fuertes y débiles que tiene cada entorno, a nuestro parecer:

	Puntos fuertes	Puntos débiles
Scratch	- Diversidad. - Personalización.	- Un tanto infantil.
Alice 3D	- 3D - No necesita conocimientos previos de programación.	- Enseña su propio lenguaje.
Khan Academy	- Interfaz sencilla. - Poder pausar un video para realizar cambios en el editor.	- Tutoriales monótonos.
CodeSpells	- Diversidad. - Enseña *Java*.	- Complejidad
Code Avengers	- *Feedback* constante	- Dificultad muy baja
Light Bot	- Puzzles para resolver.	- No enseña a programar. - Un tanto infantil.
Code Combat	- Enseña *JavaScript* - Aprendizaje más divertido con la funcionalidad multijugador.	- Falta de entendimiento de *JavaScript*. (el usuario no sabe lo que está haciendo por debajo)

Tabla 2.2: Puntos fuertes y débiles del Estado del Arte

2.3 Conclusiones

Tras utilizar estas tecnologías, nos hemos dado cuenta de que escribiendo el código los conceptos se asientan más rápido y mejor en nuestra mente. Además, el estilo de programar arrastrando “cajitas” de colores, como en *Scratch*, nos parece un tanto infantil y nuestro videojuego va orientado a un público más juvenil. Por lo tanto, hemos decidido que enseñaremos a programar escribiendo código.

También nos hemos dado cuenta de que la interfaz en general debe ser apropiada, ya que una interfaz desagradable, por ejemplo, por estar muy sobrecargada de opciones o por ser confusa, puede alejar a nuestro público. Además, actualmente la moda marca desarrollar interfaces lo más minimalistas posible. Por ello nos centraremos en diseñar una interfaz lo más sencilla posible y que resulte cómoda para nuestros jugadores.

Un punto que no debemos olvidar es la dificultad del videojuego. En el caso de *CodeAvengers* nos pasó que se nos hizo aburrido debido a la gran cantidad de ayuda que recibe el jugador para poder seguir avanzando. Nosotros no queremos que nuestros jugadores acaben el juego lo más rápido posible y que unos días después hayan olvidado lo que han aprendido. Por ello nuestra idea es guiarles parcialmente y que recurran a la documentación oficial en caso necesario. Esto puede alejar a algunos programadores con menos experiencia, pero a lo largo de la carrera hemos aprendido que al final lo que sirve es saber resolver los problemas por nosotros mismos, buscando la solución en Internet, en las *wikis* y en la documentación oficial que nos proporcionan.

Pasemos al siguiente apartado, en el que explicamos cómo es la estructura de un videojuego en general y las dos arquitecturas más extendidas de desarrollo de videojuegos. Además, estudiaremos alguno de los *frameworks* de desarrollo de videojuegos para *HTML5* que existen actualmente.

3. Desarrollo de videojuegos en HTML5

En este apartado vamos a tratar distintos aspectos referentes al desarrollo de videojuegos. Por un lado explicaremos la estructura general de un videojuego y, por otro, los módulos que componen un videojuego y las dos arquitecturas predominantes en el desarrollo de videojuegos.

Una vez hecho esto, analizaremos algunos de los motores de desarrollo de videojuegos para *HTML5* que existen en la actualidad. Hemos realizado esta tarea porque necesitamos escoger un motor para desarrollar el videojuego que enseñará a nuestros jugadores a programar videojuegos. Hemos analizado cinco de los motores actuales más conocidos escritos en *JavaScript* y, una vez escogido el que utilizaremos, hablaremos más en profundidad de él.

3.1 Estructura general de un videojuego

La mayoría de los videojuegos [14] comparten la misma estructura y esto incluye a los videojuegos desarrollados en *HTML5*. Un videojuego básico se compone de una serie de pasos definidos que es importante diferenciar para ayudarnos a la hora de modularizar el código.

La parte más importante de un videojuego es el bucle principal, pero esto no es lo único que compone un videojuego. Podríamos diferenciar las siguientes fases:

- **Inicialización:** En esta fase se realizan operaciones de inicialización del motor que estemos utilizando.

- **Carga de recursos:** En esta fase se cargan todos los recursos que van a ser necesarios en el juego. Los recursos pueden ser escenarios, *Sprites*, modelos 3D, sonido, música y demás.

- **Inicio del juego:** Esta fase es la encargada de inicializar y dar valor a todas las variables y parámetros que componen nuestro juego, y que van a variar en el transcurso del mismo, como por ejemplo, el número de vidas o la puntuación del jugador, así como la carga de los niveles.

- **Bucle principal:** Esta fase representa la espina dorsal de un videojuego. En cada iteración se llevan a cabo acciones que son fundamentales para el desarrollo del juego. Podemos distinguir tres pasos dentro del bucle:
    - Recoger la entrada del usuario.
    - Llevar a cabo la lógica propia del juego en función de la entrada del usuario y del estado actual del juego y de las distintas entidades que lo componen.
    - Transmitir al usuario el nuevo estado del juego al que se ha llegado haciendo uso de recursos gráficos y sonoros.
- **Finalización**: en esta última fase se realiza la liberación de recursos usados por el motor.

3.1.1 Arquitectura basada en componentes

También es importante hablar de la arquitectura que se va a usar a la hora de implementar un videojuego. Fundamentalmente tenemos dos opciones: utilizar una arquitectura basada en jerarquías de clases o utilizar una arquitectura basada en componentes.

El pilar fundamental de esta arquitectura es la **modularidad** [15]. Para entenderla, debemos olvidarnos del diseño orientado a objetos y herencia. La idea fundamental es que cada elemento del juego es una **entidad** independiente del resto de elementos. De este modo, cada entidad está compuesta de una serie de **componentes** que pueden ponerse y quitarse a dicha entidad en cualquier momento.

Veámoslo con un ejemplo. Supongamos que hemos creado la entidad jugador en un videojuego de plataformas y que a esa entidad le damos los componentes: salud, energía, caminar, correr y saltar. En cualquier punto del juego podríamos quitar un componente del jugador, como por ejemplo, la capacidad de saltar. O incluso añadir algún otro, como podría ser atacar dando patadas.

Como vemos, las componentes son algo así como pequeños módulos que añaden cierta funcionalidad a una entidad. Además, cuentan con una interfaz para comunicarse con el resto de componentes del diseño. En el ejemplo anterior, podemos relacionar los componentes energía y atacar, ya que para atacar suponemos que el jugador necesita suficiente energía.

Las entidades junto con los componentes que tienen, o pueden tener asignadas, aportan funcionalidad al videojuego de una manera independiente a las demás. Con este paradigma, añadir nueva funcionalidad a un juego se reduce a programar nuevos componentes que implementen esta funcionalidad, sin preocuparnos de afectar a otros componentes o entidades.

3.1.2 Arquitectura basada en objetos o herencia

La **arquitectura basada en objetos (o herencia)** se basa principalmente en el concepto de la herencia, de tal manera que las distintas funcionalidades que puede poseer un objeto de nuestro juego se obtienen mediante la herencia de las mismas a partir de otras clases, interfaces u otros objetos. Por ejemplo, supongamos un videojuego de acción en el que el personaje cuente con un arsenal de armas. Ahora, si quisiésemos añadir a nuestro jugador un arma tendríamos que hacer que heredase de un objeto de tipo jugador con arma, el cual contará con un atributo que es un objeto de tipo arma.

En los juegos con **diseño orientado a objetos** cada cosa que exista en el juego, como un jugador, un enemigo, un coche o una bala, están representados por una clase. Cada una de estas clases puede heredar de otras para tener las mismas propiedades, características o funcionalidades de la clase madre. Por ejemplo, en un juego pueden existir dos clases diferentes: Coche y Camión. Ambas tienen características comunes, así que sería lógico que en la jerarquía de clases encontrásemos una clase Vehículo con todas estas propiedades y que ambas clases heredasen de ella. A su vez, es muy probable que la clase Vehículo herede de una clase que represente objetos físicos dentro del juego, y esta a su vez herede de la clase que represente a cada objeto dentro del juego.

Pero existe un inconveniente en todo esto y es que, a medida que crece la estructura del árbol de herencia, se va haciendo más difícil mantenerla. Si hablamos de videojuegos tenemos que tener en cuenta que son aplicaciones que están constantemente cambiando sus requisitos y funcionalidades, y lo más probable es que llegado un momento tengamos que añadir una nueva funcionalidad a alguna de las clases base afectando de esta manera a toda la estructura de herencia. Si el árbol de herencia no se adapta a estos nuevos cambios, habría que reorganizar el árbol de herencia o añadir los cambios a cada nodo hoja del árbol, es decir, a las clases hijas en el nivel más inferior de la jerarquía. Ambas son malas soluciones ya que cuestan grandes cantidades de tiempo y esfuerzo.

3.2 Motores de desarrollo de videojuegos en *HTML5*

La elección del motor de desarrollo de videojuegos[5] a utilizar es una de las decisiones más importantes que hemos tenido que tomar. A la hora de elegir un motor de juego para nuestro proyecto barajamos distintas posibilidades, ya que actualmente podemos encontrar gran variedad de motores para desarrollar juegos con *HTML5*. Debido a esto, hemos decidido centrarnos en los motores más populares que cuentan con licencia gratuita.

En primer lugar hemos hecho una investigación a través de Internet para conocer las distintas alternativas que existían. Los puntos clave en los que nos hemos fijado han sido:

- Facilidad de uso.
- Buena documentación.
- Existencia de una comunidad que de soporte, tutoriales, ejemplos…
- Que formase parte de un proyecto en continua actualización y no fuese un proyecto "muerto".

Pasemos ahora a realizar un breve análisis sobre algunos de los motores de desarrollo de videojuegos en *HTML5* que existen en la actualidad.

3.2.1 Create.js

Create.js [16] es un conjunto de varias bibliotecas (escritas en *JavaScript*) y herramientas que nos ayudan a desarrollar contenidos interactivos con *HTML5*, entre ellos, videojuegos. Los módulos que componen *Create.js* son:

- ***EaselJS***: bilbioteca que ofrece soluciones directas para trabajar con gráficos e interactuar con el elemento *Canvas*. Es la parte de *Create.js* que más nos interesa a la hora de crear videojuegos o experiencias gráficas. Presenta una *API* similar a la de *Flash*, pero que utiliza las ventajas de *JavaScript*. Esta *API* utiliza un diseño orientado a objetos que facilita el trabajo con el elemento *Canvas*.
- ***TweenJS***: biblioteca desarrollada para crear *tweens* y animaciones con *HTML5*, utilizando tanto las propiedades numéricas de un objeto como

[5] https://html5gameengine.com/

las propiedades de *CSS*. La *API* que utiliza es bastante simple pero es muy potente, haciendo fácil la tarea de crear *tweens* complejos encadenando comandos.

- ***SoundJS***: biblioteca que proporciona una *API* simple y potente para trabajar con audio de una manera más abstracta que con la implementación de *HTML5*.

- ***PreloadJS***: biblioteca que sirve para gestionar la carga de assets (imágenes, sonidos, *JSON* y otros datos).

3.2.2 Quintus

Quintus [17] es un *framework* de desarrollo de videojuegos 2D que usa elementos de *HTML5* para realizar videojuegos de escritorio o móvil. Una de las principales características de Q*uintus* es que se sustenta en una arquitectura basada en componentes que, como ya hemos explicado antes, cambia el paradigma a la hora de crear un videojuego.

Quintus tiene diversos módulos que incorporan nuevas funcionalidades al motor, facilitando muchas tareas. Además de poder utilizar los módulos que proporciona como Input, *Sprites* o *Scenes*, entre otros, presenta la opción de poder crear nuevos módulos que se adapten mejor al videojuego que se esté desarrollando.

Como la mayoría de los motores actuales, posee módulos para poder cargar los archivos que se utilizarán posteriormente en un videojuego, tales como imágenes *PNG*, ficheros *JSON* o ficheros *TMX*.

Podemos destacar su módulo para crear escenas y escenarios. Una escena contiene instrucciones para poder configurar un escenario y, a su vez, un escenario contiene las distintas capas que tiene el videojuego y, por lo tanto, que se mostrará en pantalla. Los escenarios son útiles a la hora de reutilizarlos en las distintas fases del juego. El resto de módulos permiten manejar de forma eficiente el audio, animaciones, elementos de entrada y físicas en 2D.

Quintus posee un sistema de gestión de eventos que permite registrar eventos personalizados por el programador. Por ejemplo, podemos crear el evento "disparo" dentro del objeto "Pistola" de tal manera que cuando se lance el evento, nuestra pistola realice alguna acción (como disparar si tiene munición suficiente).

Una de las mayores ventajas que tiene es su diseño basado en una arquitectura basada en componentes, que hace la programación mucho más sencilla.

Por otro lado, no es un motor que se esté actualizando cada mes, sino que recibe actualizaciones un período de tiempo mayor y, por lo tanto, presenta algunas debilidades, siendo la falta de documentación la más notoria.

3.2.3 Phaser.js

Phaser es un *framework* concebido para la creación de videojuegos 2D en *HTML5* para navegadores, ya sea para teléfonos móviles o para escritorio. Soporta tanto *Canvas* como *WebGL*, y puede intercambiarlos automáticamente apoyándose en el navegador. Para el renderizado se basa en la librería *Pixi.js*[6], a la que también contribuye. Tiene soporte tanto para *smartphones* como tablets, y permite la entrada/salida por teclado, ratón, *multitouch* y *gamepad*.

Ofrece carga de recursos muy sencilla y soporta múltiples formatos. Permite la carga de mapas creados con la herramienta *Tiled*, tanto en formato *TMX* como en *JSON*. Y cuenta con *códecs* [7]de sonido como *mp3*, *ogg* y *wav*. Cuenta con soporte para el manejo de *sprites*, animaciones, grupos, cámara… Además de contar con sistemas de partículas que podemos utilizar de forma sencilla.

Phaser está siempre en continuo desarrollo. Con mucha frecuencia salen nuevas versiones y cada vez es más utilizada por la gente para crear sus propios videojuegos. *Phaser* posee una documentación bastante buena y con un apartado muy amplio de ejemplos donde puedes ver el código fuente de estos para comprender su funcionamiento. Por último, detrás de él existe una gran comunidad que se encarga de mejorar la documentación, crear ejemplos nuevos y demás.

3.2.4 Cocos 2D

Cocos 2D [18] forma parte de un conjunto de herramientas orientadas al desarrollo de videojuegos intergeneracionales, esto es, que funcionen bajo *Windows, Linux y OS X*. Los módulos que lo componen son:

- ***Cocos2d-x***: *framework* de desarrollo de videojuegos de código libre y escrito en C++. Permite no sólo desarrollar videojuegos, sino aplicaciones.

[6] http://www.pixijs.com/
[7] Especificación *software*, *hardware*, o un conjunto de ambas, que es capaz de transformar un flujo de datos para su reproducción o manipulación.

- ***Cocos2d-JS***: junto al módulo anterior, permite desarrollar videojuegos debido a que añaden la funcionalidad necesaria. Se trata de un motor escrito en *JavaScript*.

- ***Cocos2d-XNA***: *framework* de desarrollo de videojuegos 2D/3D escrito en C#.

- ***Cocos2D-Swift***: *framework* de desarrollo que permite escribir videojuegos 2D intergeneracionales. Está escrito en *Xcode* y *Objetive-C*.

- ***Cocos2d (Python)****:* módulo similar a cocos2d-x con la diferencia de estar escrito en *Python*. Se trata de un proyecto en desarrollo.

Tras esta breve introducción, nos centraremos en ***Cocos2d-js***, puesto que es el módulo basado en *JavaScript*. Cocos2d-js es un motor de código abierto con un alto rendimiento, fácil de usar y compatible con el desarrollo multiplataforma (*Android, IOS, Windows Phone, Mac, Windows...*).

Incluye *Cocos2d-html5* y *Cocos2d-x Javascript Binding* (*JSB*). *Cocos2d-html5* soporta *WebGL* y es 100% compatible con *HTML5*. Por otro lado, *Cocos2d-x JSB* es el código contenedor entre el código nativo y el código en *JavaScript*, permitiendo la conversión de las llamadas a funciones de uno a otro.

Cuenta con una extensa *API* que permite, entre otros, el manejo de *Sprites*, animaciones, partículas, temporizadores, eventos (teclado, ratón, tacto y acelerómetro) y sonido. Además, nos permite crear un juego con muy pocas líneas de código.

Una de las mayores ventajas es que se trata de un proyecto en continua evolución. Además, tiene detrás de él una gran comunidad trabajando en la documentación, ejemplos, tutoriales y videojuegos.

3.2.5 Turbulenz.js

Turbulenz [19] es un motor de videojuegos basado en *HTML5* que funciona principalmente con *Canvas*, *WebGL* y *JavaScript*. Se pueden realizar tanto juegos en 2D como en 3D. Estas son algunas de sus otras características:

- Código abierto.

- Ofrece una *SDK* que facilita el desarrollo de los videojuegos.

- Permite alojar los juegos que desarrollen sus usuarios en su página web[8].

Turbulenz nos proporciona una serie de *API's* que facilitan la codificación de los videojuegos. Las más destacadas son:

- ***Low-Level API***: grupo de interfaces que dan acceso a la funcionalidad de bajo nivel, como los gráficos, funciones matemáticas, físicas 2D y 3D, sonidos, entrada y red.

- ***Hight-Level API***: grupo de interfaces que dan acceso a la funcionalidad de alto nivel, como animaciones, manejador de recursos, peticiones a servidores y renderizado.

Además permite que se interactúe con sus servicios de una forma fácil, utilizando la *API* de servicios de *Turbulenz* o a través de su biblioteca incluida en *jslib*. Uno de esos servicios permite que se guarden los progresos de un jugador de un videojuego desarrollado con *Turbulenz* en sus servidores, mientras que otro de los servicios permite que se guarden las máximas puntuaciones, que pueden ser consultadas desde el propio juego o la web de *Turbulenz*. También podemos encontrar otros que nos permiten: acceder al perfil de los usuarios, reiniciar los datos del servidor, crear un sistema de recompensas o logros basados en medallas…

Algunos de los objetivos que ha tenido en cuenta este motor para realizar los juegos más fácilmente, son los siguientes:

- **Modularidad**: Cualquier usuario puede elegir que módulos quiere utilizar para desarrollar el videojuego.

- **Alto rendimiento**: Codificación del código eficientemente para dar un mejor rendimiento del juego.

- **Carga asíncrona**: Ninguna API debería esperar por una respuesta del servidor.

- **Carga rápida**: Reducir la cantidad de datos para cargar comprimiendo los datos eficientemente.

- **Mantenibilidad**: Código del motor legible para que se pueda mantener.

[8] https://ga.me/

Lo que más nos ha gustado de este motor es la facilidad que ofrece para implementar juegos sociales, cosa que no hemos visto en ningún otro motor. Por contra, no parece ser un motor muy activo actualmente ya que las últimas contribuciones que se han realizado son de hace medio año.

3.2.6 Conclusión

La Tabla 3.3 resume los puntos clave en los que nos hemos fijado a la hora de analizar los motores. Gracias a ella será más fácil tomar una conclusión.

	Fácil de usar	**Buena documentación**	**Buena comunidad de usuarios**	**Proyecto vivo en la actualidad**
Create.js	Sí	Sí	Sí	Sí
Quintus.js	Sí	No	No	Más o menos
Phaser.js	Sí	Sí	Sí	Sí (mucho)
Cocos2D.js	Sí	Sí	Sí	Sí
Turbulenz.js	Sí	Sí	No	No

Tabla 3.3: Resumen de las características de los motores analizados

De entre todos ellos, hemos pensado que el que más se adapta a nuestras necesidades es *Phaser*. Las características que más nos han gustado de *Phaser* son:

- Recibe **actualizaciones** de forma periódica y cada muy poco tiempo.
- Hay una **gran comunidad** de personas detrás creando tutoriales, ejemplos, completando la documentación y mejorando *Phaser*.
- La curva de aprendizaje es bastante admisible para iniciarse en la programación de videojuegos.
- Richard Davey, creador de *Phaser,* es una persona muy activa en cuanto a dar publicidad a todo aquel que trabaje con su *framework* a través de las redes sociales y su página web. Esto nos ha gustado y motivado, ya que pensamos que toda publicidad, mientras sea buena, siempre es bienvenida.

3.3 Phaser

Aunque hemos hablado anteriormente de *Phaser,* en este apartado vamos a contar de forma más detallada cuáles son sus características y su modo de uso, ya que es el motor que hemos elegido para desarrollar nuestra herramienta.

Phaser es un *framework* de código abierto basado en *Pixi.js* para el desarrollo de videojuegos. Nació el 12 de Abril de 2013 de la mano de *Photonstorm* bajo la licencia MIT. Soporta tanto *JavaScript* como *TypeScript*.

En Windows, la recomendación de *Photonstorm* es utilizar *XAMPP* debido a su facilidad de instalación y uso. Como es una herramienta que ya hemos utilizado en alguna asignatura (Ingeniería Web, por ejemplo) y tiene soporte para *GNU*/*Linux*, hemos decidido utilizarla en nuestro proyecto.

Entre sus características principales destacan:

- **Soporte** para ***WebGL*** y ***Canvas****.* Como ya hemos mencionado, *Phaser* puede utilizar para renderizar tanto *Canvas* como *WebGL* e incluso puede intercambiar automáticamente entre ambos, para abarcar un rango más alto de dispositivos.

- **Precarga de recursos**. comprueba y maneja automáticamente todos los recursos necesarios: imágenes, sonidos, *sprites*, *tilemaps*[9], *JSON*, *XML*, etc. Permite que sean cargados en tan sólo una línea de código.

- **Sprites:** los *Sprites* son los elementos fundamentales de cualquier juego. *Phaser* permite realizar gran cantidad de acciones sobre los *Sprites*, tales como darles una posición, animarlos, rotarlos, escalarlos, colisionarlos entre ellos, darles texturas, etc. Además, también cuenta con un completo soporte para manejar la entrada del usuario, pudiendo, por ejemplo, hacer *click* sobre los *Sprites*, tocarlos o arrastrarlos.

- **Grupos:** *Phaser* permite crear grupos de *Sprites* para facilitar su uso, por ejemplo, para llamar a una función sobre múltiples *Sprites* que forman un grupo. No sólo permite la colisión entre *Sprites*, sino que también permite controlar la colisión entre los distintos grupos que se hayan definido.

[9] Formato usado en videojuegos para guardar la información sobre un mapa o un nivel.

- **Animaciones:** permite el uso de *spritesheets*, pudiendo especificar el tamaño de cada sub *Sprite*. Soporta archivos *JSON* provenientes de *Texture Packer* [20] o de *Flash Cs6/CC* (tanto en formato hash como en formato *array*), y archivos *XML* de *Starling* [21]. Todos estos formatos se pueden usar fácilmente para crear animaciones para nuestros *Sprites*.

- **Físicas:** cuenta con tres sistema de físicas distintos:

    - **Arcade**: sistema para colisiones rectangulares AABB. Las colisiones AABB (*Axis Aligned Bounding Box*), se basan en encerrar a un objeto del juego en el mínimo rectángulo que lo contenga. De esta forma, comprobar la colisión entre dos objetos se reduce a comprobar la colisión entre dos rectángulos, de coste bajo computacionalmente hablando si lo comparamos con otros mecanismos de colisión.

    - **Ninja**: sistema para *tilemaps* y pendientes. Se basan en el mecanismo de detección de colisiones AABB, con la salvedad de que permite utilizar hasta 34 formas geométricas diferentes para encerrar a los objetos y calcular las colisiones.

    - **P2**: sistema de físicas para colisiones complejas. En lugar de emplear cuerpos geométricos que encierren a los objetos, las colisiones P2 construyen un cuerpo lo más preciso al *Sprite* del objeto en cuestión. Por ello, tienen un coste computacional muy elevado.

- **Partículas:** cuenta con un sistema de partículas incorporado, que permite crear gran cantidad de efectos interesantes. Se pueden crear explosiones o efectos constantes como lluvia o fuego. Todos estos efectos no sólo pueden usarse por separado, sino que pueden ser asociarlos a un *Sprite* concreto.

- **Cámara:** algo fundamental en un juego es el mundo donde se va a desarrollar. *Phaser* permite colocar los distintos objetos que forman un juego en cualquier parte de la escena y, por supuesto, cuenta con una cámara que puede situarse en cualquier parte de dicha escena. La cámara cuenta con distintos modos para seguir a un *Sprite* concreto.

- **Input:** *Phaser* proporciona una sencilla interfaz para tratar los inputs, *Phaser.Pointer*, sin que el programador tenga que preocuparse por el método de entrada, ya sea con ratón y teclado o mediante una pantalla

táctil. Además cuenta con gran cantidad de funciones para personalizar el reconocimiento de gestos en el caso en el que trabajemos con dispositivos táctiles.

- **Sonido:** soporta tanto *Web Audio* [22] como *HTML Audio* [23]. Contiene funciones específicas para lidiar con algunos problemas como el bloqueo del dispositivo, el *streaming* o el volumen.

- **Tilemaps:** *Phaser* puede cargar, renderizar y manejar las colisiones de los *tilemaps* con muy pocas líneas de código. Soporta los formatos de *CSV* y también de *Tiled*, permitiendo mapas con múltiples capas. Además, cuenta con muchas funciones para manipular los distintos tiles que conforman el mapa. Permite intercambiar tiles, reemplazarlos, eliminarlos, añadir nuevos tiles al mapa y actualizarlo en tiempo real.

- **Escalado**. *Phaser* cuenta con un manager que permite escalar el juego para que se adapte a cualquier pantalla. Podemos controlar las proporciones del *Canvas*, su tamaño mínimo y máximo, y cuenta también con opción de pantalla completa.

- **Sistema de *plugins*:** los creadores de *Phaser* han intentado mantener el núcleo del motor lo más básico posible, así que sólo cuenta con las clases más esenciales. Por ello han construido también un sistema de *plugins* para manejar el resto de funcionalidades del motor. Esto también nos permite crear nuestros propios *plugins* y compartirlos con la comunidad de *Phaser*.

Para entender el funcionamiento de *Phaser* debemos conocer el concepto de estados. Anteriormente hemos hablado de las distintas fases que podemos encontrar en un videojuego. Para *Phaser* cada una de estas fases está representada por un objeto de tipo *State*. Según los propios tutoriales que proporciona el motor, hemos observado que hay una convención a la hora de establecer cuáles van a ser estos estados. Como vemos a continuación, esta convención coincide con la estructura general de la que hemos estado hablando:

- **Boot State**: estado que define la configuración general del juego. Si se va a mostrar una pantalla de carga, como ocurre en la mayoría de videojuegos, se aprovecha para cargar aquí los *assets* necesarios.

- **Preload State**: estado en el que se cargan todos los *assets* del juego (imágenes, *spritesheets*, audios, texturas, etc). Si hay definida una pantalla de carga, se mostrará en este estado.

- **MainMenu State** (opcional): estado que muestra la pantalla de menú que se muestra antes de empezar a jugar. En este punto todos los recursos ya se han cargado en la memoria y están listos para ser utilizados en el juego.

- **Game State**: estado que contiene el bucle principal del juego. Se encarga de capturar las acciones del jugador, procesar la actualización de los elementos de la escena en consecuencia de estas acciones y dibujarlos.

Por supuesto, a parte de estos estados, que son los que consideraríamos esenciales, podemos añadir tantos estados como queramos para representar más fases distintas dentro de nuestro juego.

A su vez, cada estado tiene una serie de métodos que son necesarios para el flujo del juego, algunos de los más importantes son los siguientes:

- **Init**: es la primera función en ser llamada.

- **Preload**: se utiliza para cargar los recursos que utilizará el juego.

- **Create**: en esta función se crean los objetos que se vayan a utilizar en el juego.

- **Update**: este método es el que llamará el bucle del juego cada cierto tiempo.

- **Paused**: esta función es llamada cuando el bucle principal del juego está parado.

3.4 Conclusiones

Como hemos podido observar la estructura de un videojuego no es simple. Existen varias fases que hay que entender a la perfección para que funcione correctamente. Otro punto a tener en cuenta es la arquitectura del videojuego. Nosotros hemos hablado de las arquitecturas basadas en componentes y las basadas en herencia. Las arquitecturas basadas en componentes se fundamentan en la independencia de los objetos con el uso de componentes. Por otro lado, las arquitecturas basadas en herencia buscan justo lo contrario: buscar las relaciones entre los objetos para emparentarlos entre sí.

Después de investigar algunos de los *frameworks* o motores que existen en la actualidad, hemos elegido a *Phaser* debido a que recibe actualizaciones de forma muy periódica, tiene una gran comunidad de programadores trabajando haciendo ejemplos y mejorando la documentación; es fácil de aprender y utilizar, y por la motivación que supone que su creador publicite a las personas que trabajen con su motor.

En los dos siguientes apartados hemos tratado primero el diseño de la herramienta y, segundo, su implementación.

4. Diseño de la herramienta

Para lograr el objetivo de nuestro proyecto hemos tenido que diseñar e implementar no sólo el videojuego a través del que vamos a enseñar al usuario, sino también una serie de módulos que convierten este juego en una herramienta de enseñanza.

En este apartado vamos a diseñar todos aquellos módulos que hemos necesitado para crear nuestra herramienta.

4.1 Funcionalidades de la herramienta

Nuestra herramienta debe ser capaz de enseñar los conceptos básicos de la programación de videojuegos, utilizando como medio para este fin un videojuego. Para ello necesitamos que tenga las siguientes funcionalidades:

- Un videojuego implementado con el *framework Phaser* que presente obstáculos que deberán ser superados por el jugador programando ciertas mecánicas del juego.

- Un mecanismo para que el usuario codifique ciertos aspectos del videojuego y encontrar la forma de hacer que el código que escriba el jugador se introduzca en el juego. Este mecanismo debe ser concordante con las conclusiones que hemos obtenido con nuestro estado del arte sobre los entornos de enseñanza existentes. Recordamos que la principal conclusión que hemos obtenido es que queremos que el usuario escriba el código.

- Un mecanismo que sirva para verificar que lo que codifique el jugador es correcto sintácticamente y que cumpla las condiciones que le hemos impuesto.

- Una forma de comunicarse con el jugador cara a cara para explicarle los retos que debe superar, ciertas peculiaridades sobre la programación de videojuegos en general y algunas mecánicas para ayudarle a entender el funcionamiento de *Phaser*.

- Un mecanismo que muestre la documentación de las funciones de *Phaser*, para que el usuario pueda consultarla en cualquier momento a la hora de realizar los desafíos.

En los siguientes apartados vamos a diseñar los módulos que hemos considerado oportunos para lograr que la herramienta tenga las funcionalidades que acabamos de describir.

4.2 Interfaz de la herramienta

Lo primero que vamos a tratar es el diseño de la interfaz de usuario [24] para hacernos una idea visual de los módulos que componen la herramienta. Nos hemos apoyado en parte de lo aprendido en la asignatura Desarrollo de Sistemas Interactivos para diseñar desde un principio una interfaz agradable para el usuario, intentando seguir los principios de usabilidad [25].

En primer lugar, hemos diseñado *mockups*[10] de bajo nivel sobre papel para tener una primera idea sobre qué debería tener la interfaz. En segundo lugar, creamos los prototipos de alto nivel. Por último, decidimos realizar una evaluación con usuarios para validar el prototipo que hemos diseñado. Todo ello está expuesto en la Sección 6.

Los *mockups* son una forma rápida y muy cercana a la realidad de diseñar una interfaz de usuario con pocos recursos económicos y temporales. Es por ello por lo que hemos decidido emplearlos como primera fase en el proceso de desarrollo de la interfaz.

Como vemos en la Figura 4.10, la interfaz queda dividida en 3 secciones: la sección del **juego**, la sección del **sistema de programación** y la sección de la **documentación.**

En la sección del **juego** mostraremos el videojuego en sí. Esta vista quedará bloqueada cuando entre el sistema de programación y el jugador tenga que programar alguna mecánica del videojuego.

En la sección de **documentación** se mostrarán los enlaces a la documentación oficial de *Phaser*, así como la documentación que escribamos nosotros y que consideremos oportuna para guiar al jugador.

[10] Modelo de un diseño utilizado para la demostración, evaluación del diseño, promoción y para otros fines.

Por último, la sección del **sistema de programación** contiene el mecanismo que permite al jugador programar los aspectos del juego.

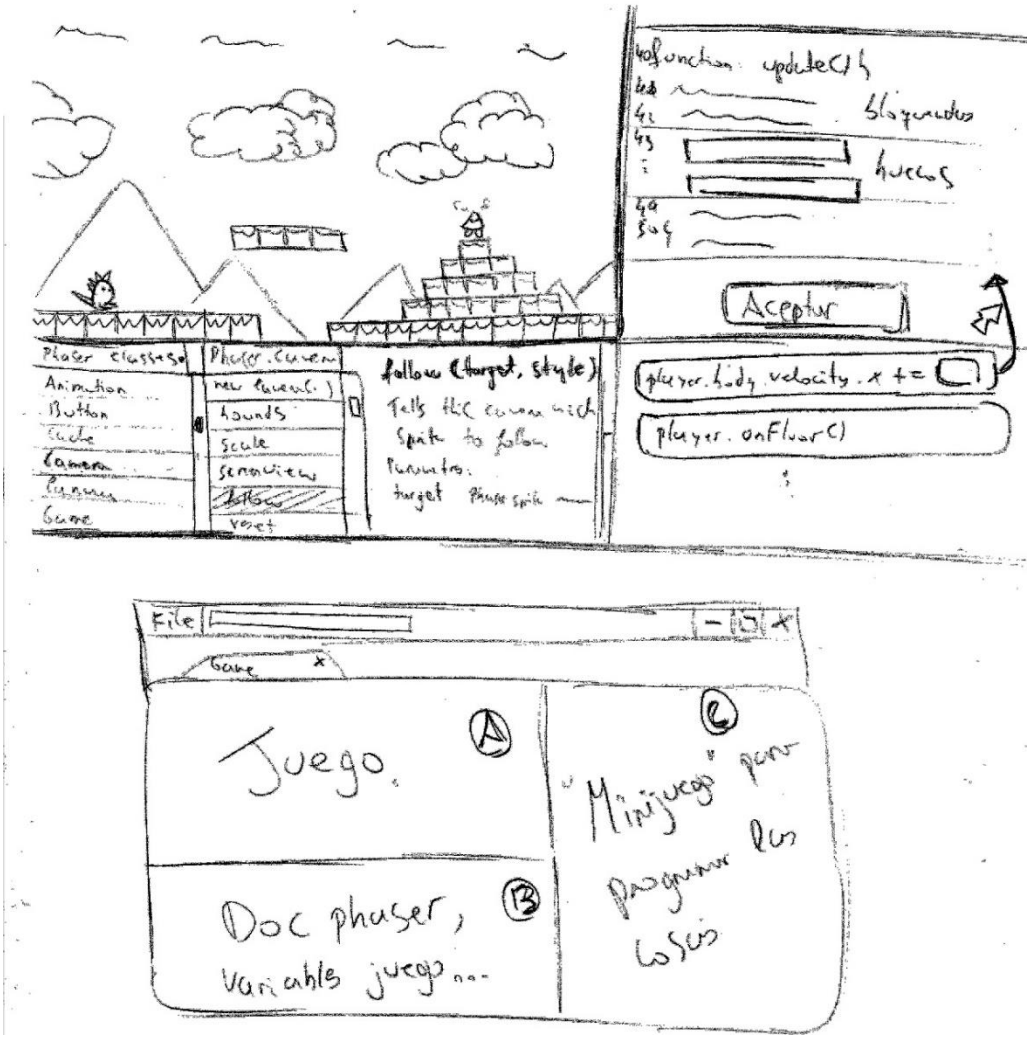


Figura 4.10: Primer *mockup*

4.3 Desafíos

Para que nuestros usuarios aprendan a programar videojuegos deberán superar una serie de desafíos, que se resolverán programando ciertas mecánicas del videojuego.

Una vez desarrollado el videojuego, el primer paso ha consistido en elegir qué cosas va a enseñar nuestra herramienta. Para ello hemos recurrido a la web oficial de los ejemplos de *Phaser*. En ella están recogidos todos los ejemplos hechos tanto por los desarrolladores de *Phaser*, como por cualquier persona que haya escrito un ejemplo lo suficientemente bueno como para ser incluido en la misma. Estos ejemplos abarcan desde lo más básico, que podría ser crear una instancia del motor y pintar un color sólido de fondo, hasta ejemplos que versan sobre colisiones avanzadas de grupos, la creación de *Sprites* en *ATLAS* o el uso del sistema de partículas.

Partiendo de estos ejemplos y de lo que hemos programado en el videojuego, hemos elegido los que nos han parecido mejores para que el usuario aprenda. Para hacer esta tarea más fácil, hemos decidido crear una hoja de cálculo que resumiera los desafíos que podríamos enseñar, así como su complejidad. Como vemos en la Tabla 4.4:

Nivel	Orden	Acciones
1	1	Caminar Izquierda
1	2	Saltar
1	3	Controlar *Walker*
1	4	Saltar sobre *Walker*
1	5	Modificar el *HUD*
1	6	Recoger Monedas
1	7	Crear puerta de fin nivel
2	1	Recoger objetos
2	2	Saltar doble
2	3	Movimiento de las plataformas movedizas
2	4	Modificar enemigo *Thrower*
2	5	Ataques a distancia
2	6	Programar un *NPC*

Tabla 4.4: Desafíos primer y segundo nivel

4.4 Diálogos

Necesitamos este módulo para comunicarnos de una forma más personal con el jugador. Con los diálogos queremos explicar las mecánicas del juego, queremos explicarle al jugador que deberá superar ciertos retos programando y queremos explicarle los elementos de los que dispone dentro de la herramienta para lograr superarlos, tales como la documentación, el editor de código y los mensajes de los test.

El jugador debe ser capaz de leer sin problemas los diálogos y éstos no deben avanzar a menos que el jugador pulse sobre un botón. Este módulo también deberá contar con un botón para saltar un diálogo en caso de que el jugador lo quiera hacer.

Este módulo podría estar dentro o fuera del juego pero, por simplicidad, hemos decidido implementar este módulo desde fuera del videojuego.

4.5 Editor de código

Una parte fundamental de nuestra herramienta es el editor de código, ya que es el medio que va a utilizar el jugador para comunicarse con el juego y resolver los desafíos. Cada vez que se comienza un nuevo desafío debe aparecer en el editor el código que se debe completar. La idea es que el jugador rellene ese código ayudándose de la documentación y de las pistas que se le ofrecen mediante los diálogos y los propios comentarios del juego. Una vez que el jugador ha completado el código puede ejecutarlo para comprobar si es correcto.

Para que la herramienta tenga el comportamiento que deseamos creemos que, a parte de las funcionalidades más básicas, como son mostrar el código y poder mandarlo al juego para ejecutarlo, nuestro editor debe contar con algunas funcionalidades más específicas que ayuden al usuario a tener una mejor experiencia:

- Contar con resaltado de sintaxis para *JavaScript*.
- Permitir que se pueda reiniciar el desafío.
- Bloquear las zonas de código que el usuario no debe editar, es decir, las partes de código que le damos ya hechas al usuario. Creemos que esto es importante, ya que así el usuario sólo se centra en la parte que tiene que rellenar.

4.6 Los test de unidad

Necesitamos este módulo para verificar que el código que ha escrito el jugador en un desafío es correcto y no sólo eso, sino que cumple los requisitos que estamos buscando en dicho desafío. Por ejemplo, si le pedimos al jugador que programe el

movimiento de un personaje no nos vale sólo con que cambie su posición horizontal, sino que tenemos que comprobar que se mueva de una forma secuencial, que se reproduzca la animación correspondiente, que no siga avanzando en caso de colisionar con algún obstáculo del mapa o enemigo o cualquier otra cosa que se nos pueda ocurrir.

También esperamos que este módulo sea capaz de decirnos qué está mal y qué está bien al ejecutar un código procedente del editor. Queremos que devuelva un texto que de una pista en el caso de que se haya ejecutado algo erróneo. En el caso de que el código sea correcto, queremos que nos diga lo que el usuario ha escrito bien.

4.7 Documentación

Necesitamos este módulo de documentación porque no queremos sobrecargar el juego con diálogos y textos que expliquen funciones de alto nivel de *Phaser*, sino que queremos que el jugador busque cómo funcionan los métodos, que parámetros reciben, qué valores devuelven e, incluso, en qué línea del motor se encuentra una función o método por si se anima a cambiar su código.

Necesitamos este módulo junto al resto de la herramienta porque no queremos que nuestros usuarios acudan a Internet para resolver sus dudas, sino que queremos que desde la propia herramienta, y sin necesidad de abrir pestañas adicionales en su navegador, puedan resolverlas. Queremos que se integre junto a la aplicación pero no queremos que sea un elemento distractor cuando la documentación no haga falta. Por eso hemos decidido que la documentación pueda mostrarse y ocultarse presionando un botón. Cuando se muestre la documentación ocupará la vista del juego, y cuando se visualice el juego se volverá a ocultar.

4.8 El videojuego

El objetivo de este apartado es definir el diseño del videojuego [26]. Para conseguirlo, hemos redactado alguno de los apartados que componen el documento de concepto de un videojuego [27], tales como las características principales del juego, el género, el público al que irá dirigido y los recursos que utilizará (imágenes, sonidos…).

Necesitamos este módulo porque, como ya hemos mencionado, enseñaremos a programar videojuegos utilizando un videojuego.

4.8.1 Concepto

El prototipo que hemos desarrollado abarca lo que sería el primer nivel del videojuego. En este primer nivel presentamos a Bersara, la protagonista de nuestra historia. Por algún motivo que el jugador desconoce, Bersara ha perdido ciertas capacidades y acciones que antes podía realizar, tales como saltar o recoger monedas, entre otras. Para ayudarla, el jugador deberá programar con *Phaser* lo que necesite Bersara.

A medida que supere los desafíos la dificultad de estos será mayor y daremos menos pistas. La idea es que el jugador se pelee por sí mismo con la documentación de *Phaser* que le ofrecemos, como ocurriría en el caso de estar aprendiendo por su cuenta cualquier otro *framework*, lenguaje de programación o tecnología.

En nuestro prototipo no profundizamos demasiado en las entrañas de *Phaser*, sino que damos una ligera introducción de sus funciones básicas y los estados que define. En este primer nivel enseñaremos al jugador a:

- Hacer que un *Sprite* se mueva a la izquierda al pulsar la flecha izquierda del teclado.
- Hacer que un *Sprite* salte al pulsar la flecha superior del teclado.
- Hacer el comportamiento de un enemigo.
- Hacer que el jugador sea capaz de matar a los enemigos.
- Mostrar un *HUD*[11] en pantalla.
- Recoger monedas y actualizar el *HUD* correctamente.
- Mostrar una puerta que le lleve al siguiente nivel. En nuestro caso, que se muestre el final del juego.

Con este prototipo queremos hacer que el usuario aprenda lo básico de *Phaser* y, sobre todo, que aprenda a utilizar la documentación para resolver sus problemas.

[11] Información que en todo momento se muestra en pantalla durante la partida, generalmente en forma de iconos y números.

4.8.2 Características principales

El juego se basa en los siguientes pilares:

- **Planteamiento sencillo**: la historia en la que se basa el juego es muy simple, pero es suficiente para que el jugador tenga un objetivo y sienta que está jugando a la vez que aprende.

- **Aprendizaje**: el pilar más importante. A través de los distintos niveles, nuestros jugadores deben ser capaces de adquirir desde las bases de la programación en *JavaScript* y *Phaser* a través de la implementación de mecánicas sencillas, hasta conocimientos más complejos sobre el motor y la programación de videojuegos con los que será capaz de desarrollar mecánicas más complejas y superar desafíos más difíciles. Es importante que este aprendizaje sea progresivo para que el jugador no se quede en blanco ante ningún desafío.

- **Ampliación**: nuestro juego debe de ser ampliable, de tal manera que siempre se puedan añadir más niveles con nuevos desafíos que ayuden al jugador a aprender nuevas mecánicas. De esta forma, si *Phaser* continúa creciendo y añadiendo mecánicas nuevas, éstas puedan ser incorporadas en un futuro sin excesiva dificultad.

4.8.3 Género

Hemos decidido que nuestro videojuego sea del género plataformas en 2D, ya que nos parece un género con el que podemos enseñar de forma sencilla las bases de la programación de videojuegos.

4.8.4 Propósito y público objetivo

Como ya hemos comentado anteriormente, el propósito de esta herramienta es el aprendizaje de los conceptos básicos, y no tan básicos, de la programación de videojuegos en *HTML5*.

Nuestro juego va dirigido a personas que no tienen por qué tener experiencia en la materia. Por ello hemos intentado diseñar una herramienta que enseñe desde lo más básico a ciertos aspectos más avanzados. Tampoco es necesario tener muchos conocimientos sobre el lenguaje que vamos a usar, por lo que pensamos que nuestro público objetivo pueden ser personas de entre 18 y 25 años que quieran dar sus primeros pasos en el mundo de la programación de videojuegos en *HTML*5.

4.8.5 Mecánicas

En este apartado vamos a explicar todas las mecánicas existentes en nuestro juego, contando tanto las mecánicas básicas que ya vienen implementadas en la versión de la herramienta, como las mecánicas que tendrá que implementar el usuario a través de los distintos desafíos.

4.8.5.1 Protagonista del juego

Vamos a explicar las distintas mecánicas que tienen que ver con el personaje principal de nuestro juego, Bersara.

Movimiento:

Bersara cuenta con los movimientos básicos de un personaje de plataformas. Puede moverse en dos direcciones (derecha e izquierda) y saltar. En esta versión del juego Bersara no podrá agacharse ni realizar ataques a distancia contra los enemigos, pero esto podría añadirse en versiones futuras.

Acciones:

A parte de moverse, Bersara puede realizar las siguientes acciones:

- **Saltar**: Bersara puede saltar cuando esta parada y también cuando se encuentra en movimiento. Si está en movimiento saltará en la dirección hacia donde se estuviese moviendo.

- **Atacar**: el salto también funciona como ataque. Cuando Bersara salta sobre un enemigo, el enemigo sufre daño.

Salud:

La salud de Bersara se representa con corazones. Al empezar el juego cuenta con tres corazones, los cuales se pueden ir perdiendo a medida que se avanza en el juego. Una vez que se han perdido los tres corazones el juego se acaba y volvemos al menú. Las maneras en las que Bersara puede perder corazones son las siguientes:

- Si un enemigo toca a Bersara. O si Bersara toca lateralmente a un enemigo.

- Si salta o cae encima de unos pinchos.

- Si cae al agua.

4.8.5.2 Monedas

Durante el juego podemos encontrar monedas que Bersara puede recoger. En esta versión del juego las monedas no tienen ninguna otra funcionalidad, las hemos incluido con el fin de poder enseñar al usuario como recoger objetos.

4.8.5.3 Carteles

Los carteles son un elemento muy importante del juego. Están repartidos por todo el nivel. Cada vez que Bersara colisiona con un cartel el juego se pausa y se lanza un desafío. Una vez que el desafío se supera el juego vuelve a su curso y el cartel que lanzó el desafío queda desactivado, de manera que el desafío no se puede volver a lanzar.

4.8.5.4 Enemigos

A lo largo del juego Bersara se encontrará con dos tipos de enemigos: los **snails** y los **slimes**. Ambos tipos tienen distinta apariencia, pero su comportamiento es el mismo. Estos enemigos caminan hacia la derecha o hacia la izquierda y cambian de dirección cada vez que chocan con algún elemento del mapa. En caso de colisionar con Bersara pueden pasar dos cosas:

- Si han sido tocados por encima significa que Bersara ha saltado encima de ellos y, por lo tanto, mueren.

- Si colisionan por los laterales producirán daño a Bersara, eliminando una de sus vidas.

4.8.6 Recursos

Los recursos son uno de los componentes más importantes de un videojuego de cara al jugador, ya que es lo primero que ve y escucha. Sin embargo, de cara al desarrollo es un concepto que queda en segundo plano, ya que la funcionalidad que se implementa es independiente de si un objeto tiene un *sprite* u otro o de si el sonido de una acción es éste o aquel.

En un primer momento la recomendación de Guillermo, nuestro coordinador, ha sido la de utilizar recursos sencillos para desarrollar lo que sería la primera versión del juego para, una vez lanzada una versión estable y completa, utilizar recursos de mayor calidad.

Siguiendo su consejo, hemos hecho las primeras pruebas utilizando las formas cuadradas de la Figura 4.11.

Figura 4.11: Primeros Sprites

Una vez el juego iba tomando forma, decidimos utilizar los recursos del conocidísimo videojuego Super Mario Bros de 1985, aprovechando que fueron liberados por Nintendo. Podemos verlos en la Figura 4.12:

Figura 4.12: Sprites de Mario

Por último, una vez desarrollamos lo que es el primer prototipo nos hemos lanzamos a buscar los *assets* definitivos. Nos han recomendado la web *OpenGameArt* [28] debido a que cuenta con una gran base de datos de recursos con licencia libre. En dicha web hemos encontramos un conjunto de recursos que nos ha gustado especialmente e, investigando un poco más, hemos dado con la web de su creador que, casualmente, publica muchos recursos similares con licencia libre [29]. El siguiente paso fue juntar los *sprites* que nos gustaron y obtuvimos lo que vemos en la Figura 4.13.

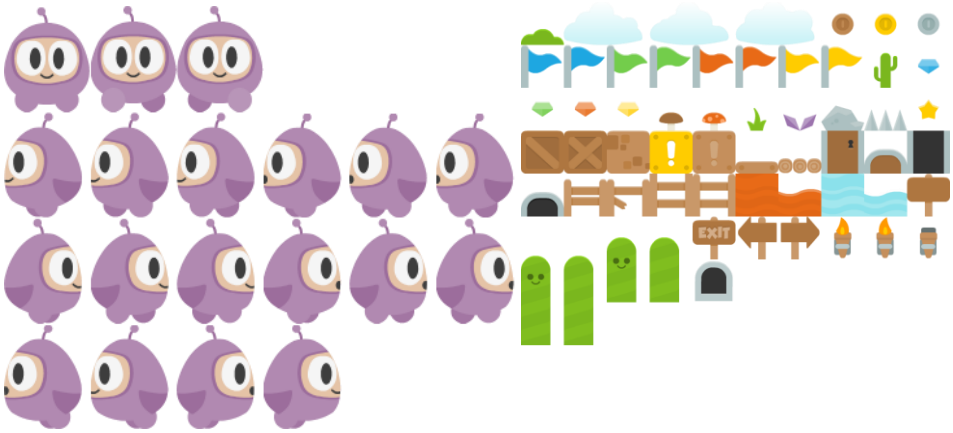


Figura 4.13: *Sprites* finales

4.9 Conclusiones

En este apartado hemos visto los módulos que necesitamos para desarrollar nuestra propuesta. No sólo necesitamos un videojuego, sino que necesitamos un sistema para gestionar los diálogos, un editor de código, los test de unidad que validen el código escrito por el jugador y una forma de mostrar la documentación al usuario. Nos hemos centrado en lo que queremos que hagan, pero no en cómo haremos que lo consigan. El siguiente apartado trata de esto último.

5. Implementación de la herramienta

Una vez tenemos el diseño de todos los módulos que componen nuestro proyecto, el siguiente paso consiste en implementarlos. Antes de nada tenemos que comentar que nuestro videojuego ya tiene nombre al fin: *Lost Code.* El resultado de esta sección es la implementación de nuestra herramienta completa, la cual está alojada en http://lauradcs4.github.io/TFG/.

5.1 Implementación de la interfaz de usuario

A partir de los *mockups* en papel y, una vez tomadas ciertas decisiones y ajustes, hemos pasado a realizar un prototipo de alto nivel con la herramienta *myBalsamiq*[12]*.* Este tipo de herramientas no permite obtener una vista más aproximada al diseño final de la aplicación, como podemos apreciar en la Figura 5.14:

Como podemos ver en la captura anterior, hemos cambiado la sección de **documentación** por una sección de **Pistas y errores**. Hemos decidido realizar este cambio porque hemos pensado que el usuario debe ver en todo momento los errores que está cometiendo mientras escribe el código.

No obstante, como consultar la documentación es algo muy importante para poder superar los desafíos, hemos decidido colocar un botón en la barra principal (situada en la parte superior de la pantalla) que muestre la documentación de la versión 2.1.2 de *Phaser*. Cuando el usuario pulsa sobre este botón, la vista del juego desaparece y aparece una vista con la documentación. Si vuelve a pulsarlo, se producirá el efecto contrario.

[12] https://www.mybalsamiq.com/

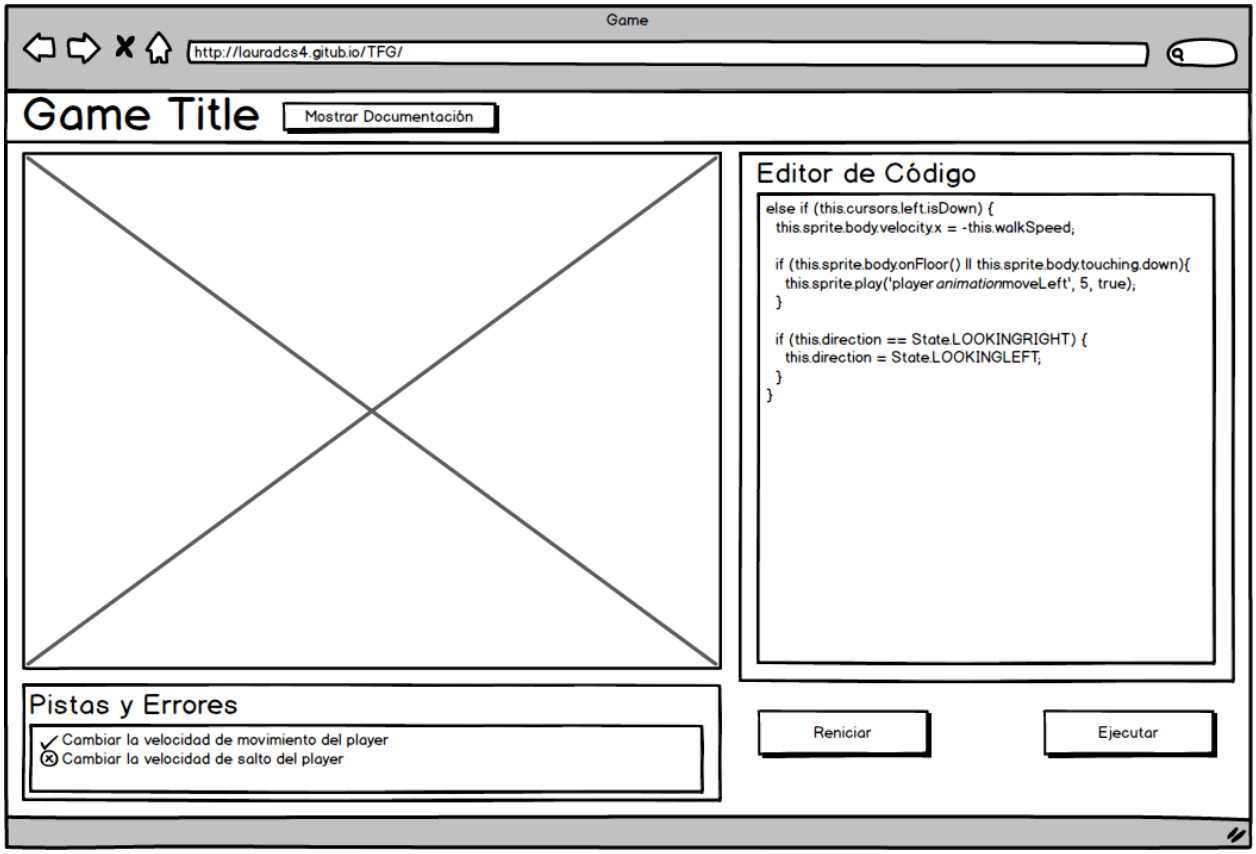


Figura 5.14: Prototipo de alto nivel

Una vez hemos realizado el prototipo en alto nivel, hemos pasado a implementarlo con *HTML5* y *CSS/CSS3* [30].

Para facilitarnos esta tarea, hemos decidido utilizar *Bootstrap*[13] para el estilizado de la aplicación. Nuestra máxima en todo momento ha sido el minimalismo. Para conseguirlo hemos utilizado colores planos y armónicos y el menor número de botones.

Como vemos en las siguientes capturas, el diseño se asemeja casi al 100% al prototipo de alto nivel diseñado con *myBalsamiq:*

En la Figura 5.15 vemos el menú principal del juego y el editor de código y los resultados de los test vacíos. También vemos la barra del menú con el título del juego (en el momento de la captura aún no teníamos un título, de ahí lo de *Game Title*):

[13] http://getbootstrap.com/

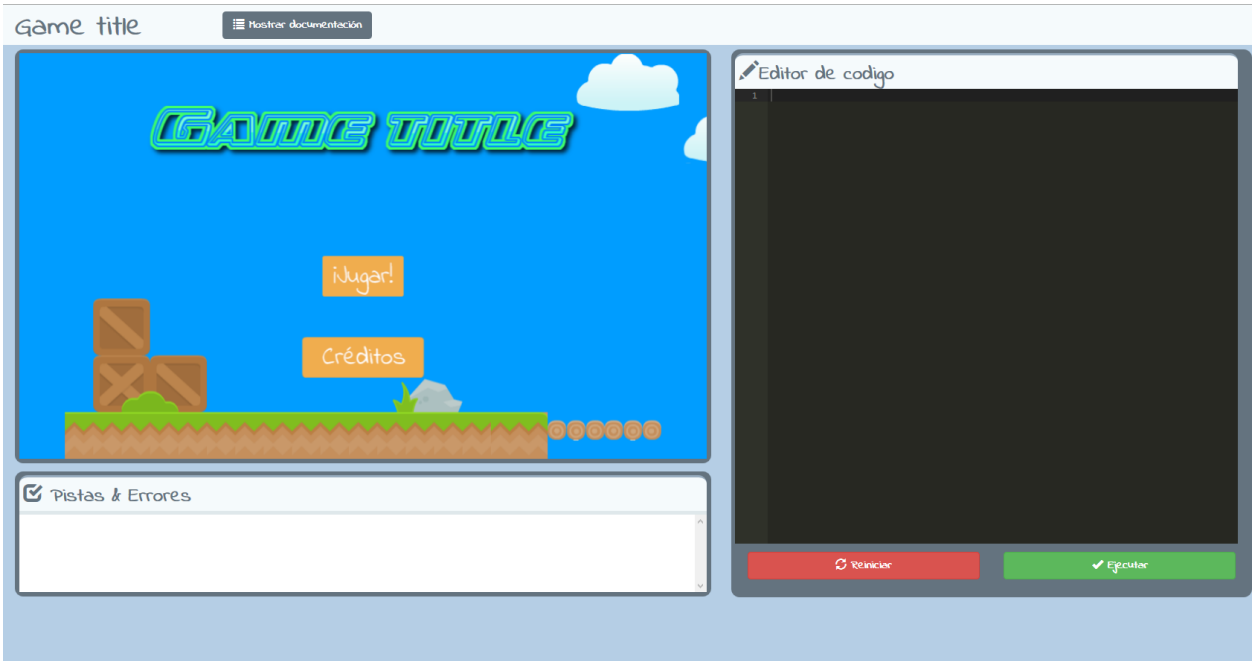


Figura 5.15: Captura de la pantalla de inicio

En la Figura 5.16 el jugador ya ha iniciado la partida y ha leído la pantalla de introducción a *Phaser.* Se muestra el primer diálogo entre Bersara y el jugador, en el que ella hace una pequeña introducción a la historia del juego y del por qué ha perdido ciertas facultades. También pide ayuda al jugador y le explica que tendrá que programar ciertas cosas usando el editor de código de la derecha de la pantalla:

Figura 5.16: Captura del juego

Por último, en la Figura 5.17, el jugador ha colisionado con el primer cartel, el juego se ha pausado, se ha lanzado el diálogo que explica el desafío y se ha cargado

el desafío en el editor para que el jugador lo resuelva. Además, el jugador ha pulsado una vez el botón ejecutar y, por eso, hay mensajes de los test.

Figura 5.17: Captura del juego, editor del código y test

5.2 Implementación del sistema de diálogos

Phaser ofrece un buen mecanismo para mostrar texto dentro del videojuego. Sin embargo, nos ha parecido que para hacer algo que quedara bien teníamos que dedicar demasiado tiempo y esfuerzo. Además, hemos leído información acerca de que *Phaser* tiene problemas para cargar fuentes externas al navegador y que, sin duda, dan un toque personal a un videojuego.

Hemos aprovechado nuestros conocimientos en *JQuery* para desarrollar una pequeña librería que nos permite cargar un elemento * * que muestra un diálogo cargado desde un fichero *JSON*. Incluso nos permite seleccionar dos imágenes que corresponden con los personajes que intervienen en la conversación.

Este sistema cuenta con dos botones: uno para avanzar en la conversación y otro para saltar el diálogo entero. Durante un diálogo la acción del juego queda pausada y el juego no volverá a la normalidad hasta que el diálogo finalice. A continuación, mostramos dos capturas de pantalla del resultado final:

La Figura 5.18 corresponde con el diálogo inicial. En todo momento es Bersara la que se comunica con el jugador y, como vemos, el jugador está representado con la imagen de un ordenador.

Figura 5.18: Captura de pantalla de un diálogo

En esta otra captura, en la Figura 5.19, mostramos el diálogo que aparece al finalizar el juego. Hemos suprimido la imagen del ordenador y Bersara aparece centrada. De esta forma nos dirigimos directamente a nuestros jugadores.

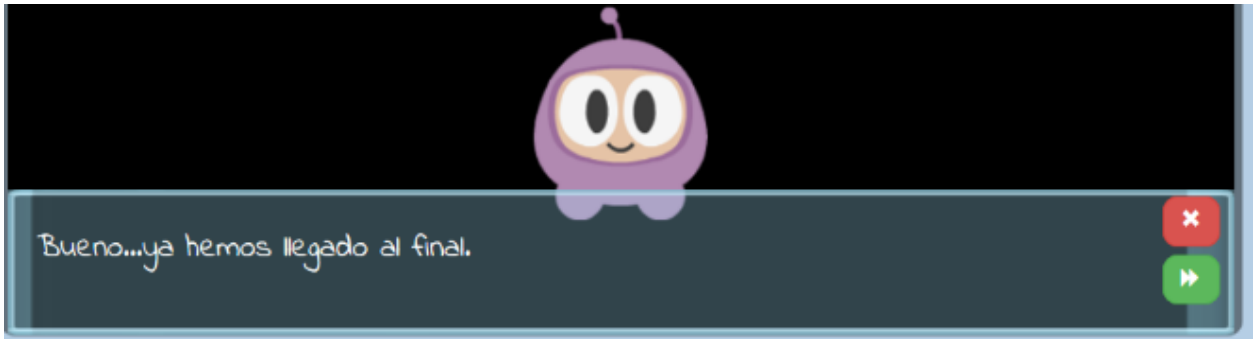


Figura 5.19: Captura de pantalla del diálogo final

5.3 Implementación del editor de código

En un principio hemos barajado la posibilidad de implementar el editor desde cero, con las funcionalidades que ya hemos especificado, pero esta es una tarea demasiado laboriosa para realizarla en el tiempo con el que contamos, por lo que hemos decidido buscar algún editor web que podamos integrar dentro de nuestra

herramienta, implementando alguna funcionalidad extra sobre este editor si fuese necesario.

Debido a la diversidad de editores de código web actuales, hemos decidido centramos entre los dos más populares: *CodeMirror* y *Ace*, siendo *Ace* el que hemos elegido por motivos que explicaremos más adelante. Antes de eso, profundicemos en cada uno de ellos.

5.3.1 CodeMirror

CodeMirror[14] es un editor de texto web implementado en *JavaScript*. Está especializado en la edición de código, ya que dispone de un número bastante amplio de modos que podemos elegir según el lenguaje que queramos utilizar y complementos que implementan funcionalidades más avanzadas que nos ayudan a escribir código más fácilmente. Podemos ver una captura de este editor en la Figura 5.20. Gracias a su *API* y a sus temas, escritos en *CSS*, el editor se puede personalizar para que encaje en cualquier aplicación. Algunas de sus características son:

- Tiene soporte para más de 100 lenguajes de programación.
- Dispone de diferentes modos según el lenguaje que queramos utilizar.
- Cuenta con autocompletado.
- Se pueden configurar distintas combinaciones de teclas para algunos comandos del editor.
- Dispone de temas que imitan algunos editores de código como *Sublime Text, Vim o Emacs*.
- Cuenta con una interfaz para buscar y reemplazar texto.
- Permite mezclar distintas fuentes y estilos dentro del mismo documento.

[14] https://codemirror.net/

```
(function($){
    'use strict';

    var Compose = function(element, options){
        options = options || Compose.defaults;

        this.markdown = options.markdown;

        this.selection = null;
        this.selecting = false;

        this.$element = $(element).attr('contentEditable', true);

        this.on = $.proxy(this.$element.on, this.$element);//now we can register event listeners directy through the
Compose object

        this.$toolbar = $('<menu>')
                        .attr('type', 'toolbar')
                        .addClass('compose-toolbar')
                        .css({
                            'position': 'absolute',
                            'top': 0,
                            'left': 0,
                            'display': 'none',
                        });
        $('body').append(this.$toolbar);

        $(document).on('mouseup', $.proxy(function(event){
            if (!this.selecting &&
                !$.contains(this.$element[0], event.target) &&
                !$.contains(this.$toolbar[0], event.target)){
                this.$toolbar.hide();
            }
            else{
                this.selectionEnd(event);
            }
            this.selecting = false;
        }, this));
```

Figura 5.20: Captura de pantalla de *CodeMirror*

5.3.2 Ace

Ace[15], al igual que *CodeMirror*, es un editor de código web escrito es *JavaScript* que cuenta con las características y funciones de editores de escritorio como *Sublime Text, Vim o TextMate*. Se puede integrar fácilmente en cualquier página web o aplicación. Algunas de sus características son:

- Tiene soporte para más de 110 lenguajes de programación, pudiéndose importar archivos *TML* provenientes de *Sublime Text* o de *TextMate*.
- Cuenta con más de 20 temas para personalizar el editor, permitiendo también importar temas desde *Sublime Text* o *TextMate*.
- Sangrado automática del código.
- Cuenta con una línea de comandos opcional.
- Se pueden personalizar las combinaciones de teclas. Cuenta con modos que proveen las combinaciones de teclas propias de *Vim* o de *Emacs*.
- Cuenta con una interfaz para buscar y reemplazar texto.
- Permite cortar, copiar y pegar texto.
- Es capaz de comprobar la sintaxis del código con algunos lenguajes, como *JavaScript* o *CSS*.

[15] http://ace.c9.io/

A continuación, en la Figura 5.21, podemos ver un ejemplo del editor *Ace* con código en su interior. Podemos apreciar el resaltado de las palabras clave y la presencia de los números de línea y de la barra de desplazamiento vertical.

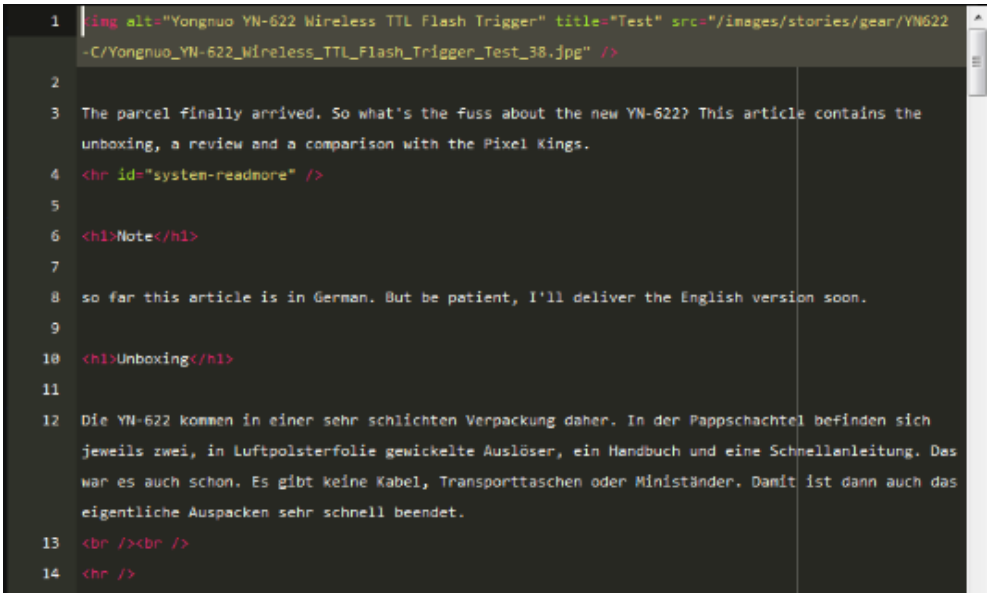


Figura 5.21: Captura de pantalla de *ACE*

5.3.3 Nuestro editor

Como podemos ver, ambos editores tienen características bastante similares, y cuentan con alguna de las funcionalidades básicas que necesitamos. No obstante, ninguno de los editores cuenta con todas las funcionalidades que queremos que tenga nuestro editor, como la posibilidad de bloquear código o poder reiniciar el contenido del editor, por lo que aunque nos basemos en uno de ellos, necesitamos ampliar su funcionalidad para lograr el resultado que queremos obtener.

Por este motivo nos hemos decantado por *Ace*, puesto que nos ha parecido más sencillo implementar estas nuevas funcionalidades con la *API* que nos proporciona. Además, el hecho de que *Ace* realice una comprobación de la sintaxis del código, en el caso de estar usando *JavaScript*, nos ha parecido muy interesante, ya que proporciona una ayuda extra al jugador.

En la parte superior del editor el jugador puede ver un título con el desafío que se está completando en ese momento, y además en la parte inferior cuenta con dos botones que el jugador puede usar para interactuar con el editor, y que implementan dos de las funcionalidades que necesitamos en nuestro editor:

- El botón **Reiniciar** devuelve el editor al estado inicial del desafío, es decir, borra el contenido actual del editor y fija de nuevo el código base que se muestra en el desafío que se está llevando a cabo en ese momento.

- El botón **Ejecutar** ejecutará el código. Para ello el código tendrá que pasar previamente el test de unidad correspondiente a ese desafío, y en el caso de pasarlos correctamente se ejecutará dentro del juego.

Además de esto nos falta una de las funcionalidades que hemos considerado más importantes y que, por otro lado, ha sido la más difícil de implementar con *Ace*: el bloqueo de código. Ha sido difícil de implementar debido a que hemos tenido que profundizar mucho en la API de *Ace* para encontrar una manera de implementar esta funcionalidad. *Ace* sí nos permite bloquear el código del editor pero con la particularidad de que sólo cuenta por defecto con la opción de bloquear todo el código, estableciendo el modo de sólo lectura en el editor. Como ya hemos explicado, esto no es lo que nosotros queremos, ya que nosotros queremos poder bloquear sólo ciertas partes del código que se muestra. En *Ace* esto se traduciría en que haya ciertas partes en modo de sólo lectura y otras en modo de lectura y escritura.

A pesar de esto, *Ace* cuenta con los mecanismos necesarios para implementar esta nueva funcionalidad. Por un lado tenemos un concepto fundamental, que son los rangos. En *Ace* podemos definir un rango utilizando un objeto de tipo *Range*. Cada rango representa una zona de código, delimitada por un inicio y un final. Tanto el inicio como el final no son más que dos coordenadas que representan una fila y una columna dentro del editor. Este es el mecanismo básico que nos va a permitir delimitar ciertas zonas de código dentro del editor. Crear un rango es algo tan sencillo como:

```
rango = new Range(2, 4, 4, 7);
```

Con esta línea estamos creando un rango que comienza en la fila 3, carácter 4, y termina en la fila 5, carácter 7. Es muy importante tener en cuenta que en *Ace* las filas y las columnas se empiezan a numerar en cero y no en uno.

Otro aspecto importante es dar a la zona de código que queremos bloquear un estilo diferente mediante el sombreado de las líneas, para que el jugador pueda distinguir visualmente qué zona de código está bloqueada y qué zona no. Esto lo conseguimos con marcadores. En *Ace* podemos crear un objeto de tipo *Marker*, que asigna a un objeto de tipo *Range* un estilo determinado. Este estilo puede estar definido en un archivo *CSS*. De este modo podemos darle a la zona que queramos bloquear, por ejemplo, un color de fondo naranja con algo de transparencia:

```
marcador = session.addMarker(rango, "readonly-hightlight");
```

Le asignamos como clase *readonly-highlight*,y de esta manera obtiene el estilo que hayamos definido para esa clase en el archivo CSS.

```
.readonly-highlight{
        background-color: #00AAAA;
        opacity: 0.1;
        position: absolute;
}
```

Por ahora hemos creado un rango y le hemos dado un estilo, pero nos queda lo más importante, que es fijar ese rango dentro del editor, es decir, dentro del documento que se está editando, para poder bloquear la entrada del usuario en esa zona. Esto también podemos hacerlo con un mecanismo que nos proporciona *Ace* que son los objetos de tipo *Anchor*. Para cada rango tenemos que crear dos objetos de tipo *Anchor*: uno en el inicio del rango y otro en el final. Esto es lo que de verdad delimita una zona dentro del editor.

```
rango.start = session.doc.createAnchor(rango.start);
 rango.end = session.doc.createAnchor(rango.end);
```

Con esto hemos creado finalmente una zona delimitada dentro del editor, sobre la que podemos llevar a cabo distintas acciones, como bloquear en ella los eventos de teclado para que no pueda editarse, que es lo que hemos hecho en nuestro caso.

```
editor.keyBinding.addKeyboardHandler( {
        handleKeyboard : function(data, hash, keyString, keyCode,
event) {

            if (hash === -1 || (keyCode <= 40 && keyCode >= 37)) {
                    return false;
            }

            if (intersects(rango)) {
                return { command:"null", passEvent:false };
            }
        }
 });
```

En la Figura 5.22 podemos ver el resultado final. Como ya hemos comentado, podemos ver en la parte superior el título del desafío que se está completando. Al área bloqueada le hemos dado un sombreado en verde oscuro, mientras que el resto sigue manteniendo el mismo estilo. El área sin sombrear es la zona donde el jugador debe escribir el código. Hemos incluido muchos comentarios en el código para facilitar un poco la resolución de los desafíos, puesto que la resolución de algunos de ellos no es trivial. También podemos ver en la parte inferior los dos botones para reiniciar y ejecutar el código.

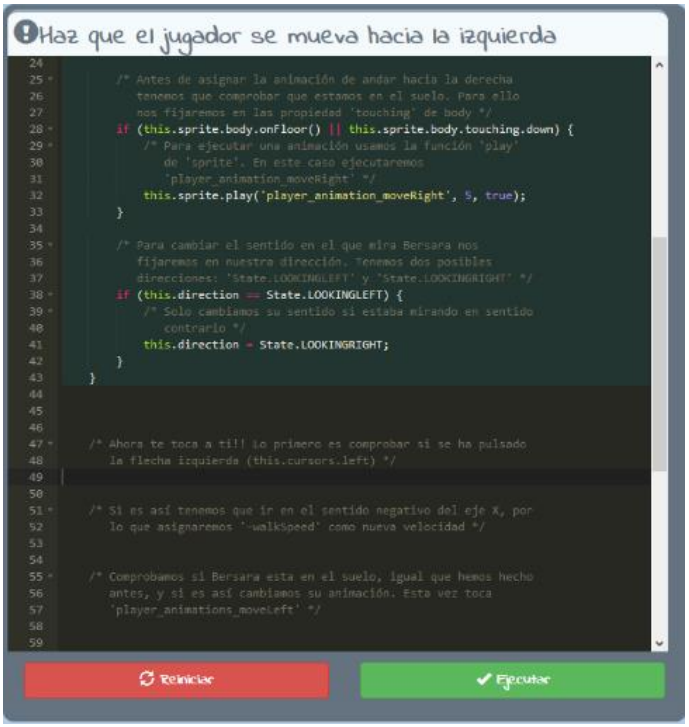


Figura 5.22: Editor de código

5.4 Implementación de los test de unidad

A lo largo de nuestra investigación hemos dado con herramientas para realizar test de unidad sobre un código de *JavaScript*, como *QUnit* [31]. Debido a que *QUnit* tiene algunas de sus funcionalidades obsoletas y muestra demasiada e innecesaria información para nosotros, hemos decidido realizar nuestro propio test de unidad al que hemos llamado *TWUnit*[16]. *TWUnit* es una herramienta para realizar test sobre código *JavaScript*.

Vamos a ver cómo funciona TWUnit con el test que hemos realizado para comprobar que el jugador puede saltar hacia la derecha:

Para ello lo primero que tenemos que realizar es crearnos una instancia de nuestra herramienta de tests de unidad, TWUnit. Al crearnos esta instancia, lo que hará TWUnit por debajo es crear una lista en la que podremos ir insertando tantos asserts como queramos.

```
tw = new TWUnit();
```

Para comprobar si el usuario ha tenido errores a la hora de escribir el código, tenemos que comprobar que este código es sintácticamente correcto y, si es así, ejecutar la función “jump” que con los tests de unidad comprobaremos si lo ha

[16] https://github.com/ZaruWright/TWUnit

realizado correctamente. En el caso de que no sea sintácticamente correcto saltará una excepción que la capturamos en el método que ha llamado a esta función.

```
eval(currentTask.target + "=" + text);

// Saltar mirando hacia la derecha
player.cursors.up.isDown = true;
player.jump();
player.cursors.up.isDown = false;
```

Una vez hemos ejecutado la función que el usuario ha implementado, tenemos que analizar los resultados de las variables para ver si ha realizado el desafío correctamente, mostremos un ejemplo de cómo construirlo con la siguiente instrucción.

```
tw.addAssert("Parece que no has puesto la velocidad correcta. La
velocidad que tienes que asignar es 'this.jumpSpeed'",
player.sprite.body.velocity.y == player.jumpSpeed,
"",
"");
```

Con el método addAssert() añadimos un assert a la lista mencionada anteriormente. Los campos de un assert son:

- Nombre del assert: nombre del test que se va a añadir.
- Condición: expresión booleana que queremos que verifique el assert.
- Comentario: texto que se mostrará si y sólo si la condición es cierta.
- Pista: texto que se mostrará si y sólo si la condición es falsa.

Vistas las características de un assert con TWUnit, el assert que insertamos en el test del salto contiene: la pista de lo que debería de hacer el usuario para completar esta partre del test satisfactoriamente, y la condición que tiene que cumplir, que en este caso se comprueba que la velocidad en el eje ‘y’ del jugador sea la misma que la variable global “player.jumpspeed” que debería de utilizar el usuario para completar esta parte del test. De esta manera si queremos que nuestro test tenga más condiciones que cumplir, basta con añadir más instrucciones de este tipo.

Una vez hayamos agregado todos los asserts que queremos evaluar en éste test, ejecutamos el método runAsserts() para establecer un valor en ellos: cierto o falso. Esta función va recorriendo toda la lista de asserts comprobando sus condiciones y mostrando el nombre del assert, y el nombre de comentario y pista ,en el caso de que tengan, en nuestro html. Si la condición ha sido correcta, TWUnit mostrará un símbolo de check verde a modo de que el test se ha realizado correctamente. En el caso contrario se muestra una equis en rojo. Si queremos utilizar la visualización de nuestros test en nuestro html tenemos que incluir el siguiente div para que funcione correctamente.

```
<div id="twunit"></div>
```

TWUnit presenta más funcionalidad de la que explicamos aquí, ya que con lo que hemos contado nos ha servido para realizar nuestros tests de unidad, pero vamos

a resaltar lo que nos parece más interesante. Si os habéis fijado, en los asserts, no hemos introducido dos de los cuatro campos que TWUnit nos proporciona a la hora de insertarlos, la razón es debido a que como TWUnit muestra de una manera gráfica los asserts que han sido correctos y los que no, decidimos obviar la parte de pistas y comentarios y poner directamente las pistas como nombre del assert, para que de esta manera, solo salga la información necesaria para poder completar un assert.

La otra funcionalidad que contiene esta herramienta de tests de unidad, es que puedes manejar varios tests a la vez y no solo uno. No hemos escogido esta característica ya que nos es irrelevante controlar varios a la vez, nos basta con tener un solo test al que podamos ir añadiéndole asserts para más tarde comprobar si son ciertos o no.

Por último, esta herramienta tiene una licencia MIT y, por lo tanto, es libre de utilizarse para otros proyectos y ser modificada. Las Figuras 5.23 y 5.24 muestran los resultados de ejecutar un test sobre un desafío:

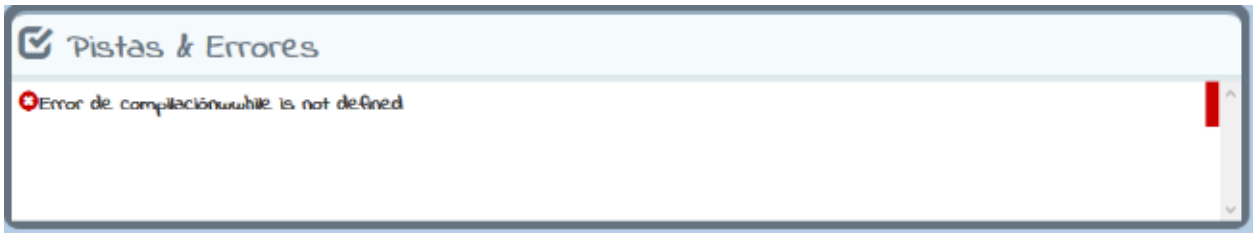


Figura 5.23: Test de Unidad sobre un desafío (I)

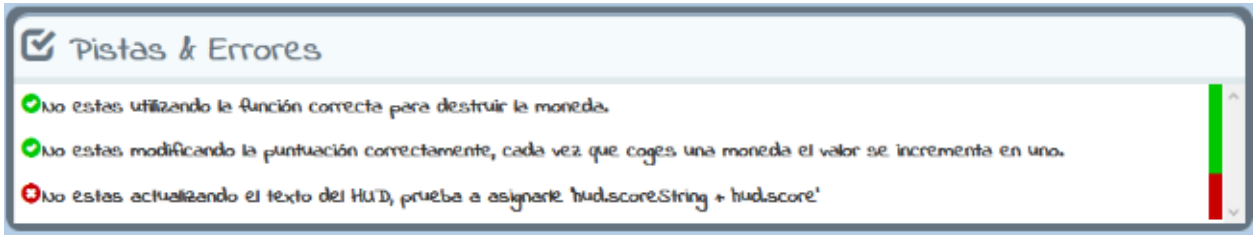


Figura 5.24: Test de Unidad sobre un desafío (II)

5.4.1 Implementación de los desafíos

Una vez implementados el editor de código y los test de unidad, vamos a ver cómo hemos implementado los desafíos que deberán resolver los jugadores para aprender.

En primer lugar hemos necesitado situar un objeto en el mapa del juego que sea relacionado por el jugador como el comienzo de un desafío. Ese objeto es un cartel de madera que, delante de él, tiene un objeto vacío que actúa como *trigger* o disparador de eventos. Cuando Bersara colisiona con un cartel el juego entra en

pausa y obliga al jugador a completar satisfactoriamente el desafío que se haya lanzado para poder continuar con el juego.

Un desafío se compone de los siguientes elementos:

- **Título** del desafío, que se mostrará como título encima del editor de código.
- **Código** del desafío. Este código se mostrará en el editor, tendrá comentarios para ayudar al jugador.
- **Función objetivo**: función o método que será reescrita con el código que ha escrito el jugador, una vez haya superado todos los test.
- **Primer rango** de valores para bloquear el código.
- **Segundo rango** de valores para bloquear el código.
- El **nombre de la función** que realizará el test para verificar el desafío.
- **Número de línea** en la que queremos que se sitúe el cursor.

Todos los desafíos estarán almacenados en un fichero *JSON.* En el Anexo 2 tenemos el ejemplo del desafío “Hacer que Bersara camine a la izquierda”. Cuando cargamos los datos del juego, se carga la información de este fichero a cada uno de los *triggers* del juego según corresponda. Cuando Bersara colisiona con uno de estos *triggers* el juego se detiene, se carga el desafío correspondiente en el editor, se destruye el *trigger* para evitar bucles y comportamientos raros; y se muestra el diálogo que explique al jugador lo que tiene que hacer. También se deshabilita en *Input* del teclado en *Phaser* para que el motor deje de capturar ciertas teclas y atajos de teclado. Una vez el jugador ha escrito el código y pulsa sobre el botón de ejecutar se ejecuta por debajo la función *submitCode():*

```
submitCode: function() {

    if(currentTask) {
        // 1- Si tenemos una tarea pendiente obtenemos el código del editor
        var text = editor.getValue();

        // 2- Habilitamos los eventos para poder lanzar eventos virtuales
        this.game.input.disabled = false;
        clearListeners.call(this);


        // 3- Testear el código
        try{
```

```
            var originFunction = eval(currentTask.target );
            if (eval(currentTask.test)) {
                eval(currentTask.target + "=" + text);
                editor.getSession().setValue("", -1);
                $('#task').html("<h3><span class=\"glyphicon glyphicon-
                      pencil\"></span>Aqui va la tarea</h3>");
                currentTask = null;
                TWUnit.HtmlInteract.htmlClear();
                testClear.call(this);
            }
            else {
                this.game.input.disabled = true;
            }
        }
        catch(e){
            console.log(e);
            tw = new TWUnit();
            eval(currentTask.target + "=" + originFunction);
            tw.addAssert("Error de compilación", true == false, "", e.message);
            tw.runAsserts();
            this.game.input.disabled = true;
        }

    }

}
```

En primer lugar se carga el código del editor:

```
var text = editor.getValue();
```

Después se habilitan los eventos del *Input* del teclado y se vacía la cola de eventos de teclado, por si se ha quedado alguno:

```
this.game.input.disabled = false;
    clearListeners.call(this);
```

Ejecutamos el test del desafío:

```
if (eval(currentTask.test)) {
```

Si el código produce alguna excepción no controlada por los test, por ejemplo acceder a un campo *null* o a una posición inexistente de un *array*, se lanzará una excepción que deberá ser capturada por el programa principal y se mostrará por la consola del navegador:

```
catch(e){
    console.log(e);
    tw = new TWUnit();
    eval(currentTask.target + "=" + originFunction);
    tw.addAssert("Error de compilación", true == false, "",
e.message);
    tw.runAsserts();
```

```
        this.game.input.disabled = true;
    }
```

A modo de ejemplo, en el Anexo 3 tenemos el código completo de cómo hemos realizado el test del primer desafío: caminar a la izquierda. Una vez completado el desafío aparecerá en el juego una estrella. Esta es la manera de darle *feedback* al usuario que hemos elegido.

5.5 Implementación de la documentación

Como ya hemos mencionado anteriormente, una de las características que más nos ha gustado de *Phaser* es que es un proyecto en continua evolución. Cuando comenzamos a desarrollar el videojuego utilizamos la versión 2.0.6. Cada vez que liberaban una actualización no dudábamos en migrar nuestro proyecto a ella. Sin embargo, hubo un punto de inflexión: tras la publicación de la versión 2.1.3 se produjeron numerosos cambios en el código del *framework* que cambiaban radicalmente el código del juego que ya teníamos desarrollado. Esto tiraba por tierra casi todo lo que ya teníamos desarrollado y, por ello, decidimos dejar nuestro juego en la versión 2.1.2.

Nuestra idea original era dejar libertad al usuario para que buscara en la documentación oficial de *Phaser* las clases, funciones y métodos que ofrece el *framework*. Ya fuera abriendo una pestaña del navegador que complementara a otra en la que ejecutara el juego, o abriendo la documentación con otro navegador. Sin embargo, la web de *Phaser* siempre ha mantenido activa la última versión de su documentación, que corresponde con la última versión de *Phaser* publicada.

Como nuestro proyecto ha sido desarrollado en la versión 2.1.2 por el motivo que hemos comentado, hemos tenido que buscar una solución a esto. Hemos decidido descargar la documentación de *Phaser* correspondiente a esta versión desde el repositorio oficial de *Github* y alojarla junto con el resto del proyecto.

Al final esto no ha resultado ser un inconveniente, ya que hemos decidido mostrar la documentación en la misma vista que el resto de la aplicación, es decir, sin usar pestañas adicionales u otro navegador. Para ello hemos utilizado el elemento *<iframe>* que nos ofrece *HTML5*, en detrimento del antiguo *<frame>.* Podemos ver el resultado en las capturas que mostramos a continuación.

En la Figura 5.25 vemos la ventana de la documentación con la página inicial cargada. En la parte izquierda vemos las clases de *Phaser* y en la derecha tenemos un índice:

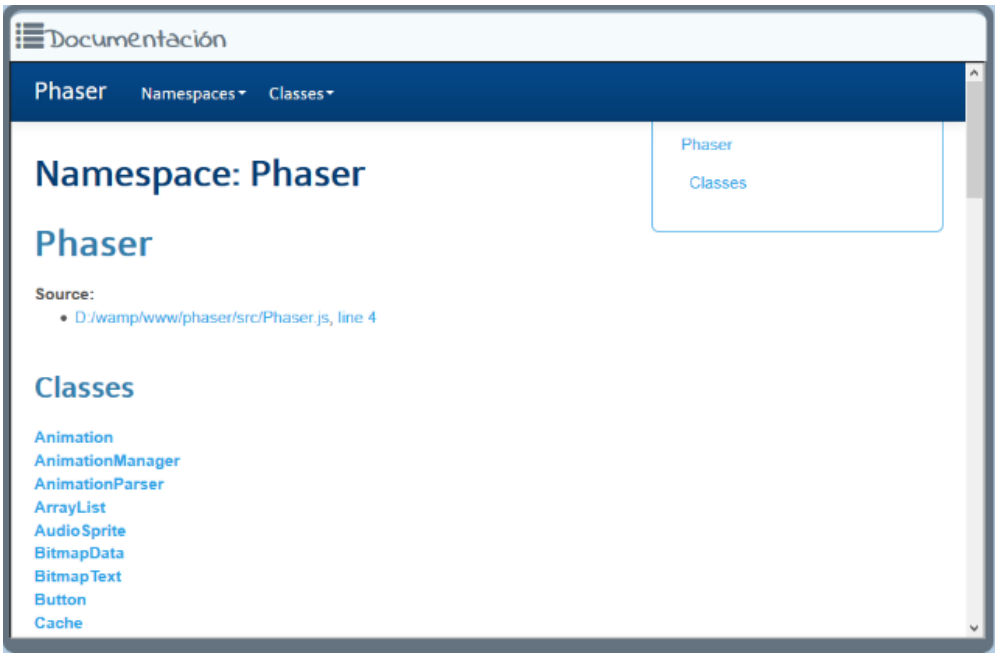


Figura 5.25: Documentación (I)

En esta otra figura, la Figura 5.26, vemos la ventana de la documentación con la clase *Phaser.Sprite* cargada. Vemos los detalles de la función *destroy*: descripción, parámetros y lugar del código fuente donde aparece definida (por si queremos editarla). En la zona derecha volvemos a ver un menú que hace las veces de índice de esta página:

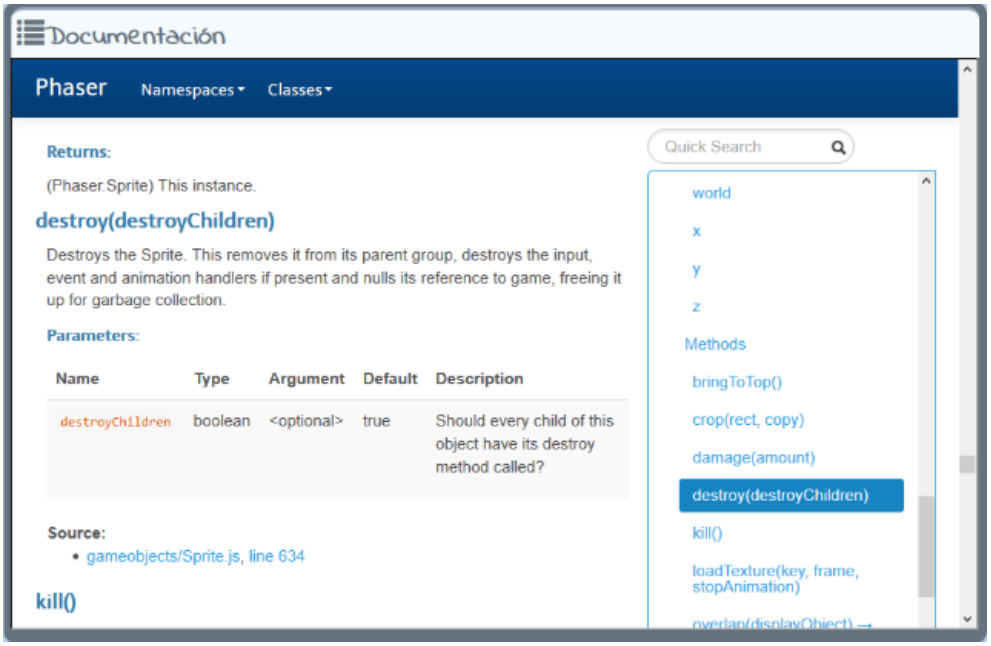


Figura 5.26: Documentación (II)

5.6 Implementación del videojuego

Phaser nos ha proporcionado una manera sencilla de organizar las fases de nuestro juego en estados en los que podremos ir pasando de uno a otro cuando los vayamos completando. Los estados que hemos implementado para nuestro videojuego son:

- **Boot**: En este estado cargamos el *sprite* que necesitamos para mostrar la barra de cargando, le damos una física a nuestro juego, ponemos el color del fondo en negro y pasamos al estado **Preload**.

- **Preload**: Aquí cargamos todos los recursos que vamos a necesitar en nuestro videojuego, tales como los mapas que utilizaremos y los assets de escenarios, enemigos, objetos etc. Mientras cargamos los recursos se mostrará una barra la cual nos da *feedback* de cuanto queda para el juego empiece. Cuando la barra termine de cargarse pasaremos al estado **MainMenu**. Podemos ver el resultado en la Figura 5.27.

Figura 5.27: Preload

- **MainMenu**: Este estado representa la pantalla de título del videojuego, donde podremos realizar dos acciones, jugar a nuestro juego o ver los créditos, cada uno de estas acciones son dos botones diferentes donde nos llevaran a sus respectivos estados, **AboutPhaser** o **ClosingCredits**. Podemos ver el resultado en la Figura 5.28.

Figura 5.28: Menu principal

- **AboutPhaser**: Tras pulsar el botón “jugar” en la pantalla de título, explicamos qué es *Phaser* y los conocimientos básicos de la arquitectura de un videojuego. Todo esto lo hacemos con la implementación de los diálogos que veremos más adelante. Tras acabar la explicación, el estado **Game** empezará. Podemos ver el resultado en la Figura 5.29.

Figura 5.29: Acerca de *Phaser*

- **ClosingCredits**: Tras pulsar el botón “créditos” en la pantalla de título, mostramos los integrantes del grupo que hemos hecho el videojuego, de dónde hemos obtenido alguno de nuestros *assets* y los agradecimientos. Tras ver los créditos regresamos al estado **MainMenu**. Podemos ver el resultado en la Figura 5.30.

Figura 5.30: Créditos

- **Game**: Este estado es el corazón de nuestro videojuego, en el cual creamos el nivel que vamos a jugar, el jugador, el *HUD* y los eventos del editor de código. También constará del bucle principal de nuestro videojuego. Podemos ver el resultado en la Figura 5.31.

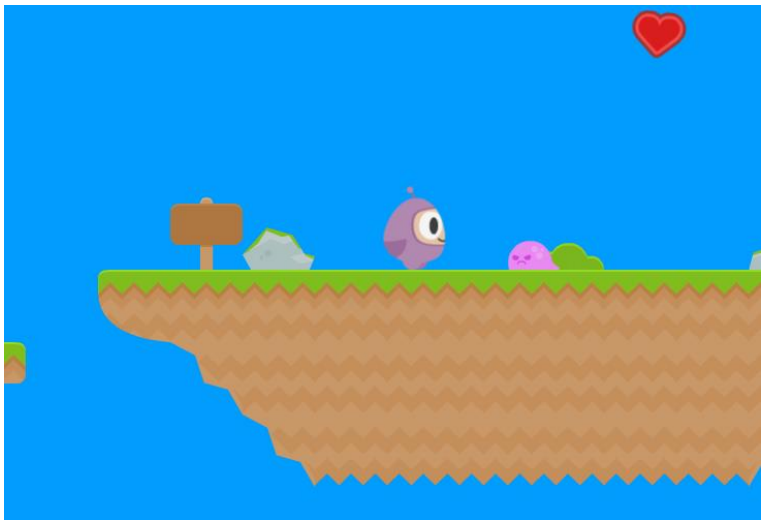


Figura 5.31: El juego

- **EndGame**: Dialogo final que cuando termina volvemos al estado **MainMenu**. Podemos ver el resultado en la Figura 5.32.

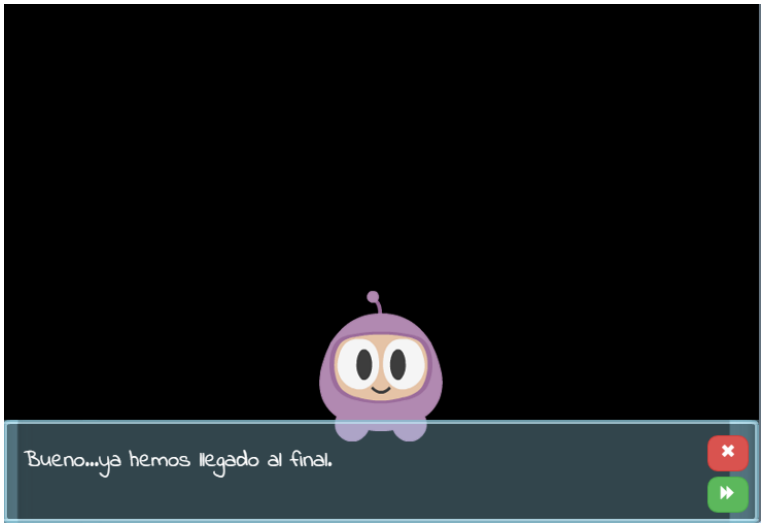


Figura 5.32: Final del juego

Pasemos ahora a explicar algunas de las clases más importantes:

La clase **Game** tiene dos métodos particulares de *Phaser*: *Create*, donde inicializamos el nivel, el jugador y los eventos de los botones; y *Update*, función que *Phaser* irá llamando periódicamente para comprobar los cambios del juego.

La clase **Player** consta también de los métodos *Create* y *Update*. En el método Create creamos al personaje asignándole físicas al cuerpo, animaciones, controles de entrada de teclado, fijar la cámara en el jugador etc. Mientras que en la función *Update* comprobaremos si el jugador ha chocado con algún enemigo, objeto o bloques vacíos (explicaremos más adelante para que sirven este tipo de objetos) y que el jugador se mueva y salte.

En la clase **Level** la función *Create* se encargará de cargar el mapa que hemos creado en *Tiled.* Dentro de esta función podemos destacar la manera de cargar las distintas capas de *Tiled*. En *Tiled* existen dos tipos de capas, la capa de baldosas y la capa de objetos. La capa de baldosas nos permite poder crear nuestro escenario a partir de los *assets* que utilicemos, mientras que la capa de objetos nos permite poder cargar después estos *assets* en el código pudiendo crear objetos de ellos para poder manejarlos. De esta manera para cargar el mapa de la capa de baldosas haremos lo siguiente:

```
this.map = this.game.add.tilemap('level1');
```

Con esto creamos un objeto *Tilemap* cargando nuestro mapa realizado en *Tiled.* Este mapa ha sido precargado en el estado *Preload* de la siguiente manera:

```
this.game.load.tilemap('level1', 'assets/maps/level1.json', null,
                    Phaser.Tilemap.TILED_JSON);
```

Después tendremos que indicar que *assets* hemos utilizado para construir nuestro mapa y añadírselos de la siguiente manera:

```
    this.map.addTilesetImage('items-sheet', 'items-sheet');
    this.map.addTilesetImage('buildings-sheet', 'buildings-
sheet');
    this.map.addTilesetImage('grass-sheet', 'grass-sheet');
    this.map.addTilesetImage('void-block', 'void-block');
    this.map.addTilesetImage('snail', 'snail_spritesheet');
    this.map.addTilesetImage('slime', 'slime_spritesheet');
```

Donde el primer argumento es el nombre que hemos utilizado en *Tiled* para nombrar a ese conjunto de *assets* y el segundo es el nombre que le hemos dado en *Phaser*. Si queremos establecer colisiones con algunos de los objetos del mapa, podremos hacerlo con rangos:

```
this.map.setCollisionBetween(2,50);
```

O diciendo exactamente el id del *sprite* con el cual queremos colisionar:

```
this.map.setCollision([82 // Caja con una equis dibujada
                       ]);
```

Por último creamos la capa con el nombre utilizado en *Tiled* y asignamos el tamaño del mundo para cuadrarlo con el tamaño de la capa.

```
this.layer = this.map.createLayer('Tile Layer 1');
this.layer.resizeWorld();
```

Para cargar objetos de la capa de objetos, valga la redundancia, haremos lo siguiente. En el caso de los enemigos, nos creamos un grupo porque tenemos varios de ellos en el mapa y nos interesa tenerlos juntos. Al crear el grupo le decimos que todos los elementos de este grupo van a tener la física ARCADE y que van a ser cuerpos:

```
slimes = game.add.group();
slimes.physicsBodyType = Phaser.Physics.ARCADE;
slimes.enableBody = true;
```

Por último, para cargar los *Sprites* creados en *Tiled* en su capa de objetos, debemos de utilizar la función *createFromObjetcs()* de la siguiente manera:

- El primer argumento será el nombre de la capa de objetos que hemos utilizado en *Tiled.*

- El segundo argumento que debemos especificar es el número de id que utiliza *Tiled* para nombrar cada uno de los Sprites utilizados en el mapa. Es decir, en nuestro ejemplo queremos cargar el enemigo Slime, que será un *Sprite* que hayamos introducido en *Tiled* al cual le dará un número de id para identificarlo y poder construir el mapa.

- El tercero será el nombre de la clave que hemos utilizado en el estado Preload para cargar la imagen del *Sprite* correspondiente, en este ejemplo la del enemigo Slime.

- Si la imagen del *Sprite* contiene varios *frames*, en la cuarta opción puedes especificar el número de *frame* que quieres usar.

- Por último, el último argumento que nos interesa es el último que ponemos, el séptimo, en el cual podremos especificar el nombre del grupo al que queremos añadir los objetos que carguemos.

```
        level.map.createFromObjects('Object Layer 1', tiledId.slimeId
,       'slime_spritesheet', 0, true, false, this.slimes);
```

Para hacer el código más legible hemos realizado una clase llamada *tileId* en el cual especificamos el número de id que tiene cada objeto, enemigo o cualquier otra cosa que carguemos como objeto.

Para acabar con la clase **Level**, tenemos la función *Update* que comprobará periódicamente las funcionalidades de los enemigos y la de los objetos que hemos cargado en la capa de objetos.

Utilizamos la clase **HUD** para almacenar la información del jugador, como las vidas y los puntos que tiene y, además, esta clase la muestra por pantalla de una manera gráfica. Utilizando los *tweens* de *Phaser* tenemos una cantidad de corazones que indican de manera visual cuántas vidas le quedan y, por otro lado, tenemos un texto en el cual mostramos la cantidad de monedas que ha recogido el usuario.

5.8 Conclusiones

En un primer momento decidimos utilizar herramientas existentes para alcanzar los propósitos que hemos definido en este apartado. No obstante y por diversas causas, hemos tenido que desarrollar por nuestra cuenta tanto la implementación de la documentación como los test de unidad.

El caso del editor de código ha sido diferente, puesto que la herramienta que hemos encontrado por Internet, *Ace,* ha satisfecho nuestras necesidades.

Para el diseño de la interfaz de la aplicación hemos utilizado parte de los conocimientos que hemos adquirido en la asignatura Desarrollo de Sistemas Interactivos. Hemos comenzado dibujando un prototipo en papel, también conocido como *mockup,* luego hemos pasado ese *mockup* junto con los cambios que hemos creído oportunos a un prototipo de alto nivel. Por último, hemos implementado dicho prototipo en *HTML*, *CSS/CSS3* y *Bootstrap*.

El siguiente paso consiste en evaluar el prototipo que hemos implementado con usuarios. Esto se recoge en el apartado siguiente.

6. Evaluaciones con usuarios

Una vez hemos desarrollado nuestra aplicación, hemos querido evaluarla para saber si cumple los objetivos que nos hemos marcado inicialmente. Queremos saber si realmente sirve para aprender las bases para programar un videojuego, si la interfaz es agradable para los usuarios, si les gusta utilizarla, si los test de unidad les ayudan y si emplean la documentación.

En primer lugar describiremos esto más formalmente con un plan de evaluación, luego analizaremos los resultados que hemos obtenido y, por último, obtendremos conclusiones de los mismos.

6.1 Plan de Evaluación

El objetivo de este apartado es describir formalmente el proceso de evaluación que hemos seguido. Una vez realizada la evaluación, hemos analizado los resultados para obtener conclusiones y detectar errores y fallos de diseño.

6.1.1 Propósito de la evaluación

El propósito de la evaluación es resolver el mayor número de errores o fallos de diseño que tenga nuestra aplicación. Además, queremos saber lo que más y lo que menos les ha gustado a los usuarios.

6.1.2 Objetivos generales

Hemos realizado una evaluación en grupo en un laboratorio de la Facultad de Informática de la Universidad Complutense de Madrid, en la que nos hemos centrado en la usabilidad de los siguientes objetivos:

- Evaluar que la aplicación funcione correctamente y no se encuentre ningún fallo de diseño que pueda llegar a ser relevante.

- Comprobar que el usuario pueda utilizar sin ningún problema la aplicación.

6.1.3 Preguntas de investigación

Las preguntas de investigación serán respondidas por los usuarios cuando finalicen la evaluación. Las preguntas son las siguientes:

- Número de desafíos que han completado.
- Que es lo que más les ha gustado.
- Que es lo que menos les ha gustado.
- Si les ha resultado útil o no la documentación.
- Si les ha resultado de ayuda o no los mensajes de los test de unidad.
- Si les ha resultado de ayuda o no los comentarios del código.

Junto a estas cuestiones, nuestros usuarios evaluados también responderán a las cuestiones que plantea el cuestionario SUS. El objetivo de esto es saber formalmente si la aplicación que hemos implementado cumple con los principios de usabilidad.

6.1.4 Requisitos para los participantes

Nuestros usuarios se han dividido en dos grupos:

- Personas con conocimientos en *JavaScript*, pero sin experiencia en la programación de videojuegos.
- Personas con conocimientos en *JavaScript*, con experiencia en la programación de videojuegos.

Todos ellos con una edad comprendida entre los 18 y 25 años.

Hemos averiguado el grado de conocimiento de nuestros usuarios sobre *JavaScript* y la programación de videojuegos con un formulario de *Google*.

Para los participantes de nuestra encuesta nos hemos centrado en estudiantes de la Facultad de Informática de la Universidad Complutense de Madrid que cumplan el perfil descrito anteriormente.

6.1.5 Diseño experimental

- La evaluación se ha realizado el día 21 de Mayo en el laboratorio 6 de la Facultad de Informática de la UCM.

- Duración total de la evaluación 40 minutos: 25 minutos para la prueba con el prototipo y 15 para responder a los cuestionarios.

Estos son los pasos que hemos seguido durante la misma:

1. Hemos dado una breve introducción de la evaluación y hemos facilitado la dirección del sitio web con el cuestionario de *Google Forms* y el enlace a nuestra aplicación.

2. Después se ha dejado a cada usuario que interactuara libremente con la aplicación, siendo el objetivo llegar al final del primer nivel.

3. Los tres hemos sido observadores y hemos tomado notas durante el proceso.

4. Una vez finalizado el tiempo de la evaluación, haya o no terminado el primer nivel, cada usuario ha completado el cuestionario SUS y ha respondido a las preguntas de investigación que hemos descrito anteriormente.

5. Procesamiento de datos y toma de conclusiones.

6.1.6 Entorno y herramienta de pruebas

Hemos realizado las evaluaciones en el laboratorio 6 de la facultad de informática. Los tres hemos hecho de observadores y moderadores. La herramienta utilizada ha sido *Google Chrome* y la aplicación estuvo alojada en un servidor al que se accedió a través del siguiente enlace: http://lauradcs4.github.io/TFG/

6.1.7 Obtención de feedback de los participantes

Los participantes han completado un formulario de *Google Forms* con preguntas iniciales acerca de su nivel de conocimiento sobre *JavaScript* y la programación de videojuegos, el cuestionario SUS y las preguntas de investigación.

6.1.8 Tareas del moderador

Pese a que los tres hemos hecho de moderadores, solo uno de nosotros ha dado la bienvenida a los usuarios y les ha explicado los objetivos y detalles de funcionamiento de proceso de evaluación.

Creemos que es buena idea llevar a cabo la técnica de *think-aloud*[17] para saber lo que el usuario piensa en cada momento durante la evaluación. Sin embargo, dada la naturaleza de la evaluación que hemos realizado, ha sido imposible llevarla a cabo.

Por otro lado, los moderadores hemos contado con un cuaderno en el que hemos escrito todos los detalles que hemos considerado relevantes durante la evaluación. Hemos realizado las siguientes tareas:

1. Dejar que los usuarios probaran nuestra aplicación teniendo un tiempo límite de 25 minutos para superar el nivel.

2. No ayudarles a excepción de que el usuario se quedara muy atascado en un desafío.

3. Medir cuánto tiempo tardaban en realizar cada desafío, aproximadamente.

6.1.9 Identificar los datos que se van a recolectar

Una vez completada la evaluación, nuestros evaluados han contestado a unas preguntas a través de un formulario de *Google Forms.* Además, hemos realizado un *debriefing* con algunos de los usuarios en el cual hemos contestado todas sus preguntas y les hemos pedido que explicaran cualquier detalle adicional que quisieran compartir con nosotros.

Los datos que hemos recogido han sido:

- Número de desafíos completados.

- Lo que más les ha gustado.

- Lo que menos les ha gustado

[17] Método utilizado para la demostración de la usabilidad de un producto informático que consiste en que el evaluado exprese con palabras lo que está haciendo durante la evaluación de dicho producto.

- Si les han servido de ayuda la documentación, los mensajes de los test de unidad y los comentarios en el editor de código.

6.1.10 Descripción de la metodología de análisis de datos

1. Al finalizar la evaluación, como ya hemos mencionado, los usuarios han completado el cuestionario SUS y han respondido a las preguntas iniciales y a las preguntas de investigación.

2. Después hemos organizado los resultados de las encuestas en forma de gráficas, hemos calculado la nota del cuestionario SUS y hemos obtenido conclusiones a partir de ellas.

6.2 Resultados de las evaluaciones

Una vez realizada la evaluación en grupo hemos procedido a organizar todos los datos obtenidos, es decir, los resultados de las encuestas y los datos que nosotros mismos, como moderadores, hemos podido recoger a través de la evaluación. Las respuestas de cada una de las encuestas realizadas se pueden encontrar en el Anexo 4.

6.2.1 Resultados de los cuestionarios

Según las instrucciones del cuestionario SUS metiendo los valores numéricos a las respuestas del cuestionario, los cuales podemos ver en la Tabla 6.5, que hemos realizado obtendremos un valor numérico. Como vemos en la Figura 6.33, en función de este valor se obtendrá una nota comprendida entre la F y la A, siendo la F la nota más baja y la A la más alta. Para obtener este valor tenemos que adecuarnos a una serie de normas impuestas, a saber (x es el valor respondido por el usuario, de 1 a 5):

- En las preguntas impares: x - 1
- En las preguntas pares: 5 - x
- Sumamos el resultado de ponderar las notas como se acaba de describir y se multiplica por 2,5.

Cuestionario	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	Impares X - 1	Pares 5 - x
1	3	3	4	4	4	2	4	2	4	5	14	9
2	3	2	4	1	5	1	4	2	4	4	15	15
3	4	2	4	2	4	1	4	2	5	4	16	14
4	3	2	2	5	5	3	4	4	4	3	13	8
5	3	2	4	2	4	3	4	2	4	3	14	13
6	5	2	4	3	4	2	3	2	4	2	15	14
7	3	2	4	2	4	1	3	4	4	1	13	15
8	4	4	4	3	4	2	5	2	4	2	16	12
9	3	3	4	4	4	2	2	2	4	4	12	10
10	3	2	3	3	3	3	4	3	3	3	11	11
11	4	3	4	2	5	1	4	2	5	3	17	14
12	4	3	4	3	4	3	4	2	3	4	14	10
13	4	1	5	1	5	3	3	1	5	2	17	17
14	2	5	1	3	5	4	2	4	3	5	8	4
15	5	2	4	1	5	1	5	1	5	1	19	19

Tabla 6.5: Cálculo SUS

$$SUS = \frac{2.5}{15} * \sum_{i=1}^{15} Impares(i) + Pares(i) = 66{,}5$$

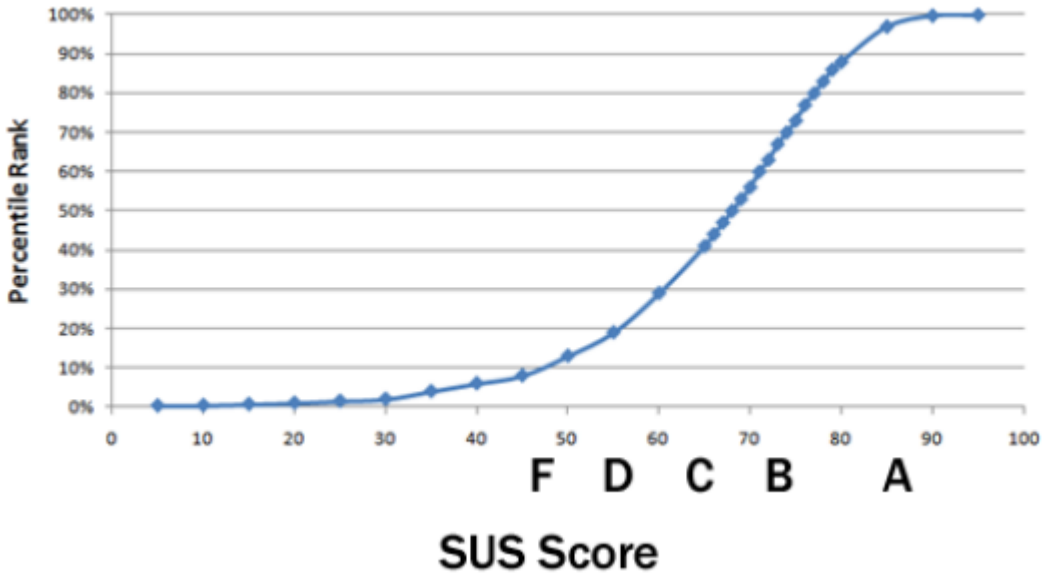


Figura 6.33: Puntuación resultados cuestionario SUS

Una vez hemos aplicado las normas anteriores hemos obtenido una puntuación de 66,5, lo cual nos da una nota de C. Por lo tanto, según el punto de vista

del cuestionario SUS nuestro prototipo cumple, por los pelos, los requisitos de un buen diseño desde el punto de vista de la usabilidad.

A partir de las preguntas de investigación hemos realizado una serie de gráficos para ver visualmente algunos de los resultados.

El gráfico de la Figura 6.34 muestra el número de desafíos que han completado nuestros evaluados. La gran mayoría han completado 1 desafío y sólo un usuario llegó a completar el tercero.

Figura 6.34: Gráfico sobre el número de desafíos completados

El gráfico de la Figura 6.35 muestra el número de usuarios que coinciden en que la documentación les ha resultado útil a la hora de resolver los desafíos.

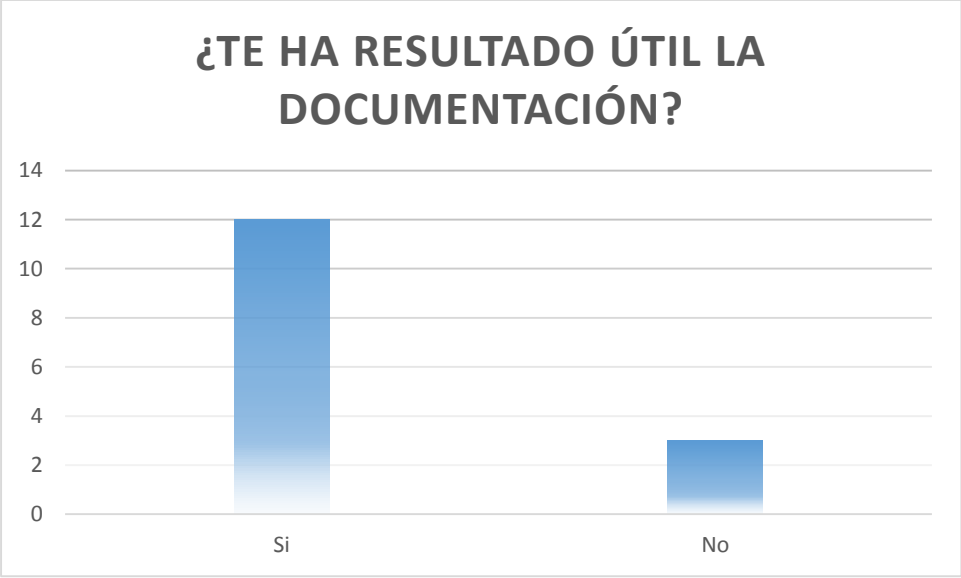


Figura 6.35: Gráfico sobre la utilizad de la documentación

En la Figura 6.36 vemos el número de usuarios que afirman que los mensajes de ayuda que se muestran tras ejecutar el código les ha resultado de utilizad.

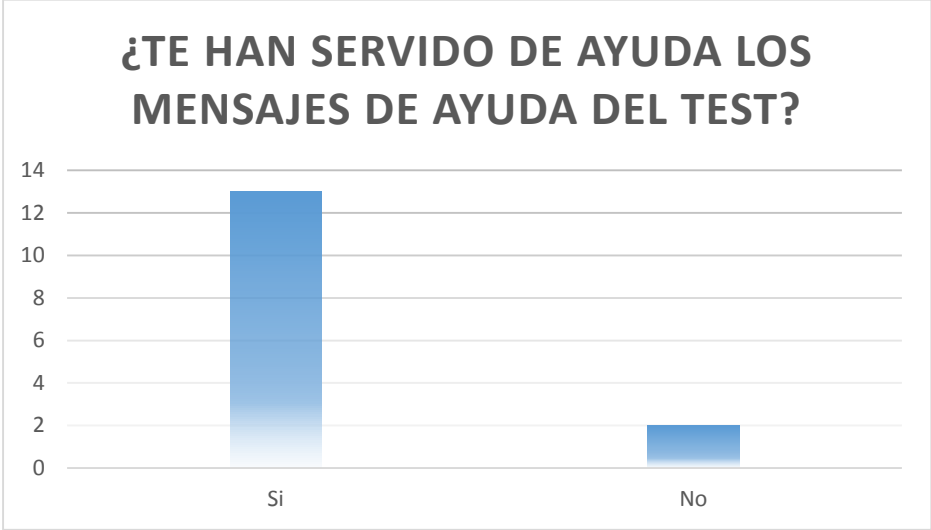


Figura 6.36: Gráfico sobre la utilizad de los mensajes de los test

Por último, en la Figura 6.37 vemos el número de usuarios a los que los comentarios del editor de código les han ayudado con la resolución de los desafíos.

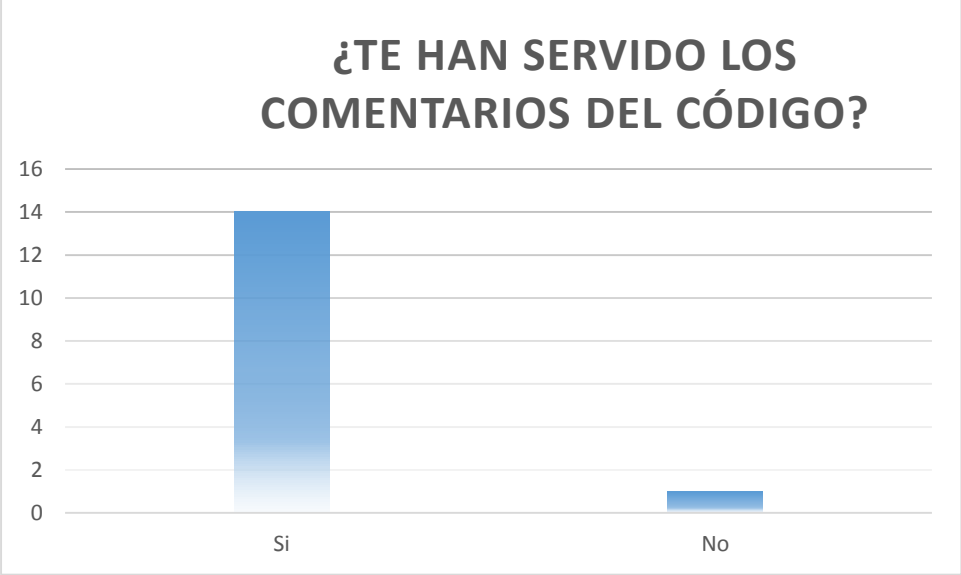


Figura 6.37: Gráfico sobre la utilidad de los comentarios en el código

Observándolos y con las otras dos cuestiones libres que hemos hecho, hemos obtenido los siguientes resultados:

- La gran mayoría de los usuarios han completado solamente el primer nivel. Esto se debe a que el tiempo de la evaluación (40 minutos) no ha sido suficiente, debido a que se tarda demasiado tiempo en empezar el videojuego en sí, debido a las numerosas explicaciones que hemos dado antes.

- La gran mayoría ha coincidido en que el videojuego tiene demasiado texto. Se dan demasiadas explicaciones al principio, momento en el que todo jugador está deseando jugar y no leer.

- La mayoría coincide en que es una aplicación agradable y fácil de usar, sin demasiada complejidad y que se aprende a utilizar rápidamente.

- Algunos participantes piensan que deben saber muchas cosas sobre la programación de videojuegos antes de poder utilizar la herramienta correctamente.

- Hemos encontrado una incongruencia y es que la mayoría (12 de 15) ha afirmado que la documentación le ha resultado útil, como vemos en la Figura 6.35 pero, sin embargo, en las respuestas a las preguntas de investigación la mayoría ha afirmado que le ha resultado útil. No solo hay que escuchar al usuario, sino ver qué ha hecho.

Gracias a las dos cuestiones abiertas, sobre lo que más y lo que menos les ha gustado de la aplicación, hemos encontrado algunos fallos de diseño y nos han comentado errores en los que nosotros no habíamos caído:

- Dentro del editor de código no han distinguido bien qué zonas de código se podían editar y qué zonas no. En parte debido a que el color utilizado para sombrear el código bloqueado se confundía con el color de fondo del editor de código.

- El editor no permite algunos atajos de teclado como ctrl+C, ctrl+V o ctrl+Z.

- Algunos han tenido dificultades para leer el texto de los test de unidad, debido a la fuente que hemos utilizado.

- En general y, aunque ya lo hemos comentado antes, la gran mayoría se ha quejado de la excesiva cantidad de texto en los diálogos.

- Algunos piensan que el texto de los test es algo confuso. Podrían mostrar mensajes más directos.

Por último y, aunque no ha quedado reflejado en las respuestas a las cuestiones, muchos de los evaluados nos comentaron que al pasar al segundo nivel echaban de menos ver lo que habían escrito en el primero, ya que no recordaban el nombre de algunas funciones o variables.

6.2.2 Observaciones de los moderadores

Como moderadores también hemos podido observar algunos aspectos a tener en cuenta:

- La mayor parte de los participantes no han abierto la sección de documentación en ningún momento. Hemos observado que esto se ha debido a que han mantenido el *scroll* vertical de la pantalla lo más abajo posible, ya que la aplicación no se ajustaba al tamaño de la ventana del navegador. De esta forma han dejado no visible la barra superior, con las opciones que permiten ver la documentación o volver al menú.

- También hemos observado que muchos de los participantes se han quedado atascados en algunos puntos del mapa, ya que no sabían hacia qué dirección debían avanzar. Por ejemplo, al empezar el juego Bersara está situada en una plataforma y sólo puede caminar hacia la derecha hasta toparse con el primer desafío. Cuando se completa este desafío ya puede caminar hacia la izquierda, pero muchos de los evaluados dudaban de hacerlo, porque pensaban que caerían por un precipicio y morirían.

- Por último, algunos de los participantes han comentado que al pasar al segundo nivel echaban de menos ver lo que habían escrito en el primero, ya que no recordaban el nombre de algunas funciones o variables.

6.3 Conclusiones

Finalmente, tras ver los problemas que tiene nuestra herramienta hemos analizado cuales podrían ser las soluciones a estos problemas. Por motivos de tiempo no hemos podido realizar todos los cambios necesarios para subsanar todos los errores descubiertos en la evaluación, ya que algunos tenían cierta complejidad, pero sí que hemos podido mejorar algunos aspectos de nuestra herramienta:

- Hemos revisado el diseño del mapa del juego, eliminando algunas zonas confusas donde los usuarios se quedaban atascados.

- Para intentar que quede más claro que zonas de código están bloqueadas en el editor y que zonas no, hemos añadido explicaciones extras en los diálogos. También hemos cambiado el color de las zonas bloqueadas para que resalte más sobre el fondo negro.

Por último hemos hecho dos fotos, con el permiso de nuestros compañeros, para dejar constancia de la realización de esta evaluación. En la Figura 6.38 tenemos una vista desde la parte trasera del laboratorio 6 y en la Figura 6.39 desde la parte delantera.

Figura 6.38: Evaluaciones en grupo en el laboratorio 6 (I)

Figura 6.39: Evaluaciones en grupo en el laboratorio 6 (II)

7. Conclusiones finales y nuestro proyecto en el futuro

En este apartado expondremos primero las conclusiones finales de nuestro proyecto y, segundo, las ideas que tenemos para nuestro proyecto en un futuro.

7.1 Conclusiones finales

Llegados a este punto podemos afirmar que hemos logrado nuestro objetivo principal y el resto de objetivos:

- Hemos desarrollado una herramienta que enseña a programar videojuegos utilizando un videojuego.
- Hemos creado un videojuego que resulta amigable y que invita a jugar.
- Hemos diseñado una interfaz que resulta agradable y que sigue la tendencia actual al minimalismo.
- Hemos implementado por nuestra cuenta los test de unidad que nos han permitido validar el código que escribe el jugador.
- Hemos utilizado el mejor método de enseñanza, a nuestro parecer, después de analizar las herramientas existentes actualmente en el estado del arte.
- Y, por último, hemos validado con una evaluación en grupo el prototipo que hemos diseñado e implementado.

No todo ha sido fácil, sino que nos hemos encontrado con algún que otro contratiempo. Puede que el mayor de ellos haya sido enseñar. Nos hemos dado cuenta de que enseñar es una de las cosas más difíciles que hemos tenido que aprender y que, para ello, hay que conocer en profundidad aquello que deseamos enseñar. A esto hay que añadir que enseñar no es lo único difícil, sino el cómo enseñar. Otro inconveniente que hemos encontrado es que nosotros no somos diseñadores gráficos, sino programadores. Por ello, la tarea de encontrar los recursos necesarios para el juego ha sido difícil.

Por otro lado, respecto al motor que hemos utilizado, *Phaser*, pensamos que ha sido una buena elección. Es cierto que al principio se nos hizo duro aprender a utilizarlo, aunque la causa de ello puede ser que nunca antes, excepto Laura, habíamos utilizado un motor. Una vez aprendimos a utilizarlo, desarrollar el videojuego fue una tarea sencilla y creemos que es una buena elección para aquellos que se aventuran por primera vez en este mundillo.

Pensamos que la interfaz de usuario de la aplicación es la ideal para programar, ya que al mismo tiempo podemos ver el código, los errores y la documentación. Esto evita distracciones al usuario y hace que la aplicación sea auto contenida, pues todo lo que nos hace falta para utilizarla está en ella. Prueba de que ha sido un buen diseño son los resultados de la evaluación en grupo. Es cierto que obtuvimos algunos errores de diseño pero, en general, todos los evaluados tuvieron buenas sensaciones. Por lo tanto, seguir un principio de diseño respecto del punto de vista de la usabilidad fue una idea acertada.

La experiencia de utilizar *Git* como sistema de control de versiones también resultó acertada. Nos ha permitido trabajar de una forma distribuida y sin los quebraderos de cabeza que provoca, a veces, tener que juntar el código. El uso de *Trello* para establecer las tareas pendientes ayudó a que trabajásemos de esta forma distribuida.

Por último, el ambiente de trabajo que hemos tenido ha sido siempre muy bueno. Gracias a esto, hemos podido realizar este proyecto de manera muy cómoda y hemos superado con creces las expectativas que teníamos en un principio.

7.2 Líneas de trabajo futuro

Actualmente, es cada vez más frecuente encontrarse con videojuegos similares al nuestro, que enseñen cualquier tipo de materia, desde las matemáticas hasta cualquier tipo de idioma.

En un futuro, puede que no muy lejano, nos gustaría explicar de una forma más completa la arquitectura de un videojuego. Pensamos que esto es bueno, ya que permitiría concebir mejor qué es un videojuego y, por lo tanto, haría posible que nuestros usuarios hicieran mejores videojuegos.

También nos gustaría implementar más niveles para obtener un videojuego completo y para que, cuando un usuario completase nuestro videojuego, pudiese desarrollar por sí mismo y sólo con la ayuda de la documentación oficial cualquier videojuego que se le ocurriese. Para ello tendríamos que ampliar las mecánicas del

juego, por ejemplo, añadir más enemigos, añadir más acciones a Bersara, crear mundos nuevos, incluir efectos con partículas…Por otro lado, también sería necesario mostrar ayudas más gráficas o incluso hacer que los mensajes de error fueran cambiando a medida que el jugador intentara ejecutar un código incorrecto, de tal manera que los mensajes cada vez diesen más pistas.

Phaser es un motor en continua actualización que cuenta con una gran comunidad, motivo por lo que le escogimos. Sin embargo, hemos tenido que congelar el proyecto en la versión 2.1.2 debido a que se produjeron numerosos cambios respecto a la versión siguiente, cambios que de haberlos tomado en cuenta, habrían retrasado en exceso la entrega del proyecto. Esto fue una decisión que no nos gustó en exceso, por ello, nuestra idea sería actualizar el proyecto a la versión más reciente de *Phaser* siempre que podamos.

Cuando comenzamos a desarrollar el proyecto nuestros conocimientos en *JavaScript* eran muy escuetos, cosa que dificultó el aprendizaje de *Phaser* y el desarrollo del videojuego. En un futuro, nos gustaría que nuestro proyecto comenzase enseñando las bases de *JavaScript*.

Otra de las cosas que nos gustaría haber enseñado es la programación de videojuegos en 3D, pero preferimos centrarnos en el 2D primero por su sencillez. Por ello, podríamos tratar de adaptar *Phaser* para que admitiera el uso de las tres dimensiones.

Creemos que todas están mejoras harían de nuestro proyecto algo realmente interesante para ser utilizado en ambientes académicos. Haríamos la enseñanza y aprendizaje de un lenguaje de programación como *JavaScript* y la de un *framework* como *Phaser* una tarea más llevadera.

7.3 Final conclusions

At this point, we can affirm that we have achieved our principal goal: we have developed a tool to teach programming videogames using a videogame. We have created a videogame that results friendly and invites the user to play. We have designed an interface that results nice and follows the current trend to minimalism. We have implemented by ourselves the unit tests that it permit us validate the code written by the player. We have use the best teaching method, in our opinion, after of analyze the existing tools currently in our state of the art. And, finally, we have validated the prototype that we have designed and implemented with a group evaluation.

Not everything has been easy, we have find with some setbacks. Teaching can be the biggest of our problems. We have noticed that teaching is one of the things more difficult that we have to learn and, for that you have to know deeply about what you want to teach. Adding that teaching is not the only difficult, the most difficult part is how to teach. Other inconvenient that we have found is that we aren't graphic designers, we are programmers. By this, the task to find the necessary resources to the game has been hard.

By other way, about to the engine that we have used, *Phaser*, we think that it has been a good choice. It is true that in the beginning was hard to learn use it, but the reason could be that Laura was the only of us that has used a game engine before. Once we learn to use it, developing the videogame was an easy task and we think that is a good choice to those people that never user a game engine.

We think that the user interface of the application is ideal to programming, because you can see at the same time the code, the errors and the documentation. This avoid distractions to the user and do that the application is self-contained, because everything we need is in it. The results that show that was a good design are the group assessment results. It is true that we obtain some design errors but, in general, all people evaluated have good feelings. Therefore, following a design principle respect of the viewpoint of the usability was a good idea.

The experience about use *Git* as version control system also was good. It has allowed us working in a distributed form and without have to join the code by ourselves. The use of *Trello* to establish the outstanding task help us to work in a distributed form.

Finally, the work environment that we have it has been very good always. Thanks to this we could make this project very comfortable and we have overcome the expectative that we have in the beginning.

7.4 Lines of future work

Nowadays, it is increasingly common to find videogames similar to ours, that teaching any kind of subject, from the mathematics to any kind of idiom.

In the future, we would like explain better the whole videogame architecture because we think that it is important. We think that is good, because it will allow to learn better what is a videogame and, because of that, it'll do possible that our users will do better videogames.

Also we would like implement more levels to obtain a whole videogame and when a user completes our game, could develop by him/herself and only with the official documentation any kind of videogame that he/she want. For it, we have to extend the game mechanics, by example, add more enemies, add more actions to Bersara, create new worlds, add effects particles… by other way, it will be necessary show more graphics helps or even doing the error messages will be changing as the player will try run a wrong code, such a way that the messages will give more tips each time.

Phaser is a continuous updating engine that has a great community, by this we choose it. However, we have to freeze the project in the 2.1.2 version because the next version has a lot of changes, if we had taken the changes, it would have delayed our project delivery. This was a decision that we didn't like so much, therefore, our idea will be update the project to the *Phaser* version more recent that we could.

When we started to develop the project our knowledge in *JavaScript* was very little, thing that do it harder leaning *Phaser* and the videogame development. In the future, we would like that our game could teach the bases of *JavaScript*.

Other of the things that we would like to teach is programming 3D videogames, but we prefer focus on the 2D firstly because was easier. Therefore, we could adapt *Phaser* to do 3D videogames.

We think that this improvements would do our project something really interesting to be used in academic environments. The teaching and the learning of a programming language as *JavaScript* and a *framework* as *Phaser* could be easier.

8. Organización del proyecto

A continuación explicaremos cómo nos hemos organizado, la metodología de trabajo que hemos seguido y lo que hemos hecho cada uno de nosotros en el proyecto.

8.1 Organización del proyecto

Hemos realizado reuniones mensuales con el director de nuestro proyecto, Guillermo. En ellas, debatíamos sobre lo que habíamos hecho en el intervalo de tiempo entre esa reunión y la reunión previa. Planteábamos las dudas que nos habían surgido e intentábamos alcanzar una solución a las mismas. Por último, comentábamos los hitos que queríamos alcanzar antes de la siguiente reunión.

En el proceso de desarrollo del videojuego, hemos empleado *Trello*[18] para marcar las tareas por hacer y las tareas finalizadas. Se trata de una herramienta web muy útil, debido a que permite organizar las tareas en forma de lista, asignar colores a las mismas, escribir comentarios...todo ello de forma dinámica y visible en tiempo real para todos los miembros del equipo, incluido nuestro director de proyecto. Podemos ver la interfaz de *Trello* en la Figura 8.40:

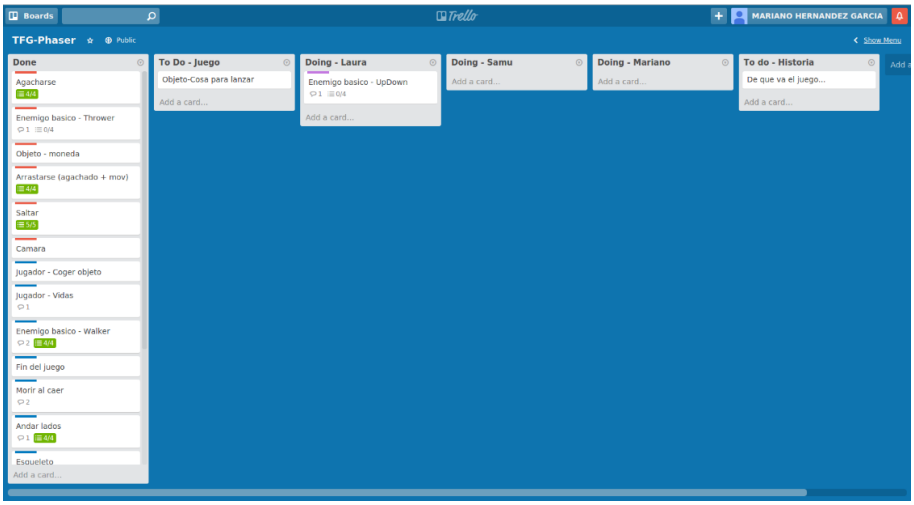
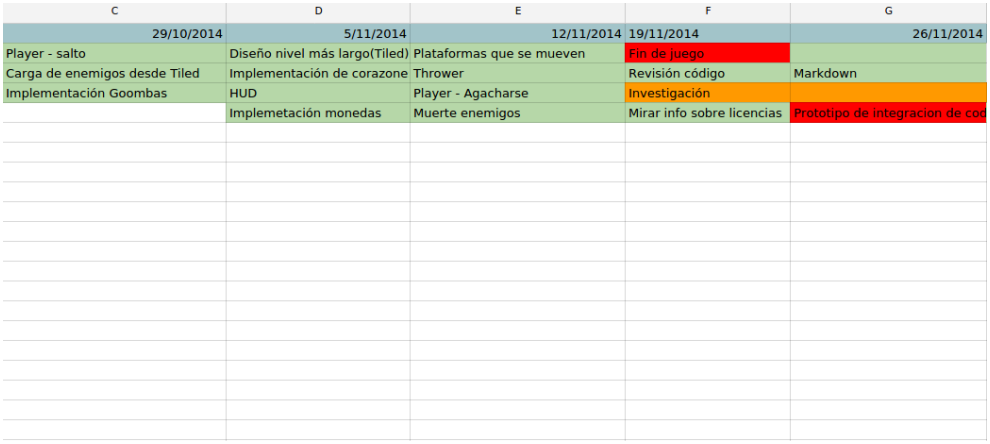


Figura 8.40: Captura de pantalla de *Trello*

[18] Plataforma web que permite la organización de proyectos en grupo.

Paralelamente y a lo largo de todo el curso, hemos seguido la metodología *Scrum* [32] para marcarnos los objetivos semanales utilizando las hojas de cálculo de *Google Drive*. Cada semana nos hemos marcado los objetivos que queríamos alcanzar: si los conseguíamos rellenábamos las celdas de color verde, si se retrasaban de color naranja y, si no se cumplían, de rojo. Podemos verlo en la Figura 8.41:



C	D	E	F	G
29/10/2014	5/11/2014	12/11/2014	19/11/2014	26/11/2014
Player - salto	Diseño nivel más largo(Tiled)	Plataformas que se mueven	Fin de juego	
Carga de enemigos desde Tiled	Implementación de corazone	Thrower	Revisión código	Markdown
Implementación Goombas	HUD	Player - Agacharse	Investigación	
	Implemetación monedas	Muerte enemigos	Mirar info sobre licencias	Prototipo de integracion de cod

Figura 8.41: Organización Scrum

En cuanto a la organización que hemos seguido a la hora de desarrollar el código, hemos decidido utilizar el sistema de control de versiones *Git* con un repositorio público en *GitHub*. En él, hemos seguido un flujo de trabajo centralizado, es decir, hemos tenido un repositorio con nuestro proyecto *.git* y todos hemos modificado ese mismo repositorio.

El repositorio ha tenido la rama *master,* una rama de desarrollo y cada uno de nosotros hemos tenido una rama de pruebas. Cuando los resultados obtenidos en alguna de las ramas de prueba han sido satisfactorios, los hemos mezclado con la rama de desarrollo. En *master* hemos tenido siempre una versión estable, aunque en ocasiones muy desactualizada, del proyecto.

8.2 Contribución al proyecto

En este apartado cada uno de nosotros explicaremos nuestra contribución personal al proyecto.

8.2.1 Laura María de Castro Saturio

El primer paso que tomamos en este proyecto fue llevar a cabo una investigación sobre los distintos motores para *HTML5* que podíamos usar. Cada uno de nosotros investigó las posibles alternativas, yo personalmente estuve investigando sobre *Phaser*, *Construct 2* y *Cocos 2D*. *Construct 2* fue un motor que descarté pronto como opción, ya que durante la investigación descubrí que para usar este motor no es necesario escribir ni una sola línea de código debido a su interfaz gráfica, por lo tanto no nos servía para nuestro propósito.

Sin embargo, *Phaser* y *Cocos 2D* son dos opciones que si tuvimos en cuenta. Finalmente, y por los motivos ya explicados, elegimos *Phaser* como motor para desarrollar el juego en el que se apoya nuestra herramienta. Una vez elegido el siguiente paso fue aprender el funcionamiento del motor. Para ello, en un primer momento, investigué los distintos ejemplos de código que se ofrecen en la propia página web del motor, los cuales abarcaban gran cantidad de temas: física, *Sprites*, partículas, entrada/salida, etc., incluso ejemplos de juegos completos.

Después de tener unas nociones básicas sobre el uso del motor decidí buscar ejemplos más completos, centrándome mayormente en tutoriales sobre juegos de plataformas, ya que fue la temática que más nos gustaba para nuestro juego. En la página web de Lessmilk encontré un listado con gran cantidad de tutoriales sobre Phaser, que abarcaban desde los aspectos más básicos del motor hasta algunos más complejos que requerían un mayor conocimiento en la materia.

Los tutoriales que me fueron de más ayuda fueron los de GameDev Academy. En esta página podemos encontrar artículos y tutoriales sobre el desarrollo de videojuegos en *HTML5*. Me fueron especialmente útiles los tutoriales sobre juegos de plataformas con Phaser, ya que en uno de ellos, realizado por el fundador de Zenva, se explicaba el uso de estados dentro de Phaser, cosa que todavía no habíamos visto en ningún tutorial, y se proponía una estructura para el código del proyecto que fue la que finalmente adoptamos para nuestro juego.

Una vez que todos teníamos ya cierta idea sobre el uso del motor decidimos desarrollar una versión básica del videojuego Super Mario Bros de Nintendo, para aplicar los conocimientos que habíamos adquirido.

Yo me encargué de crear la estructura principal del código, creando los estados básicos que tendría nuestro juego: *Boot, Preload, MainMenu* y *Game*, y las mecánicas más básicas, como el movimiento del jugador, mientras que mis compañeros realizaron algunas mecánicas extra para darle mayor jugabilidad al juego, como el comportamiento de algunos enemigos o plataformas móviles, aunque algunas de estas mecánicas no se incluyeron en la versión final del juego.

Una parte que para mí fue un poco conflictiva a la hora de desarrollar este juego fue la carga del mapa desde *Tiled*. En varios de los ejemplos y tutoriales que habíamos visto se explicaba la carga de un mapa generado con *Tiled*, o bien en formato *JSON* o bien en formato *TMX*. Pero en ninguno de ellos cargaban mapas con más de una capa, y esto era algo con lo que tuve problemas, ya que nosotros necesitábamos cargar dos capas y el motor nos daba errores al intentar hacer esto. Tras mucho buscar en algunos foros una posible respuesta sin ningún éxito, decidimos consultar a nuestro tutor sobre el tema. Él nos sugirió probar con otra versión del motor, y, efectivamente, probamos con la versión anterior a la que estábamos usando y todo funcionó correctamente.

Con este prototipo realizado, ya contábamos con los conocimientos necesarios como para poder diseñar una batería de desafíos que incluir en nuestro prototipo. Ahora necesitábamos elegir de qué manera se iban a superar. Para ello investigamos que otras herramientas del mismo tipo se podían encontrar actualmente. De las que estuvimos investigando las que más me gustaron fueron *Alice*, *Scratch* y *Code Avengers*, sobre todo este último, ya que era la aplicación que más se acercaba a lo que queríamos conseguir.

A pesar de ser una labor más compleja, decidimos que queríamos que el usuario pudiese escribir directamente el código necesario para superar los desafíos, de manera semejante a como se hace en la aplicación de *Code Avengers*. Esto requería que en la misma interfaz, a la vez que mostrábamos el juego, mostrásemos también un editor de código con el que el usuario pudiese interactuar. También nos pareció de gran ayuda incluir en la interfaz un pequeño apartado donde se mostrasen los errores del usuario, además de permitirle ver la documentación del motor cuando lo necesitase.

Nuestro compañero Mariano se puso manos a la obra y realizó varios bocetos de como podíamos integrar todos estos elementos dentro de la interfaz. Una vez que elegimos que diseño nos gustaba más, cada uno se centró en una parte de la herramienta. Yo me encargué de investigar como podíamos integrar el editor de código dentro del diseño y como comunicarlo con el juego.

En un primer momento pensé en hacerlo todo a mano, partiendo de un *textarea* y dándole estilo con *CSS*. Estuve realizando pruebas e implementé las funciones necesarias para comunicar el *textarea* con el *Canvas* del juego, de manera que el código que se escribía dentro del *textarea* se enviaba al juego y se ejecutaba.

Pero nosotros no queríamos sólo eso, sino que queríamos poder mostrar también código dentro del *textarea*, ya que la idea era mostrar ciertas funciones incompletas al usuario para que las fuese completando. También queríamos poder bloquear las partes de código que le mostrábamos para que no pudiese modificarlas y sólo se centrase en la parte a rellenar.

Hacer esto con un *textarea* desde cero era bastante complejo y pronto me di cuenta de que sí queríamos conseguir el mejor resultado la mejor opción era utilizar un editor web ya existente y adaptarlo a nuestro proyecto. Así que decidí investigar si algún editor de código web existente nos podía servir para nuestro proyecto. En mi búsqueda encontré dos posibilidades que me gustaron bastante: *Code Mirror* y *Ace*, editores de los que ya hemos hablado en sus respectivos apartados.

Estuve realizando pequeñas pruebas con ambos editores, y la verdad es que ambos ofrecen funcionalidades muy parecidas, pero a mí personalmente me pareció más sencillo trabajar con la *API* proporcionada por *Ace*. Además *Ace* cuenta con cierta comprobación de sintaxis con algunos lenguajes, como *JavaScript*, lo cual nos interesaba mucho para nuestro proyecto, porque podíamos darle una ayuda extra al usuario a la hora de editar el código.

Con *Ace* personalizar el editor fue muy sencillo, en unas pocas líneas de código pude configurar el lenguaje y el tema del editor. Lo que fue bastante complejo fue bloquear código dentro del editor. *Ace* trae funciones específicas para establecer el modo del editor, de manera que puedes deshabilitar la escritura dentro del editor muy fácilmente. Pero esto funciona sobre todo el texto que muestras en el editor, y nosotros queríamos poder bloquear sólo ciertas partes del código.

Investigué dentro de su *API*, pero no encontré una manera sencilla de hacerlo. Así que empecé a investigar en foros para encontrar alguna solución. Finalmente en un foro encontré lo que buscaba. A grandes rasgos, para poder bloquear cierta parte del texto que muestras dentro del editor tienes que definir primero el rango que quieres bloquear, fijar su inicio y su final dentro del editor bloqueando los eventos de teclado sobre esa zona.

En este foro aportaban parte del código necesario para llevar a cabo esta tarea, pero de todos modos tuve que modificarlo ya que nosotros queríamos poder tener más de una zona bloqueada, y la solución que proponían era válida sólo para bloquear una zona.

Por otro lado también necesitaba funciones para poder establecer un desafío en el editor y también para limpiar el editor. Esto último era muy importante, ya que pude comprobar que si bloqueábamos una zona de código que solapaba con una zona ya bloqueada, los rangos tomaban valores indefinidos, así que era muy importante borrar los rangos del desafío anterior antes de establecer unos rangos nuevos.

Con todo esto estuve realizando pruebas hasta conseguir que el editor se integrase perfectamente dentro de la herramienta, y que la comunicación entre el

editor y el juego funcionase de manera correcta. Para ello también tuve que realizar algunos cambios en el código del juego, pero nada relevante, sólo algunos detalles pequeños pero indispensables para el buen funcionamiento de la herramienta.

Por otra parte también necesitábamos poder guardar todos los datos referentes a cada desafío. Para ello creé un fichero *JSON* con toda esta información, de manera que al iniciar el juego cargábamos los datos del *JSON*, creando un objeto de tipo Bloque por cada uno de los desafíos escritos. Cada objeto de tipo Bloque guardaba los datos de un desafío, y a su vez representaba un objeto dentro mapa responsable de lanzar el desafío cuando el jugador chocaba con él.

Para ayudar al usuario a completar cada desafío, además de dejarle accesible la documentación de *Phaser* en cada momento y de los mensajes de ayuda que se muestran cuando el usuario falla, nos pareció bien comentar detalladamente el código que se le muestra con cada desafío.

Así que procedí a escribir estos comentarios, de manera que quedase lo más claro posible que hace cada línea de código que le mostramos y porque es necesario ese código dentro de la función. También, al principio de cada función añadí las clases de *Phaser* que se pueden necesitar consultar en la documentación para completar el desafío. Posteriormente también decidí añadir comentarios para guiar al usuario en la parte en la que tiene que escribir el código, explicando que pasos tiene que seguir y que funcionalidad tiene que conseguir.

En este punto cada uno de nosotros había integrado ya su parte, y la herramienta estaba completa. Dimos los últimos retoques y pasamos a hacer la evaluación con usuarios. Una vez terminadas las evaluaciones procedí a pasar a limpio los datos obtenidos y a realizar las conclusiones pertinentes.

Por último he realizado algunos de los cambios que pensamos tras hacer las evaluaciones, como son cambiar el color de las zonas de código bloqueadas para que tengan mayor visibilidad y cambiar el diseño del mapa para tratar de evitar que los jugadores se queden atascados.

Creo que este proyecto ha plasmado muy bien lo que somos capaces de hacer tras estos cuatro/cinco años. Personalmente para mí ha sido muy satisfactorio realizar este proyecto, tanto por el tema que abarca como por mis compañeros, ya que después de todo son mucho más que compañeros para mí, son mis amigos, y son excepcionales tanto en lo personal como en lo profesional. Lo que más me ha gustado de todo esto es que el único límite que teníamos a la hora de realizar este proyecto éramos nosotros mismos, puesto que hemos sido nosotros los que hemos decidido hasta donde queríamos llegar con ello. Y creo que aunque, efectivamente, siempre hay cosas que mejorar, podemos estar muy orgullosos de lo que hemos conseguido.

8.2.2 Samuel García Segador

En este apartado voy a contar como he ido colaborando en el proyecto a lo largo de su desarrollo y algunas de las dificultades que he ido resolviendo para poder seguir adelante.

Nada más empezar, lo primero que hice fue documentarme sobre las herramientas para poder desarrollar un videojuego, incluyendo entre estas a *Phaser*, *Quintus* y *Unity 3D*.

Cuando busqué los motores tenía un cierto interés en los que eran en 3D, por lo que me fijé mucho en *Unity*, pero la verdad, yo nunca había trabajado con un motor de este tipo y me iba a resultar más difícil poder enseñar con la herramienta a alguien.

Así que después de hablarlo y de investigarlo mucho, acordamos utilizar un motor en 2D, *Phaser*, que era lo que mejor se adaptaba a nuestros objetivos y además tenía una gran fuente de ejemplos y de documentación.

Una vez que supe el motor sobre el que íbamos a realizar el videojuego, me puse a realizar multitud de ejemplos pequeños para comprender el funcionamiento de un motor, ya que nunca había trabajado con alguno. Cuando me había familiarizado con *Phaser*, me propuse a realizar una simple implementación del videojuego de Nintendo Mario Bros para saber cómo implementar algunas de las funcionalidades que posteriormente podía tener nuestro videojuego.

Para ello me propuse hacer una versión muy reducida del Mario Bros. En ella teníamos a nuestro protagonista, Mario, que podía saltar, moverse de derecha a izquierda, recoger monedas, saltar sobre enemigos, que los enemigos pudiesen matar a Mario etc. En las primeras versiones de nuestro código del juego Laura, Mariano y yo nos asignábamos tareas distintas para realizarlas y posteriormente subirlas a nuestro repositorio de *GitHub*. Yo me he encargado de que Mario pudiese matar a un *Goomba* (uno de los enemigos en la serie de videojuegos de Mario Bros) y de un nuevo tipo de plataformas, las plataformas movedizas. Estas plataformas no tenían gravedad y se movían con una cierta velocidad hacía un lado u otro. El *Goomba* que he implementado era capaz de moverse hasta que encontrase un obstáculo, en dicho instante cambiaba la dirección y si dicho obstáculo era Mario, lo mataba y restaba vidas al contador de vidas.

Otro de los problemas con los que me he encontrado ha sido la carga de mapas con *Tiled* (en específico el formato *JSON* con el que exportábamos de *Tiled*) ya que quería poder cargar enemigos directamente del mapa y no tener que crearlos con código. Así, a la hora de diseñar el nivel sólo tendría que concentrarme en *Tiled* dejando a un lado la codificación. Por lo tanto, como el cargado "casi automático" en *Tiled* nos iba a resultar muy útil posteriormente, lo hice con resultados satisfactorios,

después de pelearme mucho tiempo con ello. Al hacerlo descubrí que en *Tiled* podíamos crear dos tipos de capas, las capas de tiles y las capas de objetos, este segundo era el que debió utilizar para poder cargar los enemigos que yo quería. Con el cargado casi automático me refiero a que para que pudiésemos cargar este *JSON* en Phaser, tenemos que modificar un archivo *JavaScript* donde tenemos todos los identificadores que *Tiled* asocia a cada tile y posteriormente en la función del respectivo nivel hacer la carga del tile de la capa de objetos que queremos cargar.

También he realizado un pequeño ejemplo con lo que se denomina *parallax scrolling* que consiste que varias de las capas del escenario se muevan a distinta velocidad consiguiendo un efecto de paralaje. Para realizar esto he tenido que jugar con la coordenada Z que tenía el mundo para poder tener así distintas escenas con la que dar este efecto.

Después de unas cuantas semanas desarrollando con *Phaser*, Laura, Mariano y yo realizamos el código que iba a tener el juego que íbamos a enseñar a programar. Al terminar, hablamos sobre algunas características que queríamos que tuviese nuestro juego a la hora de enseñar a programar al usuario. Después de esta charla me puse a investigar algunas de las herramientas que enseñaban a programar, ya fuese mediante un videojuego o por otros medios. A lo largo de mi investigación me encontré con herramientas como *Code Combat* que enseñaban a programar mediante videojuegos de una manera en la que el usuario debería de introducir código para aprender u otras como *Scratch* en las que el usuario debía hacer pseudocódigo con cajas. Mientras he ido realizando mis investigaciones, he llevado un *Word* en el que escribía párrafos, enlaces de páginas o artículos que me gustaban para después poder irlos escribiendo en esta memoria.

Al investigar el terreno sobre el que íbamos a desarrollar, teníamos 3 problemas con los que tratar: que el usuario pudiese ver la documentación de *Phaser* mientras jugaba, tener un espacio donde el usuario pudiese escribir el código que había que introducir en la herramienta y la manera en la que íbamos a enseñarle al usuario que el código que había metido era incorrecto y darle alguna pista de que era lo que tenía que cambiar del código para que funcionase.

Una vez que sabíamos que problemas teníamos que resolver, nos pusimos a resolverlos. Yo me he ocupado de la manera en la que íbamos a decirle al usuario que el código que había introducido era incorrecto. Para ello me puse a investigar sobre herramientas *JavaScript* con el que pudiera realizar test de unidad y así comprobar si el usuario había metido el código correcto.

Por la herramienta que decidí para realizar este proceso fue *QUnit*, con lo que me puse a realizar pruebas para ver como era su funcionamiento. Por el momento, yo solo quería evaluar expresiones y que me mostrase información en el caso de que fuese correcto, o en el contrario. Todo marchaba bien, pero *QUnit* mostraba más

información de la que yo quería que saliese en pantalla. Al ver esto intenté cambiar la *CSS* de la herramienta para obtenerlo a mi gusto, pero leyendo el código de *QUnit* en más profundidad averigüe que lo generaba con código, y por lo tanto si quería cambiar el estilo de la aplicación tendría que meterme dentro del código y modificarlo. Estuve modificándolo hasta un punto en el que pensé que no tenía sentido cambiar solo la interfaz y que siguiese gastando recursos aunque no lo mostrase por pantalla.

Esta fue la manera en la que decidí realizar mi propio test unitario simple que satisficiese las necesidades que queríamos cubrir. Para nuestro juego, queríamos tener una serie de objetivos que el usuario debería de completar para poder decir que un nivel es correcto. Para ello de una manera muy simple implemente una herramienta que comprobaba si una expresión dada era correcta o no y según eso mostrar código *html* en nuestra interfaz.

Al realizar mi herramienta para comprobar los objetivos que un usuario debería de cumplir para superar el desafío, me puse a realizar la codificación de algunos desafíos y solucionar algunos de los bugs que tenía nuestro código.

Uno de los mayores problemas que tenemos a la hora de comprobar el código, son los bucles infinitos, ya que nosotros comprobamos que la semántica del lenguaje sea correcto. Investigándolo mucho, llegue a la conclusión que herramientas como *MochaJs* reconocían errores de este tipo, pero el problema que teníamos que Mocha solo corría sobre *NodeJs*. Otra de las maneras que he averiguado para solucionar este problema, es cogiendo el árbol de sintaxis abstracta del lenguaje hasta llegar a un nodo que fuese un bucle infinito e introducir dentro de ese bucle una nueva instrucción *'break;'*. La herramienta que me inspiro para codificar se llama *esprima*. Por falta de tiempo y debido a que era un error que solo los más hábiles con el lenguaje iban a probar para ver si podían echar abajo nuestra aplicación, lo dejamos a parte para solucionarlo en un futuro y poder tener una herramienta cerrada.

Cuando tuvimos la aplicación cerrada, realizamos las evaluaciones de nuestra herramienta. Para llevarla a cabo hicimos una pequeña sesión para que alumnos de la asignatura 'Desarrollo de videojuegos en plataformas web' con un pequeño cuestionario para que probara nuestra aplicación. Pero antes de realizar esta evaluación, quise probarlo con un amigo con conocimientos medios de informática para ver si encontraba algún fallo en la herramienta. Gracias a esa evaluación que hice me pude dar cuenta que alguno de los test que estaba haciendo era incorrecto permitiendo que un usuario tuviese una velocidad incremental a la izquierda en el caso de moverse a la izquierda.

Después de realizar estas evaluaciones hicimos los retoques finales de nuestra memoria tal y como esta en este momento.

Como conclusiones finales del proyecto me gustaría responder a una serie de preguntas (que yo mismo me he hecho):

- ¿Qué es lo que más te ha gustado del proyecto? La verdad que ha sido un proyecto muy bonito y me han encantado todas las cosas que he realizado ya que me encantan los videojuegos y todo lo referente a ello, pero si tuviese que decir que es lo que más me ha gustado ha sido aprender como programar en un motor. Por otro lado tengo que decir que fue muy reconfortante al realizar las evaluaciones que había gente que estaba jugando a un videojuego que había realizado yo, ha sido una buena experiencia.

- ¿Qué cambiarias si lo volviera a hacer? No creo que cambiase nada, en todo caso me gustaría añadir algunos comportamientos que han quedado al aire y poder hacer más niveles.

- ¿Has aprendido algo al realizar este proyecto? De no tener ni idea de que es un motor a estar esperando a verano para poder programar en *Unity*. He aprendido un montón de cosas relacionadas con los videojuegos y la educación con ellos. Pienso que los videojuegos son una buena base para poder enseñar contenido a los demás, pero hay que dar con una buena idea para que se haga entretenido.

Como anécdota final para acabar mis conclusiones, cuando empecé el proyecto sentía cierta autoridad por el trabajo que había realizado otra gente con miedo a tocarlo. A lo largo del proyecto, he ido perdiendo este miedo ya que ha habido veces que he tenido que mirar las entrañas de *Phaser* para ver cómo se comportaba cierta función o de mirar otras herramientas para comprobar que es lo que está realizando por dentro. Por esto me siento que además de realizar como proyecto, he mejorado como programador y como persona.

Por último quería dedicar este último párrafo a mis dos compañeros de proyecto Laura y Mariano, que siempre han estado ahí y espero que siempre lo estén para ayudarme con lo que sea, como por ejemplo a escribir (risas). Espero poder realizar otro proyecto con vosotros y gracias por poder trabajar conmigo (que sé que a veces puede no ser fácil) y animarme en aquellos momentos en los que no estuve alegre.

Gracias chicos.

8.2.3 Mariano Hernández García

Una vez decidimos el *framework* de desarrollo de videojuegos que íbamos a utilizar, la primera tarea con la que empecé el proyecto fue la de aprender *Phaser*. Para ello, seguí los tutoriales que encontré en Internet, ya fuera en la propia página oficial de *Phaser* o desde tutoriales guiados creados por otras personas. Las primeras pruebas que hice consistieron en pintar un cuadrado en la pantalla y hacer que se moviera a los lados al pulsar las flechas direccionales. Una vez hecho esto, pasé a añadir enemigos, plataformas...Jugué con las distintas formas que ofrece *Phaser* para cargar mapas, tales como crear una matriz de enteros en la que cada número representa una imagen (suelo, plantas, una roca, una bloque rompible...) o la carga directa de un mapa desde ficheros *JSON* y *TMX* creados con *Tiled*.

Una vez entendí el funcionamiento básico de *Phaser*, pase con mis compañeros a desarrollar el juego Super Mario Bros utilizando dicho *framework*.

Particularmente decidí ponerme con la implementación de un enemigo muy característico de Mario: el *Thrower*. El principal inconveniente que encontré fue programar su comportamiento, ya que no es tan sencillo como el de otros enemigos. Después de meditarlo durante unos días, decidí utilizar una máquina de estados que definiera los estados que atraviesa este enemigo, que son: moverse a la izquierda, moverse a la derecha, saltar y lanzar un martillo. Esta idea resultó satisfactoria, salvo por que el comportamiento del enemigo no era idéntico al del juego original.

Una vez tuvimos el juego de Mario completamente desarrollado nos pusimos a avanzar la memoria, la cual estaba algo abandonada. En un primer momento decidimos dejar el proyecto con los recursos de Mario, pero un día encontramos en una web unos recursos que nos gustaron. Fue ahí cuando mis compañeros adaptaron estos nuevos *assets* a nuestro juego.

Pese a que no es algo que se nos suela dar especialmente bien a los miembros de nuestro gremio, me puse con el diseño de la interfaz de la aplicación, ya que es algo que me gusta bastante. En primer lugar hice un prototipo en papel, un *mockup*, y se lo presenté a mis compañeros. Tras su visto bueno y el visto bueno de nuestro coordinador, lo incluí en esta memoria en el apartado dedicado a ello. El siguiente paso consistió en trasladar ese boceto a un prototipo de alto nivel, aunque yo no fui el encargado de ello. Con el prototipo de alto nivel en la mano, me puse a implementar la interfaz gráfica de la aplicación.

Encontré serios problemas con la colocación de los elementos, ya que siempre existe el problema de los distintos tamaños de pantalla y resolución, pese a que nuestra aplicación iba dirigida a navegadores web. Para superar este bache, opté por utilizar *Bootstrap*. Al principio no todo fue fácil ya que, pese a haber oído hablar de *Bootstrap*, nunca lo había utilizado. Por ello, antes de nada, tuve que aprender a

utilizar *Bootstrap* por mi cuenta. Tras una serie de pruebas y ejemplos que hice siguiendo apuntes y tutoriales que encontraba por Internet, pasé a implementarlo en nuestro *html*.

Una vez distribuido los elementos de la aplicación, tuve que pelearme con la combinación de colores. Finalmente y tras utilizar colores elegidos bajo mi criterio, opté por utilizar una aplicación web desarrollada por *Adobe*, *Adobe Color Scheme*, que te genera la combinación de colores adecuada dado un color de base. Se lo enseñe a mis compañeros y, tras su visto bueno, me puse con la siguiente tarea: hacer que el juego, el editor de código y los test adaptaran su tamaño al tamaño completo de la pantalla.

En primera instancia traté de hacerlo con *JQuery*. El redimensionado del editor de código y del panel de los test funcionaba a la perfección con unas pocas líneas de código JavaScript y llamadas a funciones de *jQuery*. Sin embargo, fui incapaz de redimensionar el juego, ya que cuando se crea una instancia de *Phaser* hay que establecer el alto y ancho del *Canvas* y, una vez hecho esto, el motor no ofrece un mecanismo sencillo para modificar dichos valores. Navegando por la red encontré en un foro de *Phaser* un tema que hablaba sobre el redimensionado del *Canvas* a posteriori de su creación, pero implicaba añadir una función bastante extensa para ello, hecho que no me convenció. Esta cuestión estuvo rondando por mi cabeza durante varios días hasta que, finalmente, se me ocurrió una solución que podría funcionar. Se me ocurrió que se podría crear la instancia del juego una vez se hubieran cargado el resto de elementos de la aplicación, es decir, la barra del menú, el editor de código y el contenedor de los test. El problema de esto es que todos los elementos se cargaban al mismo tiempo. La solución: utilizar una función que invoca a otra trascurrido una tiempo. De esta manera calculé que pasados 100ms todos los contenedores, salvo el del juego, estaban cargados y colocados correctamente y con el tamaño adecuado. Una vez pasados estos 100ms, se puede instanciar el juego estableciendo el alto y ancho sobrante.

La siguiente tarea que realicé fueron los diálogos del juego, no escribirlos, sino buscar la manera de implementarlos. *Phaser* ofrece un mecanismo muy rudimentario para insertar texto en un juego. Básicamente sólo permite renderizar un texto y aplicarle un estilo muy básico, pero no permite añadir efectos de una manera sencilla.

Por este motivo se me ocurrió desarrollar por mi cuenta una pequeña librería que representara el diálogo entre dos personajes. Para ello utilicé *jQuery*, puesto que conocía de antemano que se pueden aplicar efectos tales como hacer que un elemento se deslice, aparezca de la nada, desaparezca... Esta librería crea un elemento * * que puede ser incrustado en cualquier código *html*. Dentro de este * * se carga una imagen que representa el contenedor de la conversación (un cuadro de texto, una nube, el típico bocadillo de los comics...). Además de esta

imagen, la librería carga dos imágenes más que simbolizan los dos miembros de la conversación, o uno en caso de que solo pasemos una imagen. Lo último que necesita es un fichero *JSON* en el que aparezca el texto de los diálogos. En un principio esta librería permitía que se configurasen distintas conversaciones con distintas imágenes y colores de los diálogos pero, ya que al final nuestro juego solo iba a hacer uso de dos imágenes y decidimos que el texto fuera siempre blanco, decidí eliminar esta configuración del *JSON* y optar por dejar la configuración como variables globales de la librería.

Una vez desarrollada la librería, pasé a hacer las pruebas de los diálogos escribiendo los diálogos. Una cosa llevó a la otra y, al final, acabé escribiendo los diálogos que aparecen en el juego. No obstante, mis compañeros también escribieron (y reescribieron) parte de la historia.

Paralelamente al diseño y desarrollo de la interfaz gráfica que yo estaba realizando, la librería de los diálogos y el texto de los mismos, mis compañeros se dedicaron a crear la librería de los test de unidad (*TWUnit*) y el editor de código (*Ace*) junto con el código que aparece en su interior correctamente formateado y con los comentarios que consideramos oportunos.

A estas alturas ya teníamos el primer nivel del juego desarrollado, el editor de código funcionando y los test de unidad dando buenos resultados. En este momento decidí implementar el menú principal y los créditos del juego, la escena que contiene la introducción a *Phaser* que sirve para guiar a nuestros jugadores y la escena final que se muestra cuando se completan todos los desafíos. Todas estas escenas las desarrollé utilizando *Phaser* ya que consideré que sería rápido, cosa que resultó ser cierta.

Una vez terminadas dichas escenas, recordé junto a mis compañeros que se nos había olvidado el tema de la documentación. Nuestra primera idea fue la de colocar un botón que, al hacer *click*, abriera una nueva pestaña del navegador que abriera a su vez la documentación oficial de *Phaser*. Pero esto planteaba dos problemas: nuestro proyecto se estancó en la versión 2.1.2 de *Phaser* por cuestiones de implementación y que abrir la documentación en otra vista diferente a la de la aplicación rompía con el principio de consistencia de la aplicación, así como resultar incómodo navegar entre dos vistas para nuestros usuarios finales. Por ello tuve otro problema asaltando mi mente. No obstante y, por suerte, recordé que en una asignatura en la que nos hablaron de las novedades de *HTML5* nos comentaron que el elemento <frame> permitía cargar cualquier web dentro de otra web pero, por desgracia, ese elemento había sido deprecado en HTML5. Esto no frenó mi curiosidad y me lancé a investigar cómo funcionaban los *<frames>.* Fue en ese momento cuando descubrí que *HTML5* contempla un elemento nuevo, los *<iframe>,* que permiten hacer exactamente lo mismo que los antiguos *<frame>* solo que con cambios y novedades que, sinceramente, no me interesaban puesto que podía haber encontrado la

solución. Y lo fue. Conseguí incluir la documentación de *Phaser* correspondiente a la versión 2.1.2 dentro de un contenedor que se puede mostrar u ocultar presionando un botón.

Ya casi en la recta final participé junto a mis compañeros en las evaluaciones, tanto la de grupo en el laboratorio 6 el día 21 de Mayo, como las individuales que realizaron compañeros y amigos míos de clase.

Por último, aunque no por ello menos importante, he trabajado revisando la memoria y he tratado de escribir todo aquello que hemos ido escribiendo en documentos sueltos lo mejor que he podido.

Creo que no hablo sólo por mí, sino que también hablo por mis otros dos compañeros. Realizar este proyecto ha sido una experiencia agradable. Es cierto que al principio se ve todo muy lejano, tienes muchos meses para elaborar un proyecto y no parece un trabajo tan extenso. Pero luego surgen los problemas, las cuestiones que hay que tratar, decisiones que, a veces, nos han podido poner a unos en contra de los otros; y muchas otras cosas que, sin duda, nos encontraremos en nuestra vida futura. Sin embargo, con el trabajo diario y nuestro esfuerzo, todas estas adversidades van desapareciendo hasta llegar a lo que hemos conseguido: crear algo de la nada a partir de nuestras propias manos y nuestro intelecto gracias a lo que hemos aprendido en cuatro años.

9. Anexos

En este apartado hemos incluido la solución de los siete desafíos que conforman el primer nivel, así como las cuestiones de las evaluaciones respondidas por los usuarios que realizaron la evaluación en el laboratorio 6.

9.1 Anexo 1: Solución a los desafíos del primer nivel

9.1.1 Hacer que el jugador camine a la izquierda

```
else if (this.cursors.left.isDown) {
  this.sprite.body.velocity.x = -this.walkSpeed;

  if (this.sprite.body.onFloor() ||
this.sprite.body.touching.down){
    this.sprite.play('player_animation_moveLeft', 5, true);
  }

  if (this.direction == State.LOOKINGRIGHT) {
    this.direction = State.LOOKINGLEFT;
  }
}
```

9.1.2 Hacer que el jugador salte

```
if (this.direction == State.LOOKINGLEFT) {
  this.sprite.play('player_animation_jumpLeft');
}
else {
  this.sprite.play('player_animation_jumpRight');
}
this.sprite.body.velocity.y = this.jumpSpeed;
```

9.1.3 Hacer que los enemigos caigan al agua y mueran

```
if (enemy.direction == State.LOOKINGLEFT) {
     enemy.body.velocity.x = -enemy.walkSpeed;
     enemy.scale.x = 1;
}
else if (enemy.direction == State.LOOKINGRIGHT) {
     enemy.body.velocity.x = enemy.walkSpeed;
     enemy.scale.x = -1;
}

if(enemy.body.y >= 980)
    enemy.kill();
```

9.1.4 Hacer que el jugador pueda saltar sobre los enemigos

```
if (enemy.body.touching.up) {
      enemy.kill();
      this.sprite.body.velocity.y = this.jumpSpeed;
      this.jumpTime = this.game.time.now + 750;
}
```

9.1.5 Mostrar el score en el HUD

```
this.scoreText = this.game.add.text(16, 16, this.scoreString +
this.score, { fontSize: '32px', fill: '#FFF' });
this.scoreText.fixedToCamera = true;
```

9.1.6 Hacer que el jugador pueda recoger monedas

```
coin.kill();

hud.score++;
hud.scoreText.text = hud.scoreString + hud.score;
```

9.1.7 Crear una puerta para salir del nivel

```
doors.enableBody = true;
```

9.2 Anexo 2: JSON del desafío caminar a la izquierda

```
{
     "goals": [
          {
          "task": "Haz que el jugador se mueva hacia la izquierda",

          "code":"function move() {\n\n\t/*** Clases clave para
ayudar a Bersara ***/\n\t//
//\n\t// Phaser.Sprite//\n\t// Phaser.Keyboard//\n\t//
Phaser.Physics.Arcade//\n\t// Phaser.Physics.Arcade.Body
//\n\t//
//\n\t/*****************************************/\n\n\t/* En
this.cursors tenemos un objeto con la información de \n\tnuestros 4
controles: flecha arriba (up), flecha abajo (down),\n\t   flecha
derecha (right), flecha izquierda (down)\n\t Si queremos saber si
alguno de los controles ha sido pulsado\n\t   podemos consultar
su propiedad 'isDown' */\n\tif (this.cursors.right.isDown) {\n\t\t/*
En el caso de haber pulsado la flecha derecha nos moveremos \n\t\t
en el sentido positivo del eje X, y por tanto nuestra \n\t\t
velocidad en la X tiene que ser positiva \n\t\tEn 'this.walkSpeed'
tenemos la velocidad que necesitamos \n\t\tpara caminar
*/\n\t\tthis.sprite.body.velocity.x = this.walkSpeed;\n\n\t\t/*Antes
de asignar la animación de andar hacia la derecha \n\t\t tenemos que
comprobar que estamos en el suelo. Para ello\n\t\t nos fijaremos en
las propiedad 'touching' de body */\n\t\tif(this.sprite.body.onFloor
() || this.sprite.body.touching.down) {\n\t\t\t/* Para ejecutar una
animación usamos la función 'play'\n\t\t\t   de 'sprite'. En este
caso ejecutaremos \n\t\t\t   'player_animation_moveRight'
*/\n\t\t\tthis.sprite.play('player_animation_moveRight', 5,
true);\n\t\t}\n\n\t\t/* Para cambiar el sentido en el que mira
Bersara nos \n\t\t   fijaremos en nuestra dirección. Tenemos dos
posibles \n\t\t   direcciones: 'State.LOOKINGLEFT' y
'State.LOOKINGRIGHT' */\n\t\tif (this.direction == State.LOOKINGLE
FT) {\n\t\t\t/* Solo cambiamos su sentido si estaba mirando en
sentido\n\t\t\t contrario */\n\t\t\tthis.direction =
State.LOOKINGRIGHT;\n\t\t}\n\t}\n\n\n\n\t/* Ahora te toca a ti!! Lo
primero es comprobar si se ha pulsado\n\t   la flecha
izquierda(this.cursors.left) */\n\n\n\t/* Si es así tenemos que ir
en el sentido negativo del eje X, por \n\t   lo que asignaremos '-
walkSpeed' como nueva velocidad */\n\n\n\t/* Comprobamos si Bersara
esta en el suelo, igual que hemos hecho \n\t   antes, y si es así
cambiamos su animación. Esta vez toca \n\t'player_animations_move
```

```
Left' */\n\n\n\t/* Y por último tenemos que cambiar el sentido de
Bersara en caso\n\t   de que este mirando en sentido contrario
*/\n\n\n\n\t/* Si no se ha pulsado nada no moveremos a Bersara. En
este caso \n\t   lo único que tenemos que hacer es asegurarnos de
que se\n\t   ejecuta la animación correcta, dependiendo de a donde
este \n\t   mirando Bersara */\n\telse {\n\t\tif (this.direction ==
State.LOOKINGLEFT){\n\t\t\tthis.sprite.play('player_animation_stand
UpLeft');\n\t\t}\n\t\telse if (this.direction == State.LOOKINGRIGHT)
{\n\t\t\tthis.sprite.play('player_animation_standUpRight');\n\t\t}
\n\n\t\t/* Y ponemos su velocidad a 0 ya que no tiene que moverse
*/\n\t\tthis.sprite.body.velocity.x = 0;\n\t}\n}\n\n",

            "target": "player.move",
            "range1": [0,0,43,0],
            "range2": [64,0,80,0],
            "test": "testMoveLeft(text)",
            "line": 49
            }
      ]
}
```

9.3 Anexo 3: Test del desafío caminar a la izquierda

```
var testMoveLeft = function (text) {

    tw = new TWUnit();

    eval(currentTask.target + "=" + text);

    // Si el jugador no está en el suelo.
    player.cursors.left.isDown = true;
    player.sprite.body.blocked.down = false;
    player.move();
    player.sprite.body.blocked.down = true;
    player.cursors.left.isDown = false;

    tw.addAssert("No podemos ejecutar las animaciones de andar si estamos
en el aire. Solo si la condición (this.sprite.body.onFloor() ||
this.sprite.body.touching.down) es verdadera ejecutaremos las animaciones",
player.sprite.animations.currentAnim !=
player.sprite.animations._anims["player_animation_moveLeft"], "", "");

    reInitMove();

    // Si el jugador está en el suelo.
    player.cursors.left.isDown = true;
        player.move();
    player.cursors.left.isDown = false;

    tw.addAssert("La animación no es correcta. Cuando Bersara camine hacia
 la izquierda ejecutará la animación 'player_animation_moveLeft'",
player.sprite.animations.currentAnim ===
player.sprite.animations._anims["player_animation_moveLeft"], "", "");
```

```
    tw.addAssert("La dirección a la que mira Bersara no es correcta. Si se
mueve hacia la izquierda su dirección debería ser 'State.LOOKIGNLEFT'",
player.direction == State.LOOKINGLEFT, "", "");
    tw.addAssert("Al movernos hacia la izquierda nos estamos moviendo en el
sentido negativo del eje X, por lo que tenemos que asignarle a Bersara una
velocidad negativa, más concretamente '-this.walkSpeed'",
player.sprite.body.velocity.x == -player.walkSpeed, "", "");


    // Comprobamos si ha puesto una instruccion del tipo If-elseIf-Else
    player.cursors.right.isDown = true;
    player.cursors.left.isDown = true;
        player.move();
    player.cursors.right.isDown = false;
    player.cursors.left.isDown = false;

    tw.addAssert("La estructura de la función parece estar mal. Asegúrate
 de que su estructura es if (Condicion para moverse a la derecha) { Nos
movemos a la derecha } else if (Condicion para moverse a la izquierda) {
 Nos movemos a la izquierda } else { Nos quedamos quietos }",
player.sprite.animations.currentAnim ===
player.sprite.animations._anims["player_animation_moveRight"] , "", "");


    // Para comprobar si ha puesto en velocidad una instrucción aditiva.
    player.cursors.left.isDown = true;
        player.move();
    player.cursors.left.isDown = false;

    tw.addAssert("No estas poniendo una velocidad constante, debes poner
'this.walkSpeed' o '-this.walkSpeed', dependiendo de hacia dónde te
muevas", player.sprite.body.velocity.x == -player.walkSpeed, "", "");

    reInitMove();

    tw.runAsserts();

    return tw.assertsOk();
}
```

Lo primero que comprobamos es que el código sea sintácticamente correcto con la función “eval” de *JavaScript* y, si es así, modificaremos nuestra función objetivo que definimos en el *JSON*, en este caso “move”, y le asignaremos el código escrito por el usuario.

```
eval(currentTask.target + "=" + text);
```

Como podemos ver ahora en el código, ejecutamos la función que nos proporciona el usuario de 4 maneras diferentes para comprobar los casos de los test, si el jugador no está en el suelo, si el jugador está en el suelo, si la instrucción introducida por el usuario es de tipo *if-elseif-else* y que la velocidad no sea acumulativa. Cada uno de estos casos se compone de unas partes fundamentales:

- Inicializar el caso de prueba, como por ejemplo, pulsar la tecla de ir a la izquierda con código para comprobar así si se está moviendo a la izquierda correctamente al ejecutar la función del usuario.

```
// Si el jugador no está en el suelo.
player.cursors.left.isDown = true;
player.sprite.body.blocked.down = false;
```

- Ejecutar la función proporcionada por el usuario.

```
player.move();
```

- Dejamos las variables o input como estaba antes de ejecutar la función.

```
player.sprite.body.blocked.down = true;
player.cursors.left.isDown = false;
```

- Añadimos a *TWUnit* los *asserts* que queramos comprobar de este caso de prueba.

```
tw.addAssert("No podemos ejecutar las animaciones de andar si
estamos en el aire. Solo si la condición
(this.sprite.body.onFloor() || this.sprite.body.touching.down)
es verdadera ejecutaremos las animaciones",
player.sprite.animations.currentAnim !=
player.sprite.animations._anims["player_animation_moveLeft"],
"", "");
```

- Reinicializamos los valores de la función, en este caso “move”, para volver a dejarla como en un principio.

```
reInitMove();
```

- Si estamos en el último caso de prueba, ejecutar los test y devolver si todos ellos han sido correctos o no.

```
tw.runAsserts();

return tw.assertsOk();
```

9.4 Anexo 4: Resultados cuestionario de las evaluaciones

9.4.1 Cuestionario 1

Preguntas previas:

1. ¿Cuál es tu nivel de conocimiento de *JavaScript*?

   Medio.
2. ¿Cuál es tu nivel de conocimiento en la programación de videojuegos?

   Bajo.
3. ¿Has utilizado alguna vez un *framework* de desarrollo de videojuegos como *Phaser*? Si has utilizado alguno, ¿Cuáles?

   ***Quintus*.**

Cuestionario SUS:

1. Pienso que me gustaría usar esta herramienta frecuentemente.

   Ni en acuerdo ni en desacuerdo.
2. Encuentro la herramienta innecesariamente compleja.

   Ni en acuerdo ni en desacuerdo.
3. Pienso que la herramienta era fácil de usar.

   De acuerdo.
4. Pienso que necesitaría la ayuda de un moderador para ser capaz de usar la herramienta.

   De acuerdo.
5. Encuentro varias funciones de la herramienta bien integradas.

   De acuerdo.
6. Pienso que hubo demasiadas inconsistencias en la herramienta.

   En desacuerdo.
7. Imagino que la mayoría de la gente aprenderá a usar la herramienta muy rápido.

   De acuerdo.
8. Encuentro la herramienta muy incómoda de usar.

   En desacuerdo.

9. Me siento seguro utilizando la herramienta.

   De acuerdo.

10. Necesito aprender muchas cosas antes de poder ponerme con esta herramienta.

    Totalmente de acuerdo.

Preguntas posteriores:

1. ¿Cuántos desafíos has completado?

   2.
2. ¿Qué es lo que más te ha gustado?

   Una herramienta educativa para aprender a programar.
3. ¿Qué es lo que menos te ha gustado?

   Quizás enfocado como demasiado infantil. No he podido copiar directamente con CTRL+C.
4. ¿Te ha resultado útil la documentación?

   Sí.
5. ¿Te han servido de ayuda los mensajes de ayuda del test?

   Sí.
6. ¿Te han servido los comentarios del código?

   Sí.

9.4.2 Cuestionario 2

Preguntas previas:

1. ¿Cuál es tu nivel de conocimiento de *JavaScript*?

   Avanzado.
2. ¿Cuál es tu nivel de conocimiento en la programación de videojuegos?

   Bajo.
3. ¿Has utilizado alguna vez un *framework* de desarrollo de videojuegos como *Phaser*? Si has utilizado alguno, ¿Cuáles?

   ***Quintus*.**

Cuestionario SUS:

1. Pienso que me gustaría usar esta herramienta frecuentemente.

   Ni en acuerdo ni en desacuerdo.

2. Encuentro la herramienta innecesariamente compleja.

   En desacuerdo.
3. Pienso que la herramienta era fácil de usar.

   De acuerdo.
4. Pienso que necesitaría la ayuda de un moderador para ser capaz de usar la herramienta.

   Totalmente en desacuerdo.
5. Encuentro varias funciones de la herramienta bien integradas.

   Totalmente de acuerdo.
6. Pienso que hubo demasiadas inconsistencias en la herramienta.

   Totalmente en desacuerdo.
7. Imagino que la mayoría de la gente aprenderá a usar la herramienta muy rápido.

   De acuerdo.
8. Encuentro la herramienta muy incómoda de usar.

   En desacuerdo.

9. Me siento seguro utilizando la herramienta.

   De acuerdo.
10. Necesito aprender muchas cosas antes de poder ponerme con esta herramienta.

    De acuerdo.

Preguntas posteriores:

1. ¿Cuántos desafíos has completado?

   2.
2. ¿Qué es lo que más te ha gustado?

   Ser yo quien programa, y ver que funcionan las cosas en el propio juego.
3. ¿Qué es lo que menos te ha gustado?

   El editor debería ser algo más completo.
4. ¿Te ha resultado útil la documentación?

   Sí.
5. ¿Te han servido de ayuda los mensajes de ayuda del test?

   Sí.
6. ¿Te han servido los comentarios del código?

Sí, mucho.

9.4.3 Cuestionario 3

Preguntas previas:

1. ¿Cuál es tu nivel de conocimiento de *JavaScript*?

   Avanzado.
2. ¿Cuál es tu nivel de conocimiento en la programación de videojuegos?

   Medio.
3. ¿Has utilizado alguna vez un *framework* de desarrollo de videojuegos como *Phaser*? Si has utilizado alguno, ¿Cuáles?

   ***Quintus*.**

Cuestionario SUS:

1. Pienso que me gustaría usar esta herramienta frecuentemente.

   De acuerdo.
2. Encuentro la herramienta innecesariamente compleja.

   En desacuerdo.
3. Pienso que la herramienta era fácil de usar.

   De acuerdo.
4. Pienso que necesitaría la ayuda de un moderador para ser capaz de usar la herramienta.

   En desacuerdo.
5. Encuentro varias funciones de la herramienta bien integradas.

   De acuerdo.
6. Pienso que hubo demasiadas inconsistencias en la herramienta.

   Totalmente en desacuerdo.
7. Imagino que la mayoría de la gente aprenderá a usar la herramienta muy rápido.

   De acuerdo.
8. Encuentro la herramienta muy incómoda de usar.

   En desacuerdo.
9. Me siento seguro utilizando la herramienta.

   Totalmente de acuerdo.
10. Necesito aprender muchas cosas antes de poder ponerme con esta herramienta.

De acuerdo.

Preguntas posteriores:

1. ¿Cuántos desafíos has completado?

   1.
2. ¿Qué es lo que más te ha gustado?

   El hecho de poder ejecutar el código y ver si lo había conseguido o en qué había fallado.
3. ¿Qué es lo que menos te ha gustado?

   Me gustaría que el editor fuese más claro. Deberían explicar mejor dónde puedo o no puedo editar. Echo en falta cntrl+c,cntrl+v y cntrl+z.
4. ¿Te ha resultado útil la documentación?

   Sí.
5. ¿Te han servido de ayuda los mensajes de ayuda del test?

   Sí.
6. ¿Te han servido los comentarios del código? **Mucho.**

9.4.4 Cuestionario 4

Preguntas previas:

1. ¿Cuál es tu nivel de conocimiento de *JavaScript*?

   Medio.
2. ¿Cuál es tu nivel de conocimiento en la programación de videojuegos?

   Medio.
3. ¿Has utilizado alguna vez un *framework* de desarrollo de videojuegos como *Phaser*? Si has utilizado alguno, ¿Cuáles?

   ***Unity 3D, Quintus*.**

Cuestionario SUS:

1. Pienso que me gustaría usar esta herramienta frecuentemente.

   Ni en acuerdo ni en desacuerdo.
2. Encuentro la herramienta innecesariamente compleja.

   En desacuerdo.
3. Pienso que la herramienta era fácil de usar.

   En desacuerdo.

4. Pienso que necesitaría la ayuda de un moderador para ser capaz de usar la herramienta.

   Totalmente de acuerdo.
5. Encuentro varias funciones de la herramienta bien integradas.

   Totalmente de acuerdo.
6. Pienso que hubo demasiadas inconsistencias en la herramienta.

   Ni en acuerdo ni en desacuerdo.
7. Imagino que la mayoría de la gente aprenderá a usar la herramienta muy rápido.

   De acuerdo.
8. Encuentro la herramienta muy incómoda de usar.

   De acuerdo.
9. Me siento seguro utilizando la herramienta.

   De acuerdo.
10. Necesito aprender muchas cosas antes de poder ponerme con esta herramienta.

    Ni en acuerdo ni en desacuerdo.

Preguntas posteriores:

1. ¿Cuántos desafíos has completado?

   1.
2. ¿Qué es lo que más te ha gustado?

   (El usuario no respondió a esta pregunta).
3. ¿Qué es lo que menos te ha gustado?

   (El usuario no respondió a esta pregunta).
4. ¿Te ha resultado útil la documentación?

   Sí.
5. ¿Te han servido de ayuda los mensajes de ayuda del test?

   Sí, bastante.
6. ¿Te han servido los comentarios del código?

   Sí.

9.4.5 Cuestionario 5

Preguntas previas:

1. ¿Cuál es tu nivel de conocimiento de *JavaScript*?

Bajo.

2. ¿Cuál es tu nivel de conocimiento en la programación de videojuegos?

Medio.

3. ¿Has utilizado alguna vez un *framework* de desarrollo de videojuegos como *Phaser*? Si has utilizado alguno, ¿Cuáles?

***Quintus, Unity*.**

Cuestionario SUS:

1. Pienso que me gustaría usar esta herramienta frecuentemente.

Ni en acuerdo ni en desacuerdo.

2. Encuentro la herramienta innecesariamente compleja.

En desacuerdo.

3. Pienso que la herramienta era fácil de usar.

De acuerdo.

4. Pienso que necesitaría la ayuda de un moderador para ser capaz de usar la herramienta.

En desacuerdo.

5. Encuentro varias funciones de la herramienta bien integradas.

De acuerdo.

6. Pienso que hubo demasiadas inconsistencias en la herramienta.

Ni en acuerdo ni en desacuerdo.

7. Imagino que la mayoría de la gente aprenderá a usar la herramienta muy rápido.

De acuerdo.

8. Encuentro la herramienta muy incómoda de usar.

En desacuerdo.

9. Me siento seguro utilizando la herramienta.

De acuerdo.

10. Necesito aprender muchas cosas antes de poder ponerme con esta herramienta.

Ni en acuerdo ni en desacuerdo.

Preguntas posteriores:

1. ¿Cuántos desafíos has completado?

2.

2. ¿Qué es lo que más te ha gustado?

La posibilidad de tener ambas vistas abiertas a la vez.

3. ¿Qué es lo que menos te ha gustado?

   No se pueden usar atajos de teclado para editar el archivo.

4. ¿Te ha resultado útil la documentación?

   Sí.

5. ¿Te han servido de ayuda los mensajes de ayuda del test?

   Sí.

6. ¿Te han servido los comentarios del código?

   Sí.

9.4.6 Cuestionario 6

Preguntas previas:

1. ¿Cuál es tu nivel de conocimiento de *JavaScript*?

   Medio.

2. ¿Cuál es tu nivel de conocimiento en la programación de videojuegos?

   Medio.

3. ¿Has utilizado alguna vez un *framework* de desarrollo de videojuegos como *Phaser*? Si has utilizado alguno, ¿Cuáles?

   ***Unity*.**

Cuestionario SUS:

1. Pienso que me gustaría usar esta herramienta frecuentemente.

   Totalmente de acuerdo.

2. Encuentro la herramienta innecesariamente compleja.

   En desacuerdo.

3. Pienso que la herramienta era fácil de usar.

   De acuerdo.

4. Pienso que necesitaría la ayuda de un moderador para ser capaz de usar la herramienta.

   Ni en acuerdo ni en desacuerdo.

5. Encuentro varias funciones de la herramienta bien integradas.

   De acuerdo.

6. Pienso que hubo demasiadas inconsistencias en la herramienta.

   En desacuerdo.

7. Imagino que la mayoría de la gente aprenderá a usar la herramienta muy rápido.

   Ni en acuerdo ni en desacuerdo.
8. Encuentro la herramienta muy incómoda de usar.

   En desacuerdo.
9. Me siento seguro utilizando la herramienta.

   De acuerdo.
10. Necesito aprender muchas cosas antes de poder ponerme con esta herramienta.

    En desacuerdo.

Preguntas posteriores:

1. ¿Cuántos desafíos has completado?

   1.
2. ¿Qué es lo que más te ha gustado?

   Me ha gustado el sistema de diálogos y guía.
3. ¿Qué es lo que menos te ha gustado?

   En el editor de texto costaba diferenciar que texto era editable y cual no.
4. ¿Te ha resultado útil la documentación?

   Sí.
5. ¿Te han servido de ayuda los mensajes de ayuda del test?

   Sí.
6. ¿Te han servido los comentarios del código?

   Sí.

9.4.7 Cuestionario 7

Preguntas previas:

1. ¿Cuál es tu nivel de conocimiento de *JavaScript*?

   Avanzado.
2. ¿Cuál es tu nivel de conocimiento en la programación de videojuegos?

   Medio.
3. ¿Has utilizado alguna vez un *framework* de desarrollo de videojuegos como *Phaser*? Si has utilizado alguno, ¿Cuáles?

   ***Quintus*.**

Cuestionario SUS:

1. Pienso que me gustaría usar esta herramienta frecuentemente.

   Ni en acuerdo ni en desacuerdo.
2. Encuentro la herramienta innecesariamente compleja.

   En desacuerdo.
3. Pienso que la herramienta era fácil de usar.

   De acuerdo.
4. Pienso que necesitaría la ayuda de un moderador para ser capaz de usar la herramienta.

   En desacuerdo.
5. Encuentro varias funciones de la herramienta bien integradas.

   De acuerdo.
6. Pienso que hubo demasiadas inconsistencias en la herramienta.

   Totalmente en desacuerdo.
7. Imagino que la mayoría de la gente aprenderá a usar la herramienta muy rápido.

   Ni en acuerdo ni en desacuerdo.
8. Encuentro la herramienta muy incómoda de usar.

   De acuerdo.
9. Me siento seguro utilizando la herramienta.

   De acuerdo.
10. Necesito aprender muchas cosas antes de poder ponerme con esta herramienta.

    Totalmente en desacuerdo.

Preguntas posteriores:

1. ¿Cuántos desafíos has completado?

   1.
2. ¿Qué es lo que más te ha gustado?

   El compilador que me permite ver los errores cometidos.
3. ¿Qué es lo que menos te ha gustado?

   El editor de texto, que se podría mejorar, no permite volver a atrás ni realizar atajos de teclado (ctrl+c, crtl+v) y permite cortar el código que viene ya realizado. También podría ser interesante revisar código realizado anteriormente.

4. ¿Te ha resultado útil la documentación?

   Sí.
5. ¿Te han servido de ayuda los mensajes de ayuda del test?

   Sí y mucho.
6. ¿Te han servido los comentarios del código?

   Sí, es muy útil para guiarte en los primeros pasos.

9.4.8 Cuestionario 8

Preguntas previas:

1. ¿Cuál es tu nivel de conocimiento de *JavaScript*?

   Medio.
2. ¿Cuál es tu nivel de conocimiento en la programación de videojuegos?

   Medio.
3. ¿Has utilizado alguna vez un *framework* de desarrollo de videojuegos como *Phaser*? Si has utilizado alguno, ¿Cuáles?

   ***Unity, Quintus*.**

Cuestionario SUS:

1. Pienso que me gustaría usar esta herramienta frecuentemente.

   De acuerdo.
2. Encuentro la herramienta innecesariamente compleja.

   De acuerdo.
3. Pienso que la herramienta era fácil de usar.

   De acuerdo.
4. Pienso que necesitaría la ayuda de un moderador para ser capaz de usar la herramienta.

   Ni en acuerdo ni en desacuerdo.
5. Encuentro varias funciones de la herramienta bien integradas.

   De acuerdo.
6. Pienso que hubo demasiadas inconsistencias en la herramienta.

   En desacuerdo.
7. Imagino que la mayoría de la gente aprenderá a usar la herramienta muy rápido.

   Totalmente de acuerdo.
8. Encuentro la herramienta muy incómoda de usar.

En desacuerdo.

9. Me siento seguro utilizando la herramienta.

   De acuerdo.

10. Necesito aprender muchas cosas antes de poder ponerme con esta herramienta.

    En desacuerdo.

Preguntas posteriores:

1. ¿Cuántos desafíos has completado?

   1.

2. ¿Qué es lo que más te ha gustado?

   Es fácil de usar.

3. ¿Qué es lo que menos te ha gustado?

   Pocos recordatorios de código.

4. ¿Te ha resultado útil la documentación?

   Bastante.

5. ¿Te han servido de ayuda los mensajes de ayuda del test?

   Sí.

6. ¿Te han servido los comentarios del código?

   Sí.

9.4.9 Cuestionario 9

Preguntas previas:

1. ¿Cuál es tu nivel de conocimiento de *JavaScript*?

   Medio.

2. ¿Cuál es tu nivel de conocimiento en la programación de videojuegos?

   Medio.

3. ¿Has utilizado alguna vez un *framework* de desarrollo de videojuegos como *Phaser*? Si has utilizado alguno, ¿Cuáles?

   Nunca, solo he probado *Quintus*.

Cuestionario SUS:

1. Pienso que me gustaría usar esta herramienta frecuentemente.

   Ni en acuerdo ni en desacuerdo.

2. Encuentro la herramienta innecesariamente compleja.

   Ni en acuerdo ni en desacuerdo.
3. Pienso que la herramienta era fácil de usar.

   De acuerdo.
4. Pienso que necesitaría la ayuda de un moderador para ser capaz de usar la herramienta.

   De acuerdo.
5. Encuentro varias funciones de la herramienta bien integradas.

   De acuerdo.
6. Pienso que hubo demasiadas inconsistencias en la herramienta.

   En desacuerdo.
7. Imagino que la mayoría de la gente aprenderá a usar la herramienta muy rápido.

   En desacuerdo.
8. Encuentro la herramienta muy incómoda de usar.

   En desacuerdo.
9. Me siento seguro utilizando la herramienta.

   De acuerdo.
10. Necesito aprender muchas cosas antes de poder ponerme con esta herramienta.

    De acuerdo.

Preguntas posteriores:

1. ¿Cuántos desafíos has completado?

   2.
2. ¿Qué es lo que más te ha gustado?

   La interfaz gráfica es muy amigable, tanto de la página como la del juego.
3. ¿Qué es lo que menos te ha gustado?

   Quizás el tipo de fuente usado, es un poco costoso de leer. Pondría pistas más directas.
4. ¿Te ha resultado útil la documentación?

   No la he usado.
5. ¿Te han servido de ayuda los mensajes de ayuda del test?

   Poco.
6. ¿Te han servido los comentarios del código?

Bastante extenso para leerlos todos.

9.4.10 Cuestionario 10

Preguntas previas:

1. ¿Cuál es tu nivel de conocimiento de *JavaScript*?

   Medio.
2. ¿Cuál es tu nivel de conocimiento en la programación de videojuegos?

   Medio.
3. ¿Has utilizado alguna vez un *framework* de desarrollo de videojuegos como *Phaser*? Si has utilizado alguno, ¿Cuáles?

   ***Quintus, Unity*.**

Cuestionario SUS:

1. Pienso que me gustaría usar esta herramienta frecuentemente.

   Ni en acuerdo ni en desacuerdo.
2. Encuentro la herramienta innecesariamente compleja.

   En desacuerdo.
3. Pienso que la herramienta era fácil de usar.

   Ni en acuerdo ni en desacuerdo.
4. Pienso que necesitaría la ayuda de un moderador para ser capaz de usar la herramienta.

   Ni en acuerdo ni en desacuerdo.
5. Encuentro varias funciones de la herramienta bien integradas.

   Ni en acuerdo ni en desacuerdo.
6. Pienso que hubo demasiadas inconsistencias en la herramienta.

   Ni en acuerdo ni en desacuerdo.
7. Imagino que la mayoría de la gente aprenderá a usar la herramienta muy rápido.

   De acuerdo.
8. Encuentro la herramienta muy incómoda de usar.

   Ni en acuerdo ni en desacuerdo.
9. Me siento seguro utilizando la herramienta.

   Ni en acuerdo ni en desacuerdo.
10. Necesito aprender muchas cosas antes de poder ponerme con esta herramienta.

Ni en acuerdo ni en desacuerdo.

Preguntas posteriores:

1. ¿Cuántos desafíos has completado?

   1.
2. ¿Qué es lo que más te ha gustado?

   Aspecto visual y la utilidad de la herramienta de ir aprendiendo mientras juegas.
3. ¿Qué es lo que menos te ha gustado?

   Que no se ha tenido en cuenta que los mensajes de ayuda se escriben sobre el botón de "x".
4. ¿Te ha resultado útil la documentación?

   Sí.
5. ¿Te han servido de ayuda los mensajes de ayuda del test?

   Sí.
6. ¿Te han servido los comentarios del código?

   Sí.

9.4.11 Cuestionario 11

Preguntas previas:

1. ¿Cuál es tu nivel de conocimiento de *JavaScript*?

   Avanzado.
2. ¿Cuál es tu nivel de conocimiento en la programación de videojuegos?

   Medio.
3. ¿Has utilizado alguna vez un *framework* de desarrollo de videojuegos como *Phaser*? Si has utilizado alguno, ¿Cuáles?

   ***Quintus, Unity*.**

Cuestionario SUS:

1. Pienso que me gustaría usar esta herramienta frecuentemente.

   De acuerdo.
2. Encuentro la herramienta innecesariamente compleja.

   Ni en acuerdo ni en desacuerdo.
3. Pienso que la herramienta era fácil de usar.

De acuerdo.

4. Pienso que necesitaría la ayuda de un moderador para ser capaz de usar la herramienta.

En desacuerdo.

5. Encuentro varias funciones de la herramienta bien integradas.

Totalmente de acuerdo.

6. Pienso que hubo demasiadas inconsistencias en la herramienta.

Totalmente en desacuerdo.

7. Imagino que la mayoría de la gente aprenderá a usar la herramienta muy rápido.

De acuerdo.

8. Encuentro la herramienta muy incómoda de usar.

En desacuerdo.

9. Me siento seguro utilizando la herramienta.

Totalmente de acuerdo.

10. Necesito aprender muchas cosas antes de poder ponerme con esta herramienta.

Ni en acuerdo ni en desacuerdo.

Preguntas posteriores:

1. ¿Cuántos desafíos has completado?

1.

2. ¿Qué es lo que más te ha gustado?

La interfaz del videojuego, los gráficos del juego.

3. ¿Qué es lo que menos te ha gustado?

Demasiado texto al principio.

4. ¿Te ha resultado útil la documentación?

Sí, bastante.

5. ¿Te han servido de ayuda los mensajes de ayuda del test?

Sí.

6. ¿Te han servido los comentarios del código?

Sí, bastante.

9.4.12 Cuestionario 12

Preguntas previas:

1. ¿Cuál es tu nivel de conocimiento de *JavaScript*?

   Medio.
2. ¿Cuál es tu nivel de conocimiento en la programación de videojuegos?

   Bajo.
3. ¿Has utilizado alguna vez un *framework* de desarrollo de videojuegos como *Phaser*? Si has utilizado alguno, ¿Cuáles?

   ***Unity*.**

Cuestionario SUS:

1. Pienso que me gustaría usar esta herramienta frecuentemente.

   De acuerdo.
2. Encuentro la herramienta innecesariamente compleja.

   Ni en acuerdo ni en desacuerdo.
3. Pienso que la herramienta era fácil de usar.

   De acuerdo.
4. Pienso que necesitaría la ayuda de un moderador para ser capaz de usar la herramienta.

   Ni en acuerdo ni en desacuerdo.
5. Encuentro varias funciones de la herramienta bien integradas.

   De acuerdo.
6. Pienso que hubo demasiadas inconsistencias en la herramienta.

   Ni en acuerdo ni en desacuerdo.
7. Imagino que la mayoría de la gente aprenderá a usar la herramienta muy rápido.

   De acuerdo.
8. Encuentro la herramienta muy incómoda de usar.

   En desacuerdo.
9. Me siento seguro utilizando la herramienta.

   Ni en acuerdo ni en desacuerdo.
10. Necesito aprender muchas cosas antes de poder ponerme con esta herramienta.

    De acuerdo.

Preguntas posteriores:

1. ¿Cuántos desafíos has completado?

   1.
2. ¿Qué es lo que más te ha gustado?

   La interfaz es muy amigable.
3. ¿Qué es lo que menos te ha gustado?

   Los diálogos.
4. ¿Te ha resultado útil la documentación?

   (El usuario no ha respondido a esta pregunta).
5. ¿Te han servido de ayuda los mensajes de ayuda del test?

   Sí.
6. ¿Te han servido los comentarios del código?

   Sí.

9.4.13 Cuestionario 13

Preguntas previas:

1. ¿Cuál es tu nivel de conocimiento de *JavaScript*?

   Avanzado.
2. ¿Cuál es tu nivel de conocimiento en la programación de videojuegos?

   Medio.
3. ¿Has utilizado alguna vez un *framework* de desarrollo de videojuegos como *Phaser*? Si has utilizado alguno, ¿Cuáles?

   ***Quintus*.**

Cuestionario SUS:

1. Pienso que me gustaría usar esta herramienta frecuentemente.

   De acuerdo.
2. Encuentro la herramienta innecesariamente compleja.

   Totalmente en desacuerdo.
3. Pienso que la herramienta era fácil de usar.

   Totalmente de acuerdo.
4. Pienso que necesitaría la ayuda de un moderador para ser capaz de usar la herramienta.

   Totalmente en desacuerdo.

5. Encuentro varias funciones de la herramienta bien integradas.

   Totalmente de acuerdo.
6. Pienso que hubo demasiadas inconsistencias en la herramienta.

   Ni en acuerdo ni en desacuerdo.
7. Imagino que la mayoría de la gente aprenderá a usar la herramienta muy rápido.

   Ni en acuerdo ni en desacuerdo.
8. Encuentro la herramienta muy incómoda de usar.

   Totalmente en desacuerdo.
9. Me siento seguro utilizando la herramienta.

   Totalmente de acuerdo.
10. Necesito aprender muchas cosas antes de poder ponerme con esta herramienta.

    En desacuerdo.

Preguntas posteriores:

1. ¿Cuántos desafíos has completado?

   2.
2. ¿Qué es lo que más te ha gustado?

   La interfaz y los personajes.
3. ¿Qué es lo que menos te ha gustado?

   Los diálogos no se ajustan bien a la ventana.
4. ¿Te ha resultado útil la documentación?

   No la he visto.
5. ¿Te han servido de ayuda los mensajes de ayuda del test?

   Sí.
6. ¿Te han servido los comentarios del código?

   Sí.

9.4.14 Cuestionario 14

Preguntas previas:

1. ¿Cuál es tu nivel de conocimiento de *JavaScript*?

   Medio.
2. ¿Cuál es tu nivel de conocimiento en la programación de videojuegos?

Medio.

3. ¿Has utilizado alguna vez un *framework* de desarrollo de videojuegos como *Phaser*? Si has utilizado alguno, ¿Cuáles?

   ***Quintus, Unity* y *Unreal*.**

Cuestionario SUS:

1. Pienso que me gustaría usar esta herramienta frecuentemente.

   En desacuerdo.
2. Encuentro la herramienta innecesariamente compleja.

   Totalmente de acuerdo.
3. Pienso que la herramienta era fácil de usar.

   Totalmente en desacuerdo.
4. Pienso que necesitaría la ayuda de un moderador para ser capaz de usar la herramienta.

   Ni en acuerdo ni en desacuerdo.
5. Encuentro varias funciones de la herramienta bien integradas.

   Totalmente de acuerdo.
6. Pienso que hubo demasiadas inconsistencias en la herramienta.

   De acuerdo.
7. Imagino que la mayoría de la gente aprenderá a usar la herramienta muy rápido.

   En desacuerdo.
8. Encuentro la herramienta muy incómoda de usar.

   De acuerdo.
9. Me siento seguro utilizando la herramienta.

   Ni en acuerdo ni en desacuerdo.
10. Necesito aprender muchas cosas antes de poder ponerme con esta herramienta.

    Totalmente de acuerdo.

Preguntas posteriores:

1. ¿Cuántos desafíos has completado?

   3.
2. ¿Qué es lo que más te ha gustado?

   Gráficos amigables, interfaz cuidada.
3. ¿Qué es lo que menos te ha gustado?

Excesiva cantidad de texto.

4. ¿Te ha resultado útil la documentación?

   Sí.

5. ¿Te han servido de ayuda los mensajes de ayuda del test?

   Algunos no son muy claros.

6. ¿Te han servido los comentarios del código?

   Sí, ayudan, pero, están orientados a gente con conocimientos de programación, personas que no los tengan pueden perderse.

9.4.15 Cuestionario 15

Preguntas previas:

1. ¿Cuál es tu nivel de conocimiento de *JavaScript*?

   Medio.

2. ¿Cuál es tu nivel de conocimiento en la programación de videojuegos?

   Medio.

3. ¿Has utilizado alguna vez un *framework* de desarrollo de videojuegos como *Phaser*? Si has utilizado alguno, ¿Cuáles?

   ***Quintus, Unity*.**

Cuestionario SUS:

1. Pienso que me gustaría usar esta herramienta frecuentemente.

   Totalmente de acuerdo.

2. Encuentro la herramienta innecesariamente compleja.

   En desacuerdo.

3. Pienso que la herramienta era fácil de usar.

   De acuerdo.

4. Pienso que necesitaría la ayuda de un moderador para ser capaz de usar la herramienta.

   Totalmente en desacuerdo.

5. Encuentro varias funciones de la herramienta bien integradas.

   Totalmente de acuerdo.

6. Pienso que hubo demasiadas inconsistencias en la herramienta.

   Totalmente en desacuerdo.

7. Imagino que la mayoría de la gente aprenderá a usar la herramienta muy rápido.

   Totalmente de acuerdo.

8. Encuentro la herramienta muy incómoda de usar.

   Totalmente en desacuerdo.
9. Me siento seguro utilizando la herramienta.

   Totalmente de acuerdo.
10. Necesito aprender muchas cosas antes de poder ponerme con esta herramienta.

    Totalmente en desacuerdo.

Preguntas posteriores:

1. ¿Cuántos desafíos has completado?

   2.
2. ¿Qué es lo que más te ha gustado?

   En mi opinión creo que este juego puede enganchar mucho a los programadores junior debido a que puedes aprender a programar divirtiéndote.
3. ¿Qué es lo que menos te ha gustado?

   El panel en el que aparecen los errores debería poderse hacer más grande. Estaría bien que te permitiera modificar todo el código que viene.
4. ¿Te ha resultado útil la documentación?

   Sí.
5. ¿Te han servido de ayuda los mensajes de ayuda del test?

   Sí.
6. ¿Te han servido los comentarios del código?

   Sí.

10. Bibliografía

1. Web oficial w3: HTML5. [Online] http://www.w3.org/TR/html5/.

2. Mozilla-JavaScript. [Online] https://developer.mozilla.org/es/docs/Web/JavaScript.

3. Williams, James Lamar. *Learning HTML5 Game programming: A hands-on guide to building online games using canvas, SVG and WebGL.* s.l. : Pearson Education, INC, 2012.

4. Crockford, Douglas. *Javascript: The good parts.* s.l. : O´Reilly, 2008.

5. Web oficial Phaser. [Online] https://phaser.io/.

6. Web oficial Scratch. [Online] http://scratch.mit.edu/.

7. Alice3D. [Online] https://www.cmu.edu/corporate/news/2007/features/alice.shtml.

8. Web oficial Khan Academy. [Online] https://es.khanacademy.org/.

9. Web oficial Code Spells. [Online] http://codespells.org/.

10. Web oficial Code Avengers. [Online] http://www.codeavengers.com/.

11. Web oficial Light Bot. [Online] http://lightbot.com/.

12. Web oficial Code Combat. [Online] https://codecombat.com/.

13. *The evolution of fantasy Role-Playing Games.* Tresca, Michael J. s.l. : Jefferson, NC: Mc Farland & Company, 2010.

14. Angelina, David Vallejo Fernández y Cleto Martín. *Desarrollo de videojuegos: Arquitectura del motor.* s.l. : Bubok (Edición física) y LibroVirtual.org (Edición electrónica).

15. Francisco Jurado Monroy, Javier A. Albusac Jiménez y otros. *Desarrollo de Videojuegos: Desarrollo de Componentes.* s.l. : Bubok (Edición física) y LibroVirtual.org (Edición electrónica).

16. Web oficial Create.js. [Online] http://www.createjs.com/.

17. Web oficial Quintus.js. [Online] http://html5quintus.com.

18. Web oficial Cocos2D.js. [Online] http://www.cocos2d-x.org/wiki/Cocos2d-js.

19. Web oficial Turbulenz.js. [Online] http://biz.turbulenz.com.

20. Texture packer. [Online] https://www.codeandweb.com/texturepacker.

21. Starling. [Online] http://doc.starling-framework.org/core/starling/textures/TextureAtlas.html.

22. Web audio. [Online] http://webaudio.github.io/web-audio-api/.

23. HTML Audio. [Online] http://dl.acm.org/citation.cfm?id=274504.

24. Alan Cooper, Robert Reimann and Dave Cronin. *About face 3.* s.l. : Wiley, cop., 2007.

25. Nielsen, Jakob. *Usability Engineering.* s.l. : AP Professional, 1993.

26. Tracy Fullerton, Christopher Swain & Steven S. Hoffman. *Game Design Workshop: A Playcentric Approach to Creating Innovative Games.* s.l. : Morgan Kaufmann Publishers, 2008.

27. Schell, Jesse. *The Art of Game Design: A Book of Lenses.* s.l. : Morgan Kaufmann Publishers, 2008.

28. Web oficial OpenGameArt. [Online] http://opengameart.org/.

29. Web oficial Kenney. [Online] http://kenney.nl/.

30. Puig, Jordi Collel. *CSS3 y Javascript avanzado.*

31. Resig, John. QUnit, js unit testing. [Online] 2008. https://qunitjs.com/.

32. Rubin, Kenneth S. *Essential Scrum: A Practical Guide to the Most Popular Agile Process.* Upper Saddle River, NJ : Addison-Wesley, 2012.