Escuela Politécnica Superior

Codemon

Grado en Ingeniería Informática

Trabajo Fin de Grado

Autor:
Pablo Requena González

Tutor/es:
Patricia Compañ Rosique

Francisco Gallego Duran

Universitat d'Alacant
Universidad de Alicante

Septiembre 2019

Objetivos

El principal objetivo de Codemon es realizar un videojuego lo más divertido posible para que alumnos de primero de distintas ingenierías (Informática, Multimedia, Telecomunicaciones) puedan practicar programación de una forma distinta, pudiendo aumentar las habilidades de una forma amigable, como es jugando.

Buscamos que los alumnos de Programación 1, busquen una motivación extra a la hora de realizar las prácticas de esta asignatura, que sea fácil aprender C/C++ y ellos mismos se peleen por investigar, aumentar los conocimientos, y aprendan.

El videojuego debe enseñar unas mecánicas mínimas, para que se puedan ir desarrollando soluciones cada vez más complejas, incluso llegar a realizar nuevos retos y mapas por otros programadores como lo que se hace en algunos de los ejemplos que comentaremos más adelante.

Se busca que Codemon parezca un videojuego, que sea divertido, entretenido, que tenga un modo historia, que sea experiencia inmsersiva y que consiga que los alumnos lo jueguen, en vez de simplemente usarlo para practicar.

Se pretende que gracias a Codemon se cree una comunidad que amplíe el videojuego, que se reciba un feedback sobre el proyecto y el juego, comprobando si realmente se siente como un juego, o como unas prácticas, que se llegue a hacer un juego multijugador, multiplataforma, realizar distintas versiones del videojuego, que se realicen mapas creados por esta misma comunidad, expandir los conocimientos posibles para jugar a videojuegos para poder utilízalo también en otras asignaturas, o implementarlo para otros lenguajes de programación.

Agradecimientos

En este apartado quiero agradecer a todas las personas que han tenido que ver en los años que llevo en informática y me han llegado hasta aquí.

Primero a mis tutores, Patricia Compañ y Francisco Gallego, sois los mejores tutores que he podido tener, os tuve en primero, y he tenido mucha suerte de encontraros en mi TFG, gracias por hacer que ame la programación tanto como lo hago. A intentar frenar mis locas ideas, estructurarlas, e intentar convertirlas en cosas posibles, a hacer este proyecto realidad con todos vuestros consejos y ayuda.

A mi familia, por aguantarme todos los días de carrera, apoyarme, darme consejos y fuerzas en los momentos más difíciles. A entender que es una carrera difícil, y que se ha convertido en una pasión.

A mis amigos, en especial a Marcos González y Marisa Risueño, que han estado ahí siempre, con ellos he aprendido a ser mejor persona, y mejor trabajador. A Gema Moreno, por ayudarme en tantos aspectos de mi vida. A Jorge Félix, por enseñarme muchas cosas y ayudarme en momentos difíciles. A todos los demás, que han sabido sacar, a veces, lo mejor de mí.

En resumen, a toda la gente con la que me he topado en mi vida como estudiante de Ingeniería Informática, y que me han ayudado, sigan o no en mi vida.

Gracias a todos.

A Danae Requena,

porque nadie mejor que tú, sabes lo

que me ha dolido esta carrera.

A Mariluz González,

porque nadie mejor que tú, sabes lo

que es sufrirme.

A Maximo Requena, ojalá pudieras vivirlo.

A todos.

Índice

Introducción

Puesto que Codemon es una idea que ya se ha intentado en otras ocasiones (por compañeros que ya realizaron Codemon como Trabajo de Fin de Grado), es ilógico habar de Codemon como una idea novedosa. En fases tempranas del proyecto, se barajó la idea de adaptar la última versión de Codemon disponible en la Universidad de Alicante. Este proyecto se podría utilizar para modificarlo y así realizar una nueva versión, extensible, escalable y utilizable. Puesto que este proyecto tiene múltiples dependencias, algunas que ni siquiera funcionan y, la documentación es un tanto escasa, se decidió rechazar esta adaptación.

Como en muchas de las habilidades que el ser humano quiere aprender, practicar es uno de los factores más importantes. Por ello, la programación, es algo que se debe practicar mucho, y para ello se ha creado Codemon. En muchos casos, el primer contacto con la programación se da en primero de carrera, en Programación 1 en varias Ingenierías. Puesto que este es el primer contacto de muchos alumnos, tener una metodología acertada y que los alumnos se interesen por la programación es algo complicado, por lo que hay una gran mayoría de estudiantes que optan por abandonar la asignatura. Con Codemon se plantea que haya un videojuego que invite a los alumnos a practicar y a poder ser, a divertirse. Codemon sería utilizado para aportar un extra en el aprendizaje de los estudiantes, ayudando a mejorar el interés de esa parte del alumnado que decide abandonar la asignatura, pudiendo complementar los ejercicios actuales, o, remplazarlos como se hace en Matemáticas 1 con PL-Man (el cual se comentará más adelante).

Codemon pretende ser fácil de utilizar, escalable, que se cree una comunidad, en la cual los alumnos puedan proponer ideas, se les incite a practicar e incluso competir entre ellos, aunque lo más importante es que se aprenda. Se plantea comprobar si los alumnos ven interesante el proyecto, se utiliza como juego y si están interesados en utilizarlo, continuarlo y extenderlo. Comprobar si Codemon se siente como un juego, o simplemente como un ejercicio más a realizar y por lo tanto la motivación que esto les puede conllevar.

Puesto que Codemon se ha inspirado en varios juegos y programas ya existentes, se han ido descartando una serie de ideas e incorporándolas con otras. Los enfoques que tienen PL-Man y *CodinGame* (comentados más adelante) son muy atractivos y, por lo tanto, se decidió inspirarse directamente en ellos. En estos dos casos, el programador tiene que realizar directamente la solución del ejercicio planteado, programando, por ejemplo, los comportamientos de un personaje en cuestión, y así resolver el puzle. Puesto que esto es algo que no queremos realizar en Codemon, se espera que el alumno a la hora de superar un nivel, y, por lo tanto, realizar un puzle, debe de programar una serie de mecánicas para ser utilizadas dentro del nivel, o puzle.

Un ejemplo bastante extenso y fácil de explicar son los *Hidden Machine (HM)*, en castellano Máquinas Ocultas (MO), o las *Technical Machines (TM)*, o Máquinas Técnicas en español (MT) presentes en los videojuegos Pokémon. Estos *HM* o *TM* son una serie de movimientos que nuestros compañeros Pokémon pueden aprender en forma de ataque. La diferencia entre los ataques *HM* y los normales (los cuales se van aprendiendo mediante experiencia y subiendo de nivel) estos son los ataques se pueden aprender mediante la elección del jugador y disponibilidad del Pokémon, los primeros pueden ser utilizados tanto en combate como fuera de él. Siendo estas, una serie de funcionalidades dentro del videojuego y pudiendo ser utilizadas por el jugador en cualquier momento; mecánicas tales como cortar árboles pequeños, destrozar rocas, empujarlas, caminar por el agua, o volar.

Estas mecánicas pueden ser directamente incluidas en Codemon, siempre y cuando hagamos algo similar a Pokémon con estas *HM* con algunas de las funcionalidades que veremos más adelante.

Como podemos ver en la figura 1, vemos un pequeño arbolito distinto a todos los demás y podemos ver como en pantalla se nos pregunta si queremos usar corte.

Este es un ejemplo de la *HM-01* Corte, la cual sirve, para cortar árboles que tienen esa forma, y puede ser directamente replicable en Codemon siempre y cuando hagamos un mapa similar.

Figure 1: Ejemplo de "corte" como mecánica programable

En la figura 2, podemos observar cómo se nos pregunta si queremos usar la *HM-04*, Fuerza, la cual sirve para empujar piedras pesadas. Este mapa podría ser directamente replicable en Codemon, siendo un posible mapa a realizar.

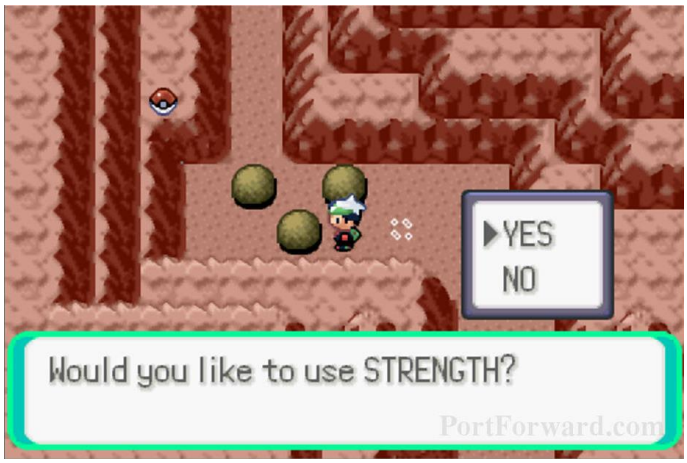

Figure 2: Ejemplo de "fuerza" como mecánica programable

Estado del Arte

El concepto de videojuego educativo o gamificación es una técnica de aprendizaje que se está extendiendo desde hace unos años. Se trata de trasladar la mecánica de los juegos al ámbito educativo o profesional, con el fin de mejorar habilidades o acciones. Puesto que es una metodología más divertida y fácil, suele generar una experiencia positiva en el usuario y da una sensación de aprender mejor o más rápidamente.
Aunque esta metodología llevada a la programación no está tan extendida, sí que hay algunas páginas web o juegos ya creados.
Estos son algunos de los ejemplos que comentaremos:

PL-Man

PL-Man es un videojuego creado por Francisco José Gallego Durán con la ayuda de varios profesores de la asignatura Matemáticas 1. Este está siendo utilizado en el primer año de Ingeniería Informática e Ingeniería Multimedia para impartir la parte de Lógica computacional en la asignatura. Se trata de un videojuego que se está utilizando para que los alumnos obtengan y practiquen los conocimientos de *SWI-Prolog* de una forma más sencilla y divertida.

Con un entorno gráfico en la consola/terminal de Linux, PL-Man tiene unas mecánicas muy similares al famoso (y clásico) juego PacMan. En el videojuego podemos controlar nuestro personaje principal identificado con el símbolo de arroba (@). El objetivo, consiste en conseguir comerse el máximo número posible de cocos, identificados con el carácter punto (.). Tendremos que huir de los fantasmas, identificados con la letra E (mayúscula, de color rojo). Podemos utilizar algunos objetos identificados con la letra O. Tendremos que programar la solución del desafío, puzle o mapa mediante una serie de reglas ya definidas que el propio juego interpretará.

La regla que analizaremos ahora mismo, se trata de una regla que realizará la acción de moverse hacia la izquierda en el momento que vea un coco, a la izquierda.

```
do(move(left)):- see(normal, left, '.').
```

Si por algún motivo no hemos podido completar el mapa (ya sea porque nos quedamos atascados, o porque entramos en un bucle infinito), el juego tiene un contador máximo de iteraciones o movimientos el cual hará que el juego siempre se detenga. Una vez acabemos el mapa, sea de la manera que sea, se nos dará una puntuación de cómo de bien se nos ha dado o no.

En la figura 3, podremos ver un ejemplo de un mapa de PL-Man, en el cual podemos identificar al personaje principal (@), unas paredes que están delimitadas por el carácter almohadilla (#), podemos ver los cocos, los enemigos, y consiste en traspasar la puerta que está identificada con los guiones bajos (_____).

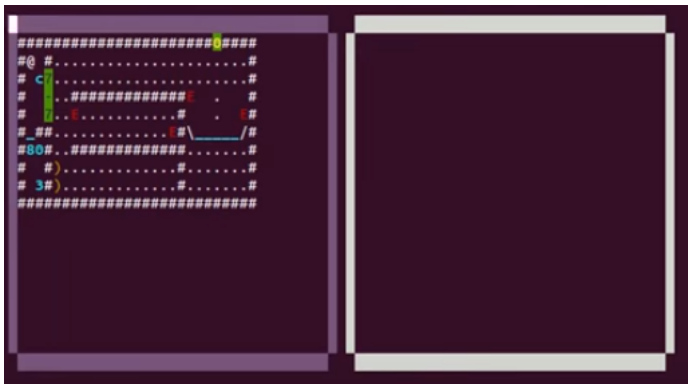

Figure 3: Ejemplo de mapa de PL-Man

Como se ha comentado antes, al terminar la ejecución de un mapa, se nos mostrarán una serie de estadísticas. En este caso, podemos ver que el mapa ha sido superado con un 100% de los cocos comidos, y que se han realizado 50 movimientos en total.

Figure 4: Estadísticas al completar un mapa

CodinGame

CodinGame es una plataforma online creada en el año 2012 por Frédéric Desmoulins, Nicolas Antoniazzi y Aude Barral. Siendo una de las webs de desafíos de programación más grande que hay mundialmente, hay una infinidad de niveles con distintos desafíos y dificultades. Además, está disponible para bastantes lenguajes de programación, e incluso el mismo desafío es normal que se pueda solucionar con distintos lenguajes, algunos que se han podido observar son *Bash, C, C++, Java, JavaScript y Go*, entre otros. Además, *CodinGame* colabora activamente con múltiples empresas bastante conocidas para intentar encontrar programadores especializados en algunos campos o simplemente que sepan realizar problemas más complejos.

Como en la figura 4, donde podemos ver que la famosa empresa Nintendo tiene un desafío patrocinado. Este mapa está calificado como "muy difícil" por el equipo de *CodinGame*, y solo ha sido resuelto por el 2% de la comunidad. El ejercicio está únicamente enfocado al lenguaje C++, y se nos asegura que, al completarlo, podremos optar a conocer al equipo de Nintendo.

En concreto, este ejercicio era de criptografía, así que, gracias a todos estos datos, podemos comprobar que la compañía Nintendo está buscando programadores en C++ especializados en criptografía.

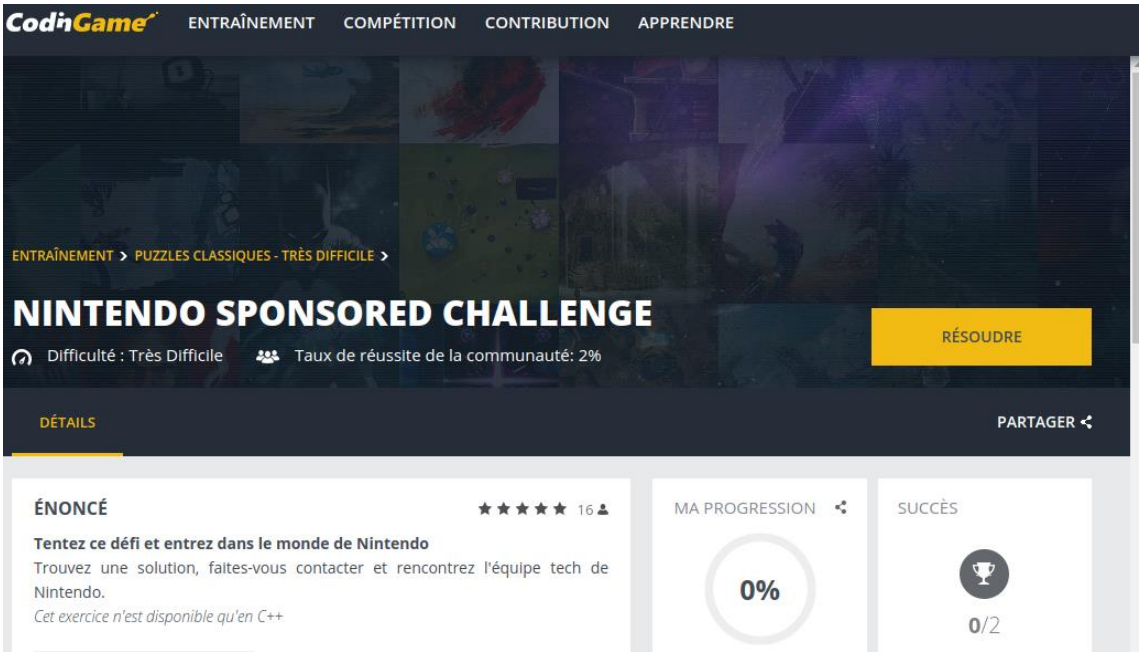

Figure 4: Puzle patrocinado por Nintendo

Las mecánicas de *CodinGame* son muy simples y sencillas. Se nos presenta un desafío que varía por dificultad, nos explican los conocimientos necesarios para poder realizar la solución y una pequeña introducción sobre este mismo. Como podemos ver en la figura 5

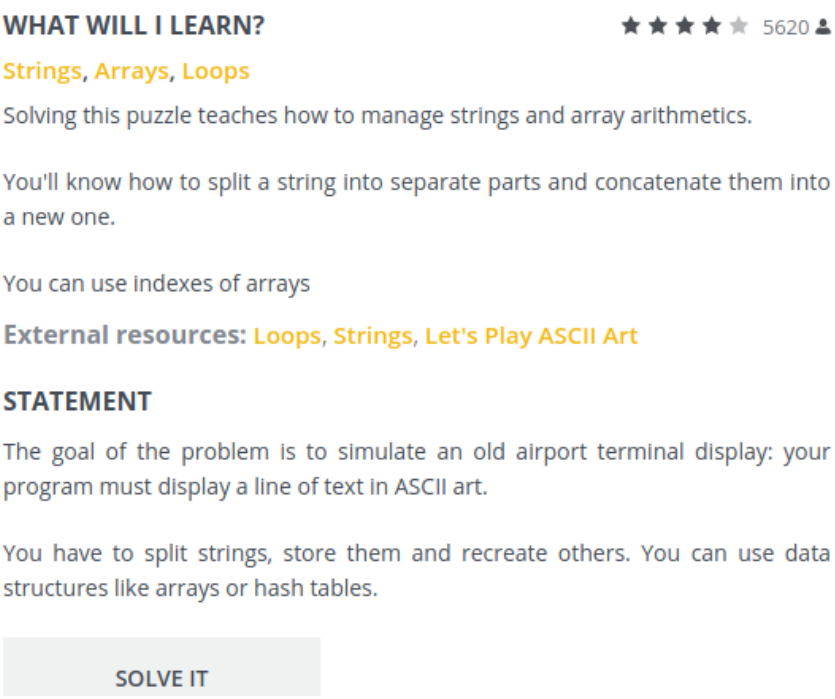

Figure 5: Puzle sencillo de CodinGame

Tras ello, se nos indicarán las normas del puzle, y la forma en la podremos hacer la solución, además de un pequeño ejemplo para poder empezar a trabajar, como muestra la figura 6.

Figure 6: Introducción al puzle

Todos los puzles de *CodinGame* se basan en la interacción directa entre el usuario y el puzle, donde por lo general se utiliza la entrada estándar para que el usuario envíe instrucciones al puzle.

```
std::cin >> x;
```

Como complemento, el puzle nos dará la información gracias a la entrada estándar de C++. Así obtendremos algunos valores, como pueden ser coordenadas, strings con pistas, etc.

```
std::cout << "25";
```

El problema es que no se ha creado un videojuego específico para aprender a programar, aunque sí se han visto casos de videojuegos que, por algún motivo, se han llevado al ámbito educativo. Minecraft, es uno de los ejemplos más famosos, ya sea por su flexibilidad de contenidos, su facilidad a la hora de exportar mecánicas o simplemente por su simpleza. Así se llevó este juego al ámbito educativo, y posteriormente se creó uno específicamente para ello.

Minecraft (Education)

Minecraft es un videojuego de tipo sandbox multiplataforma (Xbox, PS3, PS4, PC, Smartphone, etc.). Lanzado en mayo de 2009 (en fase alpha) y terminado de forma oficial en 2011, creado por Notch (Markus Persson) y posteriormente por Mojang. Siendo este uno de los videojuegos más jugados actualmente, y gracias a su gran libertad, se creó una gran comunidad. Esta comunidad se ha dedicado a realizar mapas para otros jugadores y eso ha llevado a ser utilizado en múltiples ámbitos. Teniendo en cuenta el éxito que el juego tenía entre los niños, y ya que Minecraft se había utilizado para la educación, Microsoft (la cual compró Mojang en 2014) lanzó Minecraft Education (o Minecraft EDU). Esta vertiente del juego, Minecraft EDU, sigue la estela del juego original, fácil, abierto, sandbox, solo que se separan dos jugadores de forma clara. El profesor, tendrá un perfil propio, donde tendrá herramientas usadas en el colegio/instituto como pueden ser pizarras. El otro perfil, el de los alumnos, tienen capacidades como retener características, una cámara fotográfica, etc.

Con Minecraft EDU se busca que los alumnos puedan aprender de forma más coloquial. Con clases y metodologías que fomentan algunas aptitudes como la colaboración y comunicación. De todos los elementos que tiene, los alumnos disponen también de una memoria para realizar una documentación sobre el trabajo realizado (como esta misma) la cual puede ser exportada para ser presentada. Todas las instrucciones pueden ser dadas por el propio profesor a la hora de empezar la explicación, o se pueden crear *NPC's* que tengan algunas explicaciones y hacer una especie de historieta, como en la figura 7, que el NPC nos dice "¡Hola Mundo!" y tiene un apartado de "Más información" para *URL's* para contenido adicional y/o complementario.

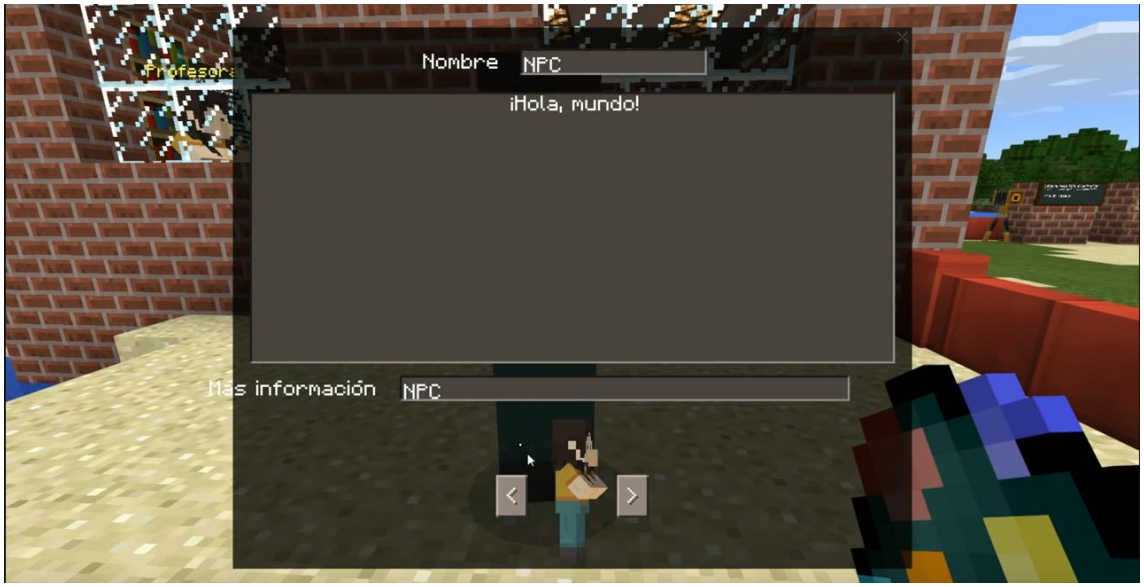

Figure 7: Instrucciones en Minecraft EDU

Minecraft EDU tiene como objetivo ser utilizado en múltiples etapas de la vida académica. Se hacen subgrupos de cursos haciendo una distinción por edades, aunque según su página web, hay cursos desde los 3 años hasta superados los 18 años. También podemos encontrar una serie de cursos por materia, podemos observar como hay sobre Matemáticas, Ciencias, Geografía, Informática, y cómo podemos ver en la figura 8, tenemos disponible la tabla periódica dentro del menú, para poder realizar, por ejemplo, actividades de química.

Figure 8: Elementos de la tabla periódica en Minecraft EDU

Además, en la versión normal del videojuego hay una serie de elementos que han servido para expandir la pasión de la informática/electrónica dentro del videojuego. Como en la figura 9, que es el elemento Redstone, el cual funciona como un cable eléctrico cuando se coloca en una superficie.

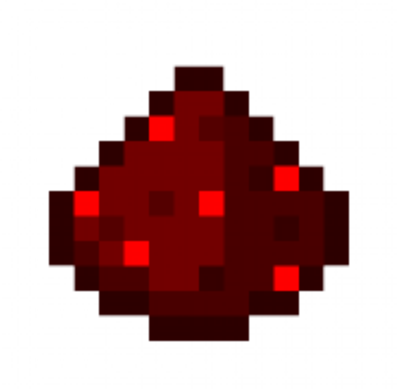

Figure 9: Redstone sin colocar en el suelo

Este elemento, junto a otros elementos como pueden ser las antorchas de Redstone (podría ser un switch de la conductividad, además de dar potencia), los repeaters (repetidores de señal, además de usarse como reloj, pestillo o un diodo o amplificador), son algunos de los elementos que podremos utilizar como en la vida misma dentro de Minecraft. Con todos estos elementos se han creado auténticas maravillas como computadoras, reproductores de música, la consola GameBoy de Nintendo, etc.

En la figura 10, podemos ver que con Minecraft es posible crear puertas lógicas, lo cual nos permitiría utilizar Minecraft dentro de una asignatura de primero de Ingeniería Informática, ratificando así lo dicho en la web de cursos +18 años. En concreto esta asignatura es Fundamentos de los Computadores (FC), en la cual se podrían aplicar las técnicas de Gamificación con Minecraft para enseñar a los alumnos de primero las puertas lógicas.

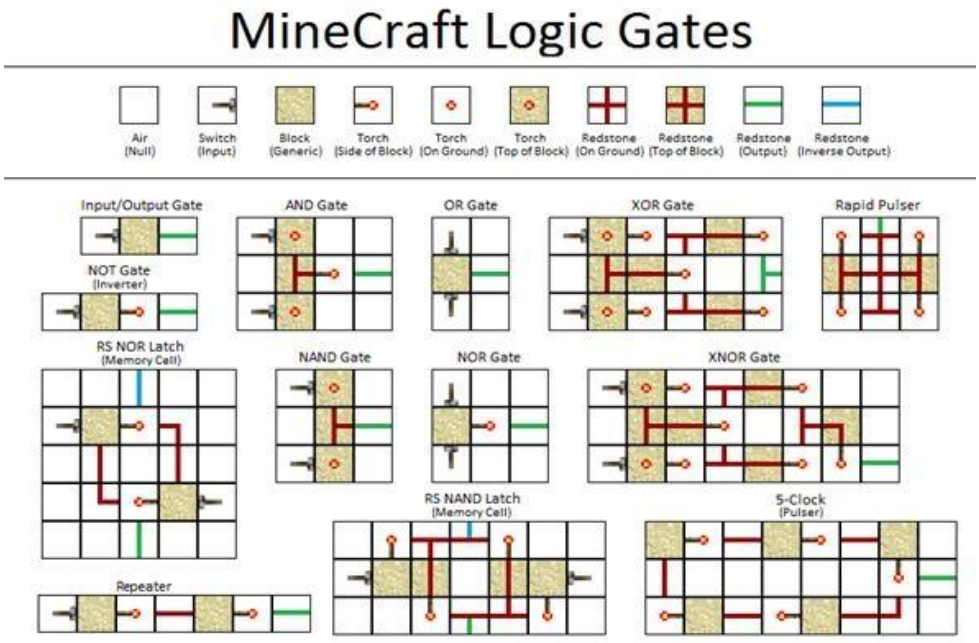

Figure 10: Puertas lógicas con Redstone

Materiales y Metodología

Herramientas

En este apartado se explicará el uso de las herramientas utilizadas para el desarrollo de esta aplicación. El lenguaje de programación que se ha utilizado C++, en concreto es estándar 17 aunque no se han empleado ventajas de este. Se ha usado la librería Simple Fast Multimedia Library (SFML) apoyándose en esta para la parte gráfica y la estructura interna.

Hemos utilizado el lenguaje JavaScript Object Notation (JSON) puesto que es uno de los formatos de texto más extendidos del mundo. Perfecto para la creación de archivos de configuración y de mapas. Elegido por su simpleza y facilidad a la hora de utilizarse, además de la fácil forma de aprenderse, añadiéndole que en la mayoría de lenguajes de programación existe alguna librería para leer archivos de este tipo. Como se quiere que Codemon tenga una comunidad, es un lenguaje perfecto sobre todo para la creación de mapas.

Se ha utilizado Makefile (Anexo 4), para la compilación del proyecto, así podemos crear un proyecto que sea independiente a cualquier IDE o compilador específico (Por ejemplo, Visual Studio y Visual C++).

Hemos contado con Valgrind, puesto que uno de los principales problemas de todo videojuego es el consumo de recursos, se ha utilizado Valgrind para comprobar el consumo de RAM e intentar mejorar, centrándose, sobretodo en este aspecto en fases tempranas de la implementación.

Se ha trabajado con Github, para realizar el control de versiones, además, se pretende utilizar en un futuro para extender el juego con las Issues de esta plataforma, y que se pueda extender el proyecto lo máximo posible (tanto a otros lenguajes, como niveles, desafíos, etc.).

Toggl, para hacer un control de las horas realizadas, así poder comprobar si se ha seguido la planificación.

Planificación

Se ha diseñado un plan para realizar el proyecto. puesto que, por ejemplo, los conocimientos de estructuración y creación de videojuegos eran un tanto escasos, por lo tanto, había que hacer un gran estudio. Esto hacía que la realización del proyecto fuera un gran inconveniente, se ha realizado el proyecto de la siguiente manera.

Un primer año de investigación previa, dónde se barajaron una serie de librerías, estructuras, y formas de realizar el proyecto. Un segundo año de implementación y creación del proyecto, el cual se ha ido compaginando con una vida laboral, por lo que era muy importante tener una planificación para intentar seguirla lo más fielmente posible para obtener al menos un prototipo jugable.

Se planificó el proyecto para tener las siguientes horas disponibles para realizar el proyecto, 5 horas semanales en el primer cuatrimestre de implementación e investigación, y 4 horas en fin de semana para realizar la documentación. Como los horarios en el segundo cuatrimestre, varían, se decidió realizar 4 horas semanales y 4 horas en fin de semana. Lo cual hacen unas 200 horas si descontamos las épocas de exámenes y prácticas.

Además, desde octubre de 2018 se ha hecho un control de las horas realizadas con la herramienta Toggl, con la cual se ha podido controlar el tiempo empleado (aproximadamente), como podemos ver en la figura 11.

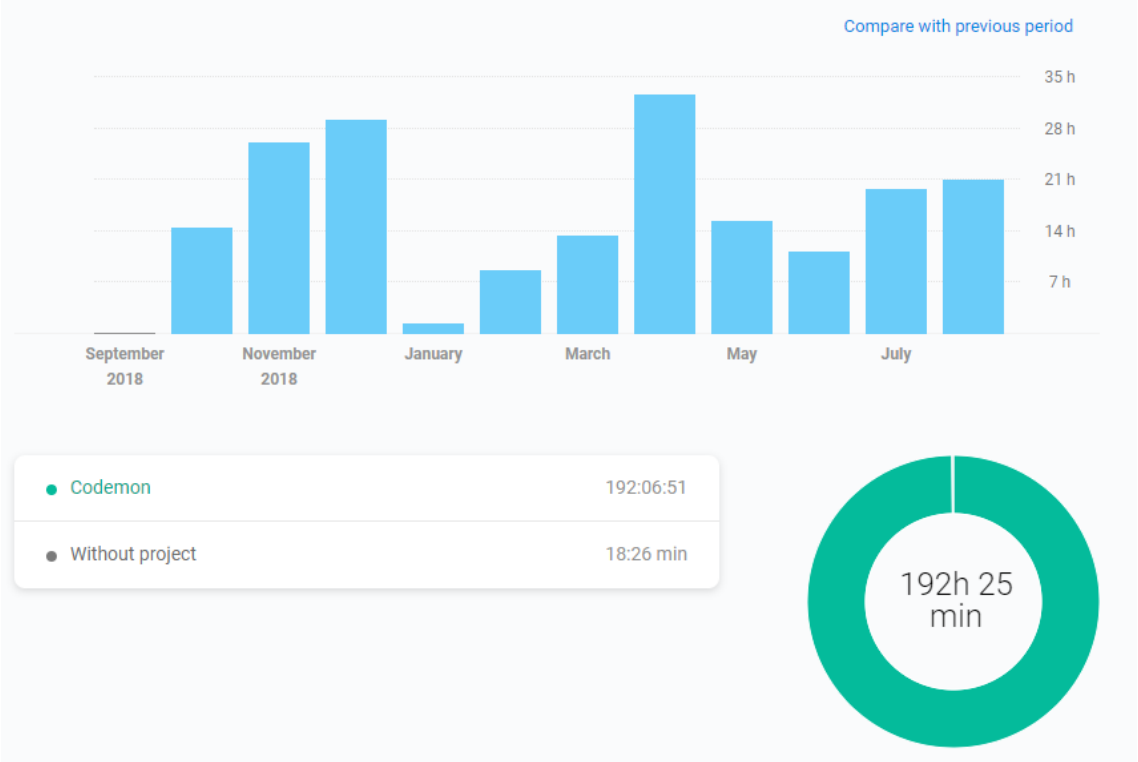

Figure 11: Dashboard de horas empleadas

Este número de horas, se corresponde entre el día 1/09/2018 hasta el 30/08/2019. Y como podemos observar, se han realizado casi 200h, que era lo planificado en un principio.

Metodología de desarrollo

La metodología utilizada para la implementación sigue las directrices de una metodología iterativa, para ello se han hecho las siguientes etapas:

Investigación previa

Esta es la etapa en la cual más tiempo se ha empleado. Se han investigado referencias para Codemon (algunas anteriormente comentadas), así como proyectos open source, videojuegos Pokémon, etc. Puesto que no había proyectos por los cuales pudiéramos basar, gracias a la bibliografía recomendada por los tutores, se decidió aprender a realizar una estructura y un *game-loop*, para realizar nuestro videojuego. El principal recurso a utilizar era el libro de *Game Engine Architecture*.

Además, hay que añadir la investigación previa para seleccionar la librería en la cual se iba a hacer el proyecto, el lenguaje, la estructura y todo lo relacionado con este mismo.

Por cada mecánica a introducir al proyecto, se ha hecho una investigación de cómo introducirlo, así como la mejor forma de hacerlo.

Prototipado

Una vez se investigaba una mecánica, y se tenía unos conceptos básicos de como tenía que ser, se pasaba a la parte de prototipado. En esta fase, si la mecánica era lo suficientemente compleja o no se podía integrar directamente, se realizaba un prototipo.

Para ello, se montaba una pequeña solución o proyecto muy básico con la funcionalidad o mecánica principal como punto principal de este. Aquí se decidía si pasaría al proyecto principal o no.

Tras esto, una vez estaba la mecánica probada, se comprobaban las dificultades para la integración, así como todas las dependencias que el prototipo podría tener y de qué forma se podría integrar en el proyecto principal.

Integración

Una vez terminadas las dos fases anteriores, se procedía a la integración. Esta es la parte más costosa, puesto que añadir una funcionalidad o mecánica nueva era un tanto complicado. En cada iteración se intentaba hacer el código más flexible para las nuevas mecánicas fueran más fáciles de integrar.

Diseño

En el presente apartado se mostrarán los mapas que se han diseñado y planteado y/o las soluciones con las cuales se ha ido trabajando desde el principio del proyecto. Además, se indicará cuales era interesante añadirlos al proyecto, o, por el contrario, rechazarlos.

El Mapa0, que podemos ver en la figura 12, es el mapa más básico que se ha podido diseñar para Codemon. Es un mapa muy sencillo dónde el jugador simplemente aprenderá como funciona Codemon, además de ser el mapa introductorio.

Figure 12: Primer boceto del Mapa0

Este mapa sería el mapa introductorio, siendo este el primer mapa del juego, pudiendo de esta forma presentar la manera en la que Codemon se comunica con el jugador/alumno.

Este mapa, consta de una única habitación con una única entidad, la cual se activará en el momento que reciba un `std::cout`, el cual hará que se active la zona de finalización del mapa, y podemos continuar.

Además, como podemos ver en la figura 13, es un mapa que se ha diseñado y planteado en varias ocasiones, puesto que debe ser el mapa inicial del juego.

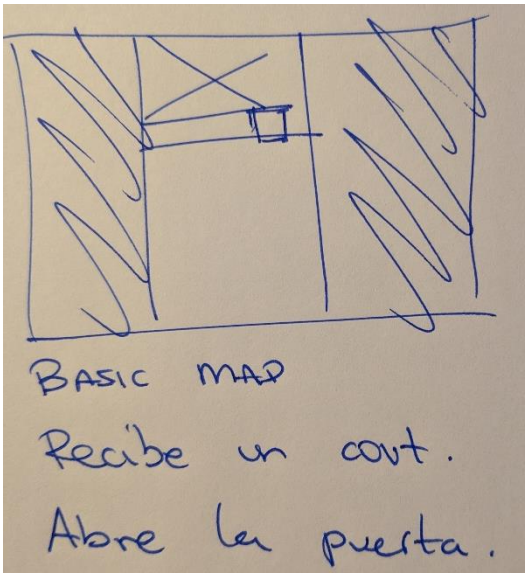

Figure 13: Boceto del Mapa0

Otros posibles mapa, dibujados en la figura 14, que se ha diseñado son mapas para trabajar los condicionales. Por ejemplo, un mapa que le comunique al jugado un número, y este devuelva si el número es par o impar, lo cual podría enseñarle al alumno nociones de números aleatorios, condicionales, y los operadores "/" y "%".

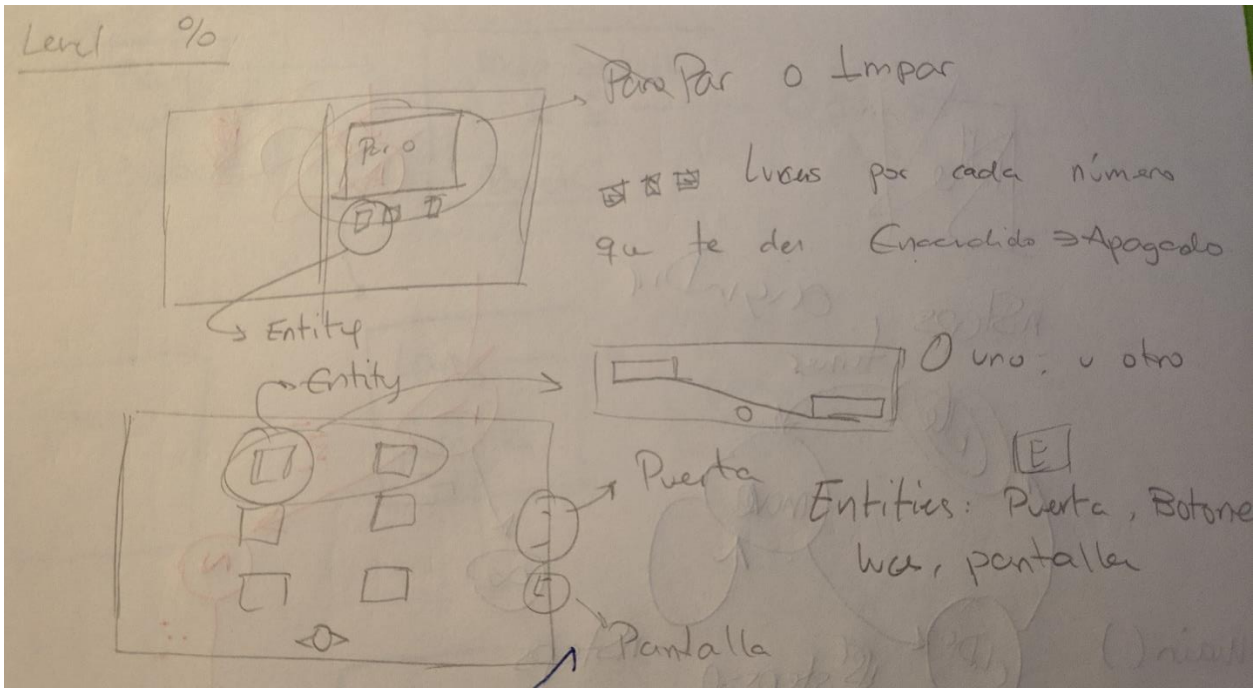

Figure 14: Mapa par o impar

Los mapas representados en la figura 15, siguen con la estela de los operadores anteriormente comentados, así como la idea de que el alumno siga practicando con los números pares e impares, así como realizar alguna prueba de números aleatorios.

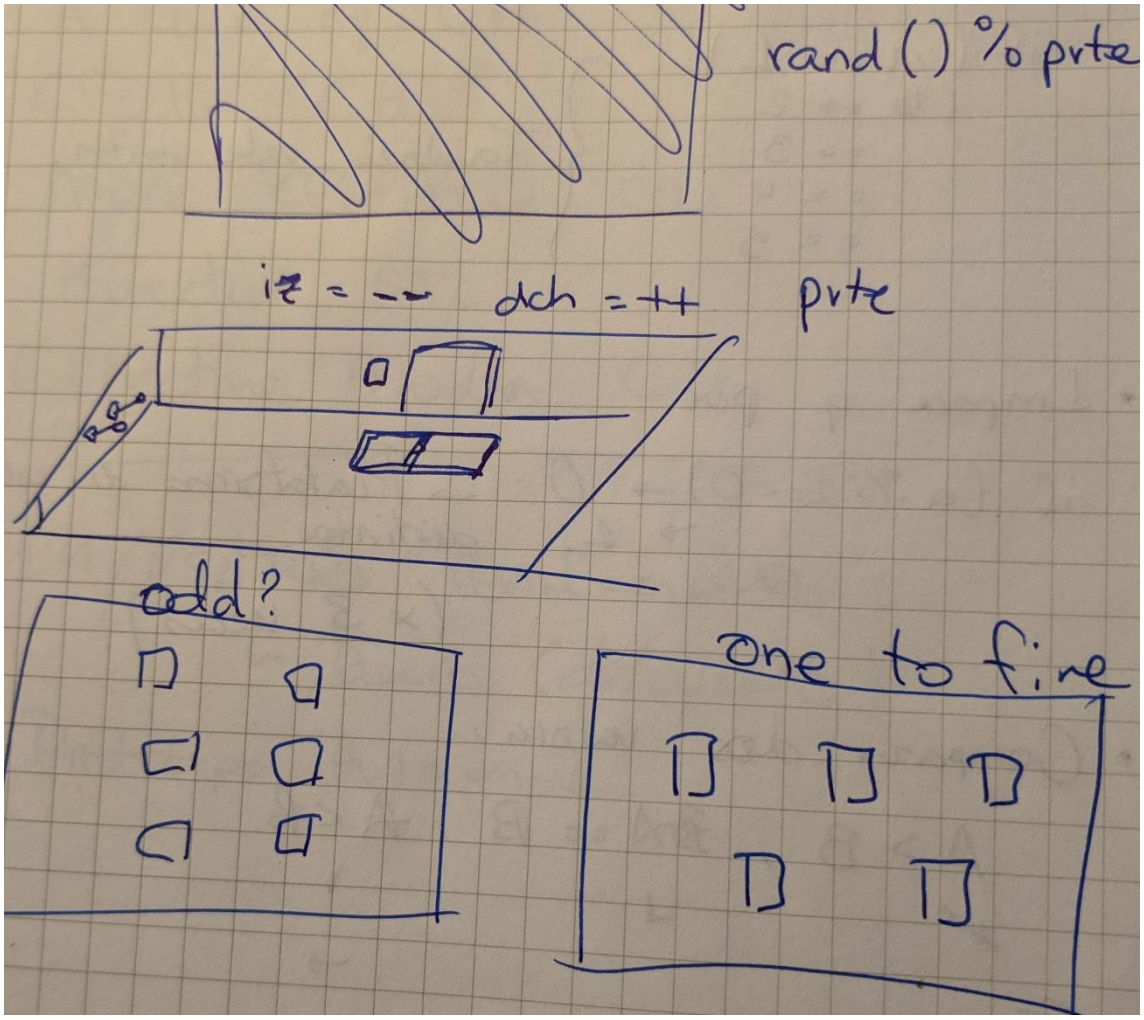

Figure 15: Mapas de random

El mapa representado en la figura 16, sería el clásico juego `Simon Says`, en el que podríamos hacer que el usuario vaya recibiendo una serie de movimientos y este los tuviera que replicar, el cual fue rápidamente descartado, puesto que sería una mecánica a realizar por el jugador, y no por las entidades, como se pretende que sea el vídeojuego, puesto que para que las entidades lo tuvieran que realizar, el código objeto del usario simplemente tendría que repetir por la salida estándar lo que le entrara por la entrada estándar.

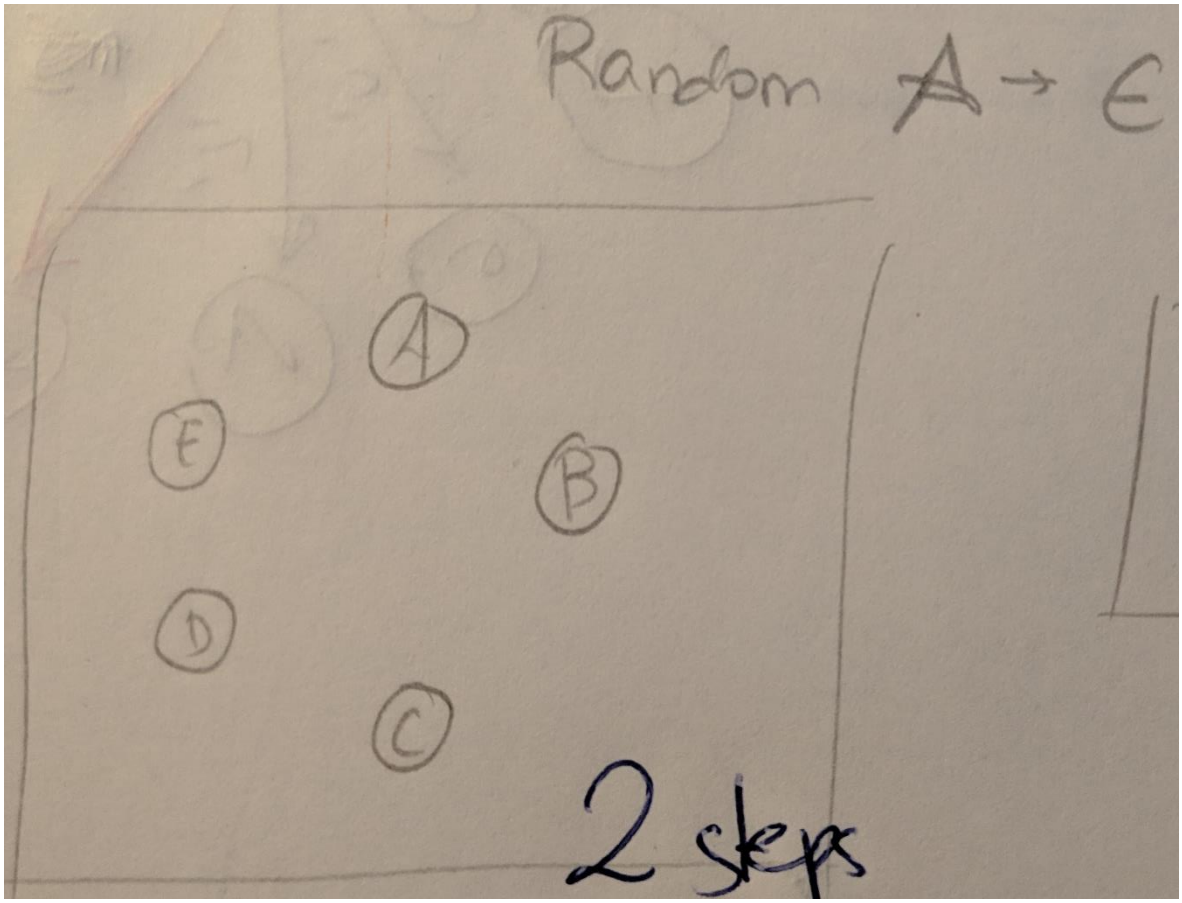

Figure 16: Mapa Simon Says

La figura 17 muestra una serie de ideas para otros mapas:

mapas sencillos para hacer comparaciones y códigos con ifs y `switches`,

- ideas para pares e impares,

ideas para hacer comparativas sobre dos valores que entren por la entrada estándar, y así indicar si uno es mayor que el otro …

En la figura 17, podemos ver una serie de ideas para más mapas:

- Mapas sencillos para hacer comparaciones, códigos con `if` y `switch`.
- Ideas para mapas utilizando par e impar como puzle.
- Ideas para hacer comparativas sobre dos valores que entren por la entrada estándar, y así indicar si uno es mayor que el otro, o menor que el otro, y según la respuesta que demos, las entidades actuen el consecuencia.

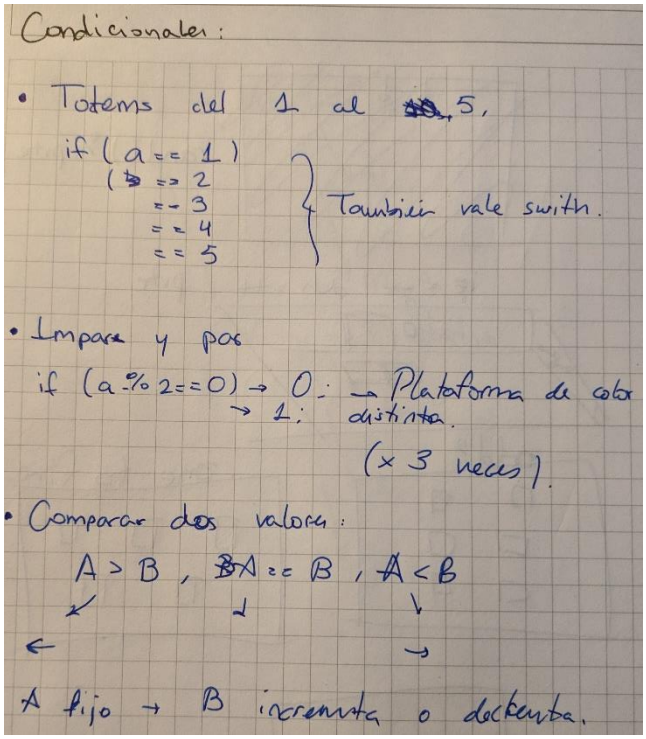

Figure 17: Información sobre mapas

Como podemos ver en la figura 18, se pretende realizar un mapa similar a los laberintos que hay dentro del juego Pokémon, siendo este, uno de los niveles más avanzados de los de `Mooving Entity` (que se comentará más adelante), pudiendo hacer así, un reto, y además utilizar una mecánica que ya está integrada dentro del juego, además, nos proporcionaría flexibilidad a la hora de realizar varios mapas, puesto que se pueden crear laberintos distintos con la misma mecánica.

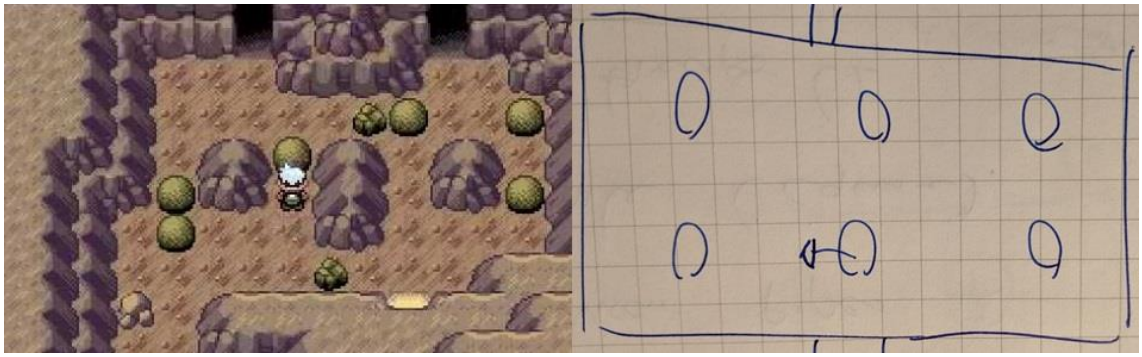

Figure 18: Mecánica de Pushing Entity.

Desarrollo

Es el presente apartado se explicará la experimentación que se ha ido realizando desde que se inició este proyecto, hasta la entrega de este mismo.

Una vez se seleccionó el tema para el trabajo de fin de grado, se empezó una lucha por aprender e intentar entender la forma en la que funcionaba un videojuego por dentro. Aprender cómo se estructura, como se crean mecánicas, en definitiva, aprender a hacer un videojuego. La creación de videojuegos no es algo que esté muy impartido en el Grado de Ingeniería Informática, puesto que solo se nos presentan unas pocas oportunidades de crear videojuegos, y no son lo suficiente como para realizar un videojuego de forma óptima a la primera.

Cuando se comenzó la investigación sobre videojuegos creados y programados en C++, la primera librería que se investigó fue SDL (Simple Directmedia Layer), en concreto su versión 2. Siendo esta una librería bastante utilizada, tiene un montón de proyectos en internet realizados con esta misma. Para aprender un poco sobre esta librería, se creó un proyecto pequeño, creando algún objeto por pantalla para tocar su parte gráfica. Esta fue rápidamente descartada puesto que la librería es de más bajo nivel y por lo tanto la curva de aprendizaje sería más lenta.

Mediante la realización de ese proyecto inicial, se conoció la librería SFML (Simple Fast Multimedia Layer) con la que finalmente se trabajó, puesto que esta librería cuenta con documentación[3] más completa, asequible, y la librería es más sencilla de utilizar.

Primero se decidió realizar un proyecto muy sencillo, no tenía por qué ser un juego, así podría conocer el funcionamiento de la librería. A partir de aquí, se plantearían unas bases para aprender a estructurar, y conocer la estructura que SFML tiene (de forma general).

Una vez se obtuvo un poco de soltura con la librería, se planteó realizar algún videojuego para conocer la estructuración de estos mismos y obtener conocimientos que después podrían ser extrapolados a Codemon, ya que todos los juegos pueden y suelen compartir ciertas técnicas como podrían ser las cargas de mapas y niveles, la carga de entidades, el guardado, el cargado, etc.

Este punto fue en el que se creó el primer prototipo del Mapa0 (anteriormente comentado), así que ya tendríamos la estructura inicial, y ya podríamos empezar a trabajar de forma iterativa.

El primer cambio sustancial que se realizó fue la lectura de mapas y configuraciones. Hasta el momento se estaba realizando la lectura de mapas desde un fichero de texto plano, el cual no seguía ninguna estructura ni ningún sentido, lo cual lo hacía difícilmente explicable, entendible y escalable. Como alternativa, se empezó a trabajar con ficheros JSON, en lugar de texto plano o XML, puesto que es un formato de fichero muy utilizado para configuraciones, además hay librerías de lectura en prácticamente todos los lenguajes, es fácil de usar, sencillo y muy ligero, muy simple de entender.

Gracias al formato JSON hemos conseguido flexibilidad a la hora de crear mapas. Además de añadirle contenido a estos mapas, puesto que en los propios ficheros de mapas se les puede añadir un array de entidades que comentaremos más adelante, en la figura 19.

Como podemos ver, hay una serie de campos en JSON, que se corresponden con una serie de datos necesarios para crear un mapa.

- *Name*: Se corresponde con el nombre del mapa, en concreto se enseñará en la fase de selección de mapa.
- *nTiles*: Se corresponde con el número de tiles que tiene el TileMap, que coincidirá con el TileMap que está etiquetado más abajo en el mismo fichero de configuración.
- *tileSizeOnMap*: Este es el tamaño del tile, se utilizará para determinar el tamaño del sprite a la hora de dibujar.
- *endMapCoord*: Es un *array* con las coordenadas de la finalización del mapa, esta zona se dibuja en un color amarillo para darle una diferencia, está puesta en X, Y, Ancho, Alto.
- *TileSet*: Esta es la ruta del tileset que vamos a utilizar para dibujar.

- *spawnCoord*: Este es el punto en el que el jugador aparecerá dentro del mapa.

- *tileMap*: Este es el aspecto del mapa a la hora de dibujarse.

```
"maps": [
    {
    "name"           : "open the door",
    "nTiles"         : [15, 20],
    "tileSizeOnMap" : 32,
    "endMapCoord"   : [256, 0, 128, 96],
    "tileSet"        : "./assets/tileMaps/tileSet.png",
    "spawnCoord"    : [315, 400],
    "tileMap": [
        [0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 0, 0, 0, 0, 2, 2, 0, 0, 0, 0, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 5, 1, 0, 0, 0, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 5, 1, 0, 0, 0, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 2, 1, 0, 0, 0, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0]
    ],
```

Figure 19: Mapa en JSON

El mapa está compuesto de un TileMap, que es una matriz de valores que marcan el Tile que se tiene que dibujar, en un principio estos se formaban por imágenes de 20x20 pixeles, pero rápidamente se cambió a múltiplos de 8, puesto que es lo más común a la hora de realizar tiles, usando los más comunes de todos, los de 32x32 pixeles. Y una serie de entidades, entre otras cosas.

Las entidades son una serie de objetos con los cuales el jugador, que es una entidad en sí, puede actuar. En los ejemplos anteriores, de Pokémon, cuando el jugador quería empujar una piedra, la piedra era una entidad, al igual que lo es el jugador. En la mayoría de casos el jugador puede interactuar directa o indirectamente con ellas.

Todas las Entidades vienen identificadas en código con un identificador de tipo `std::string`, llamado `m_type`.

- `M`: Es una entidad de tipo `MoovingEntity`, esta entidad puede ser empujada por el jugador mediante el uso de un código externo (Anexo 1).
- `T`: `TriggerEntity` es una entidad que solo sirve de activador, por lo general, está asociado a una entidad que lo activará, se utiliza, por ejemplo, como un botón en el suelo, siendo activado por una `MoovingEntity`.
- `LR`: `LeftRightMoovingEntity`, esta es una entidad enemiga que se podrá mover en estas dos direcciones un determinado número de pasos. Si se impacta con ella, el jugador, será eliminado.
- `S`: `ShootingEntity` es una entidad que dispara una serie de balas.
- `P`: Player, esta es la entidad que maneja el jugador, y por lo tanto solo habrá una de este tipo.
- `E, u O`: `Entity` o `GameObject`, son las entidades genéricas, aunque por lo general esta separación no tiene sentido, se decidió dejar así puesto que los `GameObjects` serían Entidades con las cuales el jugador no podría interactuar (Muros, paredes, suelos no pisables, etc). Y dejaríamos `Entity` para las entidades con las que el jugador sí puede interactuar, pero no tienen un subgrupo asignado.
- `A`: Activator, este es una `EntityTrigger` especial, la cual utilizaremos para el Mapa0, el cual se activará solo cuando esté el jugador tocándola, y, además, reciba algún `std::string` mediante un código externo.

Estas entidades vienen descritas en el JSON de configuración de mapas, justo detrás de la información del mapa, como podemos ver en la figura 20. Ahí podemos ver el tipo de entidad que es, la posición que tiene de *spawn*, las texturas, etc.

```
"entities":[{
    "type"     : "M",
    "position" : [320, 320],
    "texture"  : "./assets/props/button.png"
},{
    "type"      : "T",
    "activator" : 1,
    "position"  : [320, 192],
    "texture"   : "./assets/props/button.png",
    "enabled"   : "./assets/props/button.png"
}]
```

Figure 20: Entidades en configuración

Uno de los principales problemas que había en Codemon, era el cambio de mapa. Como hasta ahora solo había un único mapa, el cambio de mapa era un problema bastante importante, puesto que el código estaba acoplado. En este punto se creó un prototipo con una serie de mapas pequeños. Para poder así guardar el estado de las entidades, y que se podía cambiar pulsando las teclas del teclado numérico, como podemos ver en la figura 21, donde a la izquierda podemos ver el mapa con el fondo negro, y a la derecha el mapa con el fondo azul, donde hay una serie de entidades en color blanco, y el jugador en color rojo.

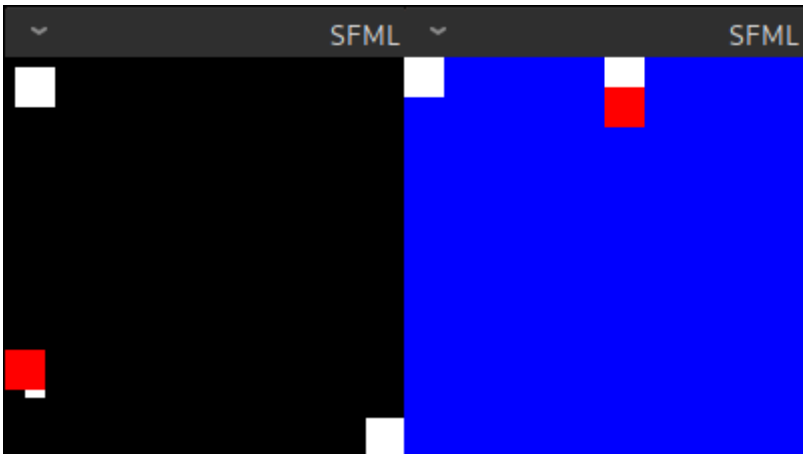

Figure 21: Prototipo de cambio de mapa.

Todos los prototipos se habían realizado sin colisiones entre entidades, puesto que era una mecánica desconocida para mí. Por suerte, en el foro de SFML[1] hay con código realizado por terceras personas que permite controlar las colisiones[2] de forma sencilla. Se integró al prototipo anterior, y así poder realizar alguna prueba con esta mecánica nueva.

Como podemos ver en la figura 18, coincidiendo con la parte izquierda de la figura 22 (el mapa), podemos ver que la entidad roja, está colisionando con una entidad, y eso se puede observar, tanto en consola, porque hay un booleano que indica si se están tocando, o no, y además, en la parte derecha, podemos observar un 13, el cual indica que es el mapa nº1, y estamos tocando la entidad nº3.

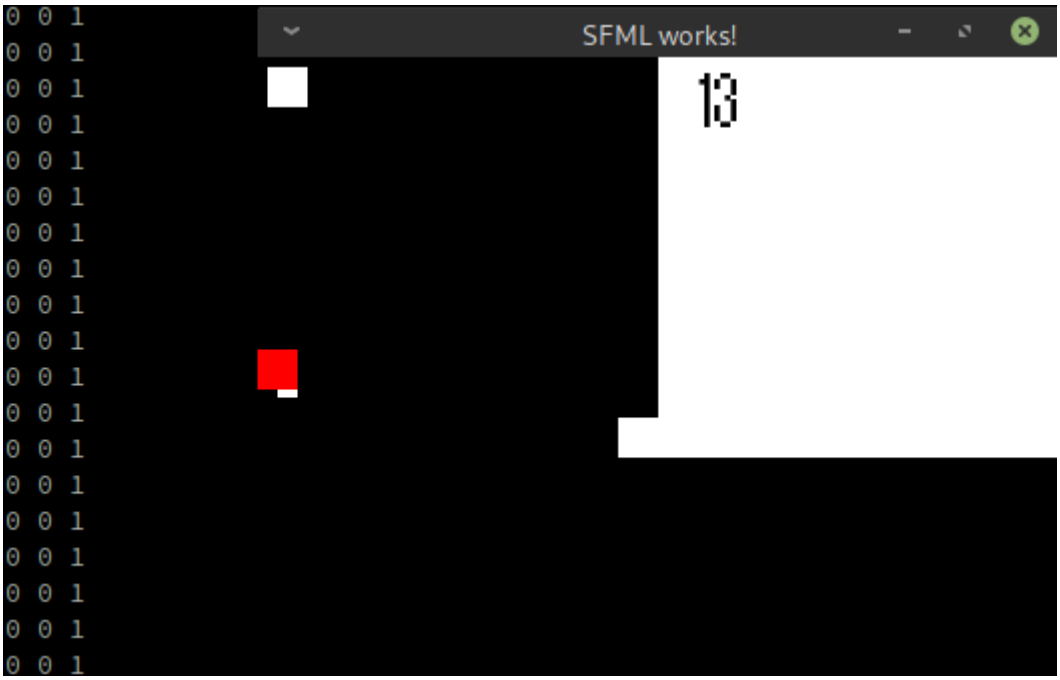

Figure 22: Prototipado de colisiones

La gestión de los recursos es un factor importante a lo que tener en cuenta. Para el prototipo principal se utilizó Valgrind, viendo que los consumos eran demasiado elevados, se decidió optimizar, sobre todo, la carga de mapas y entidades, puesto que hasta el momento se estaban cargando todos los mapas en todo momento. Se cambió la estructura de nuevo, y se pasó a leer un solo mapa (puesto que ahora son perpetuos, por el formato JSON). Además, se decidió dejar que la clase `Maps`, gestionara ella sola la carga del mapa, y jugar en él, para ello, se creó un único método de entrada público llamado `Maps::play(sf::RenderWindow&)`, así podremos realizar un control mayor por ejemplo del tiempo que se tarda en completar el mapa, para cuando queramos llevar un control de los tiempos que tardan los jugadores en realizar mapas.

Además, como en varios prototipos había realizado lógicas que no habían cambiado en ningún momento y se habían estado reutilizando en todo momento, se decidió implantarlo en el proyecto, así no habría cambios mayores en estas mecánicas, como son las lógicas de los Menús principales, de Opciones, o incluso el menú de selección de niveles.

Viendo que estaba abriendo demasiados frentes, y el tiempo cada vez era más escaso, se decidió centrarse un poco más en realizar mecánicas de juego, conseguir que este fuera un juego jugable, puesto que todos los prototipos que había realizado hasta ahora eran de una forma muy programática dónde se intentaba realizar mecánicas, pero en ningún momento se estaba pensando en que una de los principales objetivos de Codemon era que tenía que ser un juego, y que por lo tanto, debía ser divertido, o al menos intentarlo.

Puesto que había que empezar a realizar mapas, se determinó que para las primeras iteraciones centrarse en mecánicas sencillas y mapas sencillos de realizar.

Primero se dispuso hacer unos bocetos de mapas que habría que tener cuanto antes, así que se propusieron tres, con dos mecánicas distintas, como podemos observar en la figura 23.

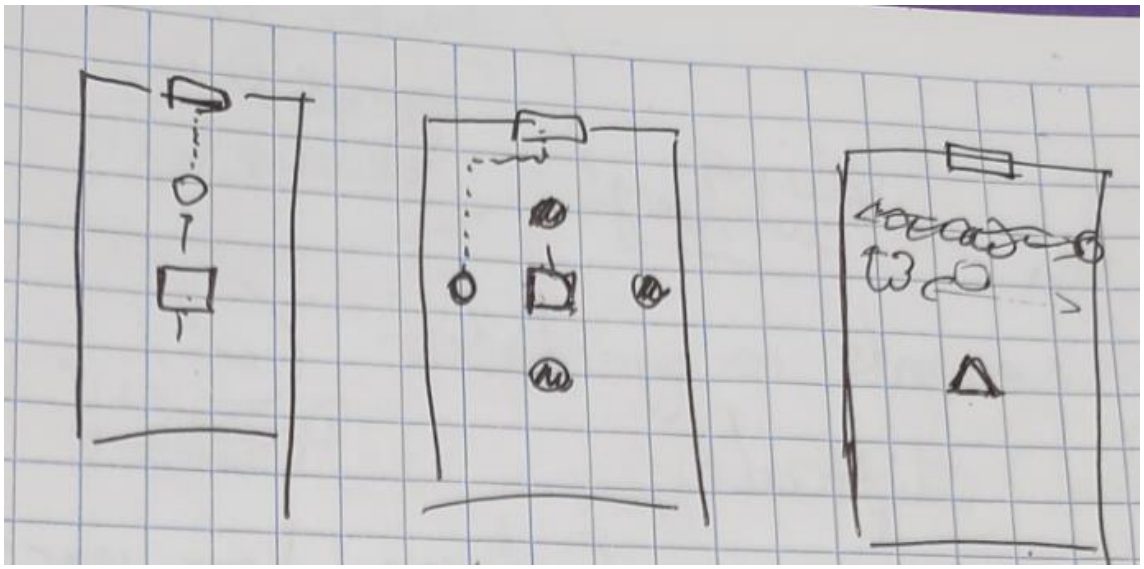

Figure 23: Mapas a realizar

Como se puede ver en la figura anterior, de izquierda a derecha, podremos ver tres bocetos de mapas distintos.

1. En el primer boceto, se decidió plantear una situación muy sencilla. Para ello, se dispondría de un par de entidades, una de tipo `TriggerEntity`, que actuaría como un botón en el suelo, y otra entidad de tipo `MovingEntity`, que podría ser una piedra que, al empujarla, en este caso en una única dirección, activara el botón, y se abriera una puerta.

2. El segundo boceto, es idénticamente igual al primero, solo que habría un cúmulo de botones, así podríamos mover la piedra en varias direcciones.

3. En el tercer boceto, se realizaría una mecánica distinta, dónde la entidad que está dibujada con un triángulo sería una entidad de tipo `ShootingEntity` que solo tendría una bala, para poder disparar a una entidad de tipo `LeftRightEntity` que se mueve un poco más arriba.

En este momento se realizó el primer mapa, correspondiente al primer boceto, presente en la figura 24.

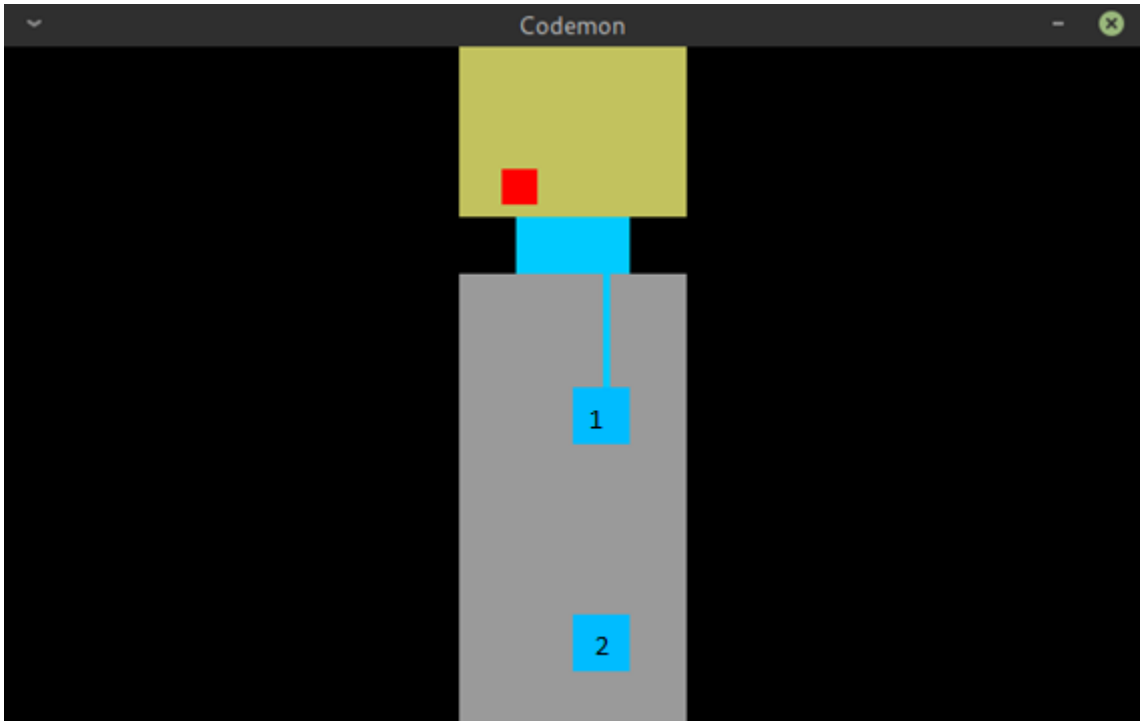

Figure 24: Prototipo del mapa Pushing Entity

Como podemos ver, hay una zona marcada con un rectángulo amarillo, que se corresponde con la zona que en el Json de assets (anteriormente comentado), está denominado con la etiqueta "`endMapCoord`", y cómo podemos ver, el jugador (entidad coloreada de rojo) está en la zona de finalización, pero el mapa no ha acabado, por lo tanto, habría que hacer alguna acción.

Como se ha explicado anteriormente, para ello, necesitaremos empujar la entidad que está marcada con un 2, al botón que está conectado a la zona azul. Como la entidad es de tipo `MoovingEntity`, tendremos que asignarle un código la cual permita moverse. Para ello presionaremos una tecla (I, anexo 2) del teclado, y por la entrada/salida estándar de la consola nos permitirá elegir que código queremos asignarle, como podemos ver en la figura 25.

```
...
1. ./assets/sourceCodes/bePush
2. ./assets/sourceCodes/move.cpp
3. ./assets/sourceCodes/say.cpp
4. ./assets/sourceCodes/say
5. ./assets/sourceCodes/bePushed.cpp
Get the code: (1-5)
1
```

Figure 25: Selección de códigos

Una vez asignemos el código, en este caso, seleccionaremos el ejecutable "`./assets/sourceCodes/bePush`", correspondiente al siguiente código, presente en la figura 26.

```cpp
#include <iostream>
using namespace std;

int main()
{
    char c;

    c = getchar();
    if (c == 'P')
    {
        c = getchar();
        if (c == ':')
        {
            c = getchar();
            if (c == 'U')
                cout << "M:U" << endl;
            if (c == 'D')
                cout << "M:D" << endl;
            if (c == 'L')
                cout << "M:L" << endl;
            if (c == 'R')
                cout << "M:R" << endl;
        }
    }

    // Importante, devolver 0 para cuando se use wait en el padre, que
    // pueda saber que el hijo ha terminado correctamente
    return 0;
}
```

Figure 26: Ejemplo de código de bePush, a realizar por el alumno.

El cual, será gestionado por la clase padre "Entity", llamado `std::string Entity::managePipe(const std::string& toChild) const;` el cual, recibirá un comando que se le mandará a el código seleccionado por el alumno mediante un par de pipes para hacer una comunicación bidireccional con este código (explicación en el Anexo 1), mediante la Entrada/Salida estándar. Así podremos pasarnos el nivel, como se puede ver en la figura 27.

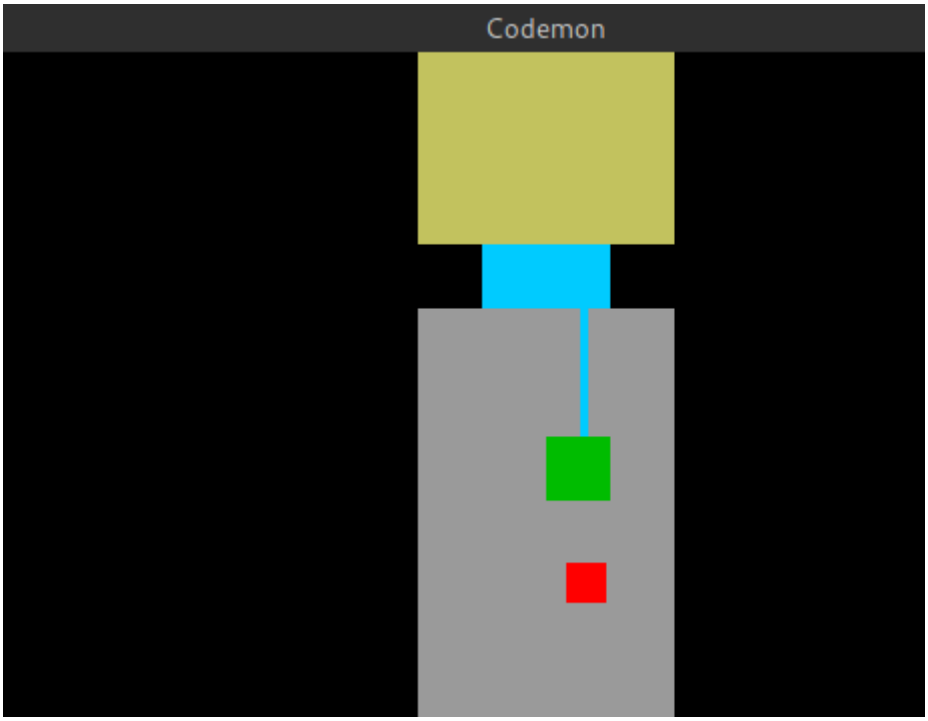

Figure 27: Prototipo con el mapa superado.

Una vez tenemos la entidad sobre el botón, este cambiará a un color verde, lo que nos sugiere que se ha activado algo (esto puede ser sustituido por una puerta que se abre, o se rompe). Cuando nos desplacemos de nuevo a la zona de finalización del mapa, el mapa acabará, y nos indicará el tiempo que hemos tardado en realizarlo.

También se realizó el segundo mapa, así se podría tener una versión más difícil y en todas las direcciones de la misma mecánica. Una vez teníamos esta mecánica, se pasó al tercer mapa, para así disponer de la mecánica de disparo y la muerte del jugador, como podemos ver en la figura 28.

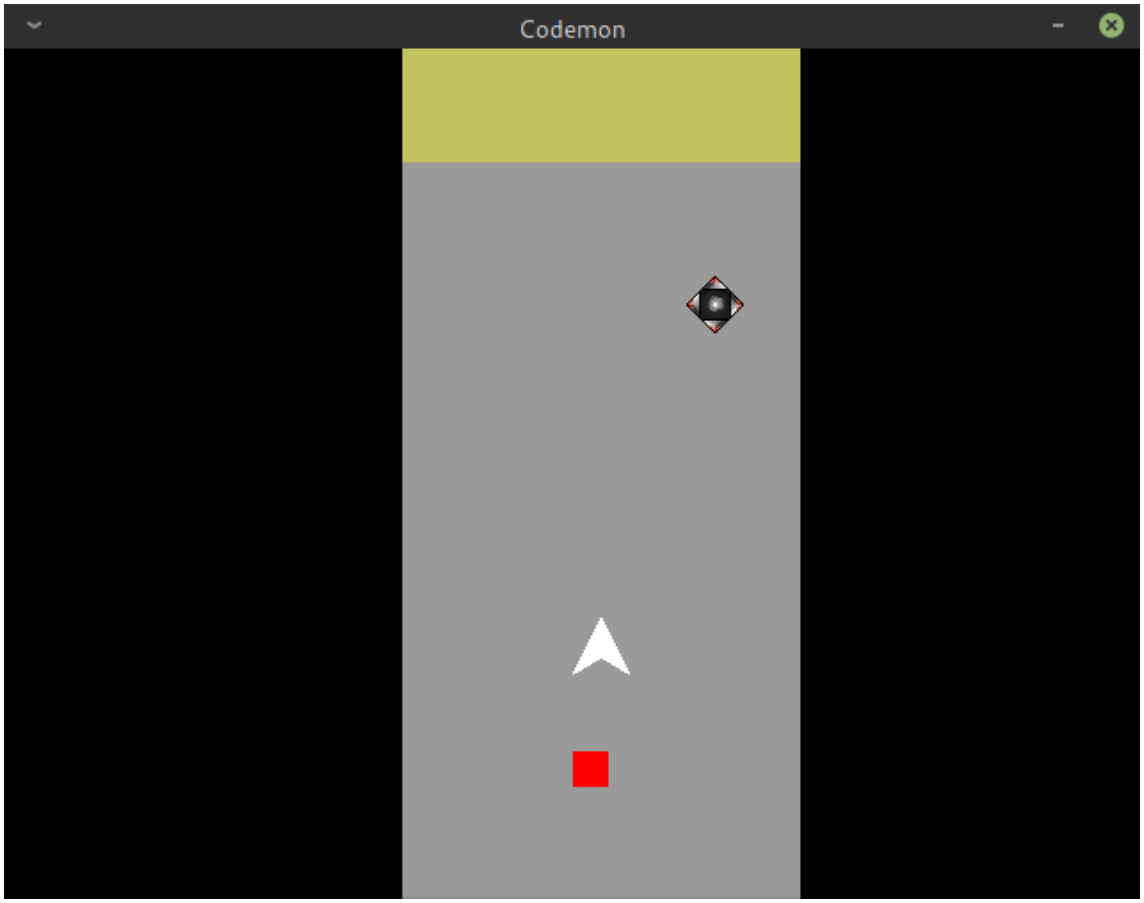

Figure 28: Shooting map.

Como podemos ver, hay un par de entidades representadas en el mapa anterior hay una entidad, que corresponde con una de tipo `LeftRightMoovingEntityque` está representada con un cuadradito con 4 pinchos, lo cual nos indicará que la entidad nos matará, y una entidad con forma de pico que nos indicará que disparará una bala que debe matar a la entidad enemiga para poder pasar (la cual si no matamos, no nos pasaremos el mapa), pero este mapa no se ha podido terminar, puesto que aún no se ha creado la mecánica de disparo, pero sí la de muerte.

Una vez nos quedamos con los tres mapas, era importante crear el Mapa0, puesto que este sería la introducción del juego, y ya había estado presente dentro de muchos prototipos, como se ha comentado anteriormente, fue el primer prototipo de todos, y así podemos verlo en la figura 29.

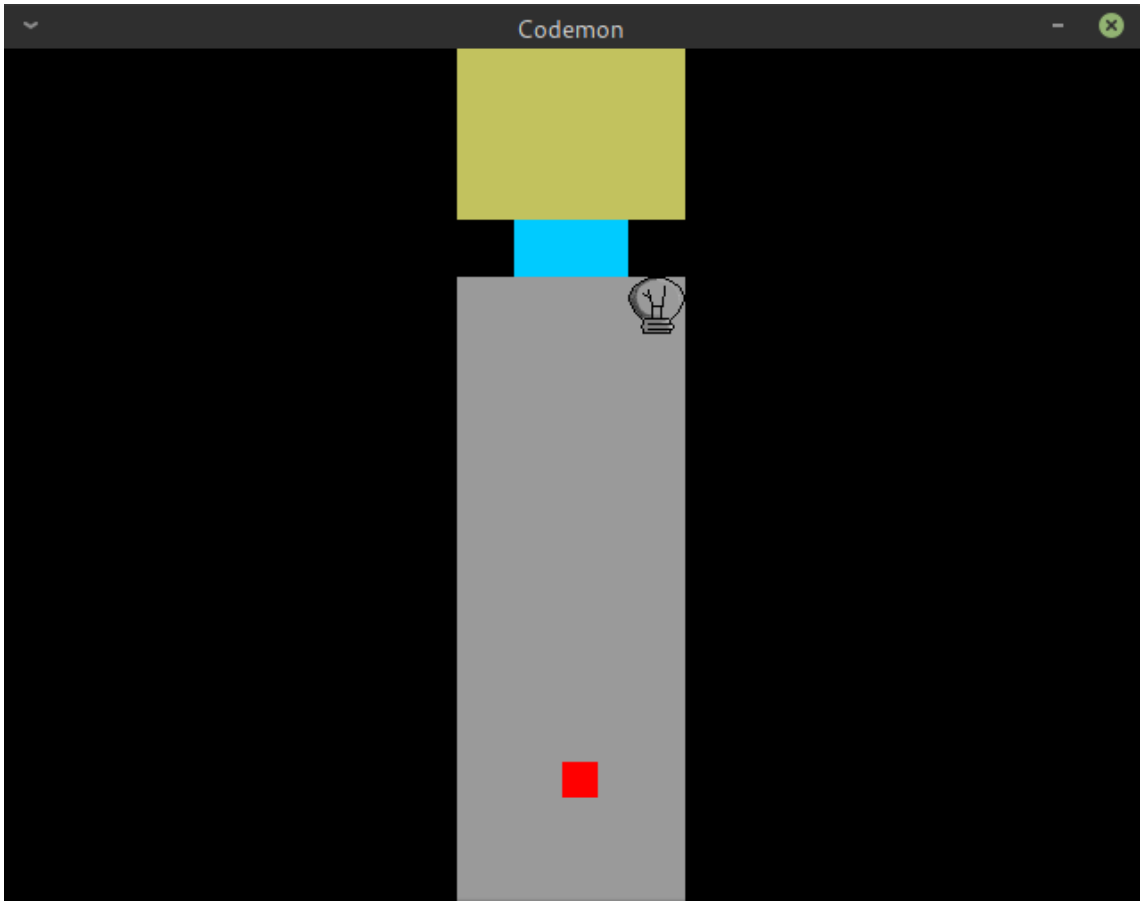

Figure 29: Mapa0.

Este mapa, consiste en una única entidad, de tipo `Activator`, que recibirá mediante el pipe de su entidad padre, y está representado con una bombilla, puesto que así podemos indicar que se ha activado, y que podremos finalizar el mapa.

Teniendo ya algunas mecánicas disponibles, y puesto que el tiempo restante no es demasiado como para poder poner más mecánicas, se decidió hacer algo de arte, para que no se viera todo como cajas, y así poder hacer de Codemon algo más parecido a un juego y no tanto a un prototipo, aunque le quede mucho por realizar.

En la figura 30 (de izquierda a derecha) se pueden observar los iconos empleados:

- **Bombilla apagada:** Correspondiente con la entidad `Activator`. Está apagada y así nos indicaría que se puede encender.
- **Bombilla encendida:** Correspondiente con la misma entidad, pero esta vez activada. Está encendida, así podemos ver que ya ha sido activada, y que no se podrá volver a activar.
- **Bala:** Correspondería con la entidad `Bullet`, pero no ha sido implementada.
- **Trigger apagado:** Este diseño corresponde con los `EntityTriggers`, cuando esté desactivado, así podremos activarlo, asemeja la apariencia a una *pressure plate* o a un botón en el suelo.

En la fila de abajo:

- **Roca:** Da la sensación de que se puede empujar, correspondiendo con el `MoovingEntity`.
- **Cañon:** Nos indicará algo que dispara, correspondiente a la entidad de tipo `ShootingEntity`, la cual dispara balas de *1…N* con el diseño anteriormente comentado.
- **Trigger activado:** Indicará que ya hemos activado el *trigger*, y, por lo tanto, no necesitaremos volverlo a activar.
- **Pinchos**: Nos da la sensación de peligrosidad, como si esta entidad nos pudiera matar, está asignado en este caso a la entidad `LeftRightMoovingEntity`.

Figure 30: Dibujos de entidades

Futuras ampliaciones

Las futuras ampliaciones de este proyecto son varias y se deja un proyecto muy abierto por si en algún momento se quisiera ampliar.

Siguiendo los objetivos originales

Estos son los objetivos que no se han cumplido en los objetivos originales.

- Crear una comunidad, a partir de los *Issues* de *Github*.
- Crear más mapas y mecánicas.
- Habilitar un modo multijugador, pudiendo crear distintos mapas, así como poder hacer la evaluación directamente en la parte del servidor.
- Crear una sala perpetua, donde cada jugador, deje, dentro del juego su propio granito de arena.

Objetivos para el producto

Estos son los objetivos que se cumplen, pero se podrían mejorar.

- Mejorar los mapas y mecánicas actuales, y extenderlos.
- Crear unos gráficos que realmente parezca que esto es un juego, y que se asemeja a Pokémon (principal inspiración del proyecto).
- Mejorar los consumos de este mismo, así como optimizar lo máximo posible el juego puesto que hay partes que aún consumen demasiado.
- Añadirle ayudas, explicaciones, etc.

Objetivos personales

Estos son los objetivos que personalmente me gustaría que tuviera el juego.

- Restructurar el juego, respetando lo máximo posible la estructura actual.
- Conseguir que sea un juego y no un prototipo.
- Hacerlo divertido.
- Añadir opciones, dificultades, poder guardar el estado de tu partida.
- Crear una clasificación de jugadores y el tiempo que tardan en pasarse cada mapa.

Conclusiones

Mediante la realización de este proyecto, se ha podido comprobar, que la realización de un videojuego es algo bastante complicado. La estructura ha sido el principal problema en la realización del proyecto, llevando así a realizar muchas menos ideas de las que tenía pensado. Esto nos indica que dentro de la carrera se debería ampliar la práctica de realizar proyectos desde cero, sin ningún tipo de código de inicio o similares.

Conseguir que un juego sea divertido, es algo que lleva mucho tiempo, y, por lo tanto, este proyecto se podría tildar de un prototipo más que de un videojuego. Los niveles que se han creado son lo suficientemente básicos como para que un alumno de primero de ingeniería lo utilice como método complementario a los actuales, por lo tanto, este juego habría que extenderlo.

Siguiendo una planificación previa, y viendo que más o menos se han cumplido los tiempos que se estimaron, reafirma que la planificación es un punto importantísimo a la hora de realizar un proyecto. Aunque el tiempo estuviera planeado, aprender que dejar objetivos atrás e ideas es algo que a veces es inevitable, y que simplemente no se pueden realizar.

Que la programación de forma eficaz en lenguajes de bajo nivel como C/C++, es algo bastante más complicado de lo que en un principio parece, que en cuanto dejas de tener en cuenta pequeñas cosas, el consumo de RAM se dispara rápidamente.

La importancia de tener a gente que te apoye en momentos de bajón dentro del proyecto, y la importancia de estar bien física y mentalmente para realizar proyectos de un tamaño más grande que una práctica, para poder continuar con este mismo y no dejarlo en el abandono.

Referencias

CONGER, D., LITTLE, R., *Creating Games in C++: A Step-by-Step Guide*. 2006. ISBN 978-0-7357-1434-2

JASON GREGORY, *Game Engine Architecture*. Tercera Edición. Florida (USA). 2009. ISBN 978-1-4398-6526-2

MEYERS, S., *Effective Modern C++.* Canada. 2015. ISBN 978-1-491-90399-5

MOREIRA, A., HANSSON, H. V., HALLER, J., *SFML Game Development*. Birmingham (UK). 2013. ISBN 978-1-84969-684-5

Web

1: Foro de SFML, https://en.sfml-dev.org/forums/index.php (Visitada en Enero 2019)

2: Código Collision.h, https://github.com/SFML/SFML/wiki/Source:-Simple-Collision-Detection-for-SFML-2 (Visitada en Diciembre 2018)

3: Documentación de SFML, https://www.sfml-dev.org/documentation/2.5.1/i (Visitada por última vez en Agosto 2019)

4: Fuente utilizada, https://www.dafont.com/es/luis-silva.d6380 (Creada por Luis Silva, *TheLouster115* en datont.com)

Anexo

Glosario

NPC *Non-player character*. Es un personaje que no está controlado por un ser humano.

Tile o tesela, es una imagen utilizada en la parte gráfica de los videojuegos, normalmente utilizada en los fondos en TileSets.

TileSet o teselas, cuando se agrupan una serie de Tiles, para realizar una única imagen, para tener, por ejemplo, cada mapa en un único TileSet.

TileMap es la posición de los Tiles del TileSet en código, dónde en el código fuente pone un 0, se representa con el Tile (0, 0) del TileSet.

Spawn es la posición en la que una entidad aparece al principio de un mapa.

Anexo 1, Pipes

El código que se ha realizado para comunicarse con un código externo se realiza mediante los pipes de Linux. Como podemos ver en la línea 14 de la figura 31, creamos dos pipes, una que comunicará el proceso padre con el hijo, y otra que comunicará el proceso hijo con el padre.

```
8       // STDIN_FILENO    0       Standard input.
9       // STDOUT_FILENO   1       Standard output.
10      // STDERR_FILENO   2       Standard error output.
11      // Creating FileDescripors.
12      // p2c : Parent -> Child information flow.
13      // C2p : Child -> Parent information flow.
14      int fd_p2c[2], fd_c2p[2];
15
16      // Create the pipes and check if they could be created.
17      if(pipe(fd_p2c) || pipe(fd_c2p))
18          std::cerr << "Error: Couldn't create pipes.\n";
19
20      // Create the child process.
21      int pid = fork();
```

Figure 31: Creación de Pipes.

Primero, crearemos el proceso hijo, y comprobaremos de que hemos creado el proceso correctamente. Tras ello, cada proceso padre o hijo realizará una serie de acciones.

Como podemos observar en la figura 32, el proceso hijo, primero guardará en dos variables la salida y entrada estándar de C++, puesto querremos recuperarla y restaurarla más adelante (línea 30).

Cerraremos la salida estándar del pipe c2p, puesto que el hijo no escribirá en este pipe, y cerraremos la salida estándar de p2c, puesto que no escucharemos de ahí.

En las líneas 37 y 38 usaremos la función `dup2` de Linux, la cual duplicará los `filedescriptors`, en nuestro caso, duplicaremos los extremos de los pipes que han quedado abiertos a la salida y entrada estándar de C++. Una vez duplicados, cerraremos la salida estándar y la entrada estándar. En la línea 45, crearemos los argumentos (en este caso, no los vamos a utilizar) y llamaremos a la función `execve`, la cual ejecutará un código seleccionado con el `std::string m_code`, que será el que ha seleccionado el usuario para ejecutar. Si el `main` del código no tiene un `return` al final, el proceso

hijo no tendrá forma de saber cuándo ha terminado de leer en el pipe, por lo tanto, no podrá terminar nunca. Tras ello, y a modo de prevención (aunque si todo ha funcionado correctamente, no debería llegar) restauraremos la salida y entrada estándar, y cerraremos las variables guardadas, y mostraremos un mensaje de error.

```
27      else if (pid == 0) // Child process.
28      {
29          // Backup std in/out.
30          int stdin_bak = dup(STDIN_FILENO), stdout_bak = dup(STDOUT_FILENO);
31
32          // Close the unused pipe channels.
33          close(fd_c2p[STDIN_FILENO]);  // Child won't write on c2p.
34          close(fd_p2c[STDOUT_FILENO]); // Child won't read from p2c.
35
36          // Duplicate the used channels.
37          dup2(fd_c2p[STDOUT_FILENO], STDOUT_FILENO); // fd_c2p_out -> std::cin.
38          dup2(fd_p2c[STDIN_FILENO], STDIN_FILENO);   // fd_p2c_in  -> std::cout.
39
40          // FD's are suplicated on std in/out, so we can close the original ones.
41          close(fd_c2p[STDOUT_FILENO]);
42          close(fd_p2c[STDIN_FILENO]);
43
44          // Exec the code with those arguments, then.
45          char *const argv[] = { const_cast<char*>(m_code.c_str()), nullptr };
46          execve(("./" + m_code).c_str(), argv, nullptr);
47
48          // This code shouldn't execute, but restore just in case that execve fails.
49          dup2 (stdin_bak, STDIN_FILENO);
50          dup2 (stdout_bak, STDOUT_FILENO);
51          close(stdin_bak);
52          close(stdout_bak);
53
54          std::cerr << "Error: " << m_code << "couln't be executed.\n";
```

Figure 32: Proceso hijo.

El proceso padre, realizará las siguientes acciones, como podemos ver en la figura 33, guardará la salida y entrada estándar, igual que el proceso hijo, también cerrará los canales sin utilizar de los pipes, en este caso, los contrarios al hijo, y lo único que hará es, mandarle el mensaje al proceso hijo (que está ejecutando el código externo) lo que hay en la variable `std::string toChild`, importante que esta variable acabe siempre con un salto de línea, y hará un `std::getline`, para leer el contenido que el hijo nos ha mandado por el pipe (recordemos que hemos intercambiado la salida y entrada estándar por los pipes, por eso este proceso funciona). Una vez se ha leído, este si DEBE restaurar la salida y entrada estándar, para poder seguir funcionando de la forma correcta.

```
56      else // Parent process.
57      {
58          // Backup std in/out.
59          int stdin_bak = dup(STDIN_FILENO), stdout_bak = dup(STDOUT_FILENO);
60
61          // Close the unused pipe channels.
62          close(fd_c2p[STDOUT_FILENO]); // Parent won't write on c2p.
63          close(fd_p2c[STDIN_FILENO]);  // Parent won't read on p2c.
64
65          // Duplicate the used channels.
66          dup2(fd_c2p[STDIN_FILENO], STDIN_FILENO);   // fd_c2p_in  -> std::cin.
67          dup2(fd_p2c[STDOUT_FILENO], STDOUT_FILENO); // fd_p2c_out -> std::cout.
68
69          // FD's are suplicated on std in/out, so we can close the original ones.
70          close(fd_c2p[STDIN_FILENO]);
71          close(fd_p2c[STDOUT_FILENO]);
72
73          // for(auto& str : toChild).
74          // {.
75              // Now, write to the child process.
76              // std::cout << str << "\n";.
77              std::cout << toChild << "\n";
78
79              // Get what the child process said.
80              std::getline(std::cin, recivedCommand);
81          // }.
82
83          // Restore std in/out.
84          dup2 (stdin_bak, STDIN_FILENO);
85          dup2 (stdout_bak, STDOUT_FILENO);
86          close(stdin_bak);
87          close(stdout_bak);
88      }
```

Figure 33: Proceso padre.

Tras esto, se devolverá el código a quien lo haya solicitado (por lo general, una entidad) y seguirá el programa ejecutándose de forma normal.

Anexo 2, Controles

En este apartado se comentarán los controles para jugar a Codemon.

- **W**: Moverá hacia arriba el selector en los menús. Moverá al jugador hacia arriba, en la fase de juego.
- **A**: Moverá al jugador hacia la izquierda.
- **S**: Moverá hacia abajo el selector en los menús. Moverá al jugador hacia la abajo, en la fase de juego.
- **D**: Moverá al jugador hacia la derecha.
- **I**: Abrirá el inventario, y nos preguntará por consola que número de código queremos realizar.
- **E**: Asignará el código asignado a la entidad correspondiente (más cercana).
- **ESC**: Retrocederemos a la pantalla anterior (menús). Saldrá del mapa cuando estemos jugando.

Anexo 3, Pantallas del juego

Esta figura 34 se corresponde con el Menú principal, en el momento que pulsamos *About*, puesto que nos mostrará información sobre el creador en la consola.

Como podemos ver, hay una fuente[4] con estilo Pokémon.

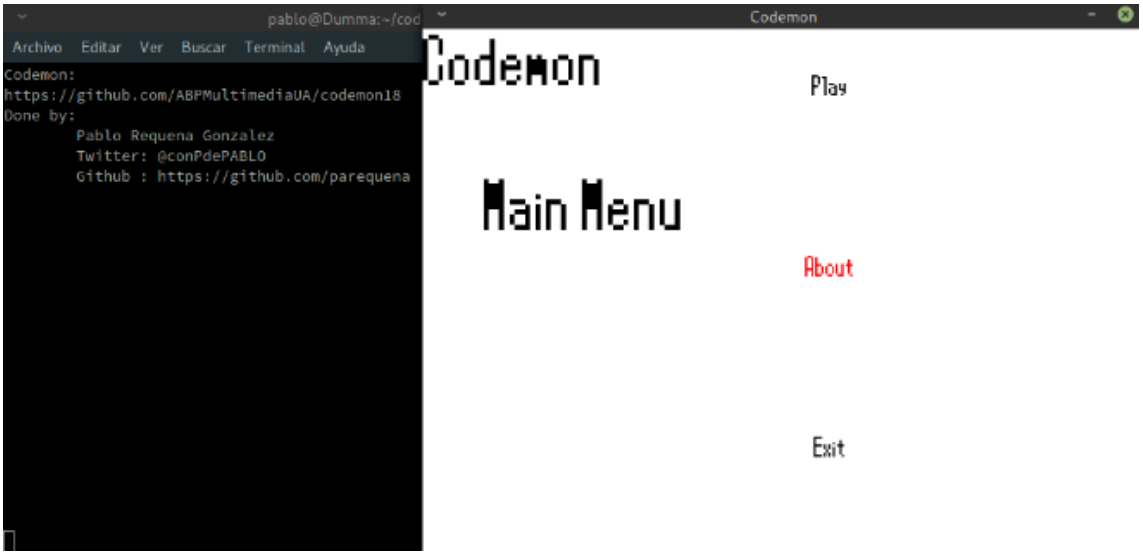

Figure 34: Main Menu

En cuanto pulsamos *Play*, veremos la pantalla de selección de mapas, presente en la figura 35.

Figure 35: Map selection

Seleccionamos el mapa, y veremos los siguientes mapas, presentes en las figuras de la 36, a la 39.

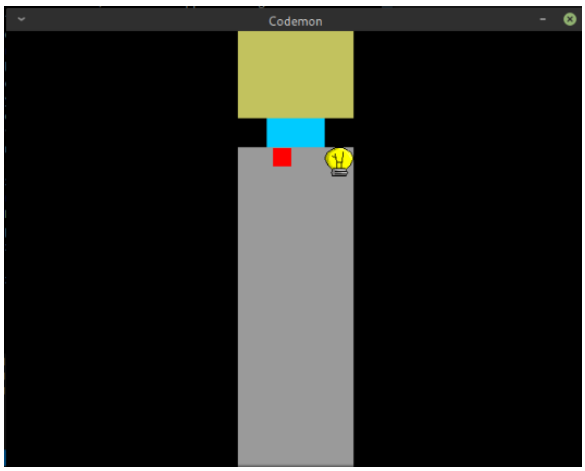

Figure 36: Map0.

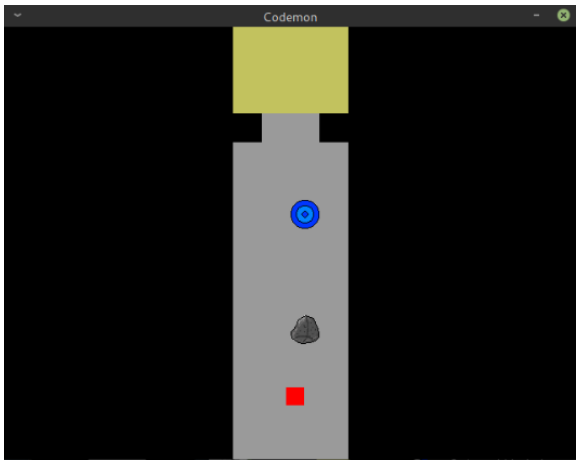

Figure 37: open the door.

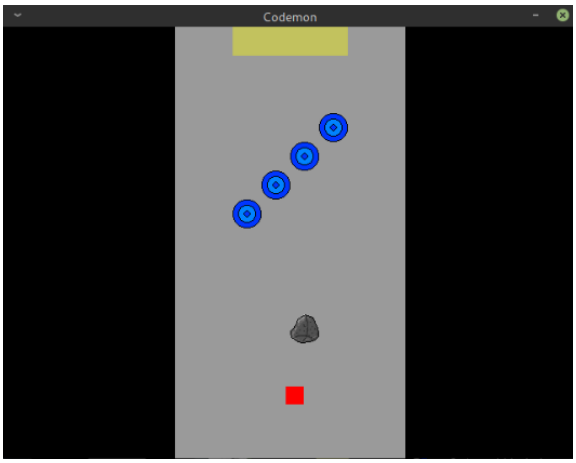

Figure 38: Multiple door

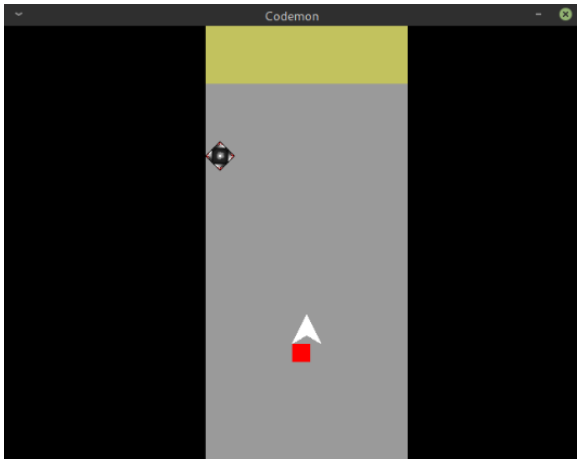

Figure 39: One shot.

Anexo 4, Makefile y estructura

En este apartado se mostrará tanto la estructura de carpetas, como el archivo Makefile que se ha utilizado para la compilación del proyecto, por si alguien quisiera replicarlo en algún momento.

Como podemos ver en la figura 40, el proyecto se corresponde con las siguientes capertas:

- **assets**: Aquí están presentes todos los archivos de configuración, los recursos, tanto mapas, como imágenes utilizadas para los `sprites`, texturas y `tilemaps`.

- **inc:** Aquí están presente todos los `headers` de C++, donde están todos los archivos "`.hpp`" de nuestro proyecto.
- **obj:** Esta carpeta contiene todos los objetos generados por el compilador de g++ utilizado en el `Makefile`.
- **prototipos:** En esta carpeta tenemos presentes algunos de los prototipos generados para la realización de este proyecto.
- **src y Main.cpp:** Aquí están todos los archivos de código de C++, los "`.cpp`".
- ***.supp**: Son archivos utilizados para la supresión de errores y warnings de Valgrind a la hora de utilizarlo.
- **pValgSupp.sh**: Un script que los mismos creadores de Valgrind tienen en su foro para generar algunos de los comandos de supresión de errores para Valgrind.
- **LICENSE y README**: Archivos que acompañan el proyecto en Github.
- **exe**: Es el ejecutable del juego, generado por el Makefile.
- **Makefile**: Es el archivo que compila el proyecto.

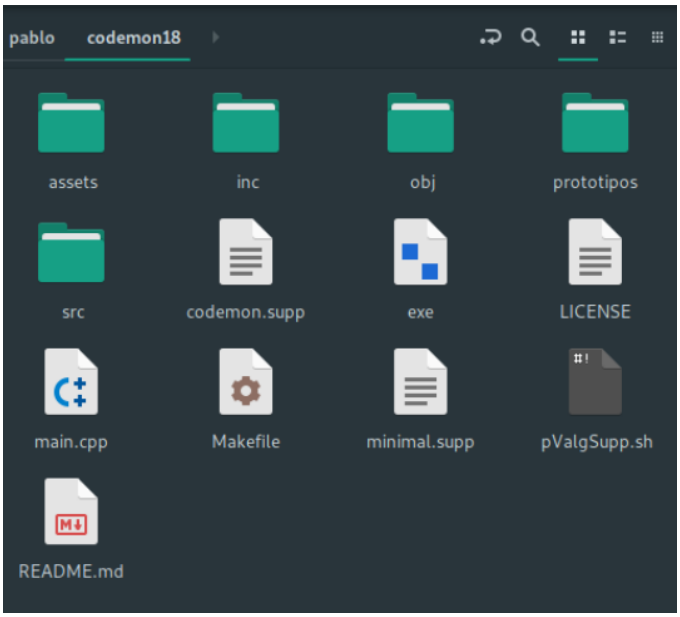

Figure 40: Estructura de carpetas

En el archivo de `Makefile`, como podemos ver en la figura 41, tenemos una serie de variables, en las cuales podemos ver, la estructura de carpetas anteriormente comentadas, en las variables `*_DIR`.

También, podemos ver unas `wildcards` para listar todos archivos dentro de las carpetas. Después podemos ver las reglas tanto como para generar los archivos necesarios para la compilación.

En la línea 21, generaremos el ejecutable, y la carpeta de obj. En la línea 23, crearemos la carpeta de los objetos. En la línea 26, crearemos el ejecutable, con todos los objetos ya creados, el `main`, y todo enlazado. En la línea 29, se creará cada uno de los objetos, con su `header`, y su archivo de código.

```
9  CC       := g++ -std=c++1z -O3
10 DEBUG    := # -g
11 OPTIONS  := -Wall -Wextra -pedantic-errors
12 LIBS     := -lsfml-graphics -lsfml-window -lsfml-system -ljsoncpp -lstdc++fs
13 SRC_DIR  := src
14 INC_DIR  := inc
15 OBJ_DIR  := obj
16 EXE_NAME := exe
17 SOURCES  := $(wildcard $(SRC_DIR)/*.cpp)
18 OBJECTS  := $(subst .cpp,.o,$(SOURCES))
19 OBJECTS  := $(subst $(SRC_DIR),$(OBJ_DIR),$(OBJECTS))
20
21 all: $(OBJ_DIR)/ $(EXE_NAME)
22
23 $(OBJ_DIR)/:
24     mkdir -p $(OBJ_DIR)
25
26 $(EXE_NAME): $(OBJECTS) main.cpp
27     $(CC) $(DEBUG) $(OPTIONS) -o $@ $^ $(LIBS) -I$(INC_DIR)
28
29 $(OBJ_DIR)/%.o : $(SRC_DIR)/%.cpp
30     $(CC) -o $@ -c $^ -I$(INC_DIR) $(DEBUG)
31
32 cls : clean
33 clc : clean
34 clear: clean
35 clean:
36     rm -rf $(EXE_NAME)
37     rm -rf $(OBJ_DIR)
```

Figure 41: Archivo Makefile