

Tradams

Videojuego de 8bits



Grado en Ingeniería Multimedia

Trabajo Fin de Grado

Autor:

Gaspar Rodríguez Valero

Tutor/es:

Francisco José Gallego Durán

Enero 2017



Universitat d'Alacant
Universidad de Alicante

Gracias a Francisco José Gallego Durán por guiarme y mostrarme el buen camino para lograr ser una gran profesional, por mostrarme lo poco que conozco y lo mucho que hay por aprender. Me ha ayudado a decantarme por la programación y le considero un gran maestro.

Agradecer también a mi grupo ABP de cuarto, pues con ellos he vivido grandes retos y los hemos superado juntos. Hemos conocido de primera mano nuestras flaquezas, pero juntos pudimos salir adelante y mostrar lo que valíamos.

Gracias a mi familia que ha confiado siempre en mí y en mi trabajo, dándome siempre su mayor apoyo y dándome todo lo que necesitará para obtener este grado. La familia es un pilar muy importante en la vida de muchas personas, y el mío ha sido el más robusto de los soportes.

Índice

<u>1. INTRODUCCIÓN</u>	9
1.1. RESUMEN	9
1.2. PROBLEMÁTICA DEL TRABAJO	10
1.3. MOTIVACIÓN	10
1.4. PROPÓSITO O FINALIDAD DEL TRABAJO	12
1.5. ESTRUCTURA DEL DOCUMENTO	12
<u>2. MARCO TEÓRICO</u>	14
2.1. TERMINOLOGIA	14
2.2. ESTUDIOS PREVIOS PLANTEADOS	16
2.2.1. DESCRIPCIÓN MAQUINA SELECCIONADA	16
2.2.1.1. ESPECIFICACIONES TECNICAS	20
2.2.1.2. FUNCIONALIDADES	20
2.2.2. PROYECTOS PREVIOS DE ESTUDIO	21
• JUEGOS PROCEDIMENTALES	21
• EFECTOS DE SPLIT SCREEN	28
• SCROLL	28
• SPLIT RASTER	30
• JUEGOS DE MAZMORRAS	31
<u>3. METODOLOGÍA DEL TRABAJO</u>	34
<u>4. CUERPO DEL TRABAJO</u>	36
4.1. INTRODUCCIÓN	36
4.2. PROYECTOS ANTECEDENTES	36
4.2.1. ARKANOID	36
4.3. DISEÑO Y ESPECIFICACIÓN	42
4.3.1. ARGUMENTO	42
4.3.2. CARACTERÍSTICAS DEL PROYECTO	42
4.3.3. PARTE PROCEDIMENTAL - PLANTEAMIENTO TEORICO	42
4.3.4. GÉNERO	44
4.3.5. AUDIENCIA OBJETIVO	44
4.3.6. ASPECTO GRÁFICO	44

4.3.7. AUDIO	46
4.3.8. PERSONAJES	46
4.3.9. ENEMIGOS	47
4.3.10. ESTRUCTURA	47
4.3.11. JUGABILIDAD Y MECÁNICAS	47
4.3.12. OBJETOS	48
4.3.13. OBJETIVOS DEL JUEGO	48
4.3.14. MENUS	48
4.4. DESARROLLO E IMPLEMENTACIÓN	48
4.4.1. ESTRUCTURA GENERAL	48
4.4.2. PATRONES DE DISEÑO	49
4.4.3. GENERACIÓN DE NIVELES	49
4.4.4. HABITACIONES	59
4.4.5. SISTEMA DE OBJETOS	63
4.4.6. ENEMIGOS IMPLEMENTADOS	71
4.4.7. INTELIGENCIA ARTIFICIAL	75
4.4.8. JUGADOR	76
4.4.9. SISTEMA DE ATAQUE	79
4.4.10. GESTION DE LA FUENTE DE TEXTO	80
4.4.11. MENUS	82
4.4.12. GRAFICOS	83
4.4.13. AUDIO	83
4.4.14. CONTROLES DEL JUEGO	84
4.4.15. REQUISITOS NO FUNCIONALES	84
4.4.16. REQUISITOS FUNCIONALES	86
4.5. PUBLICACIÓN DEL TRABAJO	90
4.6. HARDWARE Y SOFTWARE	90
4.6.1. HARDWARE UTILIZADO	90
4.6.2. SOFTWARE UTILIZADO	91
5. CONCLUSIONES	91
5.1. OBJETIVOS CONSEGUIDOS	91
5.2. OBJETIVOS NO CONSEGUIDOS	92
5.3. RESULTADOS OBTENIDOS	92

6. BIBLIOGRAFÍAS Y REFERENCIAS **93**

7. ANEXOS **94**

ASZ80 **94**

Índice de Figuras

FIGURA 1: AMSTRAD CPC 464.....	16
FIGURA 2: LOCOMOTIVE BASIC 1.1	16
FIGURA 3: IMAGEN MODO 2.....	17
FIGURA 4: IMAGEN MODO 1.....	17
FIGURA 5: IMAGEN MODO 0.....	17
FIGURA 6: PALETA DE 27 COLORES	19
FIGURA 7 : ROGUE (1980)	21
FIGURA 8- TETRIS (1984).....	22
FIGURA 9 - DIABLO (1996)	22
FIGURA 10 - KKRIEGER (2004)	23
FIGURA 11 - DWARF FORTRESS (2006)	24
FIGURA 12 - LEFT 4 DEAD (2008).....	24
FIGURA 13 - MINECRAFT (2009).....	25
FIGURA 14 - SPELUNKY (2009)	26
FIGURA 15 - THE BINDING OF ISAAC (2011).....	27
FIGURA 16 - NO MAN'S SKY	27
FIGURA 17. LOGO DE SUPER MARIO BROS (1987)	29
FIGURA 18 - EJEMPLO DE SCROLL HORIZONTAL	29
FIGURA 19 - ZONA AMPLIADA SCROLL HORIZONTAL	29
FIGURA 20 - MAPA DE POKEMON	30
FIGURA 21 - GAUNTLET II.....	31
FIGURA 22 - PEDIT5 (1975)	31
FIGURA 23 - MIGHT AND MAGIC BOOK ONE: THE SECRET OF THE INNER SANCTUM.....	32
FIGURA 24 - THE LEGEND OF ZELDA: OCARINA OF TIME (1998)	33
FIGURA 25 - PATH EXILE (2013).....	33
FIGURA 26 - GRAFICA DE COSTES.....	35
FIGURA 27 - ESTRUCTURA ENTIDAD V1.....	36
FIGURA 28 - CÓDIGO ENSAMBLADOR COLISIÓN.....	37
FIGURA 29 -COLISIÓN EJE X FIGURA 30 - COLISIÓN EJE Y.....	38
FIGURA 31 - COLISIÓN EN AMBOS EJES FIGURA 32 - NO HAY COLISIÓN.....	39
FIGURA 33 - CÓDIGO ENSAMBLADOR MOVIMIENTO.....	39
FIGURA 34 - CASOS DE MOVIMIENTO	41
FIGURA 35 - DIFERENTES TILES Y PUERTAS DEL JUEGO	44
FIGURA 36 - EJEMPLOS DE ENEMIGOS	45
FIGURA 37 - ALGUNOS OBJETOS DEL JUEGO	45
FIGURA 38- SPRITE DEL JUGADOR.....	46
FIGURA 39 - SCREENSHOT (1).....	46

FIGURA 40 - GENERACIÓN DE NIVELES V.0.25	50
FIGURA 41 - GENERACIÓN DE UN NIVEL V.0.5.....	51
FIGURA 42 - CICLO DE EJECUCIÓN DE LA GENERACIÓN DEL NIVEL V.0.75	53
FIGURA 43 - FLUJO DE EJECUCIÓN DE LA GENERACIÓN DE NIVEL	55
FIGURA 44 - SISTEMA DE INTERIORES DEL JUEGO	57
FIGURA 45 - MACRO HABITACIÓN.....	59
FIGURA 46 - DESPLAZAMIENTO ENTRE HABITACIONES.....	60
FIGURA 47 - SEUDOCÓDIGO CAMBIAR EN HABITACIONES Y COLISIONES CON EL JUGADOR.....	61
FIGURA 48 - CONSTRUCCIONES DE LAS HABITACIONES	62
FIGURA 49 - DIBUJADO DE LA HABITACIÓN JUNTO CON LAS PUERTAS.....	63
FIGURA 50 - HABITACIÓN INICIAL.....	63
FIGURA 51 - DIAGRAMA DEL SISTEMA	64
FIGURA 52 - OBJETOS IMPLEMENTADOS EN EL JUEGO	67
FIGURA 53 - HABITACIÓN CON OBJETO.....	67
FIGURA 54 - MACRO ACCIÓN OBJETO	68
FIGURA 55 - SEUDOCÓDIGO COLOCAR OBJETO	68
FIGURA 56 - SEUDOCÓDIGO DIBUJAR OBJETOS.....	69
FIGURA 57 - SEUDOCÓDIGO COLISIÓN DE OBJETOS	70
FIGURA 58 - ENEMIGO ROBOT	71
FIGURA 59 - HABITACIÓN CON ENEMIGOS.....	71
FIGURA 60 - MACRO ENEMIGO.....	72
FIGURA 61 - MACRO HABILIDAD ENTIDAD	72
FIGURA 62 - MACRO POSICIÓN ENTIDAD	72
FIGURA 63 - SEUDOCÓDIGO COLOCAR ENEMIGOS.....	74
FIGURA 64 - SEUDOCÓDIGO MAQUINA DE ESTADOS.....	75
FIGURA 65 - MACRO ENTIDAD ATAQUE	80
FIGURA 66 - FUENTE DEL JUEGO	81
FIGURA 67 - PANTALLA INICIAL.....	82
FIGURA 68- GRÁFICOS DEL MENÚ.....	83
FIGURA 69 - OBJETOS DE JUEGO	83

Índice de tablas

TABLA 1 - TABLA DE MODOS AMSTRAD	18
TABLA 2 - COLORES DENTRO DEL AMSTRAD.....	20
TABLA 3 - HABITACIONES V1.....	49
TABLA 4 - HABITACIONES V2.....	50
TABLA 5 - HABITACIONES V3.....	53
TABLA 6 - HABITACIONES V4.....	54
TABLA 7 - HABITACIONES V5.....	58
TABLA 8 - TABLA DE OBJETIVOS DE LOS OBJETOS.....	65
TABLA 9 - LISTA DE OBJETOS IMPLEMENTADOS	66
TABLA 10 - REQUISITOS NO FUNCIONALES 1 - RENDIMIENTO.....	85
TABLA 11 - REQUISITOS NO FUNCIONALES 2 - USABILIDAD	85
TABLA 12 - REQUISITOS NO FUNCIONALES 3 - FIABILIDAD.....	85
TABLA 13 - REQUISITOS NO FUNCIONALES 4 - APARIENCIA	85
TABLA 14 - REQUISITOS NO FUNCIONALES 5 - ESCALABILIDAD	85
TABLA 15 - REQUISITO FUNCIONAL 1 - MOVIMIENTO DE LAS ENTIDADES.....	86
TABLA 16 - REQUISITO FUNCIONAL 2 -DIBUJADO DE LAS ENTIDADES	86
TABLA 17 - REQUISITO FUNCIONAL 3 - LIMPIEZA DE LAS ENTIDADES.....	86
TABLA 18 - REQUISITO FUNCIONAL 4 - GENERACIÓN DE UNA HABITACIÓN	86
TABLA 19 - REQUISITO FUNCIONAL 5 - GENERACIÓN DE UN NIVEL PROCEDIMENTAL	87
TABLA 20 - REQUISITO FUNCIONAL 6 - CAMBIO ENTRE HABITACIONES	87
TABLA 21 - REQUISITO FUNCIONAL 7 - ATAQUE DEL JUGADOR.....	87
TABLA 22 - REQUISITO FUNCIONAL 8 - CARGADOR DE OBJETOS.....	87
TABLA 23 - REQUISITO FUNCIONAL 9 - GESTIÓN DE LAS FUENTES Y EL TEXTO	88
TABLA 24 - REQUISITO FUNCIONAL 10 - DIBUJADO DE LOS OBJETOS.....	88
TABLA 25 - REQUISITO FUNCIONAL 11 - MAQUINA DE ESTADOS	88
TABLA 26 - REQUISITO FUNCIONAL 12 - MODIFICAR ATRIBUTOS DEL JUGADOR	88
TABLA 27 - REQUISITO FUNCIONAL 13 - INTERACTUAR CON LOS OBJETOS DE LA HABITACIÓN	88
TABLA 28 - REQUISITO FUNCIONAL 14 - HABITACIÓN TESORO	89
TABLA 29 - REQUISITO FUNCIONAL 15 - SISTEMA DE CARGA DE ENEMIGOS.....	89
TABLA 30 - REQUISITO FUNCIONAL 16 - MENÚ PRINCIPAL	89
TABLA 31 - REQUISITO FUNCIONAL 17 - CAMBIO ENTRE NIVELES.....	89
TABLA 32 - REQUISITO FUNCIONAL 18 - CAMBIO DE PALETA DE COLORES.....	89
TABLA 33 - REQUISITO FUNCIONAL 19 – GESTIÓN DE MÚLTIPLES ENTIDADES	90
TABLA 34 - REQUISITO FUNCIONAL 20 – GESTIÓN DEL TECLADO	90

1. INTRODUCCIÓN

1.1. RESUMEN

El objetivo es desarrollar un videojuego en un Amstrad CPC 464 pasando por todas las etapas de desarrollo (diseño, implementación, testeo, producción y distribución). Este proyecto requiere de los conocimientos de la maquina a bajo nivel, es decir, conocer el funcionamiento de Hardware para sacarle el mayor rendimiento posible a la maquina objetivo.

Tradams es un juego del género *dungeon crawler* y *roguelike* en el que combates a lo largo de los diferentes niveles, contra varios enemigos y que a lo largo del juego dispones de diferentes objetos para mejorar al jugador y poder hacer frente a las diferentes situaciones.

Este proyecto cuenta con dos puntos clave, aquellos puntos que le hacen distinguirse a este proyecto del resto:

El primer punto clave de este proyecto es el lenguaje utilizado, el lenguaje ensamblador. Un lenguaje de bajo nivel que permite programar directamente las ordenes de la CPU y a la vez requiere de los conocimientos sobre el funcionamiento de la computadora. Al ser un lenguaje directo es un requisito tener mayor cuidado a la hora de que ordenes ejecutas y el orden de estas ya que no cuentas con un sistema de *debug* como con los lenguajes de alto nivel.

Existen diferentes lenguajes de ensamblador, este pertenece a un conjunto de lenguajes de ensamblador escritos en lenguaje C de programación (ASXXXX) que mediante un conjunto de configuraciones le permite adaptarse a diferentes máquinas y procesadores.

El objetivo de desarrollar en ensamblador son los conocimientos sobre las computadoras que este lenguaje aporta. Requiere de conocimientos sobre el hardware (Funcionamiento de la CPU, gestión de los bancos de memoria, ...) y la lógica matemática que conlleva y que permite conocer como las maquinas trabajan a nivel interno y como el lenguaje de alto nivel se ensambla.

El segundo punto clave que alberga este proyecto es el generador de niveles procedimentales que permite al juego generar niveles diferentes en un tiempo reducido. Este generador toma en cuenta un conjunto fijo de normas que marcan desde el número de habitaciones por nivel hasta la disposición de estas respecto al resto. Estas normas permiten generar las habitaciones y organizarlas en espacios diferentes. También tiene en cuenta el nivel en el que te encuentras para cargar y gestionar los niveles de dificultad mostrando la cantidad de enemigos pertinentes.

Las razones de realizar esta mecánica son las siguientes:

- Al ser una mecánica que nunca trabajada con anterioridad suponía un reto y más aún llevarla a cabo en una máquina de los 80, con las restricciones y limitaciones que este conlleva.
- El segundo objetivo es desarrollar un videojuego similar a The Binding of Issac, un juego conocido por sus mecánicas procedimentales y su capacidad de crear de cada partida una experiencia diferente y en mente conseguir que este proyecto mostrará una experiencia similar.

1.2. PROBLEMÁTICA DEL TRABAJO

Las mayores dificultades que encontré fueron las siguientes:

1. Las limitadas capacidades de la maquina objetivo, siendo estas las de una computadora de 1984 cuya potencia a nivel tanto de hardware como de software se hallan bastante lejos de las actuales.
2. La falta de conocimientos respecto a la propia máquina, su funcionamiento, su software y el lenguaje a bajo nivel que debo emplear para conseguir el mayor rendimiento posible y así desarrollar un proyecto de buena calidad. Este punto aporta un tiempo extra en conocer tanto el lenguaje como la computadora.

Pero también hay dificultades menores, pues al ser maquinaria antigua la cantidad de información, ejemplos y trabajos previos se ven limitados a una comunidad reducida de miembros lo que conlleva mayor tiempo de búsqueda de conocimiento respecto al ordenador seleccionado, sus funciones y sus conocimientos técnicos de la misma (Hardware y Software).

Cabe añadir que el juego tiene como objetivo el desarrollo y creación de los niveles del juego de forma procedural, creando una mecánica que nunca he desarrollado, y menos para una maquina antigua y obtener un buen rendimiento.

1.3. MOTIVACIÓN

Desde el inicio del cuarto ciclo del grado quise escoger el itinerario de Creación y Entretenimiento digital, pero no fue hasta que ya estaba inmerso en este que desconocía que apartado del mundo de los videojuegos quería ejercer y también darme cuenta lo poco que conozco respecto al mundo del entretenimiento digital y que al menos debería conocer para al alcanzar una base que me permita ser reconocido para trabajar en el mundo empresarial.

Por ello escogí este proyecto como trabajo de fin de grado, pues compartía y comparto que el conocimiento a bajo de nivel de las computadoras te ayuda a desarrollar tu agudeza como programador, enfocar diferentes soluciones y conocer en que se transforma tu lenguaje de alto nivel a la hora de ensamblarse, lo que te da una perspectiva más amplia sobre que lógicas son mejores desarrollar a la hora de programar. A la hora de programar se pueden desarrollar diferentes soluciones para un mismo problema, pero estos conocimientos te aportan soluciones que son más afines al comportamiento de la máquina, lo que conlleva un mejor resultado. Me gustaría pensar que a esta conclusión llegue yo solo, pero puedo agradecer que me la inculcará mi tutor y me llevará a realizar este proyecto.

Podría haber escogido otros proyectos de programación con motores actuales o haber realizado una propuesta propia, pero sopesando los diferentes trabajos y propuestas que tenía en mente, este proyecto cumplía en mayor medida mis objetivos que era desarrollar un bloque distinto conocimiento respecto a la programación, uno más cercano a la máquina, y a la vez un reto para medir hasta donde puedo llegar con mi nivel actual.

Hasta que no comencé con el proyecto no me percate la complejidad del trabajo y la cantidad de tiempo de aprendizaje que me iba a necesitar para afianzar unos conocimientos mínimos, pero me alegra por ello, pues creo que me hallo en el momento de enfrentar distintos proyectos que me hagan llegar a mis límites y consigan sacar tanto lo mejor como lo peor de mí.

Lo mejor que me aporto fueron aquellos momentos que puedo ver cuanto he avanzado, aplicar los conocimientos adquiridos y alcanzar nuevas soluciones. Lo peor fue que no supe valorar el enfoque del ensamblador y quise enfocarlo como un paradigma complemento distinto cuando realmente es otro lenguaje más de programación.

A lo largo de la carrera se nos ha enseñado diferentes conceptos, capacidades y funcionamientos de los computadores que, aun siendo necesarios para mi pesar, no he comprendido bien o no he visto relevantes por mi falta de experiencia, y que es ahora cuando estoy viendo lo necesarios que son.

Cabe recalcar que considero que esos conocimientos no han sido bien impartidos, no se han mostrado de forma adecuada o se han dado sin que el alumno pudiera aportar una base suficiente para asentarlos. Lo que ha propiciado que muchos compañeros de la carrera, entre ellos yo, no prestara la atención debida y por lo tanto carezca de los mismos. Por ello, este proyecto me ayuda a rescatar parte de esos conocimientos que a mi pesar ya debía tener para poder aplicarlos en el futuro.

Aprendemos en el grado sobre hardware actual cuya complejidad es alta, sin partir de una base. Creo que un mejor modelo sería conocer primero ordenadores de menor complejidad e interpolar esos conocimientos a posteriori a ordenadores actuales y a proyectos de mayor envergadura. Pues he obtenido mayor entendimiento de una computadora en este proyecto que el trabajado durante los tres primeros años de carrera.

Con este trabajo se busca conocer el funcionamiento del código ensamblador, conocer sus fortalezas y sus debilidades. Por el mismo conocimiento que aporta el código conocer como el ordenador ejecuta sus programas, desde la gestión de los diferentes registros de la CPU hasta la gestión y optimización de cada línea de código para un mejor uso de la memoria. De esta manera obtenemos una mayor percepción de cómo trabajan los programas, desde la creación del código hasta la compilación y el ensamblado de este.

Creo que estos tipos de conocimientos deberían ser impartidos a una edad temprana de la carrera pues aportan una buena base donde mejorar varios ámbitos de un ingeniero, dando una mejor base para conocer el funcionamiento de los computadores, mayor nivel para la programación y conocimientos más globales sobre la computación.

1.4. PROPÓSITO O FINALIDAD DEL TRABAJO

- Conocimientos sobre el funcionamiento de máquinas de 8 bits
- Analizar diferentes librerías, *frameworks* y *game engines* de las máquinas de 8 bits
- Aprender el lenguaje ensamblador y conocer a bajo nivel el ordenador seleccionado
- Comprender el funcionamiento del hardware de la máquina y explotarlo
- Diseñar y desarrollar un videojuego para una computadora de 8 bits
- Aplicar diferentes técnicas del mundo de los videojuegos
- Aplicar técnicas de los videojuegos de época (los 80)

1.5. ESTRUCTURA DEL DOCUMENTO

El documento se compone de 7 partes y se enseñara los diferentes procesos, métodos y cavilaciones que se han llevado a cabo con finalidad de la realización y desarrollo del proyecto.

Partimos de una introducción que mostrara un breve resumen del trabajo, problemas iniciales que conllevan el proyecto y la motivación que me ha llevado a realizar dicha tarea y el propósito de esta.

A continuación, el marco teórico el cual muestra todos los estudios e investigaciones previas al trabajo diferenciado en varios puntos. El primero de estos siendo la terminología utilizada a lo largo de la investigación y estudios previos al desarrollo del proyecto principal.

En tercera posición la metodología llevada a cabo en el proyecto siendo en este dónde se desarrollan, se muestran las técnicas y procedimientos aplicados a lo largo del trabajo.

Cuarto punto, cuerpo del trabajo, cuyo objetivo es proyectar todo el desarrollo del videojuego y mostrar de forma diferenciada su parte de diseño y especificación de la parte de desarrollo, implementación y publicación de este y como último apartado de este las herramientas utilizadas para realizar el proyecto.

Las conclusiones son el siguiente punto, en él se contienen tanto los objetivos conseguidos como los que no, y un resumen de lo conseguido. Mostrando de forma reducida en un único con punto que se ha obtenido como producto final.

Y por último las referencias y la bibliografía de la que he dado uso a lo largo de trabajo, tanto documentación física como la digital. Sin olvidarnos de la información aportada en los anexos para comprender diferentes puntos de la memoria.

2. MARCO TEÓRICO

2.1. TERMINOLOGIA

Hardware: Conjunto de elementos físicos o materiales que constituyen un ordenador, siendo este conjunto tanto los componentes en el interior de la maquina como los periféricos conectados a esta.

Software: Conjunto de rutinas y programas que permiten realizar diferentes tareas de la computadora.

Bit: Unidad mínima de información empleada en las ciencias de la computación.

Byte: Unidad de información equivalente a 8 bits.

Sprite: Elemento visual activo de la imagen en una Pantalla

Pixel: Unidad básica de una imagen en una pantalla.

Tile: O baldosa siendo esta su traducción al castellano se trata de parte gráfica de un videojuego que se utiliza para completar un escenario.

Tileset: Conjunto de Tiles de un videojuego con temática similar que busca el diseño de diferentes niveles utilizando los mismos recursos.

CPU: *Central Processing Unit*, unidad central de procesado, encargada de interpretar las órdenes del software y de realizar cálculos matemáticos.

Frame: Fotograma, imagen que se muestra por la pantalla en un momento específico.

FPS: *Frame Per Second*, conjunto de fotogramas que se muestran en pantalla por segundo.

Amstrad: Gama del Ordenador objetivo de este proyecto.

CPC 464: Modelo del Computador objetivo del proyecto.

Zilog Z80: CPU utilizada por el Amstrad CPC 464

RGB: Sistema de color basado en tres colores primarios (Rojo, verde y azul) y que hace uso de estos tres para crear el resto.

RAM: *Random Access Memory*, memoria de almacenamiento del Amstrad donde se situará el videojuego.

ROM: Memoria solo de lectura donde se halla contenido el sistema operativo del Amstrad.

SO: Sistema operativo, conjunto de órdenes y programas que controlan los procesos básicos de un ordenador y permiten el funcionamiento de otros programas.

Frameworks: Entorno de trabajo estandarizado de conceptos, prácticas y criterios para enfocar un tipo de problemática particular.

Game Engine: También llamado Motor de Videojuego, software que permite realizar diferentes gestiones tanto a nivel de gráficos como de programación de videojuegos.

HUD: Se trata de la parte de un videojuego que se encarga de mostrar el estado de este o del personaje dentro del escenario. Aportada información relevante para el jugador.

Struct: Es una nomenclatura utilizada a la hora de programar en ciertos lenguajes para definir una estructura de datos.

POO: Programación orientada a objetos.

2.2. ESTUDIOS PREVIOS PLANTEADOS

2.2.1. DESCRIPCIÓN MAQUINA SELECCIONADA

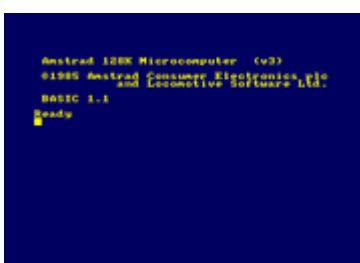


Figura 1: Amstrad CPC 464

[Esta foto](#) de Autor desconocido está bajo licencia [CC BY-SA](#)

Según Wikipedia, **Amstrad CPC 464** es el ordenador doméstico fabricado y comercializado por la empresa Amstrad Consumer Plc a partir del 1984. Fue el primer computador desarrollado por la compañía pues eran fabricantes de televisores, radios y Hi-Fi.

Dotado del **procesador Zilog Z80**, el cual alcanzaba una velocidad de 4 MHz, aunque debido al CRTC¹, durante un periodo de tiempo el procesador se halla ejecutando este, lo que finalmente obtenía un tiempo efectivo de proceso medio de 3.3 MHz, siendo el tiempo restante de ejecución empleado por el CRTC, todo ello por evitar el efecto “snowing”². Siendo la Gate Array la parte del computador de gestionar las ejecuciones entre CRTC y el tiempo de programa.



[Esta foto](#) de Autor desconocido está bajo licencia [CC BY-SA](#)

Figura 2: Locomotive BASIC 1.1

En cuanto a memoria, dispone de 64 kB de **memoria de RAM** del cual 16 kB pertenecían a la memoria de video, el modelo

¹ Controlador de gráficos 6845 CRTC (Cathode Ray Tube Controller) el cual ayuda a la generación de la señal de video del Amstrad CPC

² Efecto producido por conflictos entre la CPU y los circuitos de video al acceder a la memoria principal.

más básico. Por ello con el tiempo se creó ampliaciones de RAM hasta un total de 512 KB permitiendo una mayor gama de programas, siendo un total de 576 KB junto a los 64KB base del ordenador.

Al no poder gestionar Amstrad direcciones no mayores a 64 KB, estas ampliaciones podían consumir ciclos de ejecución al tener que cambiar entre los diferentes bancos de memoria.

Respecto a su software partió de BASIC hasta crear una versión propia, **Locomotive BASIC 1.0**. Siendo este una adaptación con mayor facilidad de uso, pero con las mismas funcionalidades que el original.

La computadora deja escoger entre diferentes modos gráficos y diferentes modos de texto, pero dependiendo del uso de un modo u otro tenían limitaciones diferentes. Un total de 3 modos gráficos.

Estos modos tenían sus limitaciones en la forma que tenía la computadora de codificar y mostrar la información por pantalla. Según los diferentes modos puede utilizar una cantidad de colores por pantalla. Cuando menor cantidad de colores más pixeles se podía codificar en un solo byte.

Partimos de los modos gráficos siendo el **Modo 2**, el modo con más resolución de estos, pero con la limitación de 2 colores máximo. Siendo una resolución de 640x200, en texto son unas 80 columnas de caracteres. Esta resolución se conseguía ya que la hora de codificar los pixeles en pantalla cada byte podía realizar una representación de cada pixel con cada bit (0 -> Primer Color y 1 -> Segundo Color).



Figura 3: Imagen Modo 2

Siendo el **Modo 1** la resolución intermedia de 320x200 y un total de 4 colores, al igual que el anterior el ordenador debe representar los pixeles posibles por byte, al doblarse la



Figura 4: Imagen Modo 1



Figura 5: Imagen Modo 0

cantidad de colores el ancho de la pantalla se ve mermado a la mitad, ya que emplearán dos bits para la representación del pixel.

Y por último el **Modo 0** con una resolución de 160x200, pero permitiendo un total de 16 colores simultáneos, es decir, 4 bits para la representación de un solo pixel. Existe un **cuarto modo**, siendo este de 160x200 con 4 colores, pero se trata de un modo no oficial.

Realmente cuando hablamos a la cantidad de pixeles que representa no significa que hallan menos en pantalla si no que al no poder representar cada uno de ellos de forma separada se ve obligado a representarlos en subconjuntos de pixeles. Siendo estos los valores de representación.

Tabla 1 - Tabla de Modos Amstrad

Modo	Numero de Colores	Color/Pixeles
Modo 0	16	1:4
Modo 1	4	1:2
Modo 2	2	1:1

Nunca se informó de las posibilidades reales del Amstrad, sobre todo respecto al chip gráfico que podía conseguir mejores resoluciones al utilizar técnicas de “overscan³”, alcanzando hasta 768x280 de resolución, aunque aumentaba la cantidad de memoria RAM a memoria de video (24 kB) no siendo esta la única forma de obtener una mayor resolución, pero sí de las técnicas más conocidas.

También se podía conseguir manejar más colores en pantalla de los asignados por el modo con técnicas como “Split screen⁴” aplicando diferentes paletas de colores a diferentes segmentos de la pantalla y otros efectos se aplicaban sobre el ráster.

³ Técnica que se trata de modificar el tamaño de la pantalla dentro del chip gráfico y así utilizar el total de esta a costa de memoria de RAM la cual será dedicada a la de video.

⁴ Técnica que hace uso del sistema de interrupciones del Amstrad para realizar un cambio de la paleta de colores a mitad del dibujado y provocando que se muestren por pantalla más de los colores posibles en sus modos.

Todos los modos de pantalla se valen de una **paleta de 27 colores** derivados del espacio de colores RGB, aunque tiempo más tarde con los modelos Plus se extendió a un total de 4096 colores y se añadió un soporte por hardware para sprites.

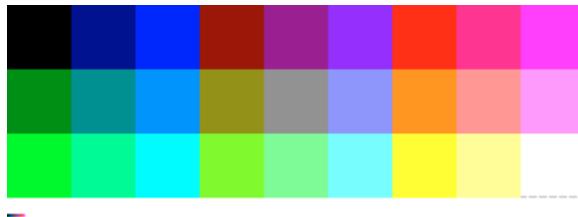


Figura 6: Paleta de 27 colores

A continuación, se puede ver la tabla que representa la **paleta de colores** del Amstrad de forma más extendida.

Firmware Number	Hardware Number	Colour Name	R %	G %	B %	Hexadecimal	RGB values	Colour
0	54h	Black	0	0	0	#000000	0/0/0	
1	44h (or 50h)	Blue	0	0	50	#000080	0/0/128	
2	55h	Bright Blue	0	0	100	#0000FF	0/0/255	
3	5Ch	Red	50	0	0	#800000	128/0/0	
4	58h	Magenta	50	0	50	#800080	128/0/128	
5	5Dh	Mauve	50	0	100	#8000FF	128/0/255	
6	4Ch	Bright Red	100	0	0	#FF0000	255/0/0	
7	45h (or 48h)	Purple	100	0	50	#ff0080	255/0/128	
8	4Dh	Bright Magenta	100	0	100	#FF00FF	255/0/255	
9	56h	Green	0	50	0	#008000	0/128/0	
10	46h	Cyan	0	50	50	#008080	0/128/128	
11	57h	Sky Blue	0	50	100	#0080FF	0/128/255	
12	5Eh	Yellow	50	50	0	#808000	128/128/0	
13	40h (or 41h)	White	50	50	50	#808080	128/128/128	
14	5Fh	Pastel Blue	50	50	100	#8080FF	128/128/255	
15	4Eh	Orange	100	50	0	#FF8000	255/128/0	
16	47h	Pink	100	50	50	#FF8080	255/128/128	
17	4Fh	Pastel Magenta	100	50	100	#FF80FF	255/128/255	
18	52h	Bright Green	0	100	0	#00FF00	0/255/0	
19	42h (or 51h)	Sea Green	0	100	50	#00FF80	0/255/128	
20	53h	Bright Cyan	0	100	100	#00FFFF	0/255/255	
21	5Ah	Lime	50	100	0	#80FF00	128/255/0	
22	59h	Pastel Green	50	100	50	#80FF80	128/255/128	

23	5Bh	Pastel Cyan	50	100	100	#80FFFF	128/255/255	
24	4Ah	Bright Yellow	100	100	0	#FFFF00	255/255/0	
25	43h (or 49h)	Pastel Yellow	100	100	50	#FFFF80	255/255/128	
26	4Bh	Bright White	100	100	100	#FFFFFF	255/255/255	

Tabla 2 - Colores dentro del Amstrad

2.2.1.1. ESPECIFICACIONES TECNICAS

- CPU: Zilog Z80 4MHz, 8 bits
- RAM: 64KB
- ROM: 32KB
- Controlador de video: Amstrad Gate-Array 40010
- Controlador de gráficos: 6845 CRTC
- Chip de sonido: General Instrument AY-3-8912, 3 canales de sonido y 1 canal de ruido blanco
- Teclado: QWERTY/AZERTY/QWERTZ de 80 teclas con *keypad* numérico y teclas para el cursor
- Monitor en color o fósforo verde.
- Unidad de casete integrada
- Zócalo para unidad de disco de 3"
- Interpretador: Locomotive BASIC 1.0

2.2.1.2. FUNCIONALIDADES

- Modos de texto
 - 20x25 caracteres
 - 40x25 caracteres
 - 80x25 caracteres
- Modos gráficos
 - Modo 0: 160x200 en 16 colores
 - Modo 1: 320x200 en 4 colores
 - Modo 2: 640x200 en 2 colores
- Paleta de 27 colores

2.2.2. PROYECTOS PREVIOS DE ESTUDIO

- JUEGOS PROCEDIMENTALES

Durante los años en los que surgieron los primeros videojuegos, allá por la década de los 60, la principal razón para añadir contenido procedural a un videojuego eran las limitaciones de memoria.

Por ejemplo, resultaba imposible añadir una gran cantidad de escenarios a un videojuego, simplemente porque no había espacio donde almacenarlo.

En los 80, apareció Rogue, juego de mazmorras donde el protagonista debería enfrentarse a diferentes enemigos y recoger diversos objetos en cada partida. Cada mazmorra se generaba de forma aleatoria, así como la posición de los objetos y enemigos obteniendo de esta manera una experiencia diferente en cada partida sin tener que recurrir a más memoria, diseñarlos o pre almacenar los escenarios.



Figura 7 : ROGUE (1980)

Unos pocos años después, en 1984 Tetris (el juego más vendido de la historia) y conocido por todos. Fue el primer juego de puzzles donde cuyo contenido de este no estaba preestablecido.

Las partidas pasaban a ser infinitas y en cada una de ellas la secuencia que iba apareciendo era diferente. Esto hizo que un simple juego de puzzles fuese re-jugable ya que cada partida ofrecía un desafío además que su jugabilidad lo volvía adictivo por tal de superarse tus propias puntuaciones.



Figura 8- Tetris (1984)

Conforme los soportes de almacenamiento de videojuegos fueron ofreciendo una mayor capacidad, las empresas se centraron más en diseñar ellos mismos el contenido de este en lugar de generarlo de forma procedimental. Esto no quiere decir que los videojuegos procedimentales desapareciesen por completo, pero sí que pasaron a tener menos importancia que en años anteriores.

En 1996, Blizzard North lanzó Diablo, un videojuego que modernizó el subgénero de los *roguelike*. Durante cada partida el escenario era generado de forma aleatoria. Además, las propiedades de los objetos también se generaban aleatoriamente, lo que hacía que cada partida que empezabas se diferenciase bastante de la anterior. Como ya sucedió en su día con Rogue, comenzaron a surgir videojuegos con ideas similares a las del Diablo, también llamados clones de Diablo.



Figura 9 - Diablo (1996)

Como curiosidad, en 2004 el grupo .theprodukkt presentó. kkrieger. Se trataba de una demo de un videojuego de acción en primera persona cuya principal característica era que consistía en un único ejecutable que solo pesaba 96 kb.

Esto se consiguió principalmente generando los elementos de forma procedural como los modelados o las texturas. El videojuego como tal no era nada que no se hubiese visto antes, pero esa capacidad de generar el contenido de este de forma procedural sí que llamó bastante la atención (el videojuego pasaba de 96 kb que ocupaba en el disco duro a unos 300 Mb que ocupaba en memoria de video).



Figura 10 - KKrieger (2004)

Llegamos a 2006, año en el que apareció el que quizás sea el videojuego con más contenido procedural que nunca se haya realizado, Dwarf Fortress. Cada vez que se empezaba una partida, se generaba un mundo entero, con su disposición del terreno, fauna, personajes, monstruos, historias, etc. Todo ello permitía al jugador vivir su aventura en un mundo rico y complejo.

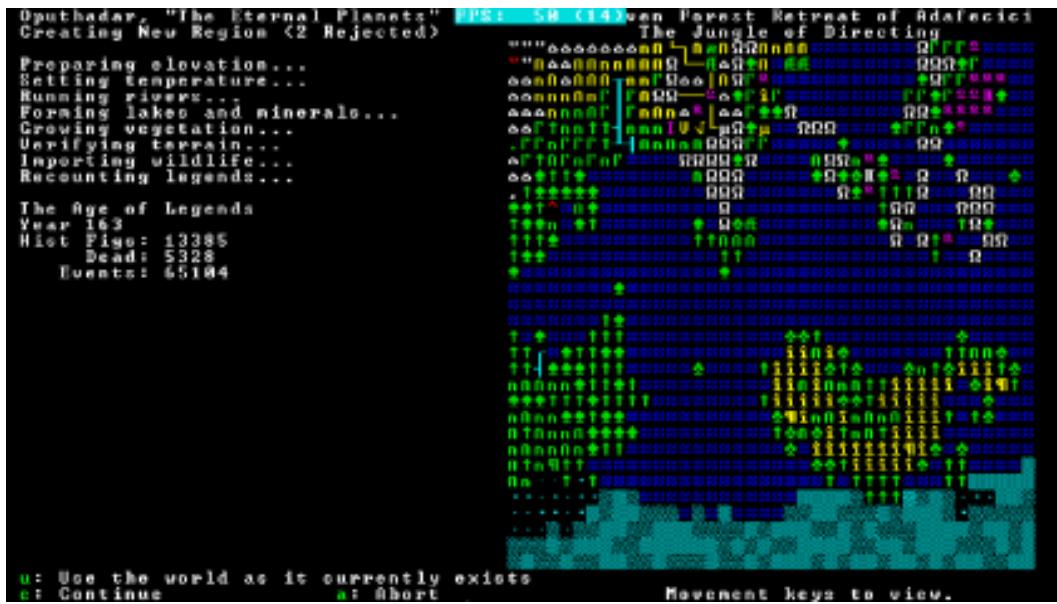


Figura 11 - DWARF FORTRESS (2006)

En el año 2008, Valve lanzó Left 4 Dead, un videojuego de acción en primera persona donde el jugador tenía que moverse por un escenario eliminando a las oleadas de zombis que iban apareciendo. El videojuego contaba con un sistema de IA que permitía modificar elementos como la cantidad de zombis o munición que te encontrabas según como fuese el transcurso de la partida.

De esta forma la partida se iba amoldando a la habilidad del jugador y cada una ofrecía una experiencia diferente (dependiendo de la propia habilidad del jugador).



Figura 12 - LEFT 4 DEAD (2008)

Llegamos al año 2009, en el cual fue lanzado Minecraft, el videojuego que volvió a elevar la popularidad de los juegos procedimentales. Cada vez que el jugador empezaba una partida, se generaba un mundo compuesto por cubos de forma aleatoria. Al jugador se le daba total libertad para explorarlo y modificarlo a su antojo, preparándose para defenderse de las amenazas que surgían cuando caía la noche.



Figura 13 - MINECRAFT (2009)

Cabe destacar que el lanzamiento de Minecraft coincidió con el auge de los desarrolladores independientes, que formaban equipos de unas pocas personas, muy lejano a los equipos de grandes producciones que perfectamente contaban con más de cien trabajadores. De hecho, Minecraft comenzó siendo desarrollado únicamente por Markus Persson, aunque posteriormente fundaría Mojang junto a otros desarrolladores.

Lo anterior es significativo, ya que llegados a este punto los videojuegos de las grandes empresas requerían de equipos muy grandes para añadir el contenido al videojuego, algo inviable para los desarrolladores independientes.

Sin embargo, la generación procedural del contenido permitía generar mundos amplios sin necesidad de tener a un equipo entero poniendo mano a mano cada elemento de este.

Durante el año 2009 también fue lanzado Spelunky, un videojuego de plataformas en 2D en el cual el jugador tenía que bajar por una cueva explorándola y enfrentándose a los peligros que acechaban. En este videojuego también creaba un escenario diferente cada vez que

empezabas una nueva partida (cosa que hacías con bastante frecuencia dada la dificultad de este).



Figura 14 - SPELUNKY (2009)

En 2011 fue lanzado The Binding of Isaac, un videojuego en el que manejabas a un niño que tenía que huir de su madre a través de las mazmorras del sótano de su casa. Este juego contenía la esencia de los *roguelikes*, ya que en cada partida tanto las habitaciones de la mazmorra como su tamaño se distribuían de forma diferente, además de los enemigos y los tesoros que te encontrabas.

La agilidad con la que avanzabas en una partida, así como las grandes cantidades de objetos que podían contenerse en cada nivel y la gran variedad de enemigos que podías encontrar proporcionaba una gran re-jugabilidad al título y una sensación diferente en cada partida.



Figura 15 - THE BINDING OF ISAAC (2011)

Y el último gran desarrollo de los juegos procedimentales fue No Man's Sky el cual busca la creación de un universo, es decir, desde los seres vivos hasta la composición de los planetas.

Ecosistemas y toda zona del juego se generaba por lógica computacional, pero a diferencia del resto este cuenta con una generación del entorno a una nueva escala donde se generan hasta un total de 18,446,744,073,709,551,616 mundos.



Figura 16 - No Man's Sky

A partir del lanzamiento de Minecraft muchos otros videojuegos con contenido procedural fueron lanzados, ya que se demostró que un equipo de desarrolladores independientes podía llegar a desarrollar videojuegos que igualaban o incluso superaban a otros videojuegos de grandes estudios en lo que a duración se refiere. Al fin y al cabo, el mantener a

un jugador durante muchas horas jugando a tu videojuego es indicador de que has hecho las cosas bien.

Se podría decir que Minecraft pertenece a ese conjunto de juegos que marco un cambio en la tendencia de los videojuegos de su época e imponiendo su estilo un gran conjunto.

- EFECTOS DE SPLIT SCREEN

Este efecto aplicado en Amstrad CPC busca la sincronización del sistema de interrupción y la sincronización vertical, y de ahí buscar lo momentos de interrupción específicos para realizar un cambio en la paleta de colores lo que provoca que se puedan utilizar diferentes paletas de colores en diferentes secciones de la pantalla.

Este efecto se aplica cuando se quieren utilizar más colores de los que el modo del Amstrad te permite. Si a la hora de dibujar el HUD querías aplicar una paleta de colores diferente, al sincronizar el sistema de interrupciones con el dibujado de la pantalla podías conocer en que parte de la pantalla se encontraba el *raster* y aplicar en ese momento el cambio.

Varios de los juegos desarrollados a lo largo de siglo XX utilizaban este efecto para ampliar la ya limitada paleta de colores de los ordenadores de época permitiendo que la parte designada al escenario del juego tuviera una gama diferente del HUD del mismo.

Dando así la sensación de tener muchos más colores en pantalla de los reales y mejorando visualmente el juego, lo que atraía más a la hora de jugar.

- SCROLL

Efecto desplaza el escenario de un juego y busca la gestión de mapas más grandes que la resolución por pantalla puede mostrar. Crea el efecto de estar desplazándose dentro del nivel cuando es el contenido el único que se mueve. Este efecto permitió la realización y montaje de entornos más grandes y más vivos.

Se debe buscar un desplazamiento suave exigiendo que el juego se optimice para no tener bajadas de *frames* ya que puede mostrar un desplazamiento brusco pudiendo provocar a la jugador mareos u otras dolencias derivadas del mareo.

Uno de los mayores exponentes de este efecto es el Super Mario Bros, creando grandes niveles haciendo uso de esta técnica. Haciendo uso del *scroll* horizontal



Figura 17. Logo de Super Mario Bros (1987)

A lo largo de la industria se utilizó este efecto para diferentes, creando variantes de este, como *Scroll Vertical*, *Scroll Libre* o *Scroll Paralelo*. Siendo más de una capa en las que se aplica el efecto.



[Esta foto](#) de Autor desconocido está bajo licencia [CC BY-NC](#)

Figura 18 - Ejemplo de Scroll Horizontal

En la imagen superior se pueden ver una imagen muy ancha que muestra una parte del mapa de Mario Bros, en el juego solo se muestra una zona menor a la de la imagen siendo el resultado el siguiente:



[Esta foto](#) de Autor desconocido está bajo licencia [CC BY-NC](#)
Figura 19 - Zona Ampliada Scroll Horizontal

Encontrándose el jugador cerca del tubo de la parte final del mapa, antes de la escalera a la bandera.

Estos efectos se han aplicado en mapas cada vez más grandes y con mayor complejidad como realizaba los juegos de Pokémon de la Game Boy mostrando el uso de *scroll libre* y que ha seguido implantado hasta los días de hoy.



Esta foto de Autor desconocido está bajo licencia [CC BY-SA](#)

Figura 20 - Mapa de Pokemon

- SPLIT RASTER

El *Split Raster* se trata de un efecto provocado por el *raster* de la pantalla cuando a mitad de dibujado se realizan cambios en la paleta de colores, este cambio se realiza mientras el *raster* se encuentra en pantalla y al realizar el cambio genera un efecto de pintado de varios colores y de continuidad entre estos.

Este efecto se puede ver en títulos como Gauntlet 2, el cual mostraba su efecto en la pantalla del Título⁵. Este juego se trataba de un juego de mazmorras y de *hack and slash* de 1986 que aplicaba varios efectos que quiero emplear en el proyecto (*Scroll libre*, *Split Raster*, *Split Screen*, ...).

⁵ Se puede ver este efecto en [Youtube](#)



Figura 21 - Gauntlet II

- JUEGOS DE MAZMORRAS

Este tipo de juegos nace a partir de los juegos de aventura y rol de tablero que busca contar la historia de los personajes que exploran una mazmorra, luchan contra diferentes enemigos y van sumergiéndose en los misterios de la mazmorra.

El primer juego de exploración de mazmorras fue pedit5, inspirado en Dungeons & Dragons, desarrollado en invierno de 1975 por Rusty Rutherford en el sistema de educación interactivo PLATO, basado en Urbana (Illinois).



Figura 22 - pedit5 (1975)

Más juegos como este aparecieron al poco como dnd también inspirado en Dungeons & Dragons y Moria (1978, inspirado en dnd). Estos eran similares a pedit5 lo que no supusieron un gran avance.

Fue en la década de los 80' cuando aparecen las primeras mecánicas extensas de los *dungeon crawlers* en videojuegos se remontan a títulos como Wizardry, Might & Magic y Megami Tensei entre otros, siendo esta década la época de oro del género.



Figura 23 - Might and Magic Book One: The Secret of the Inner Sanctum

Con el desarrollo de este género aparecieron nuevas técnicas en el género. Sus precursores que ya hemos tratado en el apartado procedural ([Rogue](#)), los cuales con la generación procedural de niveles conseguían crear experiencias diferentes, era ahora algo que se perseguía dentro del género.

Más tarde con juegos 3D con Diablo es cuando el género llegó a un nivel de experiencia mayor, ya se contaban con sistemas de *loot*, niveles y el juego contaba una historia más inmersiva apoyada por el apartado gráfico.

Con los años este género ha desembocado en otros subgéneros, siendo estos los nuevos precursores que han impulsado la evolución de este.

Durante varios años el género de Juegos de Mazmorra no ha sufrido cambios radicales o grandes avances, siendo los subgéneros los impulsores como The Legend of Zelda: Ocarina of Time, siendo un juego este de acción-aventura, pero con algunos matices y usos de las mecánicas de los juegos de mazmorras.



Figura 24 - *The Legend of Zelda: Ocarina of Time* (1998)

Con los años la cantidad de juegos publicados ha crecido de forma exponencial creando una gran cantidad de títulos de juegos de mazmorras que han avanzado en las mecánicas y gráficamente pero no han explorado más en el género de forma profunda y utilizando la mayoría de los recursos mecánicos ya creados en antaño.

Aunque el avance de la tecnología ha permitido introducir historias más completas y un universo más profundo. Estos avances han eliminado las restricciones que provocaba la memoria siendo esta ahora un recurso económico.

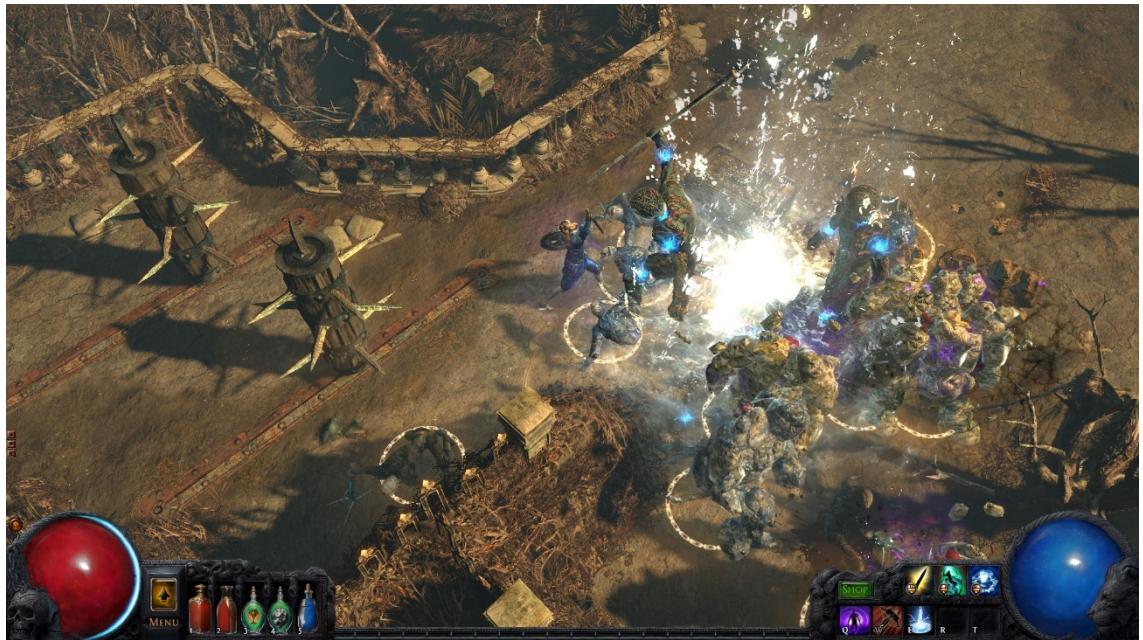


Figura 25 - *Path of Exile* (2013)

3. METODOLOGÍA DEL TRABAJO

Este proyecto parte de mi inexperiencia y desconocimiento del nivel básico del funcionamiento de una máquina, los tipos de memoria, el objetivo de cada bloque de memoria y el funcionamiento interno de un procesador. Con el objetivo de adquirir unos conocimientos básicos realice un juego más simple que me ayude asentar unas bases sobre la programación en ensamblador y realizar mis primeras funciones en este lenguaje.

Pero a la hora de programar se puede alcanzar la solución a un problema de formas diferentes, pero no siendo las soluciones mas optimas. A la hora de trabajar en proyectos en el pasado aplique una metodología que me ayudaba a conseguir soluciones más eficaces. Esta metodología se dividía en tres capas:

1^a Capa: Alcanzo la primera solución a una problemática (mecánica de juego, estructura, sistema, ...) y con todas sus funcionalidades, pero al ser una primera aproximación el código es pobre y necesita optimizarse para conseguir una mejor experiencia.

2^a Capa: En la segunda capa se busca obtener una mejor solución a la problemática partiendo de los conocimientos adquiridos. Esos conocimientos te llevan a lograr una solución más optima y de forma más organizada.

3^a Capa: En esta ultima se realiza una última evaluación buscando resolver posibles problemas vigentes y posibles mejoras buscando pulir la segunda capa y así obteniendo el resultado más optimo con los conocimientos del momento.

Todo lo anterior siempre va acompañado de un trabajo previo de planteamiento teórico sobre papel, donde se plantean diferentes diseños e interacciones que pueden llevarse acabo para resolver los problemas y de forma teórica calcular la gráfica que define cada una de estas soluciones.

Esta grafica maneja 4 ejes diferentes (tiempo, calidad, personas y coste), en mi caso tanto el coste como el número de personas se mantienen constantes pero la parte de tiempo y calidad es la que siempre tiene las fluctuaciones.

En la gráfica se puede ver los tres ejemplos más comunes a la hora de afrontar un proyecto.

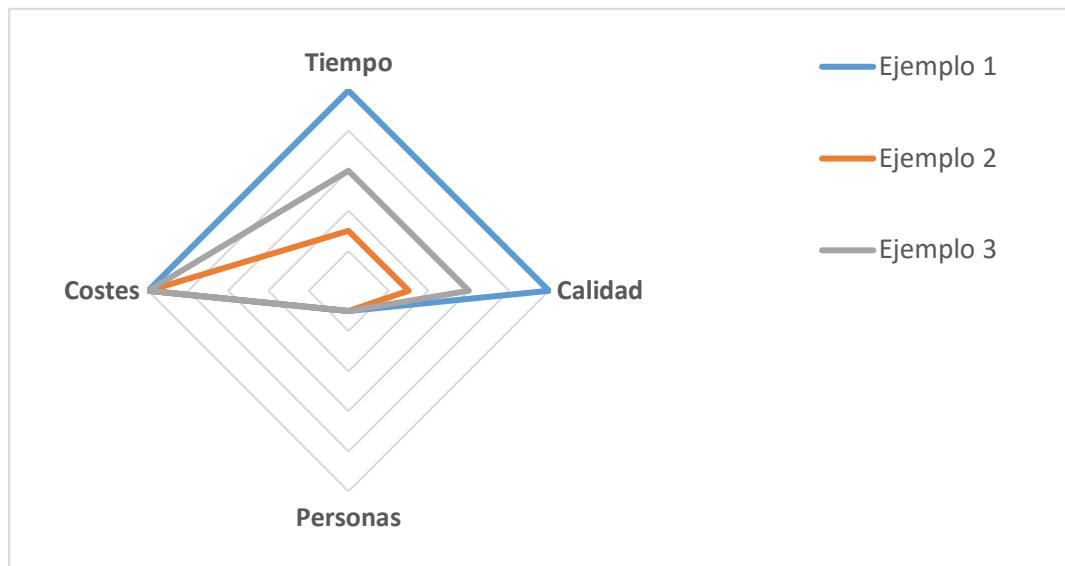


Figura 26 - Grafica de Costes

- **Ejemplo 1:** Al tener más tiempo permite conseguir una mayor calidad ya que puedes sopesar diferentes opciones y escoger de ellas la mejor, incluso revisar y optimizar tu trabajo.
- **Ejemplo 2:** El tiempo en este es el menor, lo que supone que calidad del producto queda reducida y se relega a una calidad que ayude a salir del paso.
- **Ejemplo 3:** Digamos el más equilibrado de los tres buscando una calidad estándar por una cantidad de calidad media, es decir, te deja sopesar entre diferentes soluciones, pero sin dejarte excesivo tiempo como para dedicarle demasiado a cada cuestión.

Estos ejemplos reflejan las diferentes situaciones y decisiones que he tenido que realizar a lo largo del proyecto, a la hora de afrontar cada problemática, cada mejora, cada interacción ... Consiguiendo soluciones óptimas en ciertos aspectos del juego como la Generación de niveles (**ejemplo 1**), resolver problemas de forma más pobre por falta de tiempo (**ejemplo 2**) o consiguiendo soluciones más equilibradas como las aplicadas en aspectos como el sistema cargador de objetos (**ejemplo 3**).

4. CUERPO DEL TRABAJO

4.1. INTRODUCCIÓN

A continuación, se utilizará tanto código en ensamblador como seudocódigo con la finalidad de aclarar funcionalidades y explicar de forma más amena las diferentes prácticas llevadas a cabo. También se hará uso de diferente terminología referente a ensamblador ASZ80 que se encontrará explicado en los anexos.

4.2. PROYECTOS ANTECEDENTES

4.2.1. ARKANOID

Se del juego Arkanoid, un juego en el que jugador maneja una barra en la parte inferior de la pantalla con el objetivo de destruir todos los bloques que aparecen en pantalla. Para conseguir este objetivo el juego coloca una pelota en pantalla que rebotará en el nivel y la que utilizarás para golpear los bloques.

La pelota si cruza cierto punto de la parte inferior de la pantalla el jugador perderá una vida así que el jugador también deberá evitar que la pelota alcance el límite golpeándola y redirigiéndola a los bloques para terminar cada nivel.

Para que el jugador no se acostumbre a la velocidad de la pelota, esta incrementa su velocidad a lo largo del tiempo lo que provoca que cuanto más larga sea la partida más difícil será mantener la pelota en pantalla.

1. Estructura

Todos los objetos mostrados en pantalla son considerados entidades con una misma estructura, así facilitando el uso de las mismas funciones y no teniendo que realizar las gestiones de cada una por separado. Estas entidades están organizadas de la siguiente manera:

Figura 27 - Estructura Entidad v1

```
;;=====
;;      TEntity      B1 X Position          B2 Width
;;                  B3 X Last Position
;;                  B4 Y Position           B5 Height
;;                  B6 Y Last Position
;;=====
```

Cada **B-Número** es el Byte y la posición en la estructura de la entidad. Se almacena esta información para conocer su posición actual, ancho y alto de la entidad y su última posición

porque el Amstrad no se encarga de limpiar la pantalla así que el juego tiene que encargarse de la gestión. Tanto el borrado de las posiciones anteriores como el dibujado de las nuevas.

Aunque también de esta manera te da la opción de optimizar, averiguar qué entidades se han desplazado y actualizar solo aquellas que lo requieran. Este juego es una representación reducida del juego original, teniendo solo 3 entidades distintas.

1. **Jugador:** La barra inferior de la pantalla que se encarga de golpear la pelota y permitir terminar el nivel.
2. **Pelota:** Cuadrado que rebota y se desplaza por la pantalla, cuando golpea los cuadros del nivel estos son borrados y si esta pasa por debajo del límite inferior siendo este dónde se encuentra el rectángulo del jugador, este pierde la partida.
3. **Ladrillos:** Son los rectángulos objetivo del nivel que deben ser destruidos para finalizar la partida, se destruyen con la pelota.

2. Código

En este apartado se explica las funciones más importantes y que comparten con el proyecto principal por su funcionalidad, es decir, estás formaban parte del objetivo del desarrollo de la tarea, aparte de obtener los conocimientos necesarios para iniciar el proyecto principal.

- [Colisiones](#)

Función encargada de gestionar las colisiones entre entidades. En el proyecto principal nos encontraremos con varios obstáculos (objetos, cajas, enemigos, ...) y esta función se encarga de verificar cuando hay una colisión.

Arkanoid gestiona la colisión de la pelota con las demás entidades (Pelota – Cajas y Pelota – Jugador).

Figura 28 - Código Ensamblador Colisión

```
;;=====
;; FUNCTION COLLISION_ENTITIES_ASM
;;
;; Input parameters :  IX => Pointer to First TEntity
;;                      IY => Pointer to Second TEntity
;; Return parameters: A => A = 0 (No Colision) || A != 0 (Colision)
;;=====
collision_entities_asm::
call axis_collision
inc ix ; Next Position IX (Width)
```

```

inc iy    ;; Next Position IY (Width)
inc ix    ;; Next Position IX (X LPos)
inc iy    ;; Next Position IY (X LPos)
inc ix    ;; Next Position IX (Y Pos)
inc iy    ;; Next Position IY (Y Pos)
or a     ;; Active Flags
jr nz , axis_collision      ;; IF A != 0 CHECK COLLISION
ret

axis_collision:
ld a , (ix)          ;; First Entity X/Y Pos
add 1(ix)           ;; A = X/Y Pos(A) + Width/Height(IX)
sub (iy)            ;; A = [X/Y Pos(A) + Width/Height(A)] - Second
Entity X/Y Pos(IY)
jr z , no_collision ;; No collision
jp m , no_collision ;; No collision
;; Second Check
ld a , (iy)          ;; Second Entity X/Y Pos
add 1(iy)           ;; A = X/Y Pos(A) + Width/Height(IY)
sub (ix);; A = [X/Y Pos(A) + Width/Height(A)] - Second Entity X/Y Pos(IX)
jr z , no_collision ;; No collision
jp m , no_collision ;; No collision
ret
no_collision:
ld a , #0
ret

```

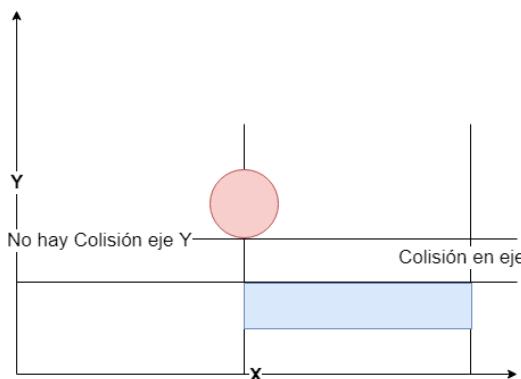


Figura 29 -Colisión Eje X

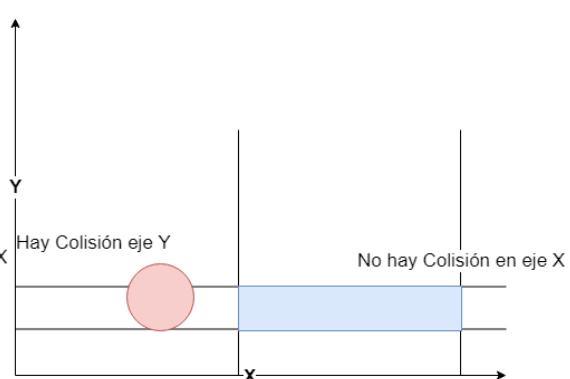


Figura 30 - Colisión Eje Y

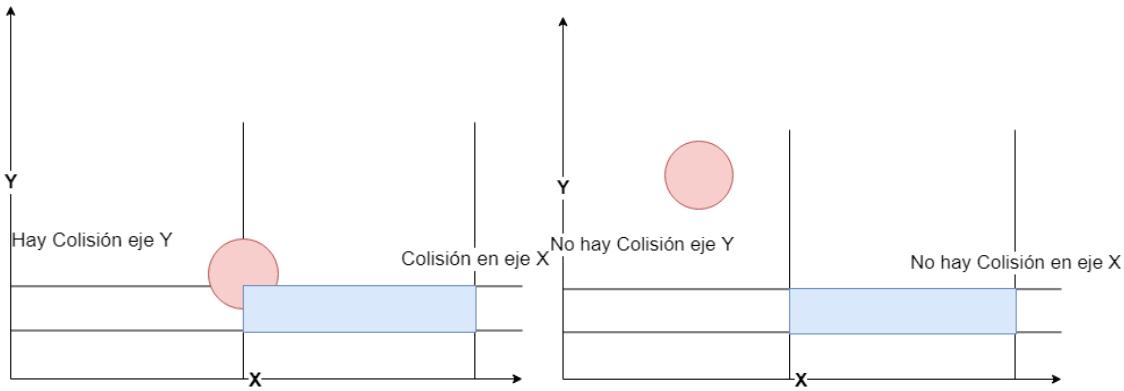


Figura 31 - Colisión en ambos Ejes

Figura 32 - No hay Colisión

Este código funciona de la siguiente manera:

1. Se le mandan 2 entidades a comparar (Pelota y Rectángulo)
 2. Se compara primero uno de los ejes, en mi caso el eje X, si sale positivo continua la ejecución
 3. Se compara el segundo de los ejes (eje Y) para ver si hay colisión.
 4. Se devuelve un número mayor que 0 si hay colisión.
- **Movimiento**

Esta función gestiona el movimiento de las Entidades, esta función es heredada en el proyecto principal tal y como se muestra en este apartado. Su funcionalidad cumple con los requisitos de proyecto y no se encontró forma de alcanzar una mejor versión mejor sin implicar un gran tiempo de desarrollo.

Figura 33 - Código Ensamblador Movimiento

```
;;=====
;; FUNCTION MOVE_ENTITY_ASM
;;
;; Input parameters : HL => Pointer to TEntity
;;                      B => X Movement
;;                      C => Y Movement
;;
;;=====
move_entity_asm::
    ld d , #Max_screen_W    ; D = MAX Screen Width
    ld e , #Min_screen_W    ; E = MIN Screen Width
    call _Movement_ASM       ; Calculate X Movement
    inc hl                  ; Next Position HL (Width)
    inc hl                  ; Next Position HL (X Last Position)
    inc hl                  ; Next Position HL (Y Position)
```

```

ld d , #Max_screen_H      ;; D = MAX Screen Height
ld e , #Min_screen_H      ;; E = MIN Screen Height
ld b , c                  ;; B = C | TO PREPARE _Movement_ASM
call _Movement_ASM         ;; Calculate X Movement
ret

;=====

; FUNCTION MOVE_ENTITY_ASM
;

; Input parameters : HL => Pointer to TEntity
;                      B => Force Movement
;                      D => Max Limit
;                      E => Min Limit
;=====

_Movement_ASM:
    ld a , b                ;; A = Mov(B)
    or a                     ;; Active de Flags
    ret z                   ;; IF A = 0
    jp p , limit_max        ;; IF A < 0
    neg                     ;; Negate A
    ld b , a                ;; B = -A
    jr limit_min            ;; IF A > 0
ret

limit_min:
    ld a , (hl)              ;; A = Pos(HL)
    sub e                   ;; A = Pos(HL) - Min_Limit(E)
limit_min_bucle:
    ret z                   ;; IF A = 0
    dec (hl)                ;; (HL)Pos = (HL)Pos--
    dec a                   ;; A = A--
    djnz limit_min_bucle   ;; IF B !=0
ret

limit_max:
    ld a , (hl)              ;; A = Pos(HL)
    inc hl                  ;; Next Position HL (Width | Height)
    add (hl)                ;; A = Pos(A) + (Width | Height)(HL)
    dec hl                  ;; Previous Position HL (Pos X | Pos Y)
    sub d                   ;; A = Pos(A) + (Width | Height)(HL) - Limit
    Max(D)
limit_max_bucle:
    ret z                   ;; IF A = 0
    inc (hl)                ;; (HL)Pos = (HL)Pos++
    dec a                   ;; A = A--
    djnz limit_max_bucle   ;; IF B !=0
ret

```

Al igual que la función de colisiones, en el movimiento se comprueban ambos ejes. El objetivo de comprobar los ejes por separado es manejar los diferentes casos que te encuentres de forma separada y consiguiendo la solución optima a cada eje. Puede darse el caso al encontrarse el personaje en el límite de unos de los ejes y te se desplace de forma diagonal, eso provocaría un movimiento en un eje y en el otro no avanzaría por que se encontraría en el límite.

Para ello la función recibe la entidad a mover y la distancia que va ha desplazarse en el mapa. Primero comienza con el eje X, comprueba que el desplazamiento no se sale de los limites establecidos y luego prosigue con el eje Y. En el caso de la entidad vaya a sobrepasar el limite de cualquiera de los ejes, esta obtiene el valor límite del mapa.

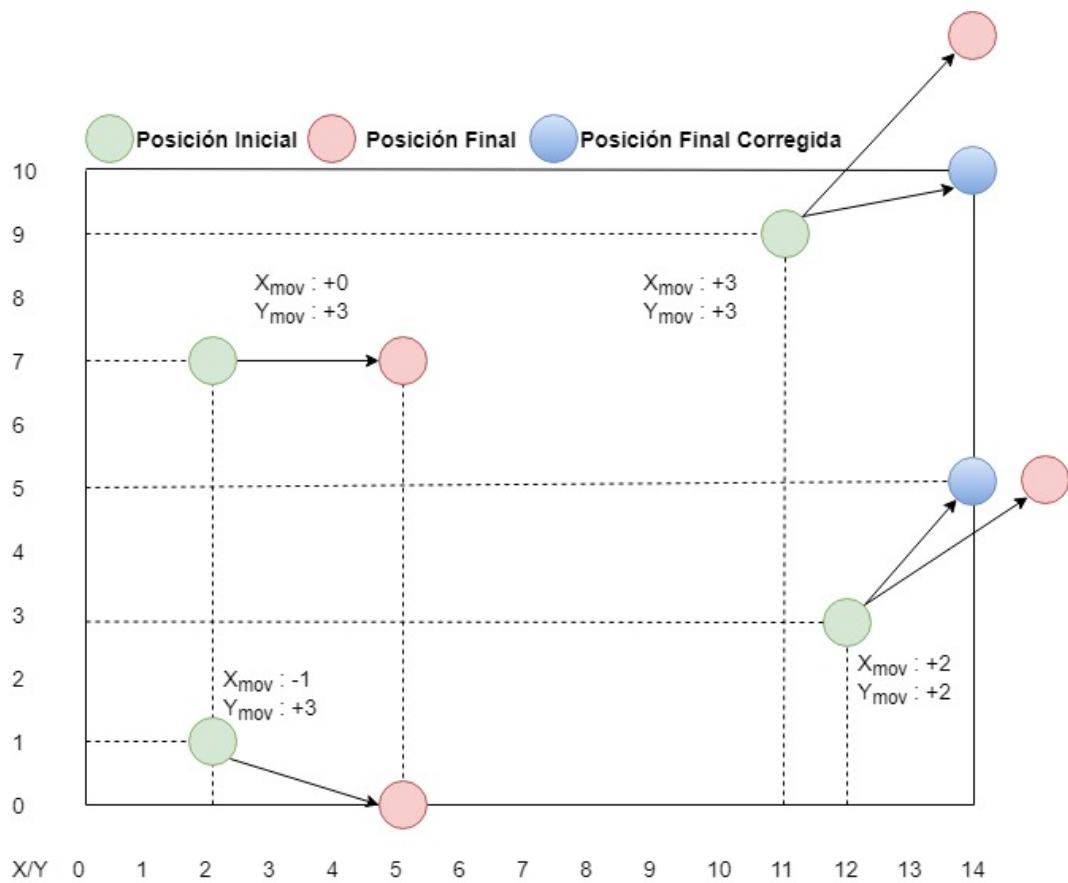


Figura 34 - Casos de Movimiento

Como se ve en la figura, si te mueves en el eje Y, al alcanzar el límite ese será tu valor, pues no está permitido avanzar más allá y se aplica las mismas normas al eje X.

El proyecto se halla inconcluso ya que antes de finalizar su desarrollo cumplió todos los objetivos por los que se llevó a cabo, siendo los objetivos del proyecto los siguientes puntos:

- Obtuve conocimientos básicos sobre la programación en ensamblador, pudiendo desarrollar las primeras estructuras de datos y la arquitectura de juego.
- Obtener funciones específicas (Movimiento y Colisión) para aplicarlas sobre el proyecto principal.

4.3. DISEÑO Y ESPECIFICACIÓN

4.3.1. ARGUMENTO

Tradams es una representación digital del propio Amstrad CPC que accede por primera vez a la red y se dispone a conocer este nuevo espacio sin conocer que este mundo se encuentra plagado de malvados virus, spyware, troyanos y demás malware que se encargaran de que esta nueva inmersión una nueva experiencia para esta vieja computadora. Por ello dispondrá de ti, el jugador para que le ayudes a obtener diferentes mejoras y actualizaciones que Tradams necesita para adaptarse y enfrentarse a estos nuevos peligros.

La web es inmensa y nunca sabes que puede depararte en sus zonas más oscuras siendo cada nivel diferente, cada habitación una incógnita y su disposición completamente. Podrás ayudar a Tradams a adaptarse a esta nueva aventura, solo está en tu mano.

4.3.2. CARACTERÍSTICAS DEL PROYECTO

El proyecto busca la creación de un juego de mazmorras cuya creación de niveles y habitaciones es procedimental. El juego también haría uso de diferentes variables para ir desbloqueando partes del juego como la cantidad de partidas finalizadas permite mayor cantidad de objetos y niveles nuevos donde derrotar más enemigos. También se mostrará el número de puntos conseguidos en cada partida a modo de reto para que el jugador conozca si ha estado mejorando o no.

4.3.3. PARTE PROCEDIMENTAL - PLANTEAMIENTO TEORICO

Un videojuego procedural es aquel que en algunos de sus apartados es generado en tiempo de ejecución siguiendo unas pautas marcadas por un algoritmo. Es bastante común recurrir a la aleatoriedad dentro de ese algoritmo para que los resultados del juego varíen de una partida a otra. Un juego procedural puede ser desde la variación de los enemigos de la partida hasta la composición y estructura de los niveles.

Lo normal es que un videojuego no sea 100 % procedural porque es difícil crear una buena experiencia de juego a partir de muchos algoritmos que generan contenido de forma

aleatoria. Por ello la parte procedural suele limitarse a unos pocos apartados, como por ejemplo la generación aleatoria de una mazmorra o el posicionamiento de los enemigos.

Las razones por las que se añaden elementos procedimentales a un videojuego pueden ser variadas, pero los objetivos principales suelen ser el aumento de la re-jugabilidad del juego y/o el ahorro de espacio en el disco duro.

En mi caso se busca la construcción procedural de los niveles partiendo de unas normas preestablecidas que deben cumplirse, estas normas están divididas en diferentes partes que cada una representa sobre la entidad/clase en la que se aplica. Todas las normas se deben cumplir a la hora de construir el nivel aunque pueden coexistir normas que modifiquen las pautas de otras entidades/clases.

Normas a nivel de Juego

- Cada dos niveles de juego se incrementará el daño de los enemigos
- Cada nivel se incrementará la cuantía máxima de enemigos siendo el límite el número hábil y gestionable por la máquina⁶.
- Cada 2 niveles será reducido el número de apariciones de ciertos objetos como: objetos de curación, monedas u objetos consumibles⁷ que puedan beneficiar al jugador sin poder alcanzar un porcentaje de cero.
- A partir del 6 nivel las habitaciones de actualización⁸ y de tienda desaparecerán del algoritmo de construcción.
- La sala de actualización no requerirá de llave solo en el primer nivel, el resto se hallarán cerradas.

Normas a nivel de Mapa

- Cada nivel contará con una **sala inicial** y una **sala de jefe Final**
- Cada nivel contará con una **sala de actualización** al menos que esta sea negada por normas del nivel de Juego.

Normas a nivel de Habitación

- Siempre ha de hallarse conectada a otra sala.

⁶ Debido a las limitaciones de la computadora debo establecer un límite dependiente del rendimiento de la máquina.

⁷ Los objetos de modificación del jugador no se hallan incluidos entre estos, es decir, aquellos que modifiquen las cualidades o los datos de combate del jugador.

⁸ Las habitaciones de actualización son aquellas salas donde el jugador pueden encontrar objetos que modifiquen sus datos o su forma de combate.

- Debe existir siempre la posibilidad de poder finalizarla

Con este conjunto inicial de normal comenzó el desarrollo

4.3.4. GÉNERO

- **Acción:** Lucha contra enemigos en diferentes situaciones
- **Aventura:** Se va avanzando por diferentes habitaciones y niveles en la que se cuenta una historia de forma oculta.
- **Rol:** El personaje obtiene diferentes objetos que mejoran sus estadísticas y dando la posibilidad de cambiar el Sprite del personaje.

4.3.5. AUDIENCIA OBJETIVO

Se busca atraer al público reducido de los amantes de los Amstrad y aquellos amantes de los juegos de mazmorras modernos como The binding of Issac o Enter the Gungeon.

4.3.6. ASPECTO GRÁFICO

Estilo **Pixel-Art**, con un total de 4 colores. El **Entorno** busca representar con una cantidad mínima de colores el ciberespacio, siendo este el entorno por el que se mueve el protagonista en busca de aventuras.

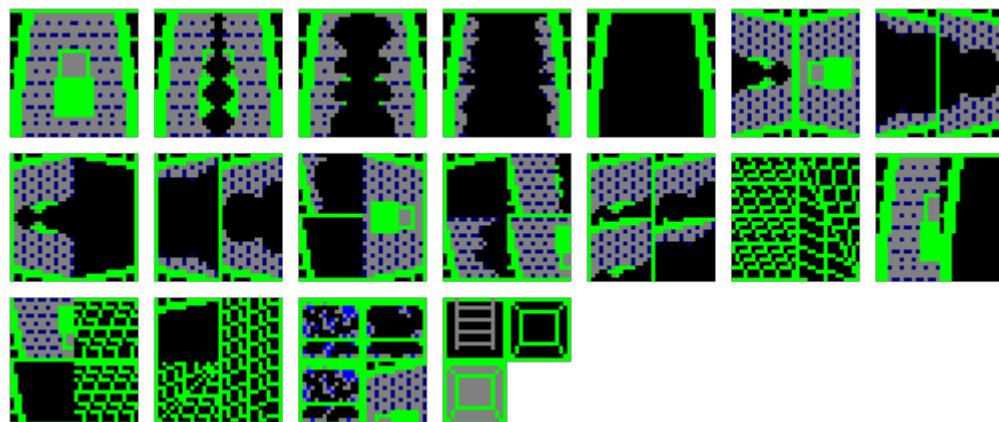


Figura 35 - Diferentes Tiles y Puertas del Juego

Enemigos: Representan todos los tipos de malware que se puede encontrar en la red (Troyanos, virus, spyware, ...).

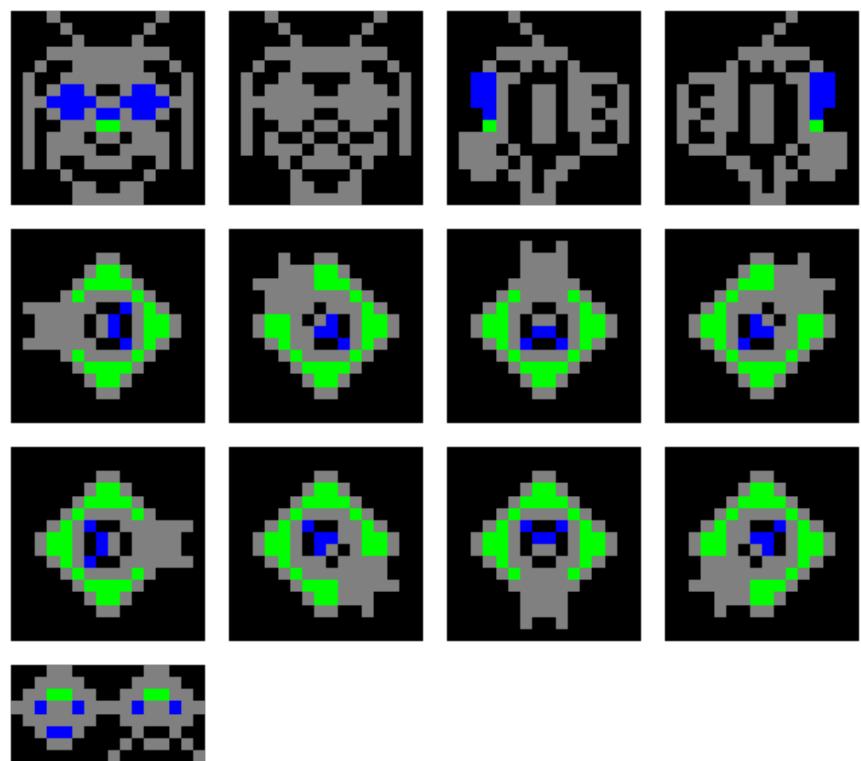


Figura 36 - Ejemplos de Enemigos

Objetos: Se tratan de periféricos y cheats aplicables al Amstrad real

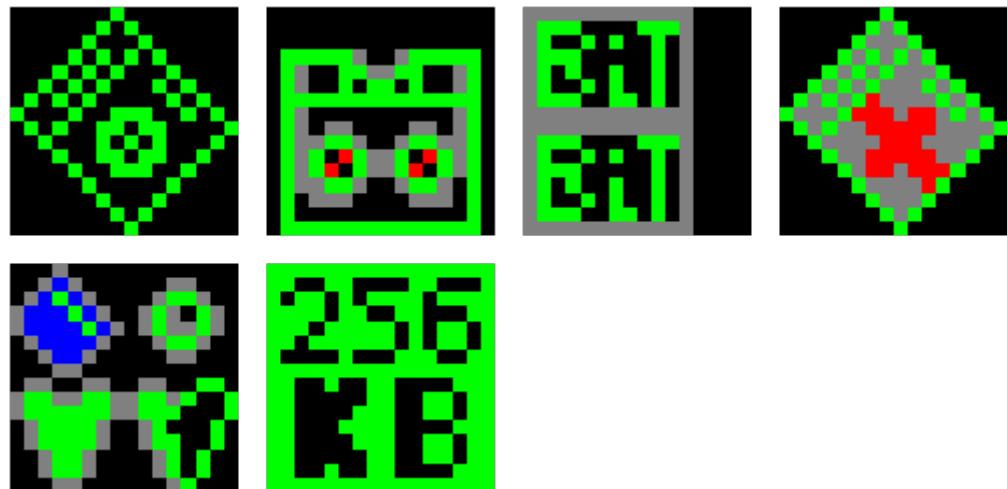


Figura 37 - Algunos objetos del Juego

Jugador: Representación de Amstrad CPC en el juego.

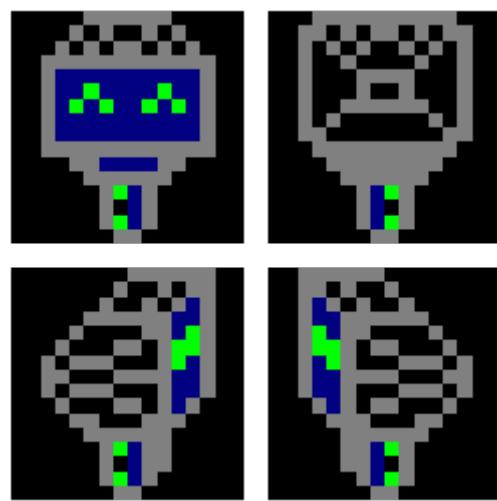


Figura 38- Sprite del Jugador

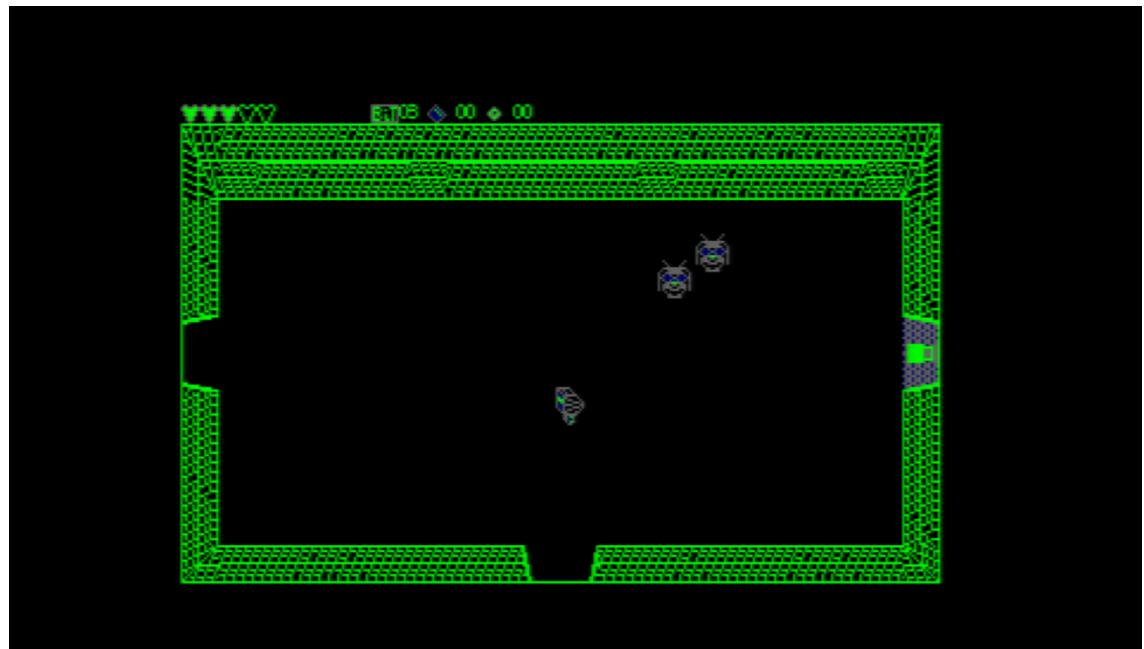


Figura 39 - Screenshot (1)

4.3.7. AUDIO

Efectos y Canciones de 8 bits aplicados con el programa Arkos Player para utilizar los canales de audio del Amstrad y generar canciones con diferentes instrumentos.

4.3.8. PERSONAJES

El personaje principal se trata de Tradams, que es la representación del Amstrad CPC 464 pero dentro del videojuego a raíz de las diferentes mejoras del protagonista puede cambiar a otros personajes los cuales son modelos más modernos al 464 entre ellos el 664, 6128 o los modelos plus.

4.3.9. ENEMIGOS

- **Robot:** Son bots que atacan de forma lineal.
- **Miners:** Son Enemigos basados en los marcianos de Mars Invaders que se encargan de dejar minar por el entorno.
- **Turret:** Torreta que se encarga de atacar constantemente al jugador albergándose en su sitio fijo.

4.3.10. ESTRUCTURA

Se trata de un juego por capas, cada capa está representada por un nivel de juego y a la vez cada nivel se halla compuesto por varias habitaciones. Cuanto más avance más capas se muestran hasta alcanzar el centro donde vislumbraras el juego al completo. Esta estructura tiene como objetivo que el jugador juegue reiteradamente y así desbloqueando más partes del juego.

Este tipo de estructura es empleada en algunos juegos procedimentales ya que como buscas que el jugador tenga que emplear tiempo en obtener la experiencia completa del juego pero que a la vez no se haga repetitivo, pueda tener diferentes experiencias y al obtener más capas del juego esas experiencias se amplíen haciendo atractivo desbloquear todo el contenido y cuanto más contenido más diferente cada experiencia.

4.3.11. JUGABILIDAD Y MECÁNICAS

En el juego debes superar cada una de las habitaciones, buscando las mejoras y utilizando diferentes objetos para vencer a los enemigos. Hay diferentes habitaciones y cada una de ellas tiene funcionalidades diferentes desde tiendas hasta habitaciones de actualización. Disponemos de objetos que nos permiten realizar acciones extra o nos aporta una mejora, estos pueden sustituirse por otros objetos que encuentres o puedes simplemente deshacerte de él.

Acciones genéricas (Enemigos y Jugador):

- **Desplazarse:** Las entidades pueden moverse en los ejes X e Y
- **Atacar:** Los ataques varían según la entidad siendo combate cuerpo a cuerpo, a distancia o mediante trampas en la habitación.

Acciones del jugador:

- **Comprar, soltar, recoger y utilizar objetos:** A lo largo de cada partida puedes encontrar diferentes objetos, pudiendo distinguirse en categorías:
 1. **Objetos Directos:** Objetos que al recogerlos se activan automáticamente como los corazones de vida.

2. **Consumibles:** Objetos que se añaden a tu inventario y podrás utilizar cuando tu deseas.
 3. **Objetos de Skills:** Objetos que modifican las habilidades del jugador.
 4. **Bits:** Moneda del juego
- **Abrir puertas:** A la hora de desplazarte por el juego encontraras puertas cerradas que se pueden abrir de dos formas distintas, utilizando una llave o superando las dificultades de la habitación.
 - **Presionar botones:** En el juego hay habitaciones que hay que presionar diferentes botones para superarlas y poder continuar.

4.3.12. OBJETOS

Son representaciones en el juego de todos los periféricos y mejoras de los computadores Amstrad representando las mejoras del personaje en el juego. Desde ampliaciones de memoria hasta lectores de casete. El jugador tiene unos valores que definen la efectividad de sus ataques, el rango de estos, cuantos puede lanzar y demás.

4.3.13. OBJETIVOS DEL JUEGO

El objetivo es conseguir pasarse todos los niveles con la ayuda de los objetos que te encuentres por el camino. En los niveles se hallarán enemigos que deberás vencer si quieres sobrevivir y al final de cada nivel espera un jefe final el cual te pondrá a prueba y vera si estás listo para continuar.

4.3.14. MENUS

Se desea desarrollar un menú principal que al perder la partida se vuelve, aparecerá las siguientes opciones:

- **Iniciar Partida:** Esta opción genera la carga del primer nivel
- **Sonido:** Activa/Desactiva el audio
- **Código:** Poner diferentes códigos para desbloquear partes del juego

4.4. DESARROLLO E IMPLEMENTACIÓN

4.4.1. ESTRUCTURA GENERAL

El proyecto divide los archivos en diferentes entidades con sus gestiones propias a modo de separación por clases, similar a programación orientada a objetos, pero en lenguajes de bajo nivel. Luego existen ciertos archivos que tiene funciones genéricas utilizadas por el resto de las

entidades, el separarlo facilita el uso por múltiples entidades, pero al ser genérico cada una de las entidades debe cumplir una estructura que queda definida por una macro común.

POO me resulta como ya que es el que utilizo a la hora de realizar los proyectos, lo que una estructura similar puede ayudarme a comprender y adaptar estos nuevos conocimientos.

4.4.2. PATRONES DE DISEÑO

Al tratarse de ensamblador no he podido aplicar los patrones de diseño típicos de lenguajes de alto nivel, pero si he intentado la forma de funcionar de estos mediante la utilización de macros a forma de *structs* y separando el dibujado, la actualización y las gestiones físicas en diferentes funciones para realizar una gestión limpia y a la hora de depurar facilita la separación de estas.

4.4.3. GENERACIÓN DE NIVELES

El conjunto de normas iniciales en este desarrollo se plantea en el apartado de diseño y especificación – parte procedimental. Con ese conjunto se mantenía una organización a la hora de incrementar la dificultad del juego, pero no determinaban como se tenía que llevar a cabo la construcción de los niveles, que esta construcción era viable y no realizar construcciones incompletas que llevaban al jugador a no poder completar el nivel por problemas estructurales y no por dificultad.

Por ello comencé con unas pocas normas que determinaban su construcción. Primero planteé el número de habitaciones distintas a construir por nivel y cuantas de cada una.

Versión 0.25 de la Construcción

Tabla 3 - Habitaciones v1

Nombre Hab.	Identificador	N.º por nivel	Función Resumida
Hab. Inicial	5	1	Inicio del Nivel
Hab. Objeto	2	1	Obtienes objeto de mejora
Hab. Tienda	3	1	Zona de compra de objetos
Hab. <u>jefe</u>	4	1	Jefe final del nivel
Hab. Conectora	1	5	Conexión entre habitaciones

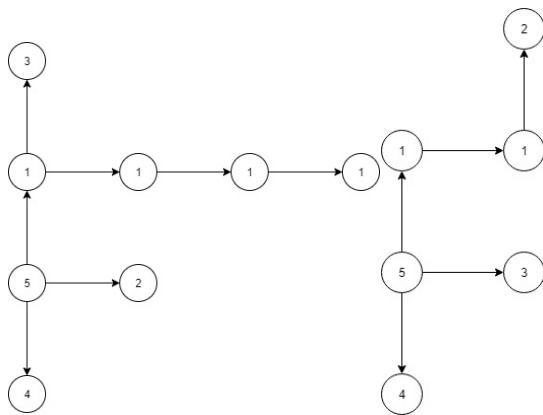


Figura 40 - Generación de Niveles v.0.25

Todas las habitaciones tenían que hallarse conectadas a una habitación conectora pues eran los nexos que continuarían la construcción, es decir, cada vez que se colocaba las habitaciones alrededor de una, una de ellas siempre tenía que ser una conectora pues sería el siguiente nodo de construcción.

Al ser un número tan reducido de habitaciones y de posibilidades la construcción siempre se llevaba a cabo, pero no era muy diferente de otras y a nivel de lógica era demasiado simple, por ello no conseguiría una experiencia diferente por partida que es el objetivo principal de este apartado.

Versión 0.5 de la Construcción

Por ello pensé en un conjunto mayor de habitaciones, con más funcionalidades y posibilidades de construcción, siendo esta.

Tabla 4 - Habitaciones v2

Nombre Hab.	Identificador	N.º por nivel	Función Resumida
Hab. Conector	1	20	Conexión entre habitaciones
Hab. Puzzle	2	1	Habitación con puzzle a resolver
Hab. Templo	3	1	Habitación con secreto
Hab. Supervivencia	4	1	Habitación de supervivencia
Hab. Oculta	5	1	Habitación oculta
Hab. Tienda	6	1	Habitación de compra

Hab. Apuestas	7	1	Habitación de apuesta
Hab. Objeto	8	1	Habitación con objeto de mejora
Hab. jefe	9	1	Habitación del jefe del nivel
Hab. Inicial	10	1	Habitación de inicio

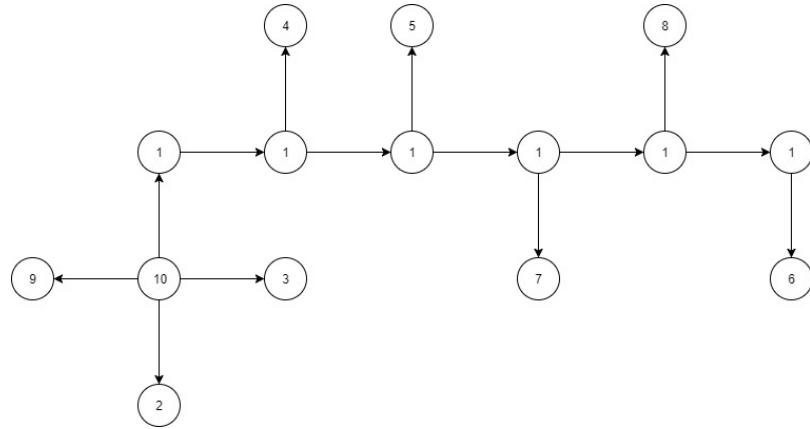


Figura 41 - Generación de un nivel v.0.5

Al crecer el número de habitaciones, creció la variedad, pero a la vez crecieron los problemas de construcción. A la hora de construir seleccionaba el habitáculo de forma aleatoria y en el caso de seleccionar una habitación agotada seleccionaba la siguiente que fuera plausible lo que provocaba pasillos largos ya que al colocar un conector se consideraba que la construcción debería continuar por ese nodo y si el primer elemento era un nodo conector continuaba por el mismo.

Eso también desembocaba que se agotaban las habitaciones de conexión y tenía que colocar el resto de las habitaciones, pero debido a la construcción y las normas no se permitía su inclusión y creaba niveles inacabados.

Había otra variante del error, los niveles están contenidos en una zona memoria a modo de matriz cuadrada y estas construcciones se quedaban bloqueadas al alcanzar las esquinas de la matriz, es decir, que ya no disponía de casillas donde construir y había conjuntos de habitaciones sin situar dentro del nivel.

Versión 0.75 de la Construcción

Por ello desarrolle nuevas normas y modifique las anteriores creando una construcción en diferentes etapas y cada una con un objetivo específico, poder manejar cada parte por separado y así gestionar sus errores de forma más centrada.

1. **Inicialización de los valores de construcción:** En este se inicializaba el generador de números aleatorios (RNG) y se reinicia los valores de la matriz por si construyes sobre una versión de otro nivel.
2. **Seleccionar e incluir habitación inicial en la matriz:** Obtener una posición aleatoria de inicio e insertar la primera habitación.
3. **Valorar las posiciones de construcción respecto a la habitación nodo:** Valorar que zonas se pueden utilizar para añadir habitaciones (si ya se hallan otras habitaciones o los límites de la matriz).
4. **Obtener que habitaciones ocuparan los espacios obtenidos:** Obtengo de forma aleatoria los valores de las habitaciones y si está ya no está disponible obtengo la siguiente a la misma.
5. **Sustituir una habitación por un nodo conector:** En esta versión las habitaciones de conexión ya no cuentan en el subconjunto de habitaciones aleatorias, ahora esta se insertará de forma sistemática entre uno de los huecos resultantes y la habitación sustituida volverá al subconjunto para ser utilizada en otra iteración
6. **Insertar los valores en la matriz:** Se insertan los valores en la matriz del nivel y se considera la habitación de conexión como siguiente nodo de construcción.
7. **Comprobar que hay habitaciones de construcción disponible:** Ahora se lleva un conteo del total de habitaciones por asignar y si este se halla en 0 se finalizará la construcción.
8. **Finalizar Construcción del Nivel:** Reinicia ciertos valores de construcción y activo los disparadores de finalización de construcción.

La ejecución se llevaba a cabo de esta manera:

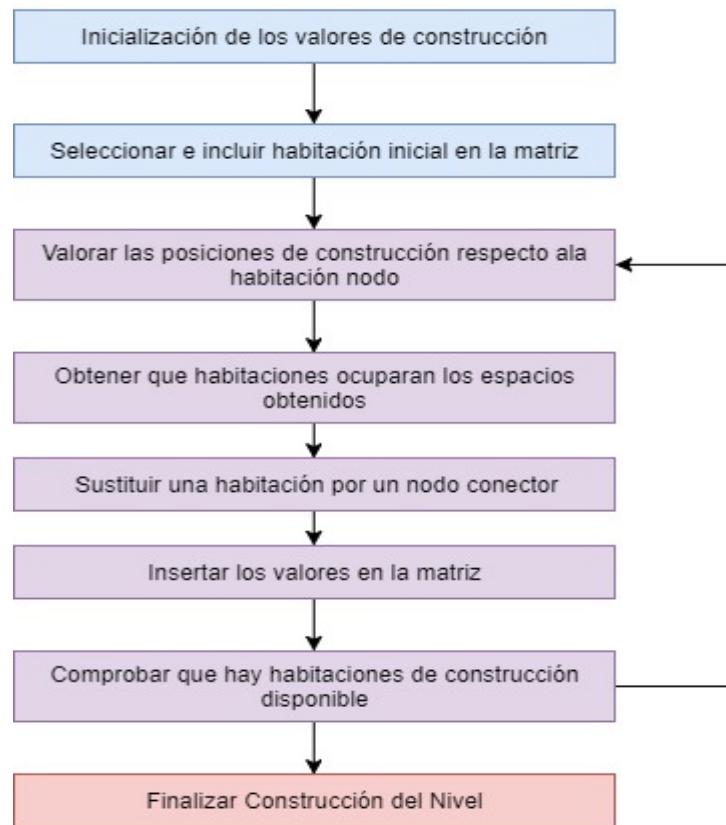


Figura 42 - Ciclo de Ejecución de la Generación del nivel v.0.75

Tabla 5 - Habitaciones v3

Nombre Hab.	Identificador	N.º por nivel	Función Resumida
Hab. Conector	1	Desconocido	Conexión entre habitaciones
Hab. Puzle	2	1	Habitación con puzzle a resolver
Hab. Templo	3	1	Habitación con secreto
Hab. Supervivencia	4	1	Habitación de supervivencia
Hab. Oculta	5	1	Habitación oculta
Hab. Tienda	6	1	Habitación de compra
Hab. Apuestas	7	1	Habitación de apuesta
Hab. Objeto	8	1	Habitación con objeto de mejora
Hab. jefe	9	1	Habitación del jefe del nivel

Hab. Inicial	10	1	Habitación de inicio
--------------	----	---	----------------------

En esta versión el sistema de conexión era simple, toda habitación adyacente se encuentra conectada, por lo que creaba un nivel que permitía acceder desde la habitación a una habitación oculta y al estar todo conectado no creaba un sistema de mazmorras muy prolífico a mi parecer, ya que si todo está conexo a ciertos objetos y ciertas dinámicas que pierden sentido como las llaves o el funcionamiento interno de algunas habitaciones

Por ello comencé a plantear un sistema de conexiones para que se generase a la vez que se construía el nivel.

Versión 0.90 de la Construcción

Añado un número mayor de habitaciones para cada tipo para que cada nivel mejore y las variaciones de un nivel a otro sean mayores. A nivel de habitaciones este sistema de tabla es el definitivo.

Tabla 6 - Habitaciones v4

Nombre Hab.	Identificador	N.º por nivel	Función Resumida
Hab. Conector	1	Desconocido	Conexión entre habitaciones
Hab. Puzle	2	2	Habitación con puzle a resolver
Hab. Templo	3	1	Habitación con secreto
Hab. Supervivencia	4	2	Habitación de supervivencia
Hab. Oculta	5	3	Habitación oculta
Hab. Tienda	6	1	Habitación de compra
Hab. Apuestas	7	1	Habitación de apuesta
Hab. Objeto	8	1	Habitación con objeto de mejora
Hab. jefe	9	1	Habitación del jefe del nivel
Hab. Inicial	10	1	Habitación de inicio

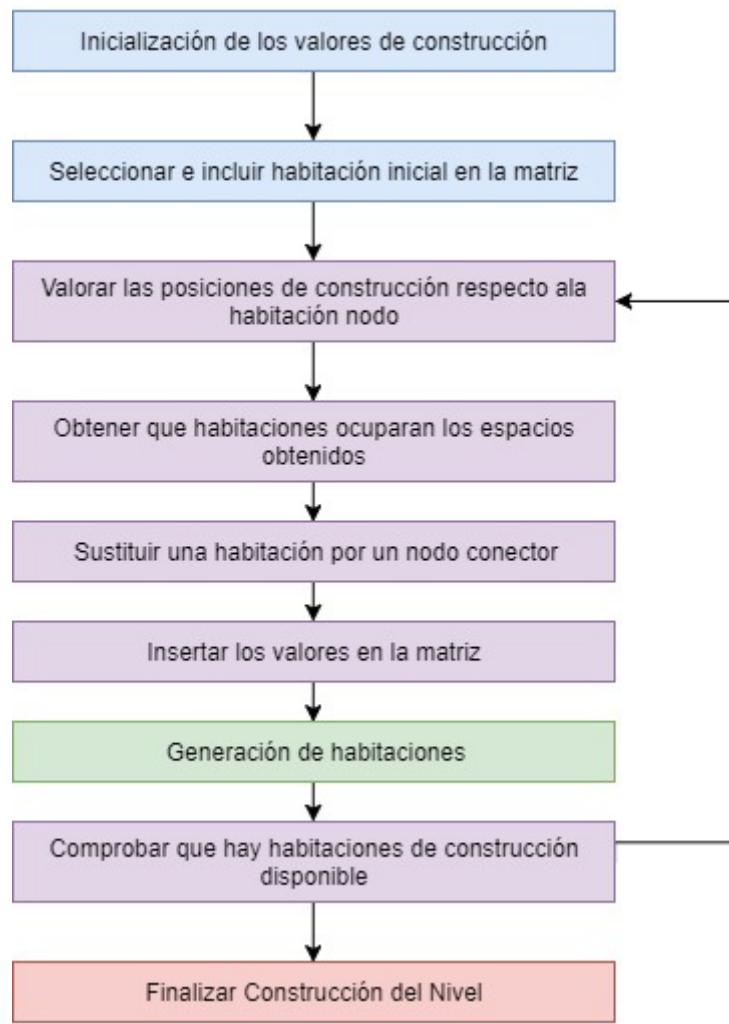


Figura 43 - Flujo de Ejecución de la generación de nivel

En esta versión se añade una nueva función a la construcción del nivel, el apartado de **Generación de Habitaciones**. Hasta ahora solo se representaba el identificador de las habitaciones en la interior de la matriz, pero no había más valores sobre cómo son o como están compuestas por ello tenía que añadir un sistema que gestionará la creación y edición de las habitaciones.

Cada vez que se genera una habitación los datos se añaden a un array donde se almacenan con el resto de las habitaciones, cada habitación está compuesta por los siguientes valores:

1. Índice de la habitación: Es la posición de la habitación dentro de la matriz

2. **Valores de Conexión de la habitación:** Este valor utiliza los primeros 4 bits para marcar que conexiones tiene la habitación, los 4 bits restantes cuales están abiertas y cuáles no.

Los primeros cuatro bits funcionan de esta manera, el primero representa la conexión inferior de la habitación, el segundo la superior, el tercer bit la derecha y el cuarto la izquierda. Siendo el segundo subconjunto de bits cuál de estas están abiertas.

Ejemplo: 0b11001000

Conexiones: 1100 (*Abajo: 1, Arriba: 1, Derecha: 0, Izquierda: 0*)

Puertas Abiertas: 1000 (*Abajo: 1, Arriba: 0, Derecha: 0, Izquierda: 0*)

3. **Identificador de Habitación:** Este byte es la suma del Identificador de la habitación más la habitación pre-generada de este identificador seleccionado. Dentro de cada categoría o tipo de habitación hay creadas varias habitaciones ya generadas para que cada nivel, aunque repita identificadores el interior de estas sea diferente.

Este sistema permite que a la hora de finalizar la generación del nivel puedo liberar la matriz y darle diferentes usos a ese conjunto de memoria para futuros cálculos o posibles gestiones de nivel.

Gestiones y diseños de habitación

A raíz del Identificador tenía que realizar un sistema que me permitiría almacenar muchos subconjuntos de habitaciones y que no ocuparan un gran espacio en memoria. Primero realice cálculos sobre cuánto podría ocupar almacenar cada uno de los tiles cada habitación utilizando un byte. A la hora de realizar los cálculos la pantalla del Amstrad en modo 1 es una resolución de 320x200.

Mis tiles son de 16x16 lo que nos da un total de 20 tiles de ancho y 12 de alto. Consumo en paredes 3 tiles de Alto y 2 de Ancho quedando un total de 18x9 tiles de 16x16

$$18 \times 9 = N^{\circ} \text{ Bytes} \times \text{Hab} = 162 \text{ Bytes}$$

Como he mencionado varias veces, uno de los principales objetivos de mi juego es que cada partida sea única al igual que cada nivel y utilizando este formato no podía tener muchas habitaciones diferentes ya que generando 32 Habitaciones se disparaba los cálculos.

$$162 \left(\frac{\text{Bytes}}{\text{Habitacion}} \right) \times 32(\text{Habitaciones}) = 5184 \text{ bytes} = 5 \text{ KBytes}$$

Y a mi parecer 32 Habitaciones son pocas, así que decidí realizar un sistema por secciones, siendo estas secciones rectangulares de 2x3 Tiles cada sección lo que optimizaba los

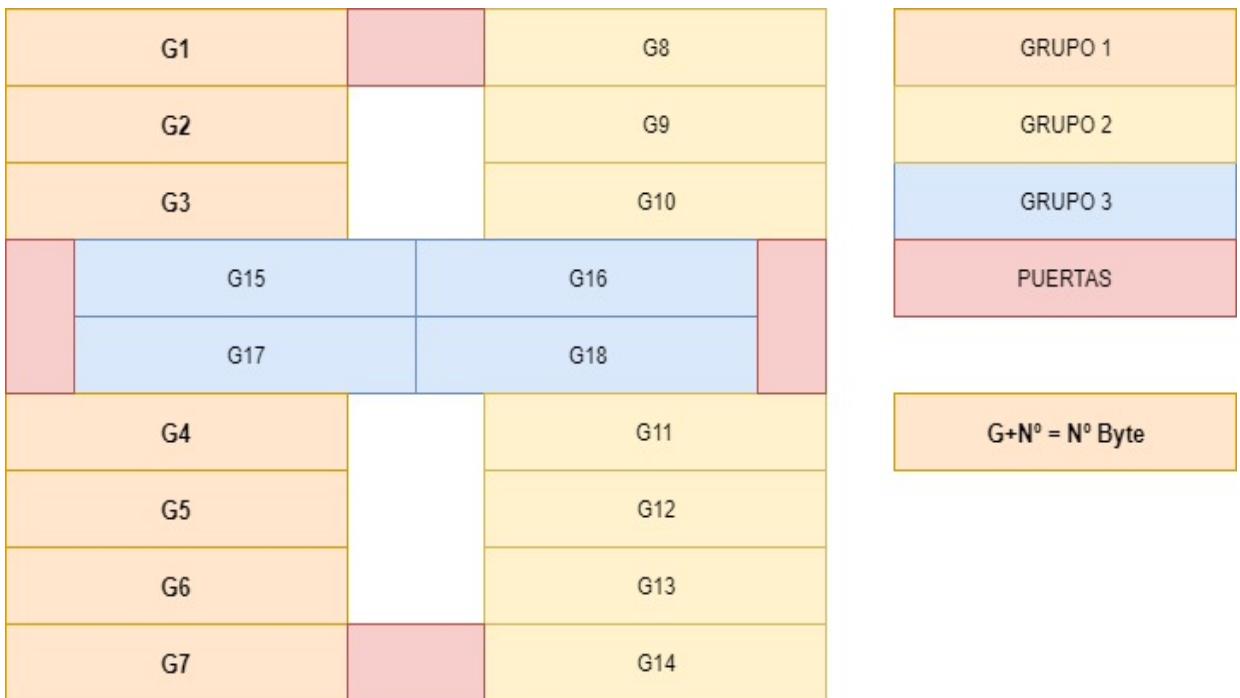


Figura 44 - Sistema de Interiores del Juego

resultados en menor consumo de memoria, pero complicaba la forma de dibujado en exceso o la forma de almacenar los valores de secciones se hacía compleja lo que podía mejorar en memoria, pero un mayor consumo en lógica computacional.

Por ello para conseguir una optimización en ambos campos desarrolle este sistema.

$$G + N \Rightarrow \text{Byte} \mid \text{En cada Byte se representan } 8 \text{ (0 = Vacío | 1 = Roca)}$$

$$18 \text{ Bytes} = 1 \text{ habitación} \quad (900\% \text{ de Mejora en Espacio})$$

De esta manera nos permite generar una gran cantidad de interiores de habitación, debido a las dinámicas y funciones de las habitaciones varían los cálculos de espacio, actualizando y ampliando la tabla de habitaciones a la siguiente:

Tabla 7 - Habitaciones v5

Nombre Hab.	Identificador	N.º por nivel	N.º de Interiores + Bytes	Función Resumida
Hab. Conector	1	Desconocido	128 2302B	Conexión entre habitaciones
Hab. Puzle	2	2	32 576B	Habitación con puzzle a resolver
Hab. Templo	3	1	8 144B	Habitación con secreto
Hab. Supervivencia	4	2	32 576B	Habitación de supervivencia
Hab. Oculta	5	3	8 144B	Habitación oculta
Hab. Tienda	6	1	1 18B	Habitación de compra
Hab. Apuestas	7	1	1 18B	Habitación de apuesta
Hab. Objeto	8	1	1 18B	Habitación con objeto de mejora
Hab. jefe	9	1	1 18B	Habitación del jefe del nivel
Hab. Inicial	10	1	1 18B	Habitación de inicio
Totales	10 hab. Distintas	13 habs. + habs. conect.	213 3832B = 3,75KB	

Así el juego podría tener una gran cantidad de habitaciones y cumpliendo parcialmente uno de los objetivos de este juego. Los cálculos aportados no son finales pues falta aportar el espacio que ocupa la propia lógica de lectura e interpretación de los interiores y la lógica añadida a diferentes habitaciones por su funcionamiento interno.

Tras una valoración final, aun teniendo la posibilidad de tener tantos tipos de interiores de habitación, valoré que no aportan mucho a la hora de la jugabilidad por lo que la cantidad se reduce.

Acceso al video de Tradams ([Video](#)) verificado 25/05/2018

4.4.4. HABITACIONES

Ahora tratamos la lógica que se encuentra detrás de las habitaciones. Las habitaciones pueden ser de diferentes tipos y cumplir diferentes funcionalidades para ello definí una macro que pudiera gestionar de forma sencilla esas variaciones.

Figura 45 - Macro Habitación

```
.macro definedRoom name
    name'_room:::
    name'_room_index:      .db #0xFF      ;; INDEX ROOM
    name'_room_connect:    .db #0x00      ;; CONNECTION VALUES
    name'_room_id:         .db #0x00      ;; ID ROOM P1
    name'_room_second_prt: .db #0x00      ;; ID ROOM P2
.endm
```

Room_index es su posición dentro de la matriz del nivel, **room_connect** las conexiones de la habitación, **room_id** el tipo de espacio que es y siendo **room_second_prt** si ya ha sido completada.

Navegación entre habitaciones

Las habitaciones solo tienen cuatro posiciones posibles respecto a otra (arriba, derecha, izquierda y abajo). En el **room_connect** como se explicaba en el punto de la generación de niveles se utilizan los primeros 4 bits (0bXXXX0000) para conocer estas conexiones y los 4 últimos bits para cuales están abiertas (0b0000XXXX).

La primera comprobación es si el **room_connect** contiene la conexión a esa habitación, la segunda comprobación también se lleva a cabo en **room_connect**, es si la puerta está abierta o no. Cuando se cumplen las comprobaciones el cambio se lleva a cabo, pero el **room_connect** es una representación de las conexiones que contiene la habitación actual pero la habitación desconoce el valor del **room_index** de las otras habitaciones.

Como comentábamos las habitaciones solo tienen cuatro posiciones posibles (arriba, derecha, izquierda y abajo) y todas las habitaciones están contenidas en una matriz. Desconocemos la posición de las demás habitaciones, pero conocemos en la que nos encontramos. En este caso se plantea como el problema matemático, para este problema

necesitamos tanto el valor actual de la habitación como el ancho y alto de la matriz para conocer los límites y conocer cuántos valores hay de diferencia entre una fila y otra.

Cuando trabajas directamente con la memoria, aprendes que las matrices es algo simulado mediante programación ya que la memoria es lineal. Ahora si queremos emular su funcionamiento necesitamos de su ancho y de su alto, en mi caso he utilizado matrices cuadradas para ahorrar una variable definición de esta.

En el proyecto se utiliza una matriz cuadrada de 8 de ancho, aunque para verlo de forma sencilla se mostrará como ejemplo una matriz 4x4 y como se representaría el desplazamiento dentro de una matriz.

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

Habitación actual = 6
Matriz 4x4

- 6 - (Subir) = 2 | -4
- 6 - (Bajar) = 10 | +4
- 6 - (Derecha) = 7 | +1
- 6 - (Izquierda) = 5 | -1

Figura 46 - Desplazamiento entre Habitaciones

Porque 4, porque si quieras bajar a la siguiente fila debes avanzar lo que vale una fila completa y de la misma manera en negativo para volver.

$$Hab_{ini} = 6$$

$$Hab_{final} = 10 \quad Hab_{act} = Hab_{ini} + Matriz_{fila} = 10 \quad Hab_{final}$$

Matriz 4x4

Ahora falta por mostrar cómo se dispara este cambio. El disparador se ejecuta cada vez que el jugador y tiene colisión con una puerta de la habitación, y hay tres posibilidades al tener una colisión.

1. La puerta está abierta. Se cambia de habitación
2. La puerta está cerrada, pero tengo llave. Se abre y se cambia de habitación.
3. La puerta está cerrada y no tengo llave. Se mantiene cerrada y no se cambia de habitación.

```

class Jugador;
class Puerta;
class Habitacion;
class Conexion;

int Colision_Jugador_Puertas (Jugador) {
    Puerta** puertas = pidoPuertas()
    Conexion* Conexion = pidoConexionHabitacion();
    int[] ValoresNavegacion = [ARRIBA, IZQUIERDA, DERECHA, ABAJO];

    comienzo del bucle, termina cuando no hay puertas, se le suma a i
        Si Conexión[i]->existe == falso {
            continua con la siguiente iteración;
        }
        Si Hay colisión(Jugador->posición, puertas[i]->posición) == no hay colisión {
            continua con la siguiente iteración;
        }
        Si Conexión[i]->abierta y Jugador->Tiene Llaves () {
            Abrir Puerta(Conexión[i])
            Quitar Llave(Jugador)
        }
        Cambiar habitación (Valores Navegación[i]) y volver;
    final del bucle;
    terminar con valor -1;
}

void Cambiar Habitación (valor navegación) {
    Habitación* actual = HabitacionActual ();
    Habitación** Habitaciones = DameTodasHabitaciones ();

    int ID Habitación = actual->ID + valor navegación;
    comienzo del bucle, termina cuando no quedan habitaciones, se le suma a i

        Si Habitaciones[i]->ID diferente ID Habitación {
            continua con la siguiente iteración;
        }
        actual = Habitaciones[i];
        Actualizar_habitacion ();

        terminar bucle;

    final del bucle;
}

```

Figura 47 - Seudocódigo Cambiar en Habitaciones y Colisiones con el jugador

La función “Actualizar_habitacion” intenta representar la ejecución del flag que se encargue de señalar que es una nueva habitación y que se debe dibujar.

Dibujado de las habitaciones

El dibujado se ejecuta una vez por habitación ya que solo dibuja los límites de esta y las puertas por encima. Se hace un único dibujado ya que al ser los límites no hay entidad que pase por encima y pueda sobrescribir esta parte de la pantalla.

Aun así, el dibujado se ha dividido en secciones para sea más rápido y contemple las zonas de dibujado como zonas reducidas de un tilemap. La división de estos dibujados solo por paredes es porque la parte interior de la habitación no contiene nada relevante que tenga que aportar en esta etapa y sería siempre los mismos valores siendo todos nulos.



Figura 48 - Construcciones de las Habitaciones

Como se puede ver en la figura superior, divido la construcción en 5 zonas (3 Horizontales y 2 Verticales) siendo estos los únicos que aportan parte gráfica a la habitación y facilitan el dibujado. Cree una función simple que recibe el largo de la construcción, el tilemap a dibujar y si esta horizontal o vertical el dibujado.

Tras realizar este dibujado continua la capa superior a las paredes que son las puertas, aquellas que se ponen por encima de estas, estas dependen del valor de conexión, pero como su posición siempre es la misma ya utilizan parámetros predeterminados de construcción y solo verifican si existe y si está abierta o no.

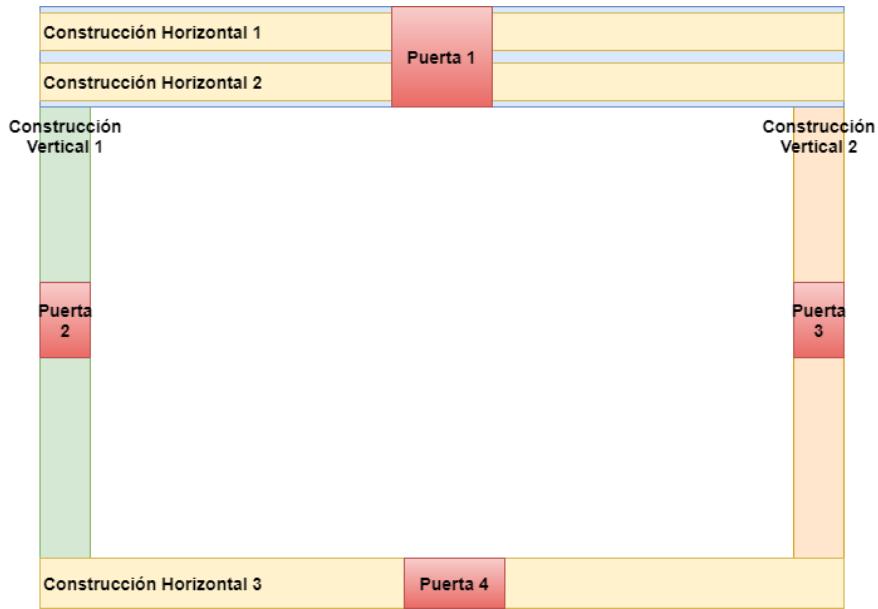


Figura 49 - Dibujado de la habitación junto con las puertas

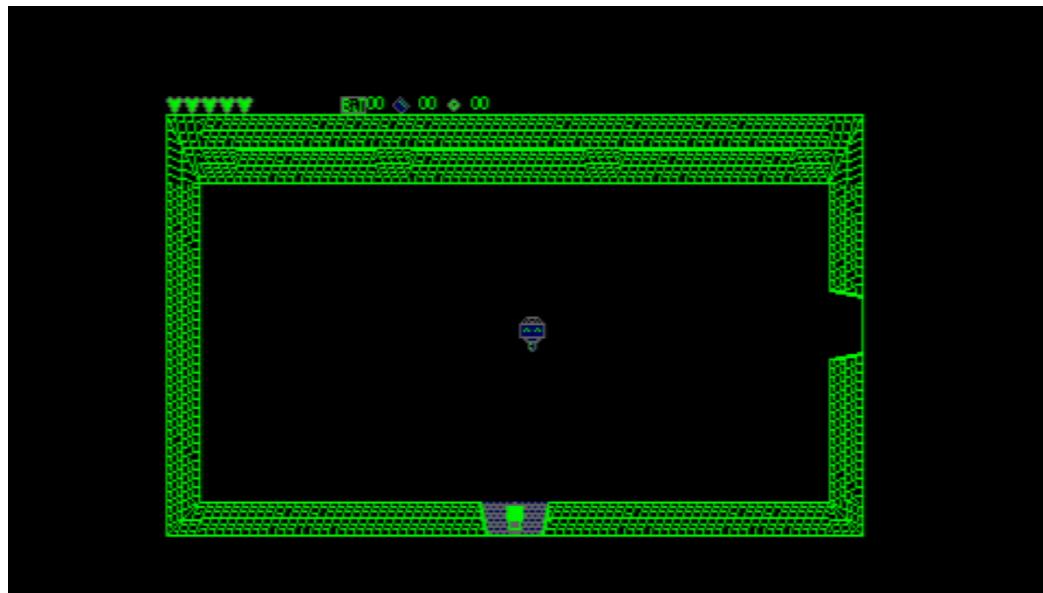


Figura 50 - Habitación Inicial

4.4.5. SISTEMA DE OBJETOS

Para obtener una variedad de experiencias diferentes de juego tenía que aportar algo más que la construcción procedural de nivel. Para ello creamos un cargador de objetos y un sistema de gestión de estos.

Los objetos tienen diferentes funciones, pero un objetivo común que el jugador tenga variedad y diferentes posibilidades dentro del juego. Estos pueden ser utilizados desde la apertura de puertas hasta la mejora de las aptitudes del jugador como la velocidad o el ataque.

La estructura de este proyecto me pareció cómoda y a lo largo del proyecto se aplicó sistemas similares o derivados del de objetos.

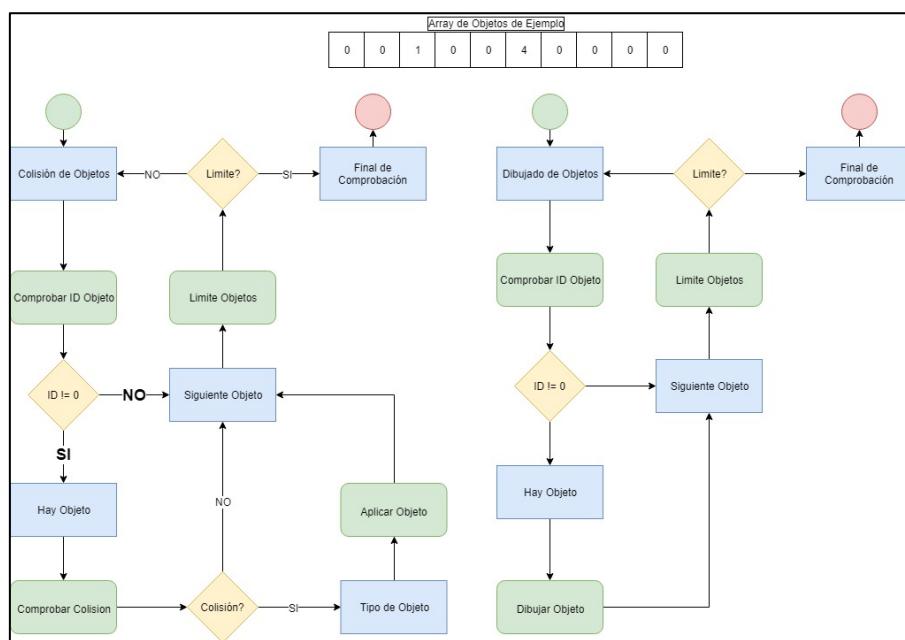
Versión 0.2 del Sistema de Objetos

En esta primera versión se crean 10 huecos en la programación siendo estos la cantidad máxima de objetos a la vez en pantalla por habitación.

Estos huecos se crean en dos partes diferentes, una la zona de posiciones donde se almacenarán las posiciones de los objetos en pantalla, su ancho y su alto. Y la segunda parte solo albergará la ID de los mismos.

Este sistema busca que solo se dibujen o se comprueben las colisiones de los objetos con ID diferentes de 0 siendo 0 que no hay objeto. El sistema trabaja con estos 10 huecos de forma similar a un Array comprobando de forma lineal que objeto a dibujar y con cuales puedo colisionar.

Figura 51 - Diagrama del Sistema



Este se trata de la primera versión del flujo del sistema de objetos. En esta versión aún no se hallaba definida la parte de gestión y acción de los objetos por lo que se dibujaba un objeto de prueba con su colisión, pero sin ningún tipo de acción.

Esta versión planteaba algunas fallas como el dibujado continuo de los objetos, aunque estos no se hubieran modificado en pantalla lo que provocaba perdida de ciclos de dibujado y sin estar planteado el sistema de acción era un sistema incompleto y no funcional.

Versión 0.6 del Sistema de Objetos

Se añadió las gestiones para que se realizará un dibujado único de los objetos y así no perder ciclos en cada iteración del juego y a la vez se le añade un sistema cercano a un cargador de objetos.

Uno de los grandes cambios que había respecto a los últimos *Dungeon Crawler* y los *Roguelike* es la cantidad de objetos que implementan y que aportan a su vez más aleatoriedad a cada partida. Por ello pensé en un sistema que añadiendo la información del objeto a una macro permitía usar ese objeto sin ningún tipo de implementación extra.

Para ello diseña una macro que contuviera toda la información respecto a un objeto y una variable que almacenará el número de objetos implementados. Esta macro contiene los siguientes valores de cada objeto:

- **ID:** Identificador del Objeto, este debe ser único entre todos los objetos.
- **Sprite:** Es la posición de memoria del grafico del objeto
- **Size:** Tamaño del objeto conteniendo tanto alto, como ancho.
- **Affects:** Hace referencia a que parte del jugador (Bolsa de Objetos o Habilidades)
- **Objective:** Busca tanto en la bolsa de objetos como en las habilidades cual va a ser modificada. Voy a listar los objetivos:

Tabla 8 - Tabla de Objetivos de los Objetos

Objetivo	Habilidades	Bolsa de Objetos
0	Velocidad de Movimiento X	Bits
1	Velocidad de Movimiento Y	Llaves
2	Vida del Jugador	Bombas
3	Vida Máxima	
4	Daño de Ataque	
5	Ratio de Disparo	
6	Velocidad de Ataque	

7	Rango de Ataque	
---	-----------------	--

- **Value:** El valor que se le añade a este.

Esta macro me facilita el trabajo a la hora de añadir objetos y aportar al juego tantos como se me ocurran.

Actualmente el juego base contiene un total de 15 Objetos diferentes con los que el jugador deberá utilizar a lo largo de la partida para mantenerse con vida.

Nombre	¿A que afecta?	Cuanto
Medio Corazón	Vida del Jugador	+ ½ Corazón
Corazón completo	Vida del Jugador	+ 1 Corazón
Bits	Moneda del Juego	+ 1 Bit
Llaves	Llaves del Juego	+ 1 Llave
Mina	Mina del Juego	+ 1 Mina
+ Vida Disk	Vida Máxima del Jugador	+1 Corazón de Vida Máxima
Espada	Daño de ataque	+1 en Daño
Ratio Fuego	Ratio de Ataque	-3 Ratio
Speed	Velocidad de Ataque	+1 Velocidad Ataque
Heal	Vida del Jugador	Cura toda la Vida
CD	Vida Máxima del Jugador	+2 Corazones de Vida Máxima
USB	Rango de Ataque	+2 Rango Ataque
Jostick	Movimiento del Jugador	+1/+2 Dependiendo del eje
Glitch	Movimiento del Jugador	+10 en Eje Y
Symbol A	Velocidad de Ataque	+2 Velocidad Ataque

Tabla 9 - Lista de Objetos Implementados

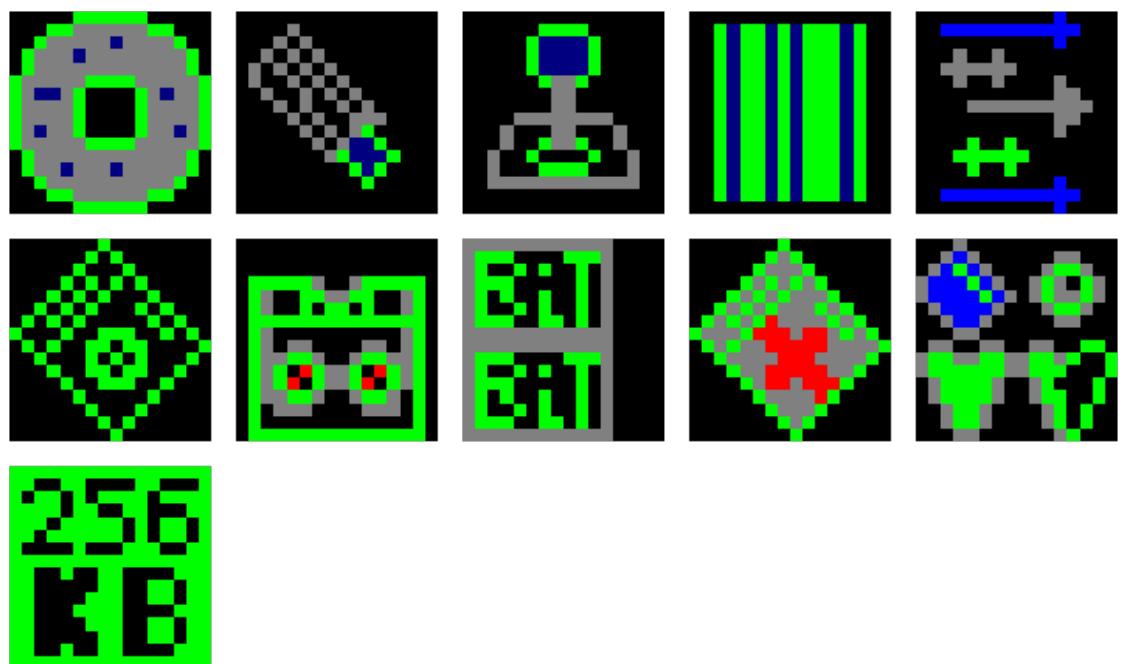


Figura 52 - Objetos Implementados en el juego

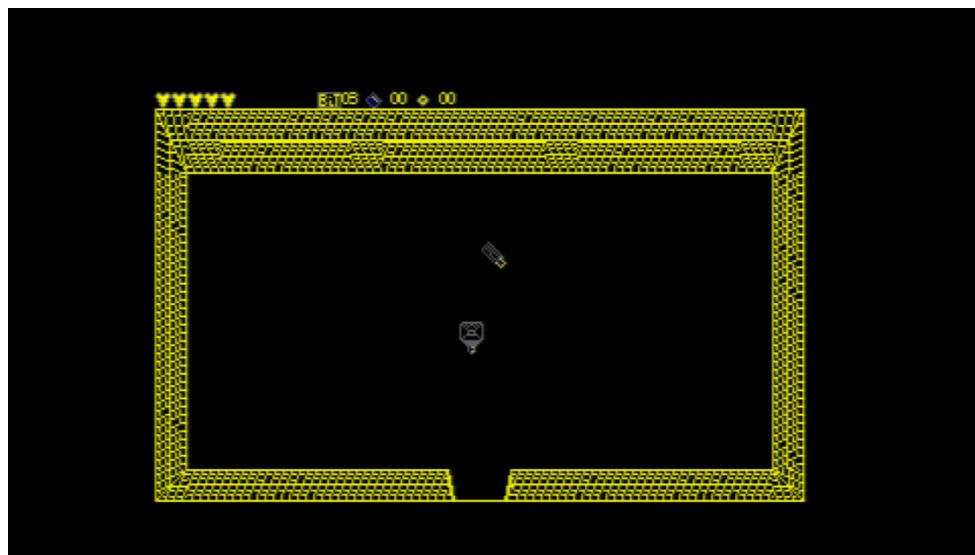


Figura 53 - Habitación con Objeto

Versión 1 del Sistema de Objetos

El sistema anterior cumplía todo lo necesario para poder gestionar y administrar los objetos de juego, pero en versiones posteriores que quería añadir una versión avanzada en la que se añadirían otro conjunto diferente de objetos que modificaría el juego de forma gráfica, pero por tiempos de desarrollo no ha podido ser desarrollado.

Funcionamiento del Sistema de Objetos

Colocar Objetos en Pantalla

Hasta ahora hemos tratado el desarrollo del sistema, que objetos contiene, como se disponen y el porqué de su disposición. Ahora vamos a tratar como estos objetos se convierten en mejoras para el jugador.

A este sistema se le añade una función que se encarga de colocar un objeto en pantalla, pero esta función requiere de ciertos valores y comprobaciones ya que por rendimiento está limitado el número de objetos en pantalla.

Esta función recibe dos bloques de datos, el primero la ID del objeto que quieras mostrar y el segundo son las coordenadas en los Ejes XY. Como mencionábamos en el párrafo anterior primero verifica si hay algún espacio donde colocar el objeto y si lo encuentra coloca la ID del objeto ya que servirá tanto como *flag* de Objeto activo como que objeto es. También contiene una última función ya que la posición que contiene el objeto también es su posición en el *array* y se utilizará para saber que macro de posición se le va a asignar.

Figura 54 - Macro Acción Objeto

```
.macro defineObjectAction name , id , sprite , size , affects , objective , value
    name'_object_action::
    name'_id:      .db id
    name'_sprite:   .dw sprite
    name'_size:     .dw size
    name'_affects: .db affects
    name'_objective: .db objective
    name'_value:    .db value
.endm
```

Esta es la macro que se encarga de recoger la información del objeto y de ella se extrae los valores del *size* ya que es necesario para situarlo en pantalla. Ahora con estos datos junto con los obtenidos con anterioridad a la hora de colocar el objeto en el *array*, obtenemos su macro de posición y le incrustamos tanto los datos iniciales del Eje X e Y como el tamaño del *sprite*.

Figura 55 - Seudocódigo Colocar Objeto

```
class Object;

void Colocar_Objeto (int ID, int [] posicion) {
    int pos_array = obtener_hueco_libre (ID);
    // Si devuelve -1 no hay hueco
    Si no hay hueco termino;
    Object* Objeto = obtener_objeto(ID);
```

```

    ObjPos* ObjetoPosicion = obtener_posicionObjeto (pos_array);
    ObjetoPosicion -> ancho = Objeto ->ancho;
    ObjetoPosicion -> alto = Objeto ->alto;
}

```

Dibujado de los Objetos

En el momento de dibujado la aplicación comprueba las 10 posiciones del *array* para ver si hay contenido algún valor diferente de 0 ya que lo contrario significa que hay un objeto activo. Al obtener el primer valor diferente lo utiliza para identificar el objeto y obtiene su macro de posición que se encuentra en la misma posición en otro *array*. En este punto tenemos tanto los valores que definen al objeto donde contiene el *sprite* y su posición que es donde dibujarlo, ya tenemos todos los parámetros necesarios para dibujarlo en pantalla.

Figura 56 - Seudocódigo Dibujar Objetos

```

class Objeto;
class ObjPos;

void dibujado_de_objetos () {

    Objeto** Objetos = DameObjetos();
    ObjPos** PosicionObjetos = DamePosicionObjetos();
    int[] ObjetosActivos = DameObjetosActivos();

    comienzo del bucle, termina cuando no quedan Objetos, se le suma a i

    Si ObjetosActivos == "No activo" {
        continua con la siguiente iteracion;
    }
    Objeto* objeto = Objetos.BuscaObjeto(ObjetosActivos[i]);
    Dibujar_objeto(objeto);

    final del bucle;
}

```

Colisión y Adjudicación de los Objetos

La última parte del sistema se encarga de la adjudicación de los nuevos parámetros al jugador y esto comienza desde el sistema de colisión. Este sistema funciona similar al sistema de dibujado, comprobando que objetos hay en pantalla el único cambio relevante es que su función final se trata de la función de colisión.

Esta función ya se encuentra sobre la memoria en la zona de proyectos antecedentes, siendo esta una función heredada de ese proyecto y su funcionalidad la misma.

Cuando al ejecutar el sistema de colisión da un caso positivo, el siguiente paso corre por parte del jugador. El jugador el contiene las funciones de asignación de valores. Se ejecuta la función que se encarga de la asignación mediante dos parámetros simples (Posición y Valor).

En puntos anteriores se trataba como se componía el sistema de carga de objetos y entre sus valores se trata del objetivo del objeto siendo este entre las habilidades del jugador y la bolsa de objetos. Como la composición de la bolsa es diferente a las habilidades se vio dividido en dos funciones que reciben los parámetros de Posición y valor, pero cada una se enfoca en un objetivo diferente.

La actualización de los objetivos es simple, busca en su contenedor el número de posición aportado por parte del objeto y le suma el valor. Al asignar el valor se coloca un valor a 0 y se limpia la entidad. Al completarse el ciclo de asignación del objeto al jugador el ciclo de vida del objeto en pantalla ha terminado y ahora debe eliminarse de pantalla y eliminarse de la zona de objetos activos.

Figura 57 - Seudocódigo Colisión de Objetos

```
class Jugador;
class Objeto;
class ObjPos;

void colision_de_objetos (Jugador) {
    Objeto** Objetos = DameObjetos();
    ObjPos** PosicionObjetos = DamePosicionObjetos();
    int[] ObjetosActivos = DameObjetosActivos();

    comienzo del bucle, termina cuando no quedan Objetos, se le suma a i

    Si ObjetosActivos == "No activo" {
        continua con la siguiente iteracion;
    }

    Objeto* objeto = Objetos.BuscaObjeto(ObjetosActivos[i]);

    Si Colision_objeto(PosicionObjetos[i], Jugador->posicion)) {
        continua con la siguiente iteracion;
    }
    Si Objeto->sitio == "BOLSA" {
        guardarObjetoBolsa(objeto);
    }
    Si Objeto->sitio == "HABILIDAD" {
        DarHabilidad(Objeto);
    }
}
```

```

        DesactivarObjeto();
        QuitarObjetoDeLaPantalla(Objeto);

    final del bucle;
}

```

4.4.6. ENEMIGOS IMPLEMENTADOS

- **Robot:** Son *bots* que atacan de forma lineal, funcionan utilizando una máquina de estados. Los estados que utiliza definen el eje y el sentido en el que debe moverse. Una vez por iteración de juego valora si un valor de sus ejes es similar al del jugador y en caso positivo actualiza al estado que le permita acercarse al jugador.

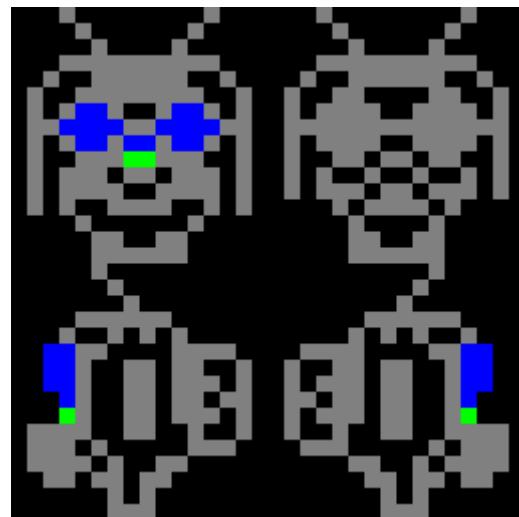


Figura 58 - Enemigo Robot

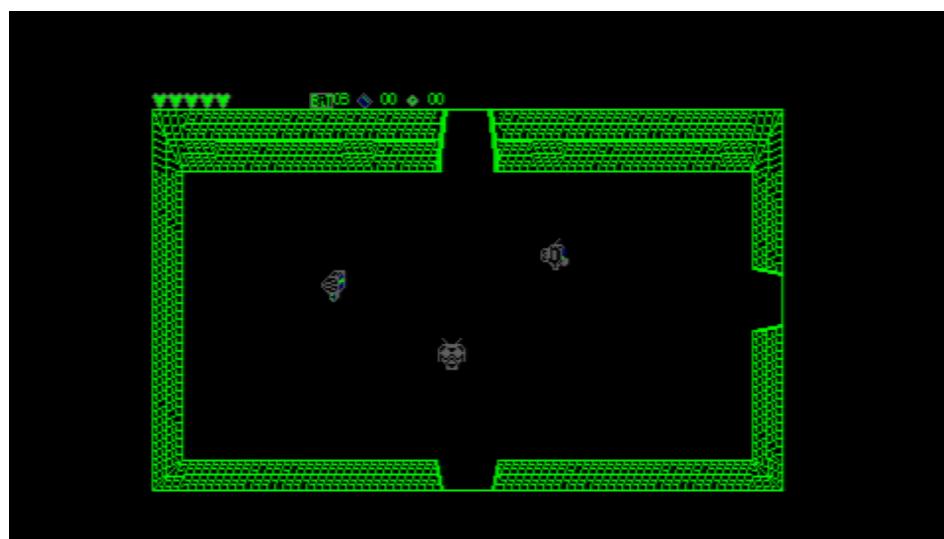


Figura 59 - Habitación con Enemigos

Los enemigos siguen unos patrones de diseño para que a la hora de implementar más tipos tenga una base en la que gestionarse. Todo enemigo está compuesto de tres macros que definen su base en el juego.

La primera macro contiene datos que nos permiten a la hora de actualizar estado si el enemigo continuó vivo o ya no hay que realizar gestiones sobre él, el *sprite* que se va a utilizar a la hora de dibujarlo en pantalla, tipo de enemigo y estado ya que todos se implementaran sobre diferentes máquinas de estados

Figura 60 - Macro Enemigo

```
.macro defineEnemy name
    name'_enemy::
    name'_active: .db #0      ;;(1)
    name'_sprite: .dw #0x0000  ;;(2-3)
    name'_type:  .db #0      ;;(4)
    name'_state: .db #0      ;;(5)
.endm
```

Esta segunda macro define las habilidades del enemigo (velocidad X, velocidad Y, vida inicial, vida máxima, daño, velocidad de ataque, rango de ataque y ratio entre ataques).

Figura 61 - Macro Habilidad Entidad

```
.macro defineEntitySkills name , spX , spY , life , MXlife , Damage , Speed , Range , Fire_rate
    name'_skills::
    name'_x_speed: .db spX
    name'_y_speed: .db spY
    name'_life:: .db life
    name'_MXlife: .db MXlife
    name'_damage: .db Damage
    name'_fire_rate:.db Fire_rate
    name'_bullet_sp:.db Speed
    name'_bullet_rg:.db Range
.endm
```

La última macro es común a la mayoría de las entidades, es necesaria para definir su posición en pantalla tanto la actual, como la anterior y su ancho y alto como entidad.

```
.macro defineEntityPosition name , x , y , x_last , y_last , width , height
    name'_position::
        name'_position_x: .db x
        name'_width:       .db width
        name'_last_x:     .db x_last
        name'_position_y: .db y
        name'_height:     .db height
        name'_last_y:     .db y_last
.endm
```

Figura 62 - Macro Posición Entidad

Funcionalidades de los enemigos

El sistema de enemigos fue implantado a posteriori del de los objetos. Eso implica que muchas de las funcionalidades parten de la base del sistema de objetos, pero al tratarse de entidades más complejas su funcionamiento se altera para aportar el soporte que necesitan.

Colocar enemigos en pantalla

Al igual que el sistema de objetos, el sistema de los enemigos empieza desde la zona de colocación, pero siendo el número enemigos más reducido que el de los objetos ya que estos necesitan una actualización más costosa y consume más ciclos del procesador.

El sistema recoge hasta un total de 4 enemigos, pero se ha optimizado para que funcione al final con 3 para que su funcionamiento sea más fluido, aunque el sistema tiene una composición que permite que en futuras optimizaciones se pueda ampliar y añadir más enemigos de forma sencilla.

La cantidad varía según el nivel siendo el primer nivel uno solo y en los siguientes ampliéndose hasta alcanzar los 3 por habitación, para mostrar una progresión de dificultad cuanto más avanzas. A la hora de colocar los enemigos la mayor diferencia al sistema son los siguientes puntos:

- Los enemigos deben poder modificar sus parámetros, a diferencia de los objetos que son una representación de los originales y no se alteran, los enemigos deben poder morir, atacar y tener sus propias decisiones, estos cambios provocan que ya no pueden ser representaciones si no copias del original lo que provoca ciclos de CPU en realizar las copias iniciales de información.
- Los enemigos a su vez simple contiene una máquina de estados que le permite crear su patrón de juego, este punto se tratará más adelante.
- Su colocación tiene que ser aleatoria, aunque no es del todo aleatoria, se le da un subconjunto de valores entre los cuales son válidas sus apariciones.

Esta colocación es convocada desde la zona de las habitaciones que dependiendo de la habitación como se explicaba en su punto. Al comenzar este sistema verifica el nivel y si este es superior al nivel máximo permitido por la Aplicación y tras verificar la cantidad comienza la transmisión de información desde el original a las copias. Este sistema se planteó de esta manera para que los valores iniciales en el sistema de habilidades siempre fueran las mismas, es decir, las del original.

Figura 63 - Seudocódigo Colocar Enemigos

```
class Enemy;
class EnemyPos;

void Colocar_enemigos_en_pantalla (int nivel) {

    int numero_enemigos = el nivel si no Maximo del juego;

    Enemy* original = obtener_enemigo_original ();

    Enemy** enemigos = obtener.todos.enemigos ();

    EnemyPos ** Posiciones = obtener.todas.Positicones.Enemgiso ();

    comienzo del bucle, termina cuando no quedan enemigos, se le suma a i
        enemigos[i]->habilidades = original->habilidades;

        Obtener_posicion_pantalla (Posiciones[i], sitioParaColocar ());

        // sitioParaColocar es la encargada de verificar que los valores están
        dentro de unos parámetros definidos

    final del bucle;

}
```

Sistema de Actualización de los Enemigos

Los enemigos al estar provistos de diferentes acciones necesitan de un sistema de actualización para que sus acciones se vean reflejadas en cada iteración. Este sistema de forma resumida controla que el enemigo está muerto y entonces lo deshabilita o en caso contrario le manda a la IA que gestione su siguiente movimiento, pero esto se reflejará en futuros puntos.

Sistema de Dibujado de los Enemigos

Este sistema prácticamente es una copia del sistema de dibujado del sistema de dibujado de los objetos, el punto principal que cambia es que los enemigos son entidades más complejas y que se desplazan por pantalla así que en este sistema también se añade la parte que se encarga de eliminar la imagen residual de la posición anterior y el desplazamiento a la posición nueva.

Sistema de Colisión de los Enemigos

Este sistema al igual que el resto es una versión más compleja del sistema de los objetos, en el caso de los enemigos tiene dos sistemas de colisión ya que hay dos colisiones que pueden sufrir los enemigos y sus consecuencias son diferentes.

El primero de estos sistemas gestiona la colisión de los enemigos con el jugador, siendo este el método que tiene el enemigo de restar salud a este. Su forma de proceder es sencilla, de forma iterada hace una comparación de colisión con cada uno de los enemigos habilitados y en el caso positivo actualiza el estado del jugador y este al HUD para que se vea reflejado ese cambio.

El segundo de estos sistemas controla las múltiples colisiones que pueden sufrir los enemigos por parte de los ataques del jugador. El jugador puede alcanzar un nivel de habilidad que permita realizar múltiples disparos en muy poco tiempo y este sistema se encarga de verificar que colisiones hay y resolverlas.

Este sistema se encarga también de gestionar cuando un enemigo recibe el impacto del ataque del jugador. Este debe restar los puntos de vida al enemigo por lo que realiza algunas tareas de actualización de parámetros y también la desactivación del proyectil que lo colisionó ya que golpeo al enemigo y debe ser deshabilitado.

4.4.7. INTELIGENCIA ARTIFICIAL

El enemigo utilizado ha uso de un sistema muy básico de máquina de estados. Estados son comprobados por cada enemigo en cada iteración del sistema de juego para verificar si mantiene el mismo, accede al siguiente o se adapta y cambia al que más le puede beneficiar.

Estos estados definen fundamentalmente el eje de movimiento y la dirección en el mismo. Estos cambios de unos a otros deben cumplir ciertos requisitos para llevarse a cabo, hay dos condicionantes principales:

1. Tras cada actualización de las acciones del enemigo verifica si su posición anterior es similar a la actual, eso quiere decir que quiso llevar a cabo un movimiento, pero los límites de la pantalla no se lo permitieron, es quiere decir que debe actualizar su estado para poder seguir moviéndose por la habitación.
2. La segunda condición que puede provocar el cambio se lleva acabo cuando al realizar el movimiento, compara sus ejes X e Y con los del jugador para buscar si coinciden en alguno y en ese caso modifica su estado a aquel que le permite perseguir al jugador.

A continuación, mostramos el seudocódigo que permite el funcionamiento de esta máquina de estados:

Figura 64 - Seudocódigo Maquina de Estados

```

void Maquina_Estados_Enemigo (Enemigo) {

    Si Enemigo->estado == "MUERTO" volver

    EnemSkills* habilidades = Obtener_habilidades_Enemigo(Enemigo);
    EnemPosi* posicionEnemigo = Obtener_Posicion_Enemigo(Enemigo);

    int[] Estados = [Derecha, Arriba, Izquierda, Abajo];
    int[] EjeMovimiento = [X, Y, X, Y];
    int[] CuantoSeMueve = [1, -1, -1, 1];
    comienzo del bucle, termina cuando no quedan Estados, se le suma a i
    Si Estados[i] == Enemigo->estado {
        Mover_en_eje(EjeMovimiento[i],habilidades->Velocidad*CuentoSeMueve[i],posicionEnemigo);
    }
    final del bucle;
    Si posicionEnemigo->anterior = posicionEnemigo->actual
        siguiente_estado(Enemigo);

    int estado = valor_un_eje_con_jugador(posicionEnemigo);
    // Si estado es mayor que -1, el jugador y el enemigo coinciden en un eje
    Si estado diferente de -1
        Enemigo->estado = estado;
}

```

Aplique una máquina de estados ya que las interacciones que tienen que tomar los enemigos no son complejas y se hallan en una habitación cerrada lo que juega a favor de los enemigos. Por ello aplique el sistema más simple que conozco para una IA, a la par que no ocupa mucho espacio. También este sistema se puede modificar para que esos cuatro estados sean diferentes dependiendo del enemigo, a si en un futuro podría modificar esta IA y adaptarla a más tipos de enemigos.

4.4.8. JUGADOR

El jugador es la entidad con las funcionalidades y la estructura más parecida al utilizado en el sistema del ABP, un sistema de captura del teclado que manda el valor al jugador y este lo interpreta.

Su sistema comienza desde el sistema de interrupciones, este sistema se ejecuta durante el tiempo de dibujado pero en el caso de este proyecto queda únicamente relegado a la detección de las teclas, aunque este sistema tiene una particularidad, su velocidad 6 veces más rápido que sistema de dibujado del monitor, eso provoca que 5 ocasiones no realizo comprobación de las teclas, es en la sexta en la que realice una limpieza de eventos de teclado para poner todas las teclas a 0 y luego compruebo cuales aún se mantienen activas.

Estos cambios realizados en la parte de las interrupciones son transmitidos en la variable llamada **actual_action**, se trata de un entero que representa la tecla presionada. En mi caso, utilizo un subconjunto reducido de teclas que cumplen las funciones del jugador así que este valor es una definición personal y no un valor por parte del sistema.

```
class teclas;

int interruption_function(){
    Si numero_interrupciones < 6 {
        numero_interrupciones++;
        vuelve;
    }

    limpiar_teclado ();
    int accion = -1;
    teclas** teclas_juego = obtener_teclas_a_comprobar ();

    comienzo del bucle, termina cuando no quedan teclas, se le suma a i
    Si comprobar_tecla(teclas_juego[i]) = true {
        accion = i;
        termina bucle
    }
    final del bucle;

    volver accion;
}
```

Esta variable alcanza hasta el sistema de actualización del jugador, en este sistema la variable **actual_action** se traduce en acciones en pantalla, estas acciones están recogidas en dos zonas bien diferenciadas:

- El sistema de movimiento
- El sistema de combate

En este apartado trataremos el sistema de movimiento, ya que el sistema de combate es un sistema básico de preparación para enviar la información al sistema de gestión de ataques siendo este un sistema externo del jugador.

El sistema de movimiento

Este sistema se encarga de gestionar las direcciones de movimiento del jugador, es un sistema sencillo por sus modificaciones y a la vez complejo por sus comprobaciones. Con esto hago referencia a las múltiples comprobaciones y actualizaciones que tiene que hacer este sistema, en casos de lenguajes de alto nivel esta complejidad queda relegada por el propio lenguaje mientras que en lenguajes de bajo nivel hay que administrar bien la información.

Este sistema primero traduce la variable de acción en la dirección en la que el jugador desea desplazarse tal cual lo traduce comprueba si el jugador se encontraba en la iteración anterior observando en la misma dirección para conocer si el *sprite* de este debe modificarse por el que se encara en esa dirección.

Ahora esta acción tiene que traducirse en movimiento en un eje, para conocer cuánto debe desplazarse utilizamos la misma macro que utiliza los enemigos de habilidades ya que se encuentre estas la velocidad en cada uno de los ejes. Siempre se gestiona la velocidad de cada eje ya que el ancho de pantalla se traduce en bytes mientras que el alto en pixeles eso conlleva que el ancho depende del modo del Amstrad y se perciba en un tamaño menor al alto (se puede gestionar mediante diferentes técnicas para realizar un desplazamiento horizontal más fluido a través de los bytes) en mi caso el jugador salta de un byte al siguiente por ello en el eje X considero una velocidad menor a la del eje Y

Esa velocidad es solo la primera parte del movimiento ya que dependiendo de la dirección esta será negada o no. Ahora utilizo una segunda macro que contiene la posición del jugador para calcular cual será la siguiente posición del jugador, durante esta función se realizan varias comprobaciones ya que las entidades tienen limitadas las zonas de movimiento, por ejemplo, no se permite que anden por encima de los muros ya que se encargan de limitar visualmente el juego y perderían entonces su sentido.

```
class PlayerSkills;
class EntityPos;
class PlayerPos;

bool movimiento_del_jugador (int actual_action) {
    int visión_actual = obtener_vision();
    if (actual_action != visión_actual ) actualizar_sprite ();
    PlayerSkills* plyskill = obtener_habilidades_player();
    int[] movimiento = [
        actual_action % 2 != 0 ? 0 : plyskill->Xspeed ,
        actual_action % 2 != 0 ? plyskill->Yspeed : 0
    ];
    PlayerPos* plypos = obtener_posicion_player ();
    return Desplazar_entidad (plypos, movimiento);
}

bool Desplazar_entidad (EntityPos* entity, int[] movimiento) {
    int[] limites_superiores = obtener_limites_maximos ();
    int[] limites_inferiores = obtener_limites_inferiores ();
    for (size_t i = 0; i < movimiento.length ; i++) {
        if( entityPos[i] + movimiento[i] > limites_superiores[i] ||
```

```

        entityPos[i] + movimiento[i] < limites_inferiores[i])
        continue;
    entity[i] = movimiento[i] + entity[i];
}
}

```

Esta última función se trata de una función genérica a todas las entidades ya que todas hacen uso de las mismas macros para identificar su posición, así ahorrando espacio en memoria, aunque también conlleva que entidades más simples puedan ocupar más espacio al verse obligadas a utilizar un conjunto obligatorio de valores.

En esta función no se muestra, pero a la hora de realizar el movimiento también se comprueba junto a los datos del ancho y alto de la entidad para comprobar los límites superiores.

boolean movimiento = limiteSuperior > entidad_n + movimiento + entidad_{ancho-alto}

Dibujado del Jugador

El sistema de dibujado por parte del jugador es sencillo, utiliza una variable que gestiona la dirección en la que mira el jugador y lo segundo la macro de posición donde se halla tanto la posición actual como la antigua. Su sistema se encarga de comprobar si cualquiera de estas dos ha sido modificada y ese caso selecciona el *sprite* que represente los nuevos valores, limpia la imagen anterior en pantalla, actualiza los valores de posición por los nuevos y dibuja el *sprite* de la dirección correspondiente.

4.4.9. SISTEMA DE ATAQUE

El sistema de ataque es aquel que se encarga de gestionar el conjunto de ataques que realiza el jugador. Parte de este sistema ya fue explicado en el punto de los enemigos, en ese punto se trató el sistema de colisión entre los enemigos y los ataques.

Tras haber tratado tanto el sistema de objetos como el sistema de enemigos, no hay mucho que aportar ya que los sistemas son idénticos y solo cambian las entidades que gestionan y la macro que contiene la información al respecto.

```
.macro defineEntityAttack name , Active, Speed , life , Direction , Damage
    attack_`name'::
    attack_`name'_active:      .db Active;; Active = 1 , Not Active = 0
    attack_`name'_speed:       .db Speed ;; Bullet Speed
    attack_`name'_life:        .db life  ;; life
    attack_`name'_direction:   .db Direction;; Direction Like Vision****

```

```
attack_name_damage:           .db Damage;; Damage
.endm
```

Figura 65 - Macro Entidad Ataque

Esta macro contiene los valores que definen el ataque del jugador, varios de los valores son sustraídos de las habilidades del jugador (Velocidad de Ataque, Vida y Daño). El parámetro **vida** es la interpretación del valor rango del jugador que se gestiona como el número de ciclos que le queda de vida al ataque.

La **dirección** se obtiene del parámetro visión del jugador y el primer parámetro (**activo**) es similar al resto de sistema, si el ataque se encuentra activo o no.

El gráfico del ataque es tan pequeño que el sistema permite 30 copias de este funcionando en pantalla, aunque después de los 20 el sistema comienza a resentirse debido al número de ciclos que consume las múltiples actualizaciones.

Este sistema comenzó intentando replicar un sistema de **display list** en un principio, ya que a lo largo del proyecto quería aplicar diferentes estilos, arquitecturas y estilos al código. Esto me daría diferentes perspectivas que aplicar en futuros proyectos, pero por problemas de tiempo, el sistema se modificó para que encajara como un sistema derivado del sistema de objetos, pero más parecido al de los enemigos ya que como comentaba este sistema necesita de una actualización algo que los objetos solo llevan a cabo cuando sufren la colisión con el jugador,

4.4.10. GESTION DE LA FUENTE DE TEXTO

Varios apartados del juego podrían haber utilizado los recursos que me aportaban tanto la maquina como el *framework* pero quería tener la experiencia y la problemática que supone esas diferentes gestiones y entre ellos el sistema de texto.

La fuente de un texto se trata del conjunto de *sprites* que se utilizan para representar el conjunto de símbolos de un lenguaje con estilo preciso. En mi caso no quería hacer uso del que contiene Amstrad, si no quería realizar la gestión por mí mismo para conocer que es necesario y que implica.

A la hora de hacer uso de las fuentes del Amstrad, la CPU hace cambio de los bancos de memoria situados desde 0x0000 hasta 0x3FFF por los de la ROM y mi juego se encuentra al completo en este banco de memoria lo que provocaría diferentes errores a la hora de hacer uso de las funciones de *String* y *Char* de CPCTelera lo que también fomento el desarrollo de una gestión propia.

También cabe destacar que no necesito todos los caracteres si no que haremos uso del abecedario solo en mayúsculas, en conjunto de números y tanto el signo de exclamación como el de interrogación, solo la parte del cierre. Esto me permite crear una fuente reducida y así la cantidad de memoria consumida en almacenar la fuente, es menor.

El sistema que conseguí no era el más óptimo, pero se adaptaba a mis necesidades y además no era un sistema complejo, lo que su lógica tampoco comportaba una gran cantidad de memoria. Nuestra fuente contendría: **ABCDEFGHIJKLMNPQRSTUVWXYZ0123456789.!?**

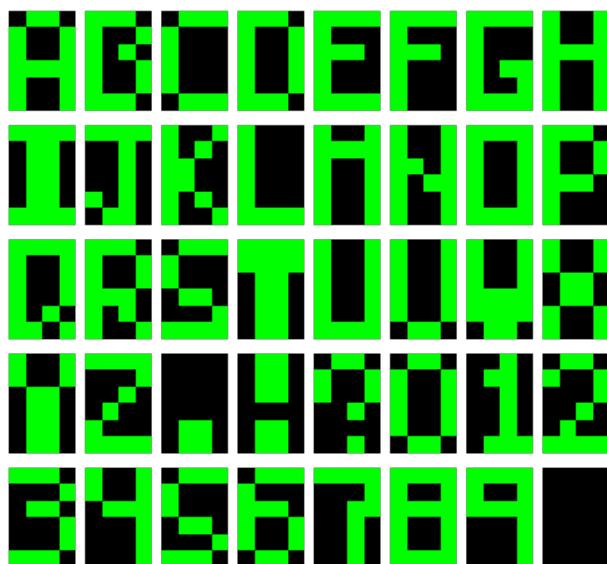


Figura 66 - Fuente del Juego

Composición de los textos

Ahora tras componer la fuente hay que escoger como hacer uso de esta. Al transformar la imagen a formato Hexadecimal, colocamos cada una de las letras en un *array* donde se encontraría toda la fuente. Cada símbolo de la fuente contará con un número que lo represente, es decir, A = 1, B = 2, C = 3 y así hasta el último símbolo. La cuenta comienza desde 1 ya que el 0 será la representación del espacio y el 99 será el final de línea.

;;	N	E	W	_	G	A	M	E	FN
New_Game:	.db 14, 5, 23, 0, 7, 1, 13, 5, 99								

Aquí tendrías un ejemplo de cómo se representaría una frase en nuestro sistema.

A la hora de gestionar el texto estático el funcionamiento es simple. Llamo a texto estático aquel que no se modifica ni se desplaza por la pantalla, pero y aquel que debe modificarse. En el caso de este juego lo que más puede requerir actualización respecto al texto, son los números mostrados en la interfaz del juego.

Este texto necesita de la conversión de un valor Decimal a un *String*, para ello hice uso de la división que, aunque el Amstrad no la tenga por Hardware se puede implementar mediante Software. En mi caso hice uso de una función publicada en CPCWiki sobre divisiones de 8 bits que te devuelve tanto el resultado como el resto de esta, justo los valores que necesitaba para representar la conversión.

4.4.11. MENUS

En un inicio quise basar el menú al sistema que recodaba que desarrollé en tercero, con la asignatura de Fundamentos de los videojuegos que mediante un *array* conocería la opción que elegía y la posición en el *array* donde se encontraba.

Por problemas de tiempo tuve que reducir el menú a un sistema simple que se encarga de visualmente mostrar cuando ha iniciado el juego o vuelves a él cuándo has perdido. Esta pantalla muestra dos *Sprites* grandes siendo una representación de Tradams más grande y con más detalles y el segundo el nombre del juego.

En la parte inferior de estos se muestra el mensaje “**Press any Key**”. Este mensaje quiere indicar la funcionalidad que encierra la pantalla. Al pulsar una tecla, leo que parte del teclado ha cambiado de valor y utilizo este disparador para construir el nivel y comenzar la partida.

A la hora de morir el jugador le devuelvo a esta pantalla ya que al ser lo primero que vio puede ser una muestra visual de que volvió al principio y puede comenzar una nueva partida solo pulsando cualquier tecla de nuevo.

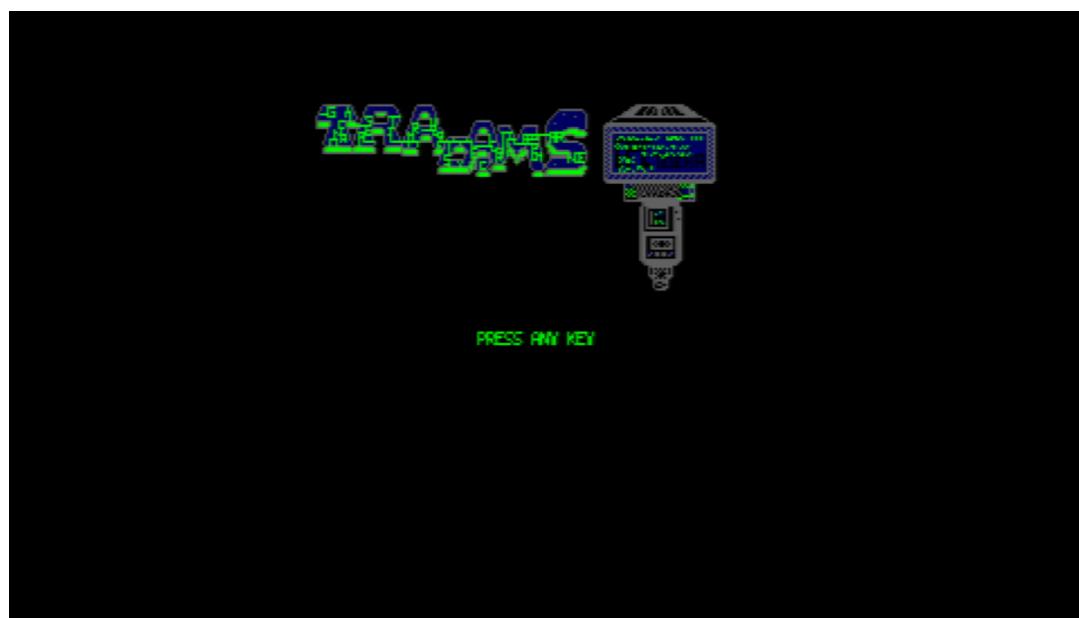


Figura 67 - Pantalla Inicial

4.4.12. GRAFICOS

Se tratan de gráficos 2D con cuatro colores distintos como se mostraba en la etapa de diseño y especificación, se han añadido más gráficos para mostrar nuevos objetos y creaciones dentro del juego.



Figura 68- Gráficos del Menú

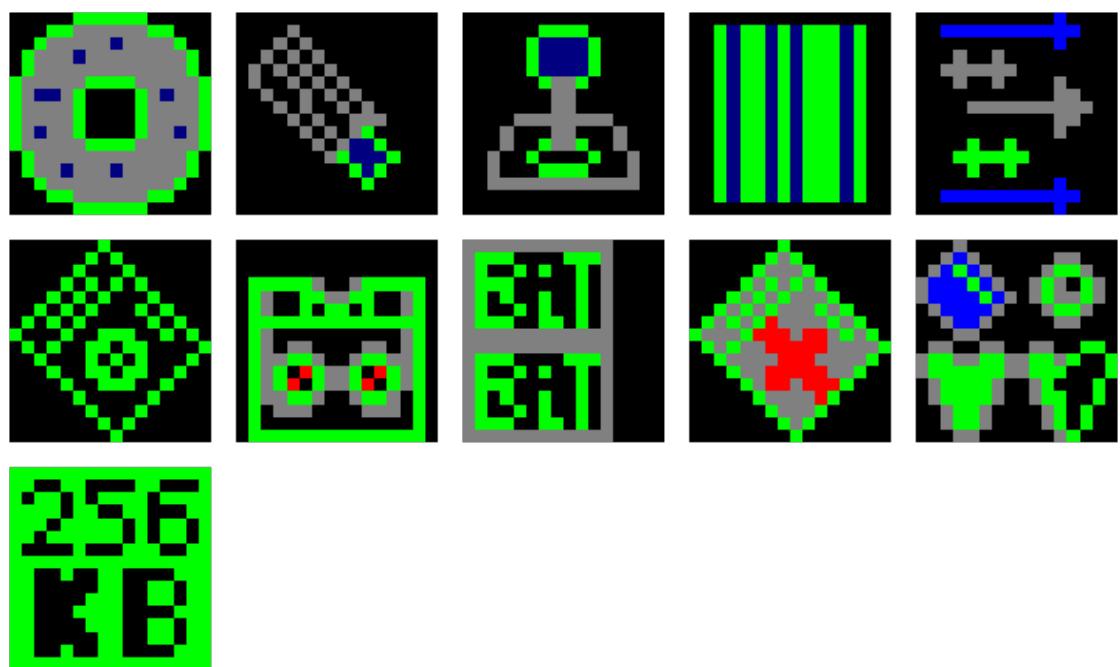


Figura 69 - Objetos de Juego

4.4.13. AUDIO

El audio por falta de tiempo no ha podido ser desarrollado ni se ha podido diseñar ninguna canción. Tuve que emplear el tiempo de este sistema y derivarlos a otros. Teóricamente el sistema derivaría del mostrado en el ABP durante la asignatura de Videojuegos 1.

Habría una parte del código en el sistema de interrupciones para ejecutar de continuo la música, pero a través de varias variables globales como las habitaciones se modificaría para adaptarse a la situación, como acceder a la habitación del jefe o una sala oculta en el nivel.

El otro canal de audio lo utilizaría para los efectos de juego como el ataque del jugador o cuando este recibe un impacto que se gestionaría de igual manera, una variable global sería modificada y habría una enumeración de todos los tipos de eventos posibles y cada uno con su sonido y a la hora de modificarlo ejecutaría cada uno de los ruidos en el momento indicado.

4.4.14. CONTROLES DEL JUEGO

- **W:** Movimiento hacia arriba
- **A:** Movimiento hacia la izquierda
- **D:** Movimiento hacia la derecha
- **S:** Movimiento hacia abajo
- **Barra Espaciadora:** Ataque del jugador

4.4.15. REQUISITOS NO FUNCIONALES

Los requisitos no funcionales son aquellas particularidades del sistema no específicas que tratan las cualidades más generales que tiene que llevar a cabo la aplicación. Cumplir estos requisitos supone buenas prácticas a la hora de realizar el proyecto

Distinguiremos los requisitos por su nivel de importancia siendo tres los niveles:

1. **Critico:** Forma parte del núcleo del juego y sin ellos el funcionamiento de este no es posible o errático.
2. **Aconsejable:** Es importante pero no necesario para el funcionamiento del juego, aunque aporta mucho a él mismo
3. **Nimio:** Es un requerimiento que baja importancia, aunque son los detalles que pueden aportar personalidad al programa.

A continuación muestran los requisitos no funcionales más importantes para el tipo de producto que se está tratando, es decir, un videojuego.

Titulo	Rendimiento
Importancia	Critico

Descripción	El juego tiene que ir fluido en todo momento siendo en un Amstrad los 50 FPS para no afectar a la experiencia de juego
--------------------	--

Tabla 10 - Requisitos no funcionales 1 - Rendimiento

Título	Usabilidad
Importancia	Critico
Descripción	El juego tiene que ser visualmente fácil de interpretar pensando en el público con menos experiencia como los jugadores nuevos.

Tabla 11 - Requisitos no funcionales 2 - Usabilidad

Título	Fiabilidad
Importancia	Critico
Descripción	Buscar que el juego no muestre errores que puedan truncar la experiencia del jugador pudiendo bloquear la partida o creando la imposibilidad de continuar.

Tabla 12 - Requisitos no funcionales 3 - Fiabilidad

Título	Apariencia
Importancia	Aconsejable
Descripción	Los juegos deben ser atractivos, buscando la inmersión del jugador dentro de este.

Tabla 13 - Requisitos no funcionales 4 - Apariencia

Título	Escalabilidad
Importancia	Aconsejable
Descripción	La arquitectura del videojuego debe realizarse de forma que sea fácil añadir más contenido al mismo.

Tabla 14 - Requisitos no funcionales 5 - Escalabilidad

4.4.16. REQUISITOS FUNCIONALES

Los requisitos funcionales referencian las diferentes funciones implementadas en el juego, es decir, todas las características que aporta el programa, pero debido a la cantidad que suelen gestionar los programas de videojuegos, estos se suelen definir de forma más genérica buscando abarcar varios de estos en el mismo requisito, pero sin perder el significado de estos.

Distinguiremos entre los mismos niveles de importancia que los requisitos no funcionales.

Titulo	Movimiento de las Entidades
Importancia	Critico
Descripción	Una función que se encargue del movimiento de las diferentes entidades del juego.

Tabla 15 - Requisito Funcional 1 - Movimiento de las Entidades

Titulo	Dibujado de las Entidades
Importancia	Critico
Descripción	Esta función se encarga del dibujado de las diferentes entidades dentro del juego

Tabla 16 - Requisito Funcional 2 -Dibujado de las Entidades

Titulo	Limpieza de las Entidades
Importancia	Critico
Descripción	Esta función se encarga de limpiar la posición anterior de la entidad.

Tabla 17 - Requisito Funcional 3 - Limpieza de las Entidades

Titulo	Generación de una habitación
Importancia	Critico
Descripción	Recoge las funciones encargadas de leer, cargar y dibujar una habitación.

Tabla 18 - Requisito Funcional 4 - Generación de una Habitación

Titulo	Generación de un nivel de forma procedimental
Importancia	Critico
Descripción	Se encarga de gestionar y crear niveles utilizando algoritmos para conseguir aleatoriedad y que cada nivel sea distinto. Siendo algoritmos sencillos de Amstrad la variedad no es grande.

Tabla 19 - Requisito Funcional 5 - Generación de un nivel Procedimental

Titulo	Cambio entre Habitaciones
Importancia	Critico
Descripción	Permite la navegación entre habitaciones siempre que estas no estén cerradas. Siendo el sistema de puertas que puedes encontrar en cada habitación.

Tabla 20 - Requisito Funcional 6 - Cambio entre habitaciones

Titulo	Ataque del Jugador
Importancia	Critico
Descripción	Se encarga de gestionar según las habilidades del jugador su ataque permitiendo que en el juego se pueda atacar.

Tabla 21 - Requisito Funcional 7 - Ataque del Jugador

Titulo	Cargador de Objetos
Importancia	Critico
Descripción	Sistema que se encarga en base unos parámetros gestionar los objetos permitiendo de forma sencilla añadir nuevos.

Tabla 22 - Requisito Funcional 8 - Cargador de Objetos

Titulo	Gestión de las Fuentes y el Texto
Importancia	Aconsejable

Descripción	Gestión de los textos mediante una fuente personalizada permitiendo crear textos sin utilizar las fuentes del Amstrad y así el juego puede mostrar mensajes por pantalla.
--------------------	---

Tabla 23 - Requisito Funcional 9 - Gestión de las Fuentes y el texto

Título	Dibujar Objetos
Importancia	Critico
Descripción	Gestión de dibujado por parte de los objetos.

Tabla 24 - Requisito Funcional 10 - Dibujado de los Objetos

Título	Máquina de Estados
Importancia	Critico
Descripción	Sistema de inteligencia Artificial aplicado sobre los enemigos.

Tabla 25 - Requisito Funcional 11 - Maquina de Estados

Título	Modificar los Atributos del Jugador
Importancia	Critico
Descripción	En el Juego al recoger un objeto debe permitir modificar los atributos del jugador.

Tabla 26 - Requisito Funcional 12 - Modificar Atributos del Jugador

Título	Interactuar con los objetos de la habitación
Importancia	Critico
Descripción	El juego debe permitir al jugador tanto recoger los objetos como abrir las puertas.

Tabla 27 - Requisito Funcional 13 - Interactuar con los objetos de la habitación

Titulo	Habitación del Tesoro
Importancia	Critico
Descripción	Crear y gestionar las habitaciones del tesoro, donde se encuentran los objetos que permiten las mejoras del jugador.

Tabla 28 - Requisito Funcional 14 - Habitación tesoro

Titulo	Sistema de Carga de Enemigos
Importancia	Critico
Descripción	Se encarga de cargar enemigos y colocarlos en posiciones aleatorias de la habitación.

Tabla 29 - Requisito Funcional 15 - Sistema de Carga de Enemigos

Titulo	Menú Principal
Importancia	Aconsejable
Descripción	Gestión y Visión del Menú Principal.

Tabla 30 - Requisito Funcional 16 - Menú Principal

Titulo	Cambio entre Niveles
Importancia	Critico
Descripción	Al encontrar la escalera en el nivel le permita continuar al siguiente nivel.

Tabla 31 - Requisito Funcional 17 - Cambio entre Niveles

Titulo	Cambio de Paleta de Colores
Importancia	Nimio
Descripción	El juego al estar limitado el número de colores, cambiar la paleta de colores en ciertas habitaciones para representar situaciones diferentes.

Tabla 32 - Requisito Funcional 18 - Cambio de Paleta de Colores

Título	Gestión de Múltiples Entidades
Importancia	Nimio
Descripción	Gestionar varias entidades en una misma habitación.

Tabla 33 - Requisito Funcional 19 – Gestión de Múltiples Entidades

Título	Gestión del Teclado
Importancia	Critico
Descripción	Gestionar las acciones realizadas en el teclado y hacerlas visibles en el juego.

Tabla 34 - Requisito Funcional 20 – Gestión del Teclado

4.5. PUBLICACIÓN DEL TRABAJO

Esta memoria se encontrará de forma íntegra en la RUA y también se dará la URL de acceso al repositorio ya que el proyecto proseguirá y continuará con su desarrollo, dando así la posibilidad a aquellos interesados de poder seguir los avances del juego.

También se mostrará en mis redes sociales el contenido ya que lo mostrare como parte de mi currículo y así demostrar que tengo conocimientos básicos sobre ensamblador y más avanzados de ensamblador ASMZ80.

4.6. HARDWARE Y SOFTWARE

4.6.1. HARDWARE UTILIZADO

Sobremesa

- Intel Core i7-5820K 3.30 GHz
- 32 GB RAM
- 12 GB VRAM – NVIDIA TITAN X
- MSI X99 GODLIKE GAMING

Portátil

- Intel Core i7 2.30 GHz
- 8 GB RAM
- 1024 MB Intel Graphics HD 4000
- 512 MB Nvidia GeForce GT 650M

4.6.2. SOFTWARE UTILIZADO

Aseprite: Programa de edición y animación de *Sprite* y herramienta de Pixel Art. Lo he utilizado para el dibujado de los diferentes *Sprite*, tiene muchas características y permite mucha personalización.

- <https://www.aseprite.org/>

Piskel: Aplicación web de edición y animación de *Sprite* y herramienta de Pixel Art. Primera herramienta utilizada para crear los primeros diseños del videojuego.

- <http://www.piskelapp.com/>

Winape: Simulador de los Amstrad CPC. Herramienta de simulación bastante completa, permite la modificación de cualesquieras de los parámetros de la CPU, ejecutar código máquina y compilar código ensamblador. Al no tener disponible a mano un Amstrad es la mejor herramienta para un reemplazo.

- <http://www.winape.net/>

Visual Studio Code: Aplicación de edición de código. Esta versión es similar a editores de código como Sublime Text, Geany, Atom y tantos otros. Tiene una gran gama de complementos a instalar, entre ellos soporte para ensamblador del Zilog Z80.

- <https://code.visualstudio.com/>

Cygwin: Herramienta que reproduce un entorno UNIX a nivel de consola y gestiona los paquetes de las diferentes librerías todo para sistemas de MS Windows.

- <https://www.cygwin.com/>

CPCTelera: *Game engine* para la gama de computadoras Amstrad CPC. Herramienta muy completa que me ha proporcionado un gran conjunto de funciones, información y soporte al proyecto.

- <http://ironaldo.github.io/cpctelera/files/readme-txt.html>

5. CONCLUSIONES

5.1. OBJETIVOS CONSEGUIDOS

Entre mis objetivos principales era conocer el funcionamiento de una maquina a bajo nivel, podría ahondar más, pero creo que he cumplido el objetivo, además este objetivo ha

aportado grandes mejoras a mi nivel de programación al conocer como optimizar las funciones a bajo nivel.

He podido desarrollar un videojuego, y aunque se trate de un juego base, es decir, que tiene zonas potencialmente ampliables, puedo estar orgulloso del resultado obtenido, es un juego funcional que ocupa menos de 16 KB, con gestión de colisiones de varios objetos por pantalla.

5.2. OBJETIVOS NO CONSEGUIDOS

Como comentaba en el punto anterior he podido realizar un juego base por ello me hubiera gustado tener más tiempo para conseguir un producto completo ya que el juego tiene un amplio rango de mejora y una gran cantidad de mecánicas a desarrollar.

5.3. RESULTADOS OBTENIDOS

- Conocer una Maquina bajo nivel
- Conocimientos sobre ensamblador
- Generación de Niveles de Forma Procedimental
- Cargador de objetos
- Gestiones de Múltiples Enemigos
- Sistema de Colisiones
- Gestiones de Múltiples Entidades
- Juego Base
- Sistema de habilidades

6. BIBLIOGRAFÍAS Y REFERENCIAS

- [1] **ClrHome:** Pagina donde probé y estudié las diferentes ordenes que permite el Zilog Z80 en ensamblador, hasta utilizando un compilador.

<http://clrhome.org/table/>

<http://clrhome.org/asm/>

- [2] **CPCtelera:** Game engine para amstrad CPC para programador tanto de C como para ensamblador.

<http://Ironaldo.github.io/cptelera/files/readme-txt.html>

- [3] **CPCWiki:** Enciclopedia digital sobre Amstrad CPC, aquí he obtenido desde datos importantes sobre la maquina hasta ideas para diferentes apartados del juego.

http://www.cpcwiki.eu/index.php/Video_modes

<http://www.cpcwiki.eu/index.php/Peripherals>

http://www.cpcwiki.eu/index.php/Amstrad_Whole_Memory_Guide

- [4] **RUA:** Repositorio Institucional de la universidad de Alicante donde he podido visualizar proyectos de años anteriores y aportarme diferentes puntos de enfoque del proyecto.

<http://rua.ua.es/dspace/handle/10045/49866>

- [5] **ASZ80:** Lenguaje ensamblador aplicado en el proyecto.

<http://shop-pdp.net/ashtml/asmlnk.htm>

- [6] **Wikipedia:** Enciclopedia digital, con acceso libre y gratuito donde he obtenido información general aportada al documento.

https://es.wikipedia.org/wiki/Amstrad_CPC

https://es.wikipedia.org/wiki/Amstrad_CPC_464

7. ANEXOS

ASZ80

Es el lenguaje ensamblador utilizado a lo largo del proyecto y del que hace uso CPCtelera. Este pertenece a un conjunto de lenguajes de ensamblador escritos en lenguaje C de programación (ASXXXX) que mediante un conjunto de configuraciones le permite adaptarse a diferentes máquinas y procesadores.

Me voy a enfocar en una de las funcionalidades que trae este ensamblador, la creación de macros como estructuras de datos del programa. Una macro te permite aunar un conjunto de valores y utilizarlos de forma similar un *struct*.

Una macro es aquella que se encuentra definida entre dos etiquetas **.macro** y **.endm**. Todo lo que este contenido entre estas dos se considera parte de la macro, pero como todo lenguaje para crear una macro debe tener la estructura bien definida para la hora de ensamblar el código se pueda traducir al lenguaje máquina.

```
.macro defineObjectValues name , id  
    name'_object :: .db id          ;; Object ID  
.endm
```

Esta sería una macro muy simple que contiene un único valor, pero como saber si su composición es correcta. La macro debe tener un nombre que la defina de forma diferente del resto (**defineObjectValues**).

Pero como se puede ver a continuación del nombre aparece **name** e **id**. Todo aquello que precede al nombre se consideran parámetros que van incluidos dentro de la macro. ¿Porque añadir **name** como parámetro? En este ejemplo esta macro define el valor de un objeto, pero puede que haga falta utilizar múltiples veces la misma macro.

El lenguaje ensamblador te permite crear etiquetas propias para hacer referencia a ciertas posiciones de memoria, funciones o para diferenciar unos valores de otros. En el caso de esta macro el nombre se utiliza para crear una etiqueta diferente en la memoria que luego durante la ejecución poder referenciar.

```
name'_object :: .db id          ;; Object ID
```

Como puedes ver el parámetro **name** va seguido de una comilla simple, esta comilla se utiliza como concatenador, es decir, une el valor de **name** al resto de la cadena y así forma la etiqueta en ensamblador.

```
.macro defineObjectValues name , id  
    name'_object :: .db id          ;; Object ID  
.endm
```

Ahora vamos a explicar el segundo parámetro (**id**), como se puede ver la ID se asigna directamente sin concatenador, su asignación es directa, pero delante de él encontramos otra etiqueta (**.db**). Como puedes reconocer todas las etiquetas propias del lenguaje ASZ80 comienzan por punto y luego la orden para que la reconozca.

A lo largo de este trabajo se puede ver dos etiquetas muy utilizadas, que son:

1. **.db**: Se encarga de definir un byte (8 bits), es decir, lo que continua tras esta orden es el valor que se quiere guardar en memoria.
2. **.dw**: La función de esta etiqueta es igual a la anterior pero almacenando 2 bytes (16bits)

Ahora volviendo a la macro, **id** se trata de un valor de 8 bits que se almacenara en la macro. Hasta ahora hemos tratado como se compone una macro ya que esto lo que hace es definir la macro en el código, pero a la hora de ensamblarlo este no se incluye en la memoria ya que solo se traduce las diferentes llamadas a esta macro para incluir la estructura en el código.

¿Pero cómo se llaman estas macros? Es muy sencillo, si ya has trabajado con lenguajes de alto nivel, la forma de llamarlo es similar a cuando invocas una función, en el caso de la macro nombras la macro y le mandas los valores que necesita.

```
defineObjectValues a , 1      ;; 1
```

El resultado de esta llamada sería lo siguiente en el lenguaje ensamblado.

```
a_object:: 0x01
```

Al igual que los lenguajes de alto nivel, este lenguaje ensamblador te permite definir variables o valores globales e incluirlos en otros ficheros, aquellos valores son los que sus etiquetas de definición (a_object) van precedidos de doble dobles puntos (::), estos son los

globales y luego pueden ser referenciados en otros ficheros, mientras que solo un conjunto de puntos dobles (:) serían consideradas etiquetas locales.

A la hora de ensamblar el código, todo este va a parar al mismo banco de memoria, pero se crean estas formalidades para tener un orden y una organización a la hora de trabajar con el código si no este se volvería ilegible y no sería sostenible.