Escuela
Politécnica
Superior

Desarrollo de un Game Engine de 8 bits

Grado en Ingeniería Informática

Trabajo Fin de Grado

Autor:
Pablo Bey Cabrera
Tutor/es:
Francisco José Gallego Durán

Septiembre 2018

Universitat d'Alacant
Universidad de Alicante

Desarrollo de un Game Engine de 8 bits

Autor
Pablo Bey Cabrera

Directores
Francisco José Gallego Durán
Departamento de Ciencia de la Computación e Inteligencia Artificial (DCCIA)

Grado en Ingeniería Informática

Escuela Politécnica Superior

ALICANTE, 6 de septiembre de 2018

Preámbulo

Desde siempre la informática y los videojuegos me han llamado mucho la atención. Al tener los conocimientos suficientes en mi primer año de carrera, empecé a hacer mi primer juego con una herramienta que lo único que podía hacer era dibujar lineas en una pantalla sin mucho éxito por la lentitud de la herramienta. Aún así había aprendido mucho al dedicarle tantas horas a programarlo. No fue hasta que al buscar otras herramientas vi en la biblioteca de la Universidad de Alicante un libro bastante desactualizado de la herramienta OpenGL. Este libro sirvió como las bases de programación de juegos. Cuando acabe mi primer año conseguí hacer un motor gráfico muy básico usando, enteramente esta nueva herramienta. Seguí trabajando cada año, aprendiendo más sobre esta herramienta con información más actualizada y, llegado al punto, me di cuenta que esto era a lo que quería dedicarme. Aprender, mejorar, crecer como desarrollador de videojuegos y poder llegar a conocer mucho sobre herramientas gráficas como OpenGL, DirectX y el actual Vulkan para poder aprender más.

Vi esta propuesta de trabajo de fin de grado y no lo pensé dos veces. Al hablar con mi tutor me di cuenta de lo mucho que necesitaba conocer las máquinas profundamente para programar. Aún soy muy inexperto y necesito más conocimiento, tanto teórico como práctico y este proyecto me ayudará a dar el primer paso de muchos.

Agradecimientos

En primer lugar debo de dar las gracias a mi familia que siempre han sido los primeros que han apostado por mí cuando nadie lo hacía, invirtiendo no solo su tiempo sino sus vidas en mí. Mi pareja que siempre ha estado allí apoyándome o simplemente estando cuando la necesitaba, mejorando mi vida.

Diego Lezcano aconsejándome en mis penurias y riéndonos de las pequeñas tonterías, desde los primeros años hasta 2025. Javier Pascual, escuchando mis notas de voz excesivamente largas y dándome ánimos a cada paso que daba. Daniel Matinez Albaladejo, lo único que puedo decir es que me alegro mucho de haberte tenido de compañero todos estos años, especialmente cuando hemos estado codo con codo programando. Javier Palomares Crespo, porque me has aportado nuevas perspectivas y uno de los momentos más cómicos de mi vida que no podré olvidar. Sin que falte Gonzalo Martínez Font por todo el feedback que me ha dado en este proyecto que me ha ayudado a mejorarlo.

A la biblioteca de la universidad de alicante por ayudarme todos estos años con materiales tan interesantes como educativos.

Por supuesto a mis gatos que hacían las veces de jardines zen y me ayudaron a concentrarme.

Muchos profesores antes de mi vida universitaria me dejaron marca y han hecho que sea la persona que soy a día de hoy, como mención especial debo nombrar a Jordi, Verónica y María Jesus, si no hubieran estado ahí no estaría aquí.

A Manuel Belso Alarcon por empujarme a dar mis primeros pasos en la programación y animarme a hacer esos pequeños proyectos que ahora me parecen muy básicos pero entonces eran hazañas. También a Antonio Jorge Pertusa Ibáñez por su dedicación con sus alumnos y darle un vuelco a mis esquemas sobre la programación.

Antonio Soriano Paya, que me ayudó a entender los computadores de una manera que no creí posible y a conseguir que me fascinase por la informática a bajo nivel, y a Andrés Fuster Guillo que con sus clases y su conocimiento disfruté aprendiendo y haciendo prácticas de la arquitectura de computadores. A Bernardo Ledesma Latorre por ponernos retos que nunca nadie nos había hecho para convertirnos en unos ingenieros informáticos capaces. También a Nacho Viché Clavel por su dedicación a sus alumnos y su amabilidad cuando más la necesitaba.

Mi grupo de TAES que conocí por pura suerte y con el que tuve mucha sinergia me alegro de haber estado con vosotros, Ramses, Adel y Javier. Sin faltar mi grupo de SGI, Alejandro, Andrés y Jaume con los que he pasado buenos ratos y que escuchaban mis tonterías y aprietos. Sin embargo, no puedo olvidarme de BytesTheDust, aún mis compañeros, Adrián y Yerai, están oyéndome de fondo programar de lo mucho que nos

esforzamos en sacar MoonMan adelante, fue una experiencia muy enriquecedora.

A Francisco Moreno Seco y María del Pilar Arques Corrales por enseñarme las técnicas necesarias para abrirme las puertas a algo que desconocía que quería saber y comprender, pero sobretodo por hacerme ver la programación de una manera diferente.

Por último quiero dar gracias a la persona con la que este proyecto no hubiera sido posible y ese es mi tutor Francisco José Gallego Durán, que tanto en Matemáticas 1 como Razonamiento Automático como siendo tutor de este proyecto me ha encaminado hacia un destino que antes tenía sin definir, con todas esas reuniones y siempre dándome buenos consejos.

Por todas estas personas, y más, he llegado hasta aquí.

A mi familia por haber invertido en mí,
y a mi pareja por estar a mi lado.

La ciencia no es sino una perversión de sí misma
a menos que tenga como objetivo final
el mejoramiento de la humanidad

Nikola Tesla

Índice general

Índice de figuras

Índice de tablas

Glosario

ROM: "Read-Only Memory" o memoria de solo lectura es un tipo de memoria que como su propio nombre indica solamente se permite lectura, no escritura.

RAM: "Random Access Memory" o memoria de acceso aleatorio permite acceder(leer y escribir) a diferentes zonas de la memoria sin que exista una diferencia de respuesta entre ellas.

Z-80: El Zilog Z-80 es un procesador de 8 bits creado por la empresa Zilog, Inc. en 1976. Para más información visitar el apartado del Z-80.

CISC: "Complex Instruction Set Computing" o conjunto complejo de instrucciones computables.

R-R: Diseño hardware Registro a Registro se realizan operaciones únicamente entre registros.

R-M: Diseño hardware Registro a Memoria se pueden llegar realizan operaciones entre registros y entre registros y valores en memoria.

M-M: Diseño hardware Memoria a Memoria se realizan operaciones entre registros, entre registros y valores en memoria, y entre valores en memoria y otros valores en memoria.

AGD: Arcade Game Designer, motor de juego multiplataforma para máquinas de 8 bits.

1. Introducción

1.1. Las máquinas de 8 bits

La época de los 8 bits se caracteriza por una capacidad de memoria muy limitada, que se consideraría ínfima en nuestros días y cargas mediante cintas magnéticas extremadamente lentas, tardando, por ejemplo, varios minutos en cargar 64 KB. En esta época salieron grandes computadores como las máquinas MSX, Amstrad CPC, ZX Sprectrum, Commodore 64... etc. Las tres primeras tienen un procesador muy conocido, hecho por la empresa Zilog, Inc., denominado con el nombre de Z-80. Un procesador que se usaría en videoconsolas tan famosas como la Sega Master System o la Game Gear, e incluso en una versión modificada del mismo para la Game Boy. Este procesador es en el que se enfocará este proyecto. En concreto en el sistema Amstrad CPC 464, debido a su amplio uso en el mundo de los 8 bits, la buena documentación existente y, sobretodo, por ser un buen punto de inicio para el aprendizaje del desarrollo de motores.

1.2. Importancia en la actualidad

La relevancia a día de hoy de realizar un motor de juego en máquinas de 8 bits, con la falta de recursos que supone, es precisamente por esta limitación. Al intentar aprovechar al máximo estos recursos se tiene que usar el ingenio en pos de encontrar la forma más eficiente y óptima de diseñar y programar, de este modo, para aplicar todos estos conocimientos aprendidos en máquinas actuales. Aún así no tiene relevancia únicamente por esta razón, sino que hay todo un mercado de videojuegos retro. Videojuegos creados en máquinas obsoletas pero con el carisma y personalidad de esa época.

1.3. Motor de juego

Tal y como está presente en el título del proyecto, se desarrolla un motor de juego para máquinas de 8 bits. Los motores de juego se caracterizan por facilitar el desarrollo de videojuegos, ahorrando trabajo rutinario y/o ofreciendo herramientas de alta utilidad. Para ello, se estudiará la máquina en la que se realizará este mismo motor, el Amstrad CPC 464, para poder ofrecer un producto completo. Este motor de juego ofrecerá muchas herramientas de este tipo que facilitarán el desarrollo de videojuegos a cualquiera que quiera empezar a crearlos para esta máquina, así como para los que quieran aprender sobre el desarrollo de videojuegos en máquinas retro.

El enfoque del game engine, al cual se ha decidido llamar xuexi, es para juegos con visión cenital, es decir, juegos en los que la vista es desde arriba y el personaje se

desplaza en las cuatro direcciones de la pantalla: arriba, abajo, izquierda y derecha. Un gran ejemplo de este tipo de juego es “The Legend of Zelda” de NES, el cual, estableció las bases de los videojuegos en esta perspectiva. Es por ello que se tendrá en cuenta a la hora de desarrollar el motor.

2. Objetivos

El objetivo de este proyecto es mostrar el desarrollo de un game engine, o motor de juego, enteramente en ensamblador, para máquinas de 8 bits con cada uno de los componentes que se consideren necesarios. En pos de probar estos mismos componentes, se deben crear una serie de prototipos y pruebas demostrando su funcionalidad.

A la hora de desarrollar componentes, se busca un conocimiento interno de las máquinas que ayude a solucionar los problemas de la manera más eficiente posible, teniendo en cuenta la programación a bajo nivel.

Se debe analizar la máquina que soportará el motor de juego, documentarse sobre proyectos similares y explicar cada una de las decisiones que se toman durante el diseño e implementación de cada uno de los componentes y características del motor.

En este listado se especifican cada uno de los objetivos inherentes al desarrollo de un motor de juego de 8 bits:

Generales

- **Máquinas de 8 bits:** Conocer el funcionamiento interno de la máquina en la que se trabajará.
- **Motor:** Desarrollar los componentes del motor, integrarlos al mismo y demostrar su funcionamiento.
- **Optimizaciones:** Aprovechar el conocimiento de la máquina para realizar optimizaciones a bajo nivel que mejoren el funcionamiento del motor.
- **Funcionalidad:** Demostrar el funcionamiento de todos los componentes mediante prototipos y pruebas.

Específicos

- **Marco teórico:** Análisis tecnológico de la máquina de 8 bits elegida para el motor.
- **Limitaciones:** Diseñar teniendo en cuenta siempre las limitaciones técnicas de la propia máquina.
- **Estudio de arte:** Análisis de proyectos similares al nuestro para aprender de ellos.
- **Diseño:** Diseñar cada uno de los componentes con el resto de objetivos en mente, explicando por qué se ha tomado una decisión de diseño y no cualquier otra.

Con todos estos objetivos se realiza el proyecto, aprendiendo, mejorando y en definitiva creciendo como ingeniero y desarrollador.

3. Marco Teórico

3.1. Conceptos básicos

Hay una serie de conceptos básicos que se deben tener en cuenta a la hora de desarrollar a bajo nivel. Sin la comprensión de estos, resultará imposible comprender la máquina y todos sus componentes:

- **Bit:** Un bit solo tiene 2 valores posibles, activado(1) y desactivado(0).
- **Byte:** Un byte es un conjunto de 8 bits. Una cantidad limitada de valores pueden ser representados dentro de 8 bits, si tenemos 2 posibles estados y 8 posiciones se tendrían 2^8 combinaciones pudiendo representar del 0 al 255 usando el sistema binario en lugar del decimal.
- **Valores con y sin signo:** Cuando un conjunto de bits tiene signo se reserva el bit más significativo de este conjunto para indicar si es positivo(0) o negativo(1).[1] Si un byte no tiene signo podrá representar del 0 al 255 pero si lo tiene será del -128 al 127.
- **Bit más/menos significativo:** En un número binario los bits más significativos son los que tienen más valor y por ende los que se encuentran más a la izquierda, siendo los menos significativos los de menor valor, estando a la derecha.
- **Doble precisión:** Una única precisión hace alusión a la unidad básica de la arquitectura, en este caso el byte. Por lo tanto la doble precisión no es más que 2 bytes o 16 bits funcionando como una única unidad.
- **Instrucciones:** Ordenes que hacen referencia al código binario que el procesador puede entender y posteriormente ejecutar.
- **Array:** Un array o cadena es un conjunto de elementos contiguos del mismo tipo.
- **String:** Un array de caracteres.
- **Puntero:** Un puntero es un valor destinado a almacenar la dirección de un elemento para apuntarlo/señalarlo posteriormente.

[1] En este proyecto se hará referencia normalmente al complemento a 2 en caso de ser negativo.

3.2. Máquinas de 8 bits

Las máquinas de 8 bits hacen referencia a computadores con arquitecturas en las que los registros, el direccionamiento y la CPU se basan en la unidad de los 8 bits. En muchas de estas máquinas, a pesar de tener arquitecturas de 8 bits, algunos de sus registros podían ser de 16 bits, al igual que sus direcciones para poder direccionar 64KB de memoria.

3.3. Zilog Z-80

3.3.1. Historia

Zilog, Inc., es la empresa responsable de la creación del procesador Z-80. Su historia se remonta a finales del año 1974 cuando Federico Faggin, después de trabajar en el desarrollo del procesador Intel 8080, dimitió para entonces fundar la empresa Zilog y en julio de 1976 publicar el Zilog Z-80 en el mercado. Este procesador era compatible a nivel binario con el Intel 8080 debido a que Federico Faggin trabajó en el desarrollo del mismo. [Landley, 1998]

El Z80 competía donde la familia de procesadores MOS Technology 6502 dominaba el mercado de los procesadores de 8 bits, sin embargo, el Z-80, sus derivaciones y sus clones se llevaron una gran porción del mercado llevándoles a ser unas de las familias de procesadores más utilizadas de esta época. Todo ello se ve en la referencia [The Cambridge Centre for Computing History, 1978].

3.3.2. Especificaciones técnicas

Arquitectura: 8 bits. CISC, (Complex Instruction Set Computing, o conjunto complejo de instrucciones computables).

Reloj: La frecuencia del reloj con la que se realiza un ciclo de ejecución es de 4 MHz.

Memoria: Tiene capacidad de direccionamiento de hasta 16 bits, lo que equivale a tener un máximo de 2^{16} ó 64K bytes de memoria. Aunque se puede añadir más como se verá más adelante en el apartado 3.4.3.

Formato: El formato Little-Endian es usado por el Z-80 provocando que los datos con más de un byte, como direcciones de memoria, se almacenen ordenados del byte menos significativo al más significativo.

Alimentación: Se requiere una alimentación de 5V.

3.3.3. Particularidades de la máquina

Las particularidades de la máquina son uno de los apartados más a tener en cuenta, si queremos sacar todo el potencial de la misma. Si se entiende como funciona a este nivel conseguiremos ganar esos ciclos y memoria que se necesitan para una mayor eficiencia.

Registros

Los registros son la memoria integrada en la CPU con la que, sacrificando en espacio, se gana en velocidad. Estos registros pueden almacenar datos de una memoria externa y escribir cualquier dato almacenado a la misma. Al ser tan veloces en comparación con la memoria externa, se usan generalmente para llevar a cabo los cálculos que podrían ser significativamente más lentos de usar únicamente una memoria externa.

En el Z-80 tenemos registros de 8 y 16 bits aunque existen parejas de registros de 8 bits, que al combinarse aumentan su precisión a 16 bits.

Como se puede ver en la referencia [Joseph C. Nichols, 1984], existen diferentes tipos de registros entre ellos los registros de propósito general, ordenados en la Tabla 3.1, para poder apreciar las parejas de registros y los registros de propósito específico en la Tabla 3.4. En estas mismas tablas se explica las características de los registros:

15 0 bit

A	F
B	C
D	E
H	L

Tabla 3.1.: Registros de propósito general

- **Doble precisión (AF, BC, DE, HL):** Parejas de registros que nos permiten trabajar con datos de doble precisión.
- **Acumulador (A):** Cuando se realiza cualquier operación aritmeticológica de 8 bits, el primer operando siempre está en A y, al procesar la operación, A se sobrescribe con el resultado de la operación, permitiendo acumular los resultados y seguir operando con ellos.
- **Marcadores (F):** El registro de marcadores (o flags) contiene en cada uno de sus bits condiciones del procesador en base al valor resultante de una operación, generalmente aritmeticológica. Cada operación, afecta a este registro de una manera diferente, por lo que es importante conocer la documentación del set de instrucciones[2].

bit	7	6	5	4	3	2	1	0
Flags	S	Z	F5	H	F3	P/V	N	C

Tabla 3.2.: Marcadores del registro F

- **S :** Indica el bit de signo de un valor, es decir, almacena 1 si es negativo y 0 si es positivo.

[2]Ver la documentación del set de instrucciones del Z-80 en http://clrhome.org/table/

- **Z :** Si el valor es cero este bit es activado.
- **F5 :** Copia el bit 5 de los 8 bits más significativos del valor. Flag no documentado.
- **H :** El bit de "Half Carry" nos indica si ha habido un acarreo intermedio en una operación de 8 bits, es decir, si el bit 3 de los dos operandos genera acarreo. En el caso de que sea una operación de 16 bits es activado con el acarreo del bit 11 de los operandos. Esto es debido a que internamente el Z-80 realiza las operaciones aritmeticológicas de 4 en 4 bits es por ello que se guarda siempre el último "Half Carry".
- **F3 :** Copia el bit 3 de los 8 bits más significativos del valor. Flag no documentado.
- **P/V :** Este bit tiene dos condiciones y en función de la instrucción se tiene en cuenta una u otra. P, indica la paridad del valor. Sin embargo, V indica "overflow".
- **N :** Indica si la última operación fue una resta.
- **C :** Indica si ha habido acarreo en el valor, tanto si es por una operación de 8 bits como de 16.

15 0 bit

A'	F'
B'	C'
D'	E'
H'	L'

Tabla 3.3.: Registros de propósito general alternativos

- **Registros alternativos (AF', BC', DE', HL'):** Una serie de registros adicionales que se pueden intercambiar por sus equivalentes (AF, BC, DE, HL) sin tener que perder los datos de los registros, pudiendo volverlos a intercambiar si se considera necesario.

15 0 bit

I	R
IX	
IY	
SP	
PC	

Tabla 3.4.: Registros de propósito específico

- **Vector de Interrupciones (I):** Indica el estado de las interrupciones del procesador.

- **Refresco de la memoria (R):** Es un contador que indica el refresco la memoria.
- **Registro índice (IX, IY):** Almacenan una dirección en memoria facilitando el acceso a los datos contiguos a esa posición de memoria.
- **Puntero de la pila (SP):** Almacena la posición actual de la pila en memoria.
- **Contador de Programa (PC):** Indica la posición en memoria de la siguiente instrucción a ejecutarse.

Arquitectura e instrucciones

Las arquitecturas CISC se caracterizan por un conjunto de instrucciones amplio y complejo, ofreciendo operaciones en los que los operandos pueden ser datos situados en la memoria en lugar de los registros. Hay un total de tres tipos de diseños de procesador los Registro a Registro (R-R), los Registro a Memoria (R-M) y por último los Memoria a Memoria (M-M), sus nombres nos indican qué tipo de conjunto de operandos pueden llegar a estar en una operación además de que sus propiedades son acumulables, es decir, los procesadores con R-M trabajan también con R-R, y los M-M con R-R y R-M.

El orden en el que están mostrados los tipos es de menos CISC a más CISC siendo el Z-80 del tipo R-M. De esta manera hay algunos operandos donde sus valores están localizados en memoria y otros en registros. Todas estas características y particularidades son clave a la hora de desarrollar nuestro motor de juego, por lo que se tratarán estas características relacionándolas con las herramientas que posee el desarrollador, las instrucciones. Por ello se reitera en que se deben conocer estas herramientas con detenimiento[3].

Pila

El concepto de pila consiste en un conjunto de elementos uno encima del otro, permitiendo retirar o añadir un elemento de la pila en la parte más alta. Por ejemplo, si se desea un elemento que no está situado en la parte más alta se deben retirar previamente los elementos encima del mismo. Este concepto se traslada al Z-80 con el puntero de la pila o registro SP, el cual apunta a la parte más alta de la pila situada en la memoria.

Los elementos de la pila son datos de 16 bits, los cuales se pueden retirar o añadir a través de los registros de propósito general y los registros de propósito específico IX e IY. Si se retira un elemento, el registro SP es incrementado 2 veces, sin embargo no eliminará el valor retirado de la memoria, solamente lo almacenará en el registro elegido. Si se añade, el registro SP es decrementado 2 veces con el elemento de 16 bits añadido en la nueva dirección de memoria. Al haber únicamente un puntero que nos indica la parte más alta de la pila, no se conoce la parte más baja de la misma y tampoco en que punto empezaría a sobrescribir los datos del programa debido a una gran acumulación de inserciones. Por lo tanto mantener una correcta estrategia de retirada e inserción es relevante si no se desea afectar a zonas de memoria adyacente.

[3] Ver la documentación del set de instrucciones del Z-80 en http://clrhome.org/table/

Uno de los usos principales de la pila es almacenar la dirección de retorno en las llamadas a las rutinas, siendo las mismas partes del programa con un propósito específico que al usarse reiteradamente se encapsulan en lo que se llama rutina, pudiendo acceder a ellas en diferentes situaciones sin tener que repetir el mismo código.

Interrupciones

Una interrupción es una pausa del trabajo que pueda estar realizando el procesador en pos de dar los recursos de la CPU a un dispositivo externo temporalmente para, entonces, retornar al trabajo previo. Normalmente, durante la ejecución de cualquier instrucción, se comprueba si se ha recibido una señal de interrupción. Si se ha recibido al terminar la instrucción se provoca la interrupción, este proceso puede llevar a interrupciones anidadas.

Según se explica en [Joseph C. Nichols, 1984] las interrupciones pueden verse clasificadas en 2 grupos:

- **Interrupciones enmascarables:** son interrupciones que a través de software pueden ser desactivadas. El procesador Z-80 tiene un pin específico para recibir la señal de activación de este tipo de interrupciones llamado $\overline{INT}$, abreviatura de "Interruption".
- **Interrupciones no enmascarables:** son interrupciones que no pueden ser evitadas bajo ninguna circunstancia. El Z-80 tiene un pin específico para recibir la señal de activación de este tipo de interrupciones llamado $\overline{NMI}$, acrónimo de "No Maskable Interruption". Sin embargo, no se tendrán en cuenta ya que en nuestro sistema objetivo no se usan, aunque se pueden llegar a utilizar en otros dispositivos externos.

También se explica que las interrupciones en el Z-80 tienen diversos modos que indican el comportamiento que tendrá el procesador en caso de recibir una señal de interrupción:

- **Modo NMI:** el único modo provocado por una interrupción no enmascarable, realiza una llamada a la posición de memoria 0x0066.
- **Modo 0:** realiza una llamada a la posición de memoria transmitida por el bus de datos.
- **Modo 1:** realiza una llamada a la posición de memoria 0x0038.
- **Modo 2:** realiza una llamada a una posición de memoria representada con sus 8 bits más significativos en el registro "I" y los ocho bits menos significativos en el bus de datos del procesador.

3.3.4. Técnicas de programación

Las diversas optimizaciones que se pueden realizar a un programa son tanto en memoria como en rendimiento. Dependiendo de las necesidades, la programación puede

cambiar drásticamente de un tipo al otro en pos de la eficiencia. Con el Z80 en mente, se realizan una serie propuestas simples para aumentar la eficiencia de los programas:

- En los bucles en los que se usa un solo registro como contador se debe procurar usar el registro B con la instrucción DJNZ *. Se ahorra 1 byte y 3 ciclos por iteración del bucle.
- En lugar de usar CP #0, se debería usar OR A. Tiene el mismo efecto esperado y, aunque también se podría usar con AND A, si se usa el flag H o "Half Carry Flag" en algún aspecto del código puede dar error, ya que causa la activación de este flag. Se ahorra 1 byte y 3 ciclos.
- En lugar de usar LD A,#0, se debería usar XOR A. Tiene el mismo efecto esperado pero si se usa el flag H o "Half Carry Flag" en algún aspecto del código puede dar error, ya que causa la desactivación de este flag. Se ahorra 1 byte y 3 ciclos.

3.4. Amstrad CPC 464

3.4.1. Historia

En 1984 saldría el Amstrad CPC 464, diseñado por un grupo de ingenieros en Reino Unido captó gran parte del mercado de empresas informáticas como Commodore y Sinclair llegando a vender tres millones de unidades junto con el Amstrad CPC 6128. El éxito de Amstrad, especialmente en Europa, se atribuye a su precio económico, por vender todos los periféricos como un mismo producto y también porque muchos programas y periféricos fueron desarrollados para este sistema.

En el desarrollo del producto, existía un prototipo llamado ARNOLD que fue creado alrededor del procesador MOS 6502 con el objetivo de competir en el mercado, ya que se usó en el BBC Micro, la Commodore 64 y el Apple II. Sin embargo, fue cambiado definitivamente por el Z-80 al ser la opción más viable, lo cual redujo los costes como explica [Smith, 2014].

3.4.2. Especificaciones técnicas

Procesador: Zilog Z80 a 4MHz, sin embargo, tiene un rendimiento real de 3,3MHz debido a que la memoria RAM es compartida con el circuito de vídeo.

Memoria: Por defecto 32KB de ROM y 64KB de RAM.

Pantalla: Monitor de tubo de rayos catódicos.

Teclado: Con unidad de casete integrada

Puertos externos: Joystick, impresora, unidad de disco, salida de vídeo y audio.

Alimentación: Sin el monitor el cpc 464 necesita una alimentación de 5V.

3.4.3. Particularidades de la máquina

En las particularidades del Z-80 se vieron muchos conceptos que se complementan con el Amstrad CPC 464, ya que el procesador forma parte del propio Amstrad. Gracias a estos nuevos conceptos, se completa la información necesaria para desarrollar el motor gráfico.

Memoria

El Amstrad CPC tiene dos tipos de memoria ROM y RAM. Por defecto una de 32KB y otra de 64KB respectivamente, sumando un total de 96KB de memoria. Sin embargo, como se vio en el bloque del Z-80, el procesador solamente puede direccionar hasta 64KB.

Este problema es arreglado por lo que se conoce como bancos de memoria, que consiste en insertar varias memorias de hasta 64KB en la máquina y seleccionar entre una u otra en función de lo deseado, compartiendo las mismas direcciones de 16 bits. De esta manera, se puede intercambiar entre ROM y RAM o ampliar la memoria de la máquina.

La memoria ROM contiene almacenado todo el firmware de la máquina, permitiendo acceder a él usando sus rutinas en caso de deshabilitarse. En la Tabla 3.5 podemos ver cómo están dispuestas las diferentes memorias:

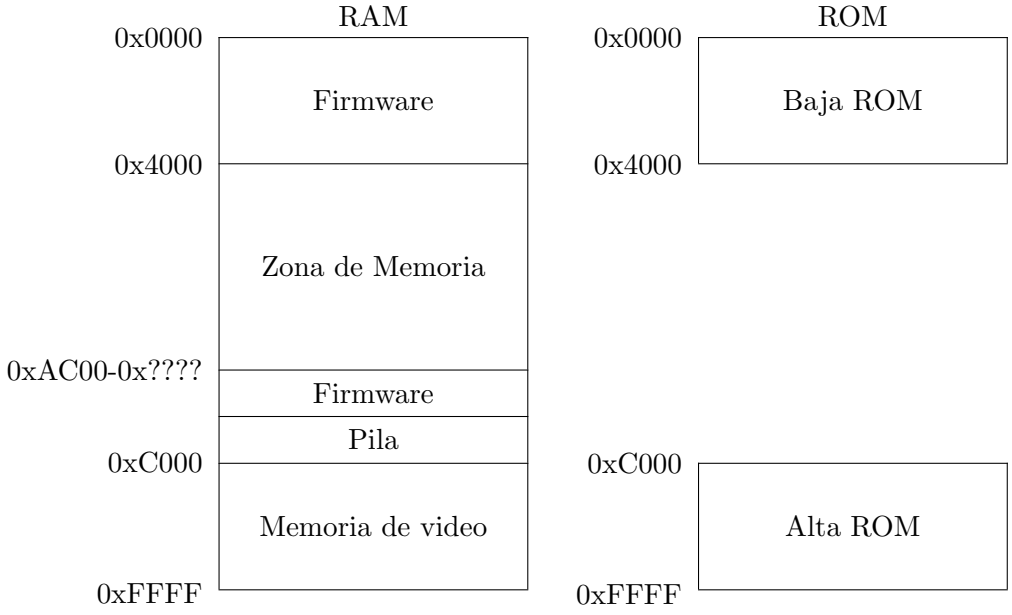

Tabla 3.5.: Estado aproximado de la ROM y RAM sin el firmware desactivado

Video

La imagen mostrada en pantalla tiene origen en la memoria de vídeo, como se puede ver en la Tabla 3.5, localizada por defecto entre la posición de memoria 0xC000 y 0xFFFF,

ofreciendo 16 KB de memoria.

La memoria de vídeo está organizada de una forma concreta, separada en 200 lineas con 80 bytes de largo. Sin embargo no son 200 lineas seguidas sino que se agrupan en 8 bloques de 2048 o 0x0800 bytes. En cada uno de estos bloques se encuentran 25 lineas de 80 bytes y cada una de estas lineas saltan de 8 en 8, de manera que el bloque 0xC000-0xC7FF contiene las lineas 1,9,17 ... 192. Cuando se pasa al siguiente bloque, 0xC800-0xCFFF, se empieza por la segunda linea y así sucesivamente hasta llegar al octavo bloque, 0xF800-0xFFFF.

Con este modelo se puede observar que los 5 bits más significativos de una dirección en la memoria de vídeo, dicen en qué linea se está verticalmente y los 11 restantes en qué byte y píxeles se está horizontalmente.

Si se han hecho los cálculos adecuados, se puede observar que si hay 25 lineas de 80 bytes se han ocupado $25 * 80 = 2000$ o en hexadecimal $0x19 * 0x50 = 0x7D0$ por cada bloque de 0x0800 bytes. Por esto hay 0x30 bytes sobrantes por cada bloque. Esta decisión de diseño hace que esos 0x30 bytes que sobran puedan ser aprovechados con cualquier tipo de dato.

El Amstrad CPC 464 tiene 3 modos por defecto, cada uno de ellos ofrece un uso diferente de este espacio de memoria:

- **Modo 0:** 160x200 de resolución con 16 colores al mismo tiempo en pantalla. En un byte hay dos píxeles alineados horizontalmente. Los 4 bits impares representan el color del primer píxel y los 4 bits pares representan el color del segundo píxel.

bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0
píxel 0 (bit 0)	píxel 1 (bit 0)	píxel 0 (bit 2)	píxel 1 (bit 2)	píxel 0 (bit 1)	píxel 1 (bit 1)	píxel 0 (bit 3)	píxel 1 (bit 3)

Tabla 3.6.: Modo 0, 2 píxeles por byte

- **Modo 1:** 320x200 de resolución con 4 colores al mismo tiempo en pantalla. Modo por defecto, en un byte hay 4 píxeles alineados horizontalmente pero en este modo se separan los 4 bits más significativos y los 4 bits menos significativos formando parejas de esta manera, el bit 4 del primero se empareja con el bit 4 del segundo, dando el código del color del primer píxel. Después, el bit 3 y el bit 3 para el segundo píxel, y así sucesivamente.

bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0
píxel 0 (bit 1)	píxel 1 (bit 1)	píxel 2 (bit 1)	píxel 3 (bit 1)	píxel 0 (bit 0)	píxel 1 (bit 0)	píxel 2 (bit 0)	píxel 3 (bit 0)

Tabla 3.7.: Modo 1, 4 píxeles por byte

- **Modo 2:** 640x200 de resolución con 2 colores al mismo tiempo en pantalla. En un byte hay 8 píxeles alineados horizontalmente, es decir, cada bit representa el color de cada uno de los píxeles.

bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0
píxel 0	píxel 1	píxel 2	píxel 3	píxel 4	píxel 5	píxel 6	píxel 7

Tabla 3.8.: Modo 2, 8 píxeles por byte

Todos estos modos ocupan el mismo espacio de 80x200 bytes, por lo tanto en el modo 0, si en un byte se representan 2 píxeles, la resolución es de (2*80)x200, y el mismo proceso se puede aplicar al resto de modos: modo 1 (4*80)x200 y modo 2 (8*80)x200. Por lo tanto si se realiza un eje de coordenadas respecto a los bytes y no a los píxeles, independientemente del modo que se use, se accede al mismo byte pero no al mismo píxel.

Esta misma memoria de vídeo se muestra en pantalla con el monitor de rayos catódicos, pero a través del raster, un patrón que recorre de izquierda a derecha y de arriba a abajo cada uno de los píxeles de la pantalla.En la Figura 3.4.3 se puede ver en profundidad este patrón.

El raster recorre la memoria de vídeo byte a byte y píxel a píxel, para entonces pintarse en pantalla lo que representan estos bytes.

Esta información no es en balde, ya que la posición del raster no es un dato a menospreciar, porque si se modifica algún dato en una zona de la memoria de vídeo y en ese fotograma el raster ya ha pintado esa zona, no se podrá ver esa modificación en el fotograma.

3.5. El desarrollo de videojuegos

El desarrollo de videojuegos tiene su propio lenguaje y conjuntos de conceptos que se deben tener claros antes de adentrarse en el estudio de motores de juego que haya en la actual escena retro. Entre ellos se encuentran:

- **Sprite:** Un mapa de bits que contiene la información de cada uno de los píxeles que pueden ser representados en pantalla como una imagen.
- **Frame(Fotograma):** En este contexto representa uno de los sprites que se usan para una animación, aunque también puede hacer referencia a la tasa de fotogramas por segundo de un monitor.
- **Tile:** Un tile o baldosa es un sprite de anchura y altura determinados que tiene un identificador asociado.
- **Tileset:** Un tileset es un conjunto de tiles de exactamente el mismo tamaño. Los identificadores de cada tile se asocian respecto de un conjunto.
- **Tilemap:** Un mapa de identificadores que hacen referencia a los tiles de uno o varios tilesets, cuya función es la de ahorrar espacio en memoria, al poder repetir tiles usando únicamente lo que ocupa un identificador, en lugar de un sprite con el tamaño de la memoria de vídeo.

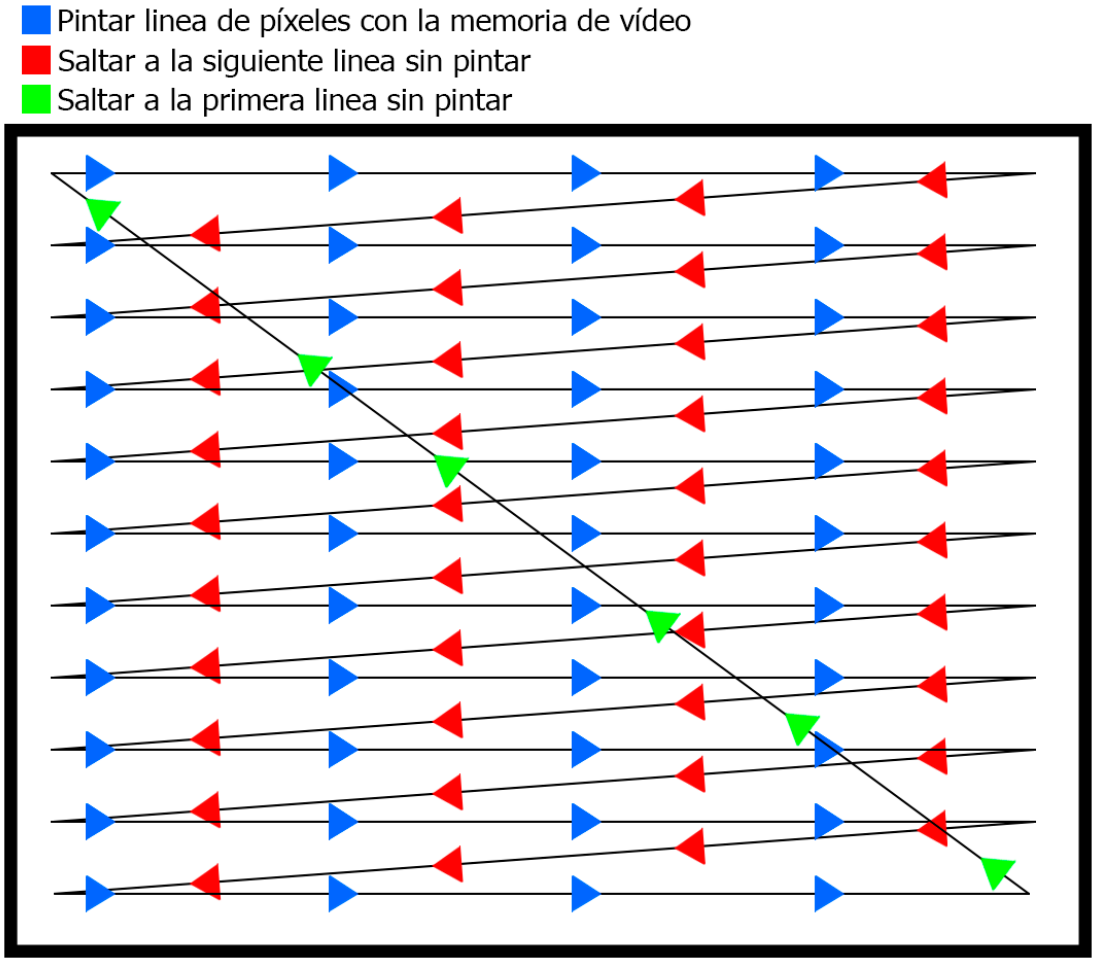

Figura 3.1.: Patrón de escaneo de pantalla, Raster.

4. Estado del arte

El análisis de otros proyectos similares ayudan a enfocar el propio proyecto; a comprobar cuales son características necesarias, a diferenciarse del resto de proyectos y aprender de ellos. Se han escogido dos motores de juego relevantes en la escena retro, el Arcade Game Designer con su propio editor y capacidad de diseñar para Amstrad CPC, ZX Spectrum, Sinclair Timex y Acorn Atom, y el MK2 para el ZX Spectrum a los que se analizarán comentando sus características más relevantes y diferenciándolos.

4.1. Arcade Game Designer

Creado en la década del 1980 inicialmente para la ZX Spectrum, aunque posteriormente se portabilizaría al Amstrad CPC, el Arcade Game Designer es un motor de juego que proporciona las herramientas necesarias para realizar sprites, bloques, pantallas, etc. con muy poco o nulo código por parte del usuario, usa un editor acompañado con un lenguaje limitado basado en BASIC para realizar lógica de juego.

4.1.1. Características

Pantalla

Se puede modificar el tamaño de las pantallas/habitaciones dividiendo en celdas de un carácter, 8x8 píxeles, hasta 32 celdas de ancho y 24 de alto. Este es el primer ajuste que se debe determinar antes de empezar a diseñar cada pantalla, porque cualquier cambio a este tamaño puede llevar al cambio de cada una de la pantallas ya diseñadas o en el caso del editor a la eliminación de todas las pantallas.

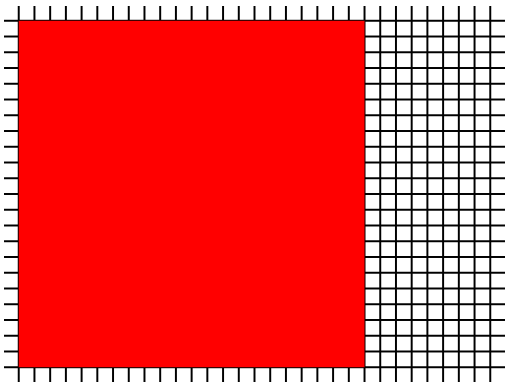

Figura 4.1.: Editor del AGD para el tamaño de pantalla con 22x22 celdas de tamaño.

Se tiene una paleta por defecto que se puede modificar a gusto del usuario afectando a cada uno de los gráficos en pantalla, sin embargo hay un elemento de la paleta que se usa específicamente para el fondo, en el caso del editor es el color de la esquina inferior izquierda.

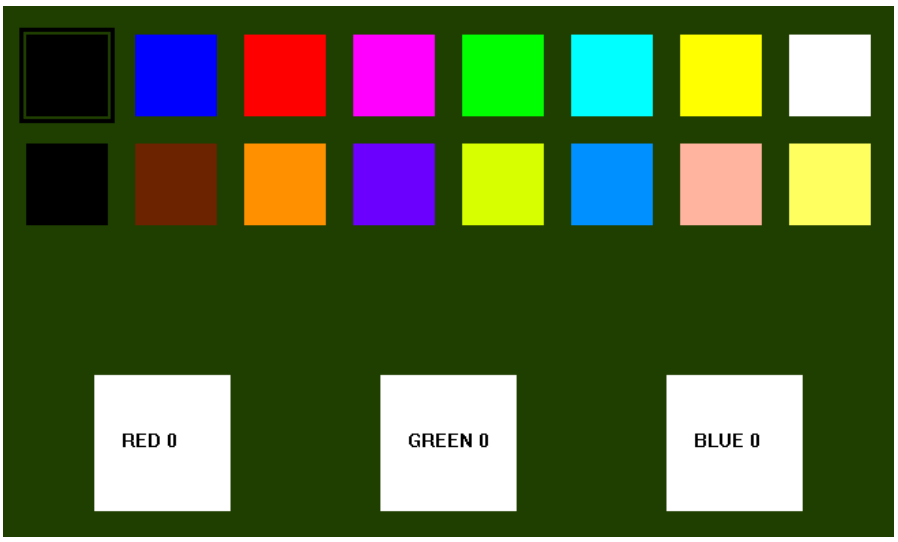

Figura 4.2.: Editor del AGD para la creación y modificación de paletas.

Se pueden diseñar pantallas directamente en el editor seleccionando bloques o sprites previamente creados, pero tienen que estar situados respecto de las celdas en las que se subdivide la pantalla. Incluso se pueden situar no sólo los sprites sino los lugares en los que pueden aparecer.

El mapa se diseña sobre una matriz de habitaciones con el conjunto y combinaciones de pantallas que el usuario desee hasta un máximo de 16 pantallas de ancho y 10 de alto en el editor, cada una de las pantallas definidas tiene un identificador empezando por el 0 que se usa para situarlas en el mapa pudiendo repetir las mismas habitaciones, además se debe definir la pantalla de inicio en la que empezará el juego. Las pantallas que sean adyacentes son accesibles desde su lado correspondiente mientras no se impida el camino, es decir, si tocas el limite izquierdo de la pantalla iras a la pantalla adyacente por la izquierda siempre que haya una pantalla definida en esa posición del mapa.

Además mediante código se pueden modificar el color de los bordes de la pantalla de la máquina y se puede limpiar la pantalla con un comando heredado de BASIC (CLS).

Bloques

Los bloques, elementos en principio inmóviles, ocupan una celda del mapa por lo tanto son de 8x8 píxeles. Se crean con los colores anteriormente escogidos de la paleta. Hay diversos tipos de bloque con cada uno sus propiedades acordes a su nombre:

- **EMPTYBLOCK:** Únicamente un elemento visual, en principio no tiene impacto

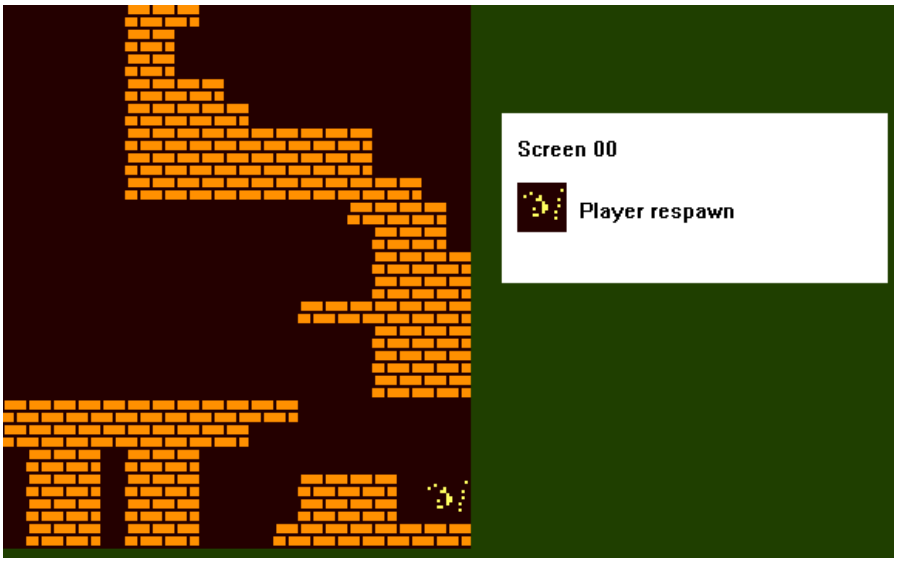

Figura 4.3.: Editor del AGD, seleccionada la posición de reaparición.

en el juego.

- **PLATFORMBLOCK:** Una plataforma flotante, para el movimiento de un sprite si la dirección es hacia abajo al intentar colisionar con el bloque.
- **WALLBLOCK:** Para el movimiento en cualquier dirección de un sprite al intentar colisionar con él.
- **LADDERBLOCK:** Al colisionar con él actúa como una escalera pudiendo subir o bajar por ella en la pantalla sin gravedad.
- **FODDERBLOCK:** Este bloque puede ser eliminado mediante código con el comando "DIG".
- **DEADLYBLOCK:** Te hace perder la partida o restar la vida si colisionas con él.
- **CUSTOMBLOCK:** Se aplica una función personalizada mediante código.

Sprites

Los sprites están predefinidos con un tamaño de 16x16 píxeles, se crean con los colores anteriormente escogidos de la paleta y se les puede añadir varios fotogramas aunque se tienen que controlar por código, posteriormente se pueden asociar 4 sprites cada uno representando a una dirección del espacio en 9 posibles conjuntos de sprites siendo el primer conjunto el jugador, además se puede asociar un tipo de comportamiento a cada uno de los conjuntos, entre esos tipos hay:

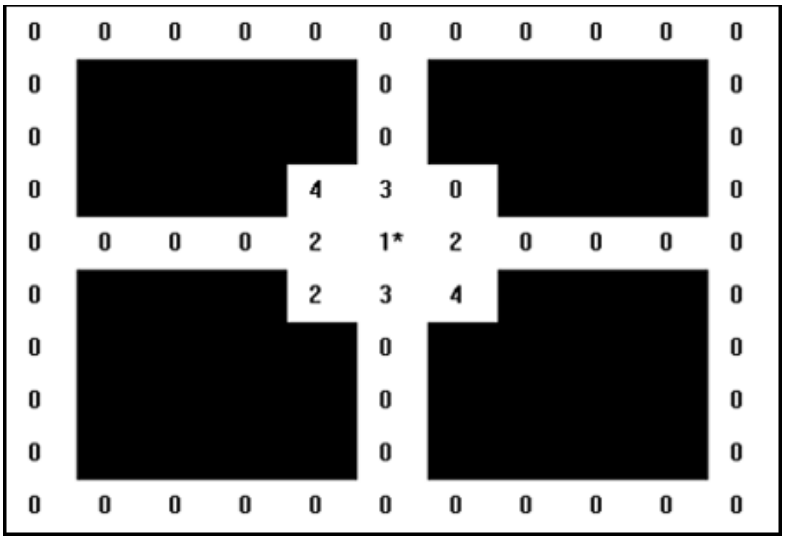

Figura 4.4.: Editor del AGD, diseño del mapa, el asterisco es la pantalla de inicio.

- **Empty template:** Ningún comportamiento asociado.
- **Cybernoid-style:** Controles de izquierda, derecha, disparar y arriba.
- **Platformer:** Controles de izquierda, derecha y salto.
- **Ladders & levels:** Controles de izquierda, derecha y, arriba y abajo para las escaleras.
- **Static Collectable:** Un coleccionable que al entrar en colisión con el jugador se elimina y después se añade a la puntuación del jugador.
- **Bouncing nasty:** Para enemigos que se mueven en diagonal cuando colisiona cambia la dirección a otra diagonal.
- **Horizontal patrol:** Para enemigos que se mueven de izquierda a derecha y de derecha a izquierda, cuando colisionan se intercambia la dirección entre una de las dos.
- **Vertical patrol:** Para enemigos que se mueven de arriba a abajo y de abajo a arriba, cuando colisionan se intercambia la dirección entre una de las dos.

Mensajes

Los mensajes son texto que se imprime en la pantalla carácter a carácter pero para poder imprimirlo, se necesita una fuente de texto y el Arcade Game Designer tiene una por defecto que puedes personalizar como quieras e imprimir los 96 caracteres predefinidos ya sean individualmente o como cadenas de caracteres.

Partículas

El AGD tiene un motor de partículas integrado que permite crear partículas con movimiento en las 4 direcciones espacio, borrarlas cuando se desee e incluso hay una herramienta predefinida que junta varias partículas a las que se les da un movimiento en forma de explosión llamado TRAIL.

Entrada y Salida

Las entradas de teclado están definidas y mediante código se puede dar funcionalidad a cada una de las teclas para los sprites, bloques, etc. además de que se pueden cambiar las teclas por otras que desee el jugador.

La música se puede añadir acompañada de sonidos SFX a través de código para activarlos cuando el usuario lo requiera.

Lógica arcade

Como lógica arcade se recoge todos los elementos propios de un juego arcade que ya proporciona el motor de juego, entre ellos estan la posibilidad de tener objetos escondidos, en una pantalla o que actualmente llevas pudiendo verlos en el menú proporcionado por el motor o en otro lugar definido por el usuario.

Las colisiones tienen comportamientos ya definidos, véase los bloques, pero se proporciona la herramienta al usuario que puede definir comportamientos específicos de las colisiones con elementos definidos.

La puntuación del jugador, qué bonuses obtienes al recoger objetos, resetear la misma puntuación, la cantidad de vidas que tienes, ganar o eliminar a todos los enemigos, y perder provocando el reinicio del juego o de la pantalla es lógica que facilita el motor de juego agilizando el trabajo de cualquier juego arcade.

El juego además proporciona por defecto una tabla de saltos que consiste en almacenar los cambios de posición que tiene que sufrir un sprite, normalmente el del jugador, cuando salta, además permite la modificación de la tabla.

Este motor facilita también funciones que proporcionan valores aleatorios y las funciones que cree el usuario pueden ser asociadas a eventos del juego como ganar, perder, objeto recogido, etc.

4.2. MK2 (La churrera)

MK2 es un motor de juego creado por el grupo *Mojon Twins*, es sucesor de su anterior proyecto MK1 o "La churrera". Hecho modularmente con el lenguaje C y con la ayuda de un conjunto de herramientas para hacer juegos específicamente en ZX Spectrum facilita el desarrollo en máquinas de 8 bits para juegos de plataformas o visión cenital, por ello a diferencia del AGD no tiene editor propio pero sí un lenguaje basado en BASIC para hacer la lógica del juego. Este motor servirá para dar más perspectiva sobre los motores de juego y comprender cuales son los elementos que diferencian a MK2 de otros motores e incluso el propio AGD.

4.2.1. Características

Fondo de pantalla

Este motor usa un fondo de tilemaps de 2 capas, una para el tilemap principal con el fondo base de la habitación y otra que se superpone para añadir más tiles a la habitación.

Los tiles en MK2 no solo tienen una función meramente visual, sino que dependiendo del tipo de tile se puede colisionar con él y provocar un efecto u otro. Hay una cierta variedad de tipos y cabe destacarlos, están el tile traspasable que su único propósito es visual, el traspasable-matante que quita vida al jugador pero no bloquea su camino, el traspasable-ocultante que si el jugador se queda quieto en ellos se esconde de los enemigos, la plataforma que solo tiene sentido en los juegos de plataformas porque cuando el movimiento es hacia arriba no detiene al jugador pero sí que lo hace cuando es hacia abajo, el obstáculo que detiene al jugador en todas las direcciones y por último los interactuables que consisten en cerrojos que se abren si el jugador posee una llave y bloques que pueden ser empujados lateralmente si es un juego de plataformas o en todas las direcciones si es de visión cenital.

Tipos de tiles

MK2 solamente soporta tilesets con 16x16 píxeles cada tile y de 16 o 48 tiles, 16 parece una cantidad innecesariamente pequeña pero de esta manera se pueden representar los 16 tiles con un número de 4 bits en lugar de 8 bits lo que provoca que en 1 byte se puedan introducir 2 tiles resultando en que los tilemaps ocupen la mitad, además de que este motor al usar solo 16 tiles tiene una opción que permite hacer sombras automáticas que generan la ilusión de tener más tiles en memoria como se ve en las Figura 4.5, si se activa hará que los tiles obstáculo afecten a los tiles de su alrededor y se dibuje un tile alternativo, si lo hay, que se encuentra en la misma columna del tile original pero en la tercera fila del tileset.

La posición en la que se encuentra cada tile en el tileset, de izquierda a derecha y arriba a abajo, tiene cierta relevancia porque el tile 0 es el tile de fondo que se usa cuando no hay tile definido pero el propósito de los tiles del 1 al 13 pueden tener el comportamiento que desee el usuario, sin embargo los tiles del 14 al 19 son reservados del motor con un comportamiento definido, el 14 y el 15 son el tile empujable y el tile de cerrojo respectivamente, si no se usa ninguna de estas dos características se pueden usar estos tiles para lo que el usuario decida sin ninguna restricción, el tile 16 es un objeto para recarga la vida del jugador al colisionar con él, el tile 17 si se decide activar son los coleccionables que el jugador recogerá durante el juego, el tile 18 representa las llaves que al recogerlas pueden ser usadas para abrir los cerrojos y por último el tile 19, el fondo alternativo, se activa de forma aleatoria en lugar del tile 0 en pos de que el fondo no tenga un patrón definido.

Como tenía AGD, MK2 también tiene una fuente que puedes personalizar a tu gusto, eso sí, cada letra tiene su posición y tamaño respectivos por lo que salirse del patrón que se ve en la Figura 4.6 daría a errores cuando se quisiera mostrar texto por pantalla. Esta fuente se puede añadir a la misma imagen del tileset configurando la exportación

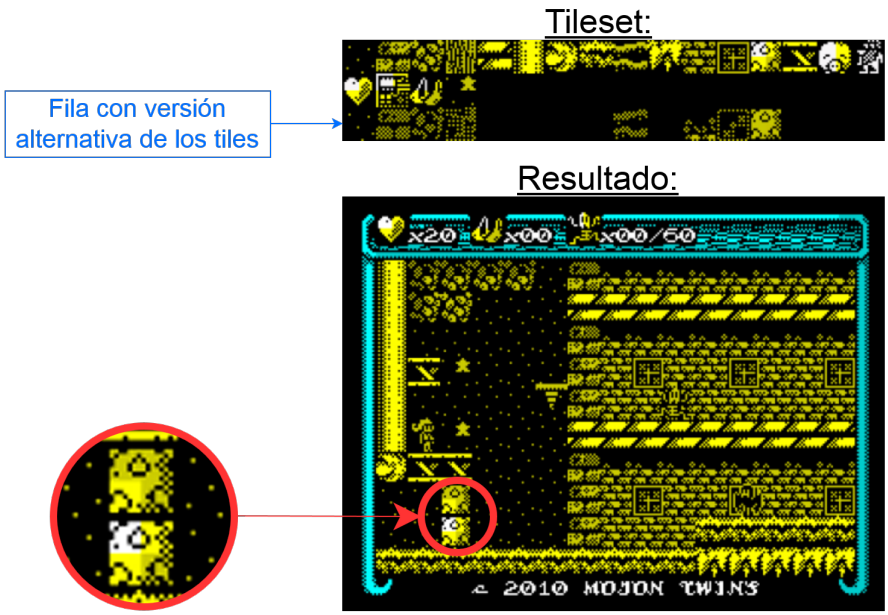

Figura 4.5.: Sombras automáticas y ejemplo de su uso.

del proyecto.

Figura 4.6.: Base para el uso de fuentes.

Pantallas y mapas

Los mapas se realizan en una versión modificada de Mappy, una aplicación de creación de mapas, que con un tileset y una sola imagen realiza todas las pantallas que se hayan escogido y sus interconexiones pertinentes, por supuesto para conseguirlo divide la imagen en bloques del tamaño de un tilemap completo en ZX Spectrum, es decir, en tilemaps de 15x10 tiles. En la Figura 4.7 se puede ver cómo el programa realiza esta división con las seis pantallas separadas por unas finas lineas azules. El usuario debe tener cuidado al introducir un mapa con muchas habitaciones ya que puede no sobrar la memoria por lo que el usuario en principio deberá hacer una estimación de un tamaño limitado, y no definido, de pantallas para entonces si sobra espacio añadir más.

El motor además permite realizar una pantalla de inicio, el marco de juego en el que se pueden añadir los objetos, la vida y las llaves que posee el jugador a cada momento de la partida, y por supuesto una pantalla del final que se muestra cuando haces el juego.

Figura 4.7.: Mapa de MK2 hecho con Mappy.

Sprites

Este motor maneja 5 sprites de 16x16 píxeles que se sitúan en un conjunto que contiene, en este orden, el personaje que usará el jugador, 3 enemigos y si es un juego de plataformas, una plataforma móvil o si es de visión cenital, otro enemigo, sin embargo, por cada sprite le acompaña una máscara de recorte que si encuentra un color específico el píxel del sprite no es dibujado para permitir que los sprites no sean cuadrados y a través de ellos se pueda ver el fondo, en la Figura 4.8 se puede ver la máscara pero se ven más sprites de los esperados porque como se puede observar en las Tablas 4.1 y 4.2 hay diversos frames con los que se realizan las diversas animaciones del motor. Además se pueden añadir unos sprites adicionales para tener balas, enemigos pisables, explosiones y sprites con comportamientos personalizados por el usuario.

La zona colisionable de estos sprites puede ser 16x16 o 8x8 píxeles ya que no siempre los sprites van a tener exactamente la forma de un cuadrado de ese tamaño, es por ello que si se decide usar una zona colisionable de 8x8 se puede colocar en el centro del sprite o en la parte más baja del mismo.

Figura 4.8.: Conjunto de sprites para un juego de plataformas

Enemigos e IA

Hay diversos tipos de enemigos que pueden estar en el juego y cada uno tiene un comportamiento que lo diferencia del resto. El sprite al que están asociado se puede ver reflejado en las Tablas 4.1 y 4.2 aunque si se añade código se pueden cambiar los tipos de enemigo a otros, que con los tipos básicos son:

1. **Vertical:** Con un punto de inicio y final se desplazan de un lado hacia el otro en vertical.
2. **Horizontal:** Con un punto de inicio y final se desplazan de un lado hacia el otro en horizontal.
3. **Diagonal:** Con un punto de inicio y final se desplazan en diagonal pero no se asegura que pueda llegar al punto de final.
4. **Plataformas móviles:** Con un punto de inicio y final tanto en vertical como en horizontal que en los juegos de visión lateral sirve como plataforma móvil.
5. **Perseguidores:** Estos enemigos aparecen fuera de la pantalla y pueden atravesar paredes para atacar al jugador aunque si se esconde puede evitar ser atacado.
6. **Personalizados:** Creando su lógica de juego se puede dar el comportamiento que se desee.
7. **Perseguidores sin descanso:** Estos enemigos aparecen dentro de la pantalla sin poder atravesar las paredes y te persiguen para atacarte, aunque se maten en un tiempo vuelven a aparecer con temporizadores que el usuario puede definir.

Exportador

A diferencia del AGD el MK2 al no tener un editor propio la exportación y compilación debe ser realizada por el usuario a mano con una serie de comandos definidos por los desarrolladores que reciben una serie de datos del juego entre ellos se pueden encontrar

Conjuntos de sprites para juegos de plataformas	
0	Personaje principal. Derecha. Andando. frame 1
1	Personaje principal. Derecha. Andando/Parado. frame 2
2	Personaje principal. Derecha. Andando. frame 3
3	Personaje principal. Derecha. Saltando
4	Personaje principal. Izquierda. Andando. frame 1
5	Personaje principal. Izquierda. Andando/Parado. frame 2
6	Personaje principal. Izquierda. Andando. frame 3
7	Personaje principal. Izquierda. Saltando
8	Enemigo de tipo 1. frame 1
9	Enemigo de tipo 1. frame 2
10	Enemigo de tipo 2. frame 1
11	Enemigo de tipo 2. frame 2
12	Enemigo de tipo 3. frame 1
13	Enemigo de tipo 3. frame 2
14	Plataforma móvil. frame 1
15	Plataforma móvil. frame 2

Tabla 4.1.: Orden de conjuntos de sprites para juegos de plataformas

Conjuntos de sprites para juegos de visión cenital	
0	Personaje principal. Derecha. Andando. frame 1
1	Personaje principal. Derecha. Andando. frame 2
2	Personaje principal. Izquierda. Andando. frame 1
3	Personaje principal. Izquierda. Andando. frame 2
4	Personaje principal. Arriba. Andando. frame 1
5	Personaje principal. Arriba. Andando. frame 2
6	Personaje principal. Abajo. Andando. frame 1
7	Personaje principal. Abajo. Andando. frame 2
8	Enemigo de tipo 1. frame 1
9	Enemigo de tipo 1. frame 2
10	Enemigo de tipo 2. frame 1
11	Enemigo de tipo 2. frame 2
12	Enemigo de tipo 3. frame 1
13	Enemigo de tipo 3. frame 2
14	Enemigo de tipo 4. frame 1
15	Enemigo de tipo 4. frame 2

Tabla 4.2.: Orden de conjuntos de sprites para juegos de visión cenital

dar los datos de inicio de los mapas, su tamaño, etc.; las posiciones de cada enemigo y jugador y la adición de lógica por parte del usuario mediante scripts en el lenguaje del motor.

5. Metodología

La metodología que usará el proyecto es una metodología ágil que se desarrolla por iteraciones, a diferencia de otras metodologías se realizan las fases principales del desarrollo; planificación, análisis, diseño, implementación y pruebas, en una iteración para después hacer otra iteración con las mismas fases pero con otros objetivos, así sucesivamente hasta el objetivo final o hasta que se decida finalizar el proyecto.

5.1. Iteración

Las Iteraciones son fases que se subdividen en las subfases de planificación, análisis, diseño, implementación y pruebas, haciéndose simultáneamente excepto planificación. En cada iteración se deben de tener claros los objetivos, tanto los principales como los secundarios. A partir de la fase de planificación cada fase de las iteraciones tienen asociadas una serie de tareas que se coordinan para conseguir los objetivos antes mencionados como pequeños hitos a realizar.

Planificación	Análisis
	Diseño
	Implementación
	Pruebas

<———— X semanas ————>

Tabla 5.1.: Una iteración con una duración de X semanas.

5.2. Planificación

La planificación es la subfase en la que se define todo el trabajo que se realizará en la iteración. En primer lugar se definen las características que conseguirán los objetivos planteados para la iteración, en segundo lugar una característica se subdividirá en "historias" con cada uno de los componentes a implementar de la característica y por último una historia se subdivide a su vez en tareas pequeñas con la cantidad de trabajo medido en horas y con cada una de las implementaciones que se deben de hacer para completar ese componente.

Característica	⟶	Historia	⟶	Tarea

Tabla 5.2.: Modelo seguido en la planificación.

5.3. Análisis

Esta subfase requiere analizar la tarea o problema y comprenderla para extraer todas las posibles subtareas o nuevas tareas que requiera y de esta manera observar cómo afectan unas a otras para actuar en consecuencia. La experiencia influencia mucho a realizar con éxito esta subfase ya que cuanta más experiencia más fácil es encontrar tareas adicionales o mal estimadas en su cantidad de trabajo, errores que se hayan cometido en las especificaciones, tareas en una "historia" que no le corresponde y tareas duplicadas que pueden llevar a gastar nuestra cantidad de trabajo innecesariamente.

5.4. Diseño

El diseño comprende la resolución del problema, no implementándolo, sino con cómo funcionará, cuales serán cada uno de los elementos que intervendrán en el proceso y compararlo con otros posibles diseños para entender por qué se ha escogido esa decisión de diseño y no otra. En primer lugar se dividen las tareas y subtareas para resolver los problemas de manera más sencilla para posteriormente identificar la relación entre estas divisiones, una vez se tienen identificadas se tiene que especificar la funcionalidad de las tareas y también las relaciones entre otras tareas y por último definir cada uno de los componentes necesarios para la finalización de las tareas.[Sommerville, 2005]

En el proceso de diseño es importante recalcar que siempre se debe abordar los problemas con enfoques diferentes permitiendo escoger el mejor, apoyarse en diseños que ya se hayan hecho para encontrar la mejor manera de proceder, tener diseños versátiles a los que se puedan realizar cambios en pos de poder mejorar o cambiar el mismo, poder evitar errores a través de un diseño bien definido y tener en cuenta que un diseño no es una referencia para el código sino una serie de procesos y arquitecturas que ayudan a resolver un problema.

5.5. Implementación

La implementación como su mismo nombre indica, implica la realización de la solución al problema o tarea. Al haberse diseñado la solución de la tarea previamente se aplicaría al proyecto para entonces comprobar si la estimación de la cantidad de trabajo en horas ha sido correcto, si ha estado por debajo o por encima de las horas estimadas se anotaría con el objetivo se ser más precisos en futuras estimaciones. Si no se han realizado los diseños correctamente, es decir, no contemplan los suficientes casos o no se especifica lo suficiente, se habrá arrastrado el problema de diseño y se tendrá que solucionar en esta subfase.

5.6. Pruebas

Al tener una implementación terminada es inevitable que haya errores, que pueden provocar que no funcione el componente pero que en casos concretos sí, es por ello que las pruebas son una parte tan importante del desarrollo y se deben establecer un conjunto de comprobaciones para cada uno de los componentes, en pos de encontrar estos errores y solventarlos. Las pruebas se realizan a varios niveles, se parte de la tarea implementada para comprobar el funcionamiento del componente, para entonces ir comprobando que su integración al resto de componentes relacionados no da ningún problema y por último comprobar el funcionamiento en global con todo el proyecto.

6. Cuerpo del documento

6.1. Iteración 1. Núcleo del motor

Conseguir el núcleo de lo que será el motor es el objetivo principal de la primera iteración del proyecto, componiéndose el núcleo de características como el control del personaje principal con teclas y/o joystick, creación de mapas e interconexión entre los mismos, punto de inicio del juego y punto de fin o destino, y definición de bloques colisionables y no colisionables. Aunque antes de poder realizar todas estas características se necesita una base sobre la que trabajar.

6.1.1. Base del motor

Bucle de juego

Una de las primeras características del motor es ser capaz de borrar, dibujar y actualizar entidades en un bucle de juego. Se tiene que realizar en este orden, "borrar-actualizar-dibujar", porque si cambiamos el orden por ejemplo a "borrar-dibujar-actualizar" justo después de actualizar se borraría no lo que hemos dibujado si no la supuesta entidad que se dibujará, con el primer orden siempre borramos lo que se ha dibujado.

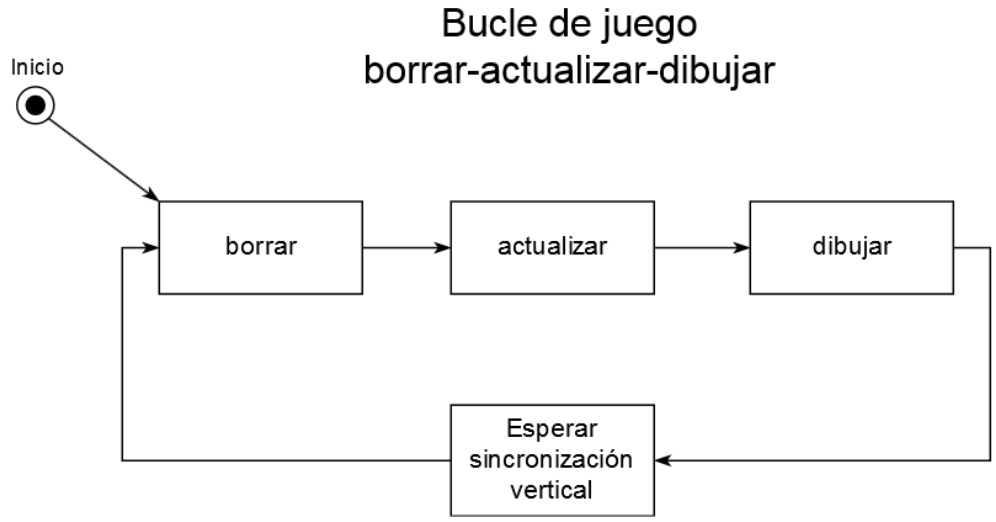

Figura 6.1.: Bucle de juego utilizado en el motor.

Como se puede observar en las Figuras 6.2 y 6.1 hay un estado adicional, esperar a la sincronización vertical, con el que el motor se sincroniza con el raster, esto se realiza

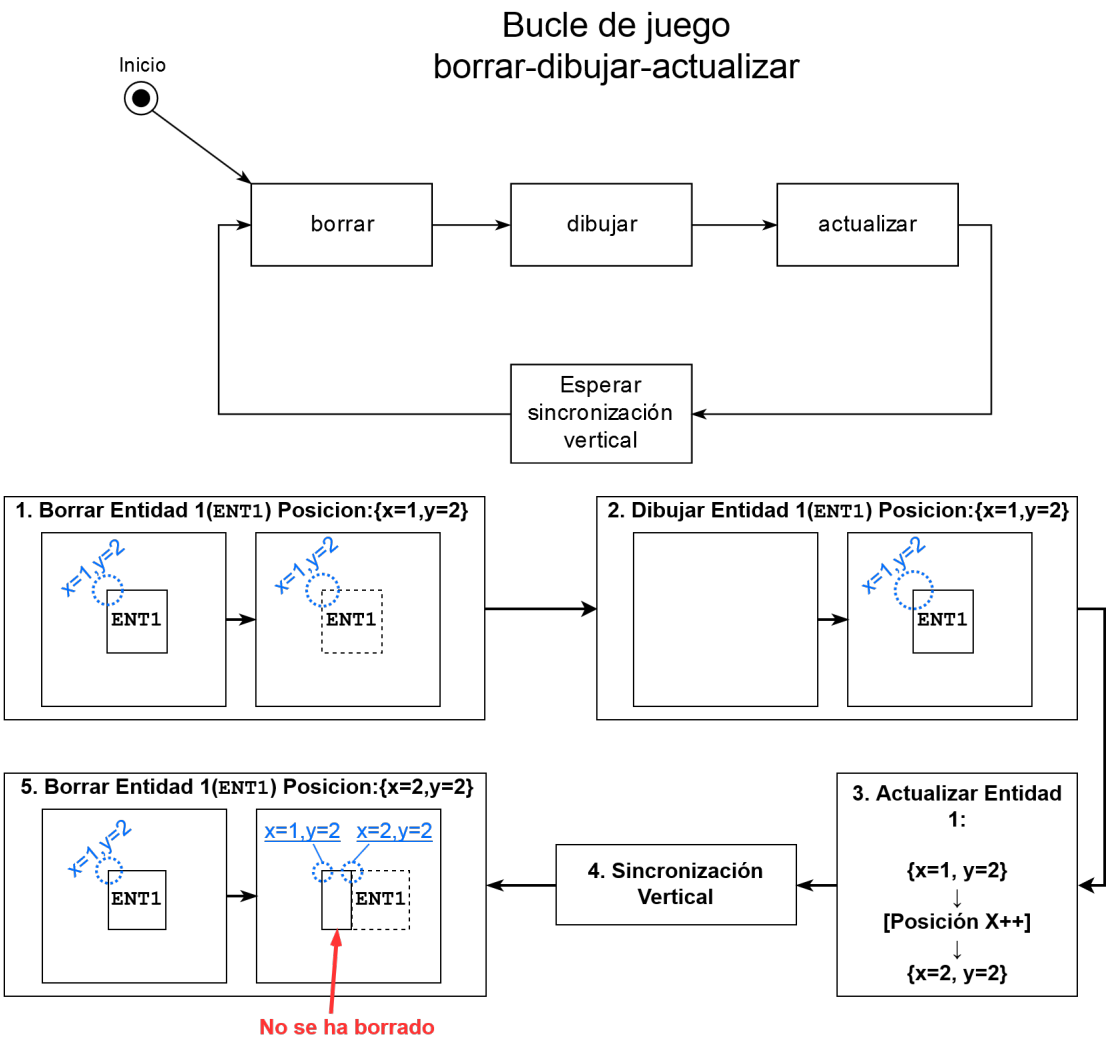

Figura 6.2.: Bucle de juego borrar-dibujar-actualizar con un ejemplo de su problema de funcionamiento.

en pos de evitar que se borre y dibuje sin tener en cuenta el raster y por lo tanto que en algunos frames se empiece otra vez a borrar antes de mostrar todo lo dibujado observándose un parpadeo ya que el monitor del Amstrad CPC es un pantalla de rayos catódicos que se recorre píxel a píxel. El problema en "borrar-dibujar-actualizar" podría solucionarse almacenando la última posición, para borrar con esa posición, y aunque se perdería en memoria se ganaría en dibujar antes para intentar estar por delante del escaneo del raster.

Modos

Hay tres modos documentados en el Amstrad CPC, haciendo un repaso de ellos están el modo 0 con 160x200 de resolución y 16 colores simultáneos en pantalla, el modo 1 con 320x200 y 4 colores simultáneos, y por último el modo 2 con 640x200 de resolución y 2 colores. Entre estos 3 modos se decidió escoger el modo 0 a pesar de su baja resolución porque en comparación con los otros modos tiene 16 colores lo que hace que sea visualmente atractivo.

Los elementos que se sitúen en pantalla tendrán el eje de coordenadas situado no respecto de la resolución, 160x200, sino de los bytes que ocupa la memoria de vídeo, 80x200 bytes, por lo que las posiciones en X serán del 0 al 80 y en Y del 0 al 200.

Entidad

La entidad debe ser el conjunto de datos más básico para poder construir a partir de ella de forma que los bloques no sean elementos a parte de la entidad sino parte de la propia entidad, porque sino se caería en la redundancia ya que muchos datos coincidirían. La entidad más básica debe tener un número identificativo, la posición en los ejes de abscisas y ordenadas además de su tamaño en los mismos ejes, es decir, cuadraras o rectangulares. Por limitaciones técnicas las entidades siempre tendrán esta forma aunque se pueda visualizar con un sprite que tenga una forma diferente.

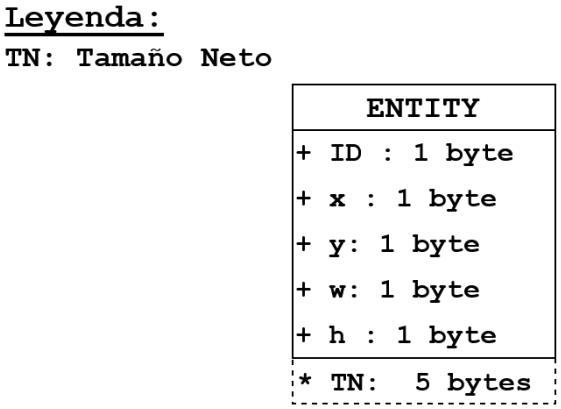

Figura 6.3.: Modelo de entidad básica o "Entity".

Como se ha dicho anteriormente las entidades se podrán visualizar con un sprite aprovechando su posición y tamaño, pero no lo añadiremos al modelo de entidad básica ya que podrán haber entidades que no sean visibles como por ejemplo muros invisibles, disparadores, etc. Esto implica que se hará un modelo de entidades visibles que herede los datos de la entidad básica para construir sobre ella.

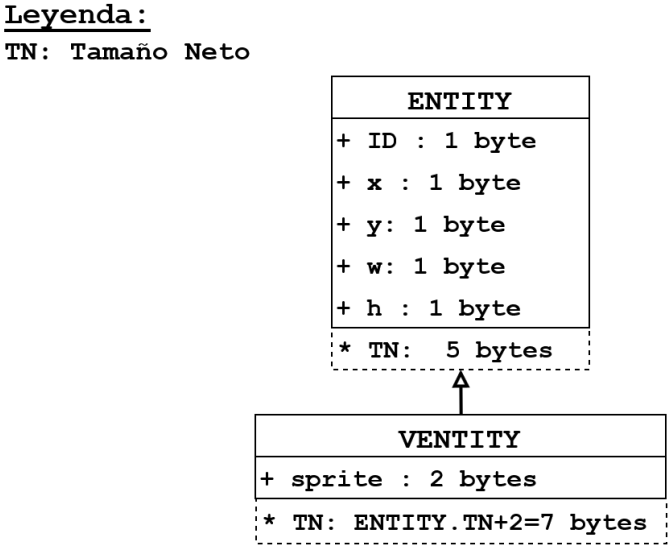

Figura 6.4.: Modelo de entidad visible o "Visible Entity".

A la hora de mostrar entidades de diferentes tamaños e incluso duplicados con cada uno sus variables independientes ¿Como usar esta serie de entidades de diferentes tamaños en el borrado, dibujado y actualizado? Lo primero que necesitamos para responder esta pregunta es dónde, en concreto en qué parte de la memoria pondremos esta serie de entidades. Se almacenará al final de toda la memoria del programa donde se pueden guardar temporalmente todas las entidades en ejecución de una pantalla, de esta manera se podrán hacer copias de una misma entidad con sus datos completamente independientes en una pantalla. Se debe tener cuidado al almacenar muchas entidades ya que puede llegar a ocupar el espacio de la pila o incluso de la memoria de vídeo si no se tiene cuidado, es por ello que se pondrá un limite de memoria razonable que el usuario no podrá traspasar.

Una vez dejado claro "dónde" ahora podemos responder "cómo", y se diseñaron 3 modelos que barajaremos comprobando sus ventajas y desventajas:

- Array estático: este modelo consiste en un array de entidades de diferentes tipos y tamaños, cada entidad conoce su tamaño para continuar al siguiente elemento, en este array no se puede eliminar ni añadir entidades y si se pudiese añadir se haría al final del array. Tanto para el borrado como el dibujado como la actualización se debe recorrer este array de entidades provocando que se tenga que recorrer 3 veces aunque algunas entidades no se dibujen, borren y/o actualicen.

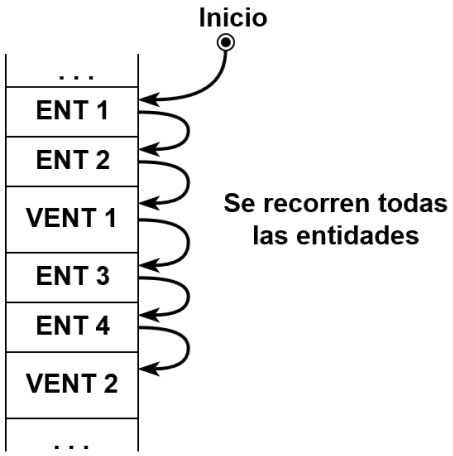

Figura 6.5.: Diseño del array estático.

- Array dinámico: este modelo muy parecido al estático permite la eliminación de entidades ahorrando recursos a la hora de recorrer el array de entidades ya que en el momento en que no se necesita una entidad se elimina, esto se consigue no haciendo consciente a la entidad de su propio tamaño sino almacenando al final de cada entidad del array un puntero al siguiente elemento. De la misma forma que el estático, este array de entidades tiene que recorrerse 3 veces aunque algunas entidades no se dibujen y/o actualicen. En el caso de insertar para evitar problemas como los de las Figuras 6.6 y 6.7 se tendría que reservar memoria siempre teniendo en cuenta el tipo de entidad que ocupe más, gastando mucha más memoria.

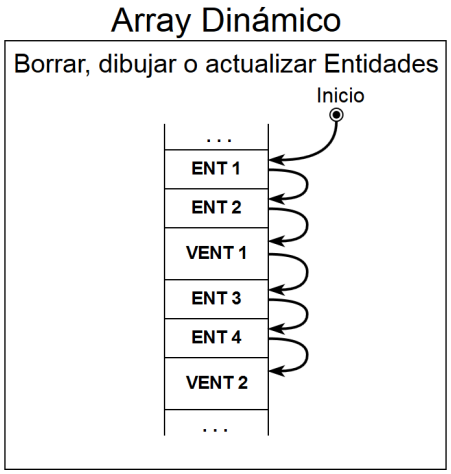

Figura 6.6.: Diseño del array dinámico.

- Arrays por categorías: este modelo entiende que existen tres categorías principales

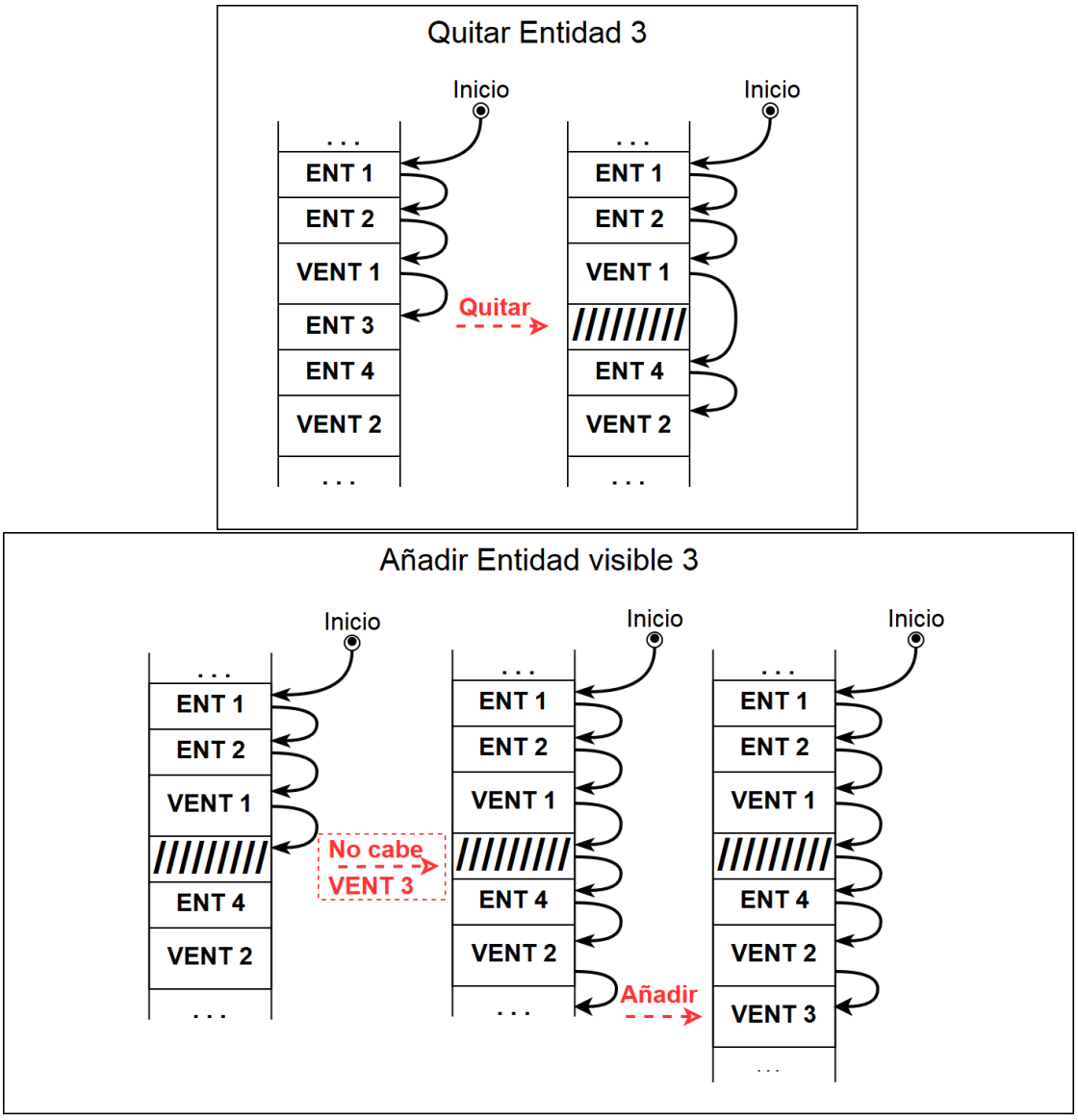

Figura 6.7.: Ejemplo de quitar y añadir entidad con array dinámico.

en las cuales no participan todas las entidades, si son dibujables, si son colisionables y si son actualizables. Cada una de estas categorías esta acompañada de un array de direcciones a entidades para dibujar/borrar/colisionar/actualizar, de esta manera si hay muchas entidades dibujables solo se tienen en cuenta en el momento del borrado y dibujado pero si hay pocas actualizables no tienes que recorrer absolutamente todo el array de entidades, si añades entidades dentro del limite dado ya sea por el usuario o por defecto se añade al final del array o en un hueco sin problema.

Estos arrays pueden tener 3 valores: una dirección a una entidad, un final de array (0x0000) y por último un hueco resultado de borrar una entidad del array (0xFFFF), un detalle que puede llamar la atención es por qué el final de array es 0x0000 y por qué el hueco es a su vez 0xFFFF, esto es debido a como se recorre este tipo de array. En este algoritmo siempre se lee la parte alta de cada dirección para comprobar si se ha llegado al final del array o si se ha llegado a un hueco, este es el caso de una inserción pero no dista mucho de cualquier tipo de recorrido. Esta optimización se debe a que se quiere ahorrar espacio y tiempo. Provocado por esta decisión de diseño las entidades nunca podrán estar entre 0x0000 y 0x00FF además de 0xFF00 y 0xFFFF aunque el último no es un problema muy grave ya que gran parte de él es memoria de vídeo, sin embargo, es un detalle muy importante a tener en cuenta.

El diseño del Array estático es muy lento ya que tiene que comprobar futilmente cada entidad a pesar de no realizar alguna de las 3 acciones. El Array por categorías nos permite añadir en ejecución la pantalla de manera desordenada sin poder borrar las entidades solo inutilizarlas y el Array dinámico nos permite borrar y meter de manera desordenada a coste de mucha memoria. El diseño que tiene menos desventajas es el Array por categorías ya que ni es tan lenta como la primera, aunque más difícil de llevar a cabo, ni ocupa tanta memoria como la tercera.

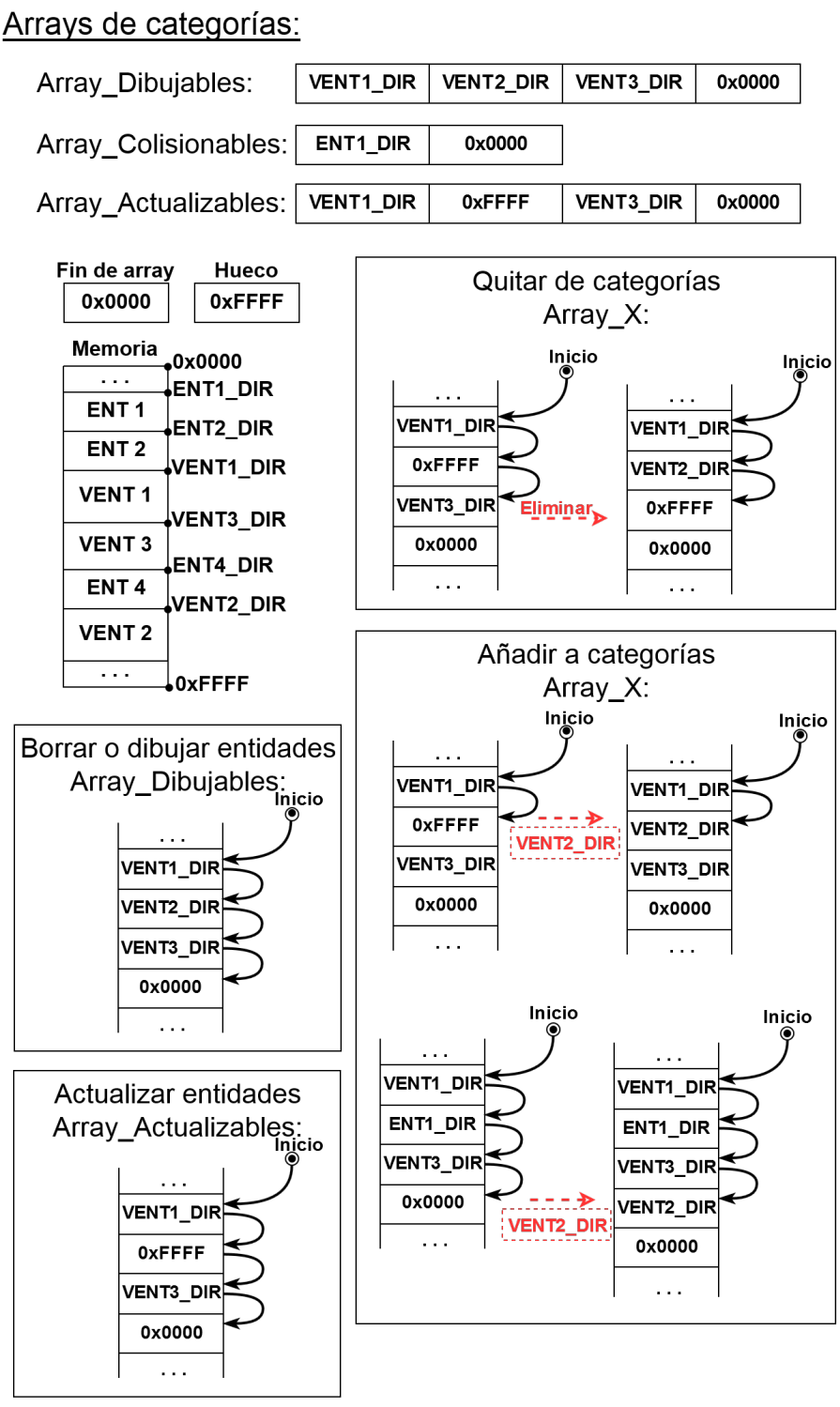

Figura 6.8.: Diseño del Array por categorías.

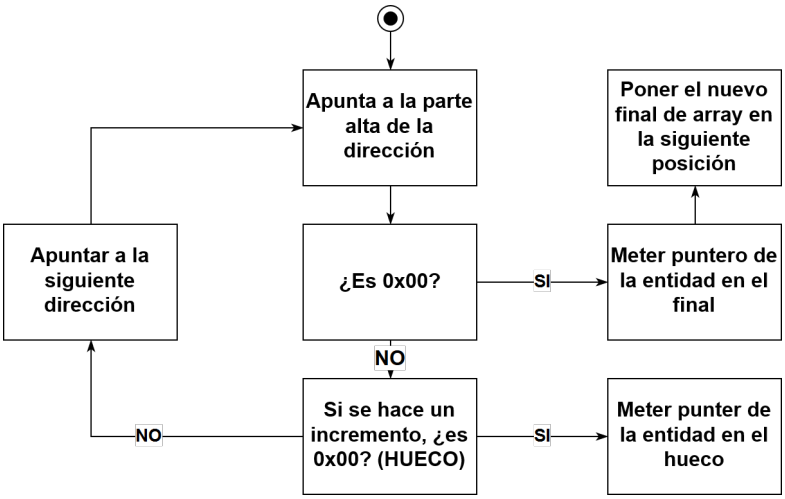

Figura 6.9.: Algoritmo del recorrido de un array de una categoría en la inserción.

6.1.2. Control del personaje principal

Cada movimiento de los personajes debe estar supervisado para no superar los límites de la pantalla y no colisionarse con las entidades colisionables además de posiblemente en un futuro poder saber la orientación del personaje. Esto está planteado así ya que en primer lugar si no nos movemos primero no podemos colisionar con nadie, sin embargo, antes de movernos se debemos comprobar que no se sale de los limites de la pantalla ya que ello puede llegar a comportamientos indefinidos. Se actualiza la posición entonces podemos comprobar si toca los bordes de la pantalla y justo después si colisiona con alguna entidad y parar el movimiento en el eje que genera el problema. El movimiento tiene que estar separado por ejes, es decir, moverse en X y moverse en Y son dos herramientas porque si una entidad solo se mueve en un eje no realizar las comprobaciones que se hacen en el otro eje.

Con cada movimiento que se haga se comprueba si has tocado alguno de los bordes de la pantalla y qué borde de la pantalla, de esta manera ya tendremos preparado el movimiento por el mapa. Por último con ayuda de CPCtelera asociamos la activación de las teclas W, A, S y D que como en la Figura 6.11 vemos su equiparación a las cuatro direcciones a las que puede moverse el personaje. Se podría dar la posibilidad de cambiarlas por otras en el futuro como el AGD, con cada una de las cuatro direcciones a las que puede ir el personaje.

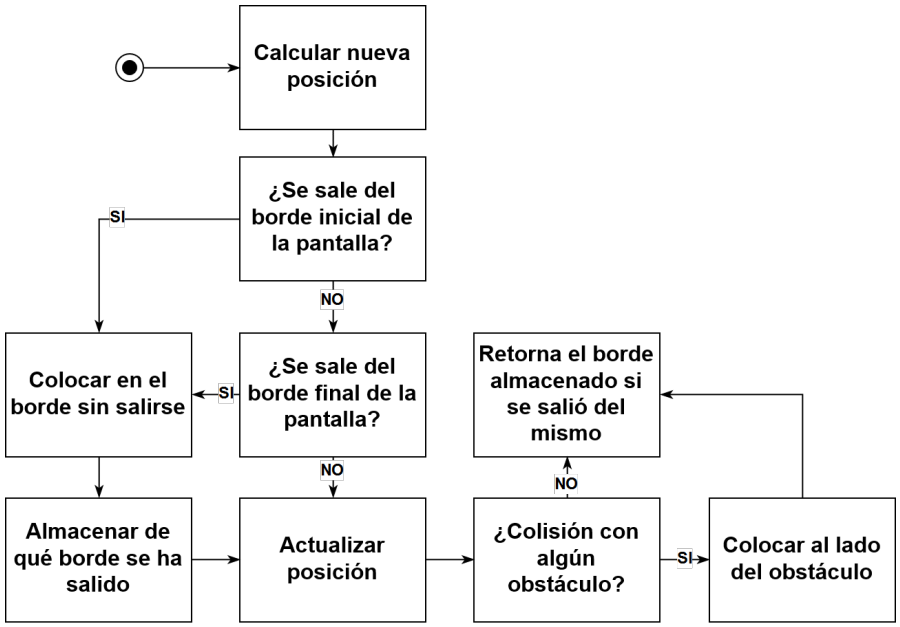

Figura 6.10.: Algoritmo de movimiento en cualquier eje.

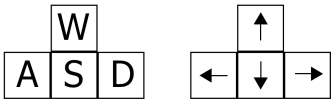

Figura 6.11.: Uso de las teclas W, A, S, D.

6.1.3. Mapas y pantallas

Cada pantalla o habitación deberá tener asociado un identificador, un mapa de tiles y un puntero a las entidades que están en la pantalla con sus posiciones asociadas ya que el usuario necesita poder colocarlos en posiciones diferentes en función de la habitación. AGD usa un mapa en forma de matriz de tamaño máximo con los identificadores de las habitaciones y hay asociada una pantalla de inicio, en este caso se usará una opción parecida con un mapa de tamaño definido por el usuario que si todos los mapas juntos generan más de 160 habitaciones dé un error a la hora de pasarlo al motor ya que el AGD solo tiene un mapa de 160 habitaciones y me parece una cantidad razonable pero este número podría bajar o subir en función del espacio que considere necesario para el jugador.

Además se podrán crear varios mapas y cambiar entre ellos pero solo uno puede ser el inicial. Como las características del núcleo del motor exigen el mapa tendrá también un punto inicial y final, y como matriz que es requiere una anchura y una altura.

Entonces, si se ha tocado un borde en concreto el personaje se trasladará a la habitación adyacente respecto de ese borde, manteniendo tu posición en el eje Y si es a las habitaciones izquierda y derecha, y manteniendo tu posición en el eje X si es a las habi-

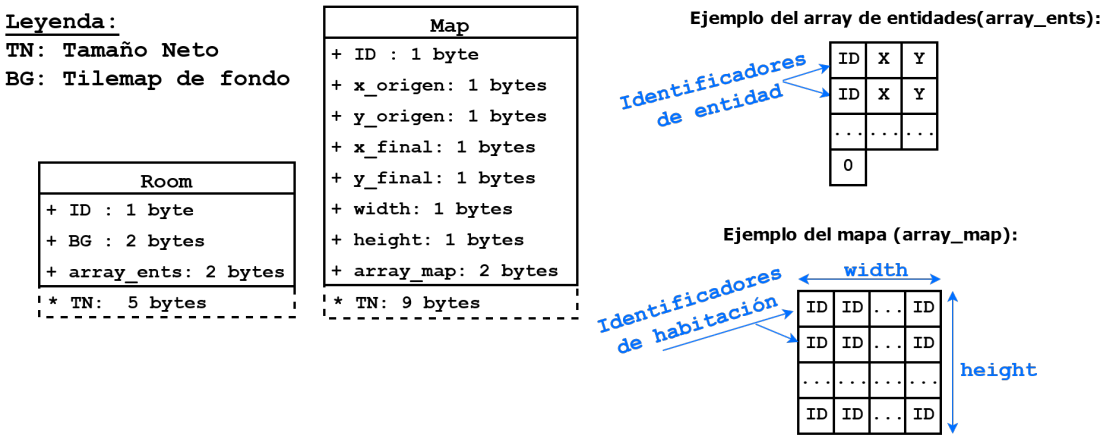

Figura 6.12.: Diseño de las habitaciones y mapas.

taciones superiores e inferiores, en el caso el personaje intente acceder a una habitación fuera de los límites del mapa se volverá a la habitación anterior para evitar comportamientos indefinidos. Además si llega a la habitación final salta un evento al que se le puede asociar rutinas predefinidas o definidas por el usuario.

6.1.4. Cargar Pantalla

El array de entidades de cualquier habitación contiene una serie de conjuntos de tres elementos en este orden: Identificador de Entidad, posición en X y posición en Y, para poder conocer cuando se finaliza el array se necesita un valor nulo así que se reserva un identificador nulo con valor 0 para que en el momento en el que un identificador sea 0 implique el fin del array.

Para ahorrar memoria cada una de las entidades/habitaciones de los arrays de habitaciones/mapas respectivamente usan sus identificadores y no sus posiciones a memoria como se puede observar en la Figura 6.12 ya que los identificadores ocupan un único byte y las posiciones a memoria 2 bytes, si repetimos mucho una misma entidad/habitación en alguno de los arrays se perderá 1 byte de memoria por cada entidad/habitación que se haya repetido. La manera de solucionarlo se realiza con 3 arrays de 2 bytes cada elemento uno de entidades, otro de habitaciones y por último uno de mapas. En cada array se tienen almacenadas las direcciones de memoria según el identificador empezando por el 1, es decir, la dirección del identificador 3 estará en la posición 2 del array por el identificador nulo mencionado anteriormente, este diseño se puede ver reflejado en el ejemplo de la Tabla 6.1.

A raíz de estas decisiones de diseño los identificadores asociados a cada entidad/habitación/mapa serán únicos, pero el usuario nunca manejará sus números sino nombres que después serán ordenados del 1 a N y asociados a su entidad/habitación/mapa para colocar sus direcciones en la posición del array correspondiente.

Una vez aclarado el modelo que vamos a seguir para las habitaciones y los mapas

Arrays de identificadores				
Identificadores	ID 1	ID 2	ID 3	ID 4
ID Entidades	ENT1_DIR	ENT2_DIR	VENT1_DIR	VENT2_DIR
ID Habitaciones	ROOM1_DIR	ROOM2_DIR		
ID Mapas	MAP1_DIR			

Tabla 6.1.: Ejemplo de los arrays de identificadores

se debe definir finalmente el algoritmo de carga de mapas y habitaciones con todas las decisiones de diseño que se han realizado.

Se ha tomado la decisión de que solamente se puede tener un único tileset para facilitar el desarrollo, en principio no debería resultar en ningún problema ya que al final todos los tiles que se usen van a tener que almacenarse en memoria de una manera u otra. Donde podría ser un problema esto es que en el futuro usemos compresión para ahorrar memoria de manera que haya diferentes tilesets por mapa pero aún dista de que se introduzca esta característica.

Tenemos una serie de categorías de entidad en las que se indica si son [colisionables/dibujables/actualizables] en un solo byte pero la cuestión es como funcionará y dónde se almacenará este byte.

En primer lugar las categorías de una entidad se almacenan en un byte en el que la activación del bit 0 implica que la entidad será dibujable, el bit 1 será colisionable y el bit 2 actualizable. Se plantea de esta forma por si en el futuro se desean hacer más categorías.

Bit	bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0
Ca	Para ampliaciones					actualizable	colisionable	dibujable

Tabla 6.2.: Esquema de un byte de categorías

En segundo lugar podemos almacenar estas categorías de si son [colisionables/dibujables/actualizables] de las entidades según el ID en un array igual que las entidades, las habitaciones y los mapas en vez de en las entidades como tal porque al copiar las entidades perdemos 1 byte por entidad lo que hace que con muchas entidades se ocupe mucho espacio y no ganamos en rendimiento haciéndolo. Esta manera aunque con bastante acoplamiento debería ser funcional pero cuando añadimos una entidad en la pila de entidades al final de la memoria del programa para conocer cuanto se va a copiar de la entidad principal también necesitamos conocer el tamaño por lo que al final se opto por otra opción que consigue el mismo efecto al añadir el byte de categorías y el byte del tamaño de los datos principales a copiar justo antes de declarar los propios datos principales y a la hora de copiar la entidad solamente copiar los datos principales. Se hará así por el aumento de rendimiento que supone tener los datos cercanos y no tener que acceder por un ID pero sobretodo por bajar el acoplamiento del programa y por lo tanto que los posibles futuros cambios sean más fáciles de realizar.

El número máximo de un identificador por tipo (entidad/habitación/mapa) que puede

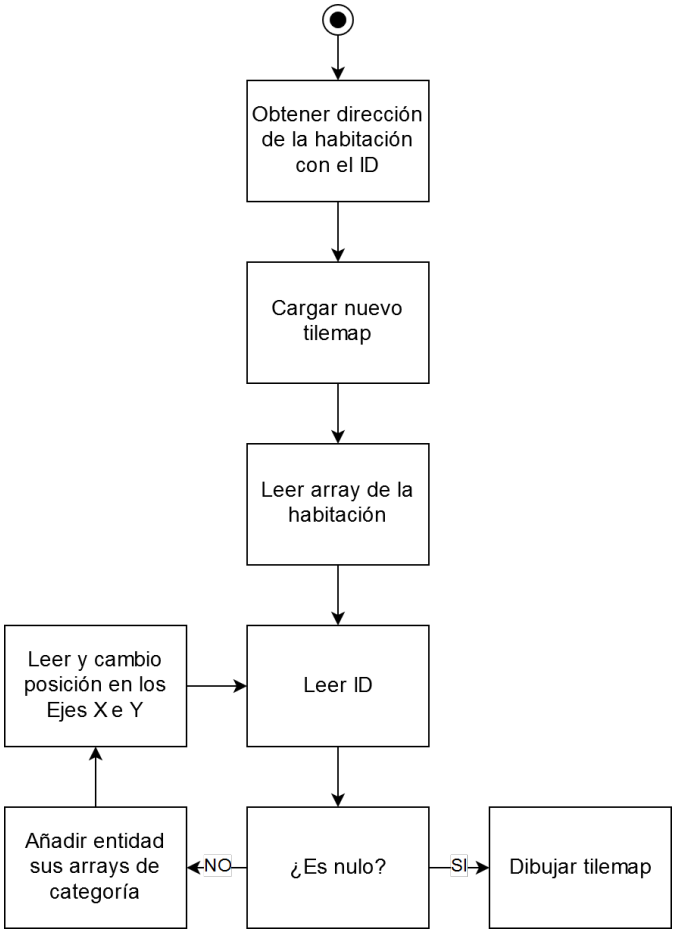

Figura 6.13.: Diseño de la carga de una habitación.

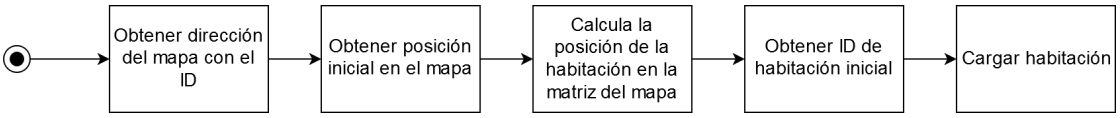

Figura 6.14.: Diseño de la carga de un mapa.

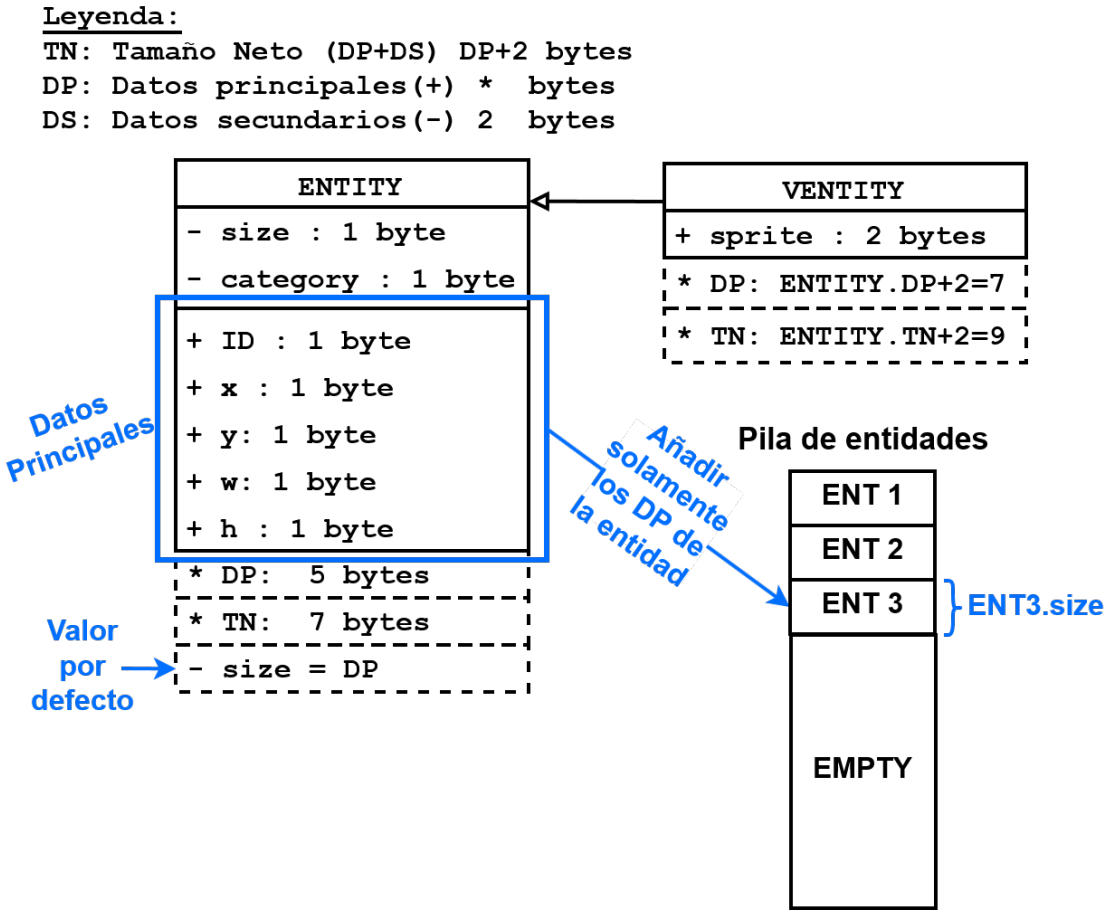

Figura 6.15.: Nuevo diseño de Entidad con un ejemplo de la utilidad del cambio.

haber es 127 esto es así como decisión de diseño ya que el bit 7 que falta se necesita libre, porque cuando se calcula una dirección de un array a partir de un identificador [1] se multiplica por 2 desplazando, el identificador menos uno, un bit a la izquierda. Se hace así para optimizar porque las direcciones de los elementos de estos arrays ocupan 2 bytes ya que la posición destino siempre será $destino = origen + (Identificador - 1) * 2$ y así evitar lentitud por hacer doble precisión, sin necesidad, ya que no habrá tantas entidades en memoria, se resuelve de esa forma. Es por todo ello que en lugar de haber un máximo de 160 habitaciones habrán un máximo de 127.

6.1.5. Bloques colisionables y no colisionables

La categoría que he estado usando llamada colisionables creo que se aleja de su objetivo principal que es obstaculizar el movimiento, por ello creo que un nombre adecuado para la categoría es obstáculos, posteriormente se podrá hacer una categoría llamada colisionables que pueda colisionar con todos elementos pero se sopesará en posteriores iteraciones.

Bit	bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0
Ca	Para ampliaciones					actualizable	obstáculo	dibujable

Tabla 6.3.: Segundo versión del esquema de un byte de categorías

Con los bloques el usuario no necesita conocer las categorías ya que por su naturaleza los bloques colisionables no se actualizan, se dibujan y son obstáculos, en futuras versiones quizás se pueda habilitar el cambio de categorías por parte del usuario pero con tipos de entidades tan específicos en principio no se necesitará. Después de analizar estas categorías se discernió que un bloque colisionable no es más que una "Visible Entity" con la categoría de obstáculo añadida por lo que hay que tenerlo en cuenta y un bloque colisionables es solo una "Visible Entity".

Finalmente para poder asociar el movimiento del personaje a una entidad se ha realizado un tipo de entidad que es actualizable que contiene el evento que se tiene que hacer cuando se actualice de esta manera no hay necesidad de poner a mano el evento haciendo responsable a la entidad actualizable o "Updateable Entity (UEntity)" mejorando la mantenibilidad del motor y ahorrando espacio ya que no hay que introducir una condición por identificador porque sólo el salto condicional ocupa 3 bytes. Por último tendríamos este esquema en la Figura 6.16.

Además los bloques no se acaban aquí ya que algo que se podría tener en cuenta para próximas iteraciones es hacer bloques invisibles pero siendo obstáculos como muros invisibles o bloques movibles que solo pueda mover el jugador.

[1] Para más información del array mirar Tabla 6.1

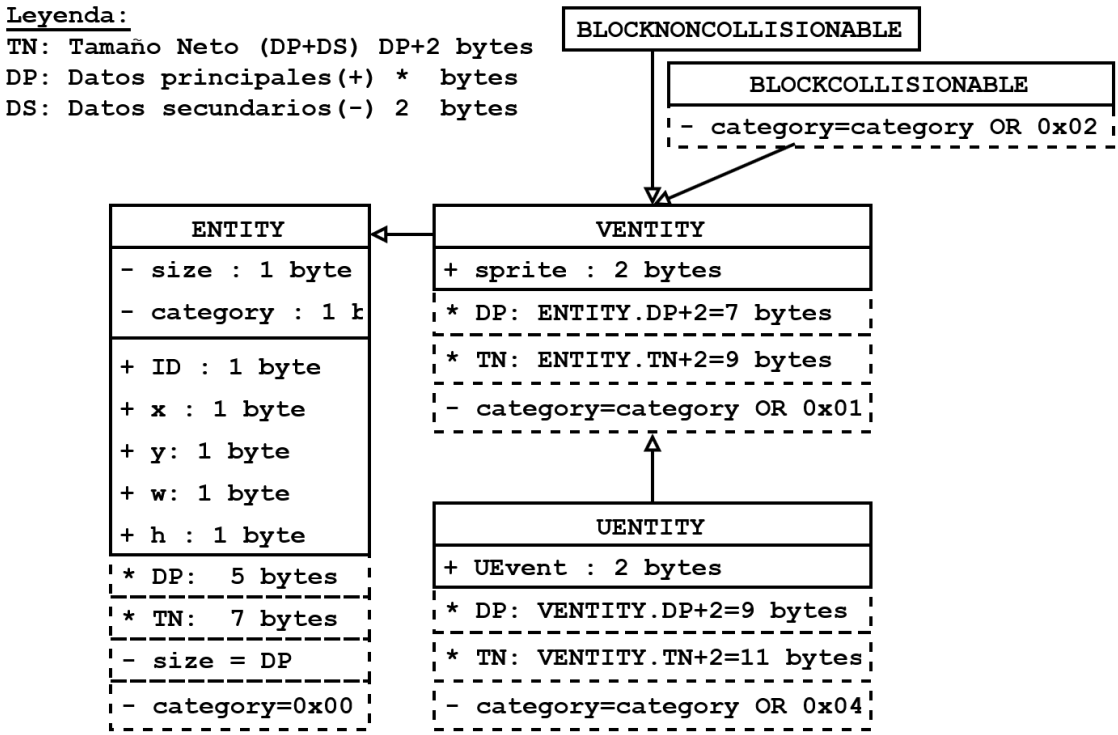

Figura 6.16.: Diseño final de las entidades en esta iteración.

6.2. Iteración 2. Exportación y otras características

Llegado a este punto se necesita empezar a pensar en cómo el usuario va a realizar su juego, sin embargo, aún hay que agregar características para tener un motor completo, la más urgente es la adición de enemigos en el juego aunque si la duración de la iteración lo permite además se quiere añadir doble buffer, objetos recogibles, mapas de durezas, aceleración de entidad y animaciones, las cuáles se querrán explicar posteriormente. Esta iteración estará dedicada a añadir estas nuevas características dentro de lo que sea posible aunque gran parte de ella a permitir que el usuario que pueda realizar su juego evitando tocar código, que lo primordial sean los datos.

6.2.1. Enemigos

El jugador es tanto la persona que juega como su representación en el juego, una entidad controlable, y el enemigo es una entidad que controla la propia máquina, en teoría su propósito es entorpecer o poner un reto al jugador mientras que él intenta avanzar en el juego. Esto en el motor quiere decir que los dos son del tipo "Updateable Entity" pero con eventos que realizan acciones diferentes, es decir, mientras que el evento del jugador provoca que avance por el escenario o realice acciones a través de la teclas, el evento del enemigo provoca el avance por el escenario pero en función de una serie de órdenes predefinidas que se podrían llegar a considerar inteligencia artificial.

Patrones de movimiento

Los enemigos pueden tener muchos tipos de movimientos predefinidos que se proporcionarán al usuario, entre ellos se pueden recalcar los vistos tanto en AGD como MK2, el movimiento vertical, el movimiento horizontal y el movimiento diagonal, además MK2 posee una pequeña IA que consiste en perseguir al jugador si está en su zona de visión. Por ahora solamente se desarrollará el movimiento en vertical y horizontal pero no se descarta añadir los anteriormente comentados en futuras iteraciones.

Como se puede ver en la Figura 6.17 el movimiento vertical y horizontal cuando toca un borde de la pantalla cambia su dirección a la contraria. Como es lógico este comportamiento se tiene que repetir en caso de que colisione con un objeto. Entonces para añadir estos comportamientos al motor necesitamos que a cada movimiento se nos diga si hemos chocado con el borde de la pantalla o si hemos chocado con algún obstáculo. Se requiere esta diferencia ya que para los enemigos chocar con los bordes de la pantalla o con un obstáculo no supone ninguna diferencia, sin embargo, para el jugador tocar los bordes de la pantalla significa cambiar de habitación y chocar con un obstáculo parar el movimiento en esa dirección, por lo que se actualizará el algoritmo de movimiento para satisfacer esta necesidad.

Al hacer los dos eventos, movimiento vertical y horizontal, para las entidades actualizables que serán enemigos se pudo observar que si se querían distintos tipos de enemigos con distintas velocidades se tendrían que implementar un evento entero por cada entidad que quisiera mover el usuario a una distinta velocidad, es por ello que para evitar este

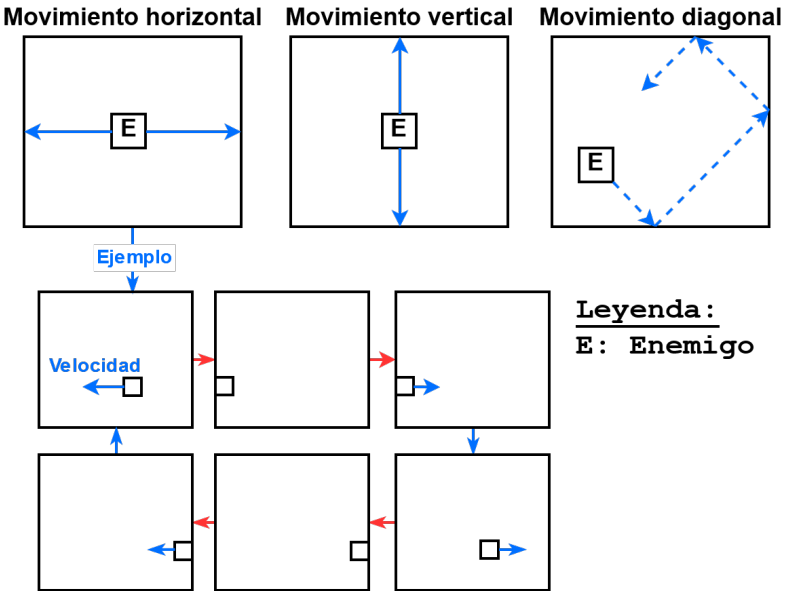

Figura 6.17.: Patrón de movimiento horizontal, vertical y diagonal por la pantalla con ejemplos.

problema hay que crear un nuevo tipo de entidad que tenga la velocidad en los dos ejes del espacio, el nombre sería "Moveable Entity (MEntity)" o entidad móvil. Esta entidad tendrá 1 byte por cada eje y cuando se quiera mover sumará o restará la velocidad a su posición en función de si se dirige a una posición mayor o menor en los ejes de coordenadas respectivamente. Cuando ya tenemos las entidades móviles definidas se puede añadir un poco de abstracción haciendo los dos tipos de enemigos de movimiento horizontal y vertical que tienen bloqueados los movimientos que no usan al valor cero como se puede observar en la Figura 6.19.

6.2.2. Doble buffer

Un buffer consiste en un espacio reservado de memoria con un propósito específico, en este caso se hace referencia al buffer de la memoria de vídeo. El doble buffer en este proyecto consiste en dos espacios de memoria de vídeo del mismo tamaño, 2 bloques de 16KB, que serán usados para que de mientras se muestra la memoria de vídeo de un buffer hacer los cálculos y los dibujados necesarios en otro para que al acabar intercambiarlos y así sucesivamente.

La razón de añadir esta característica no es más que para evitar el parpadeo u la desaparición de entidades a la hora de dibujar, algo que no desea ningún desarrollador de videojuegos. En muchas ocasiones si la carga de trabajo en cada iteración del bucle de juego es muy grande, el monitor que muestra la pantalla píxel a píxel la memoria de vídeo se puede adelantar al dibujado provocando que no se vea lo dibujado por haber empezado la pantalla borrando e incluso que parpadee si el monitor sobrepasa la

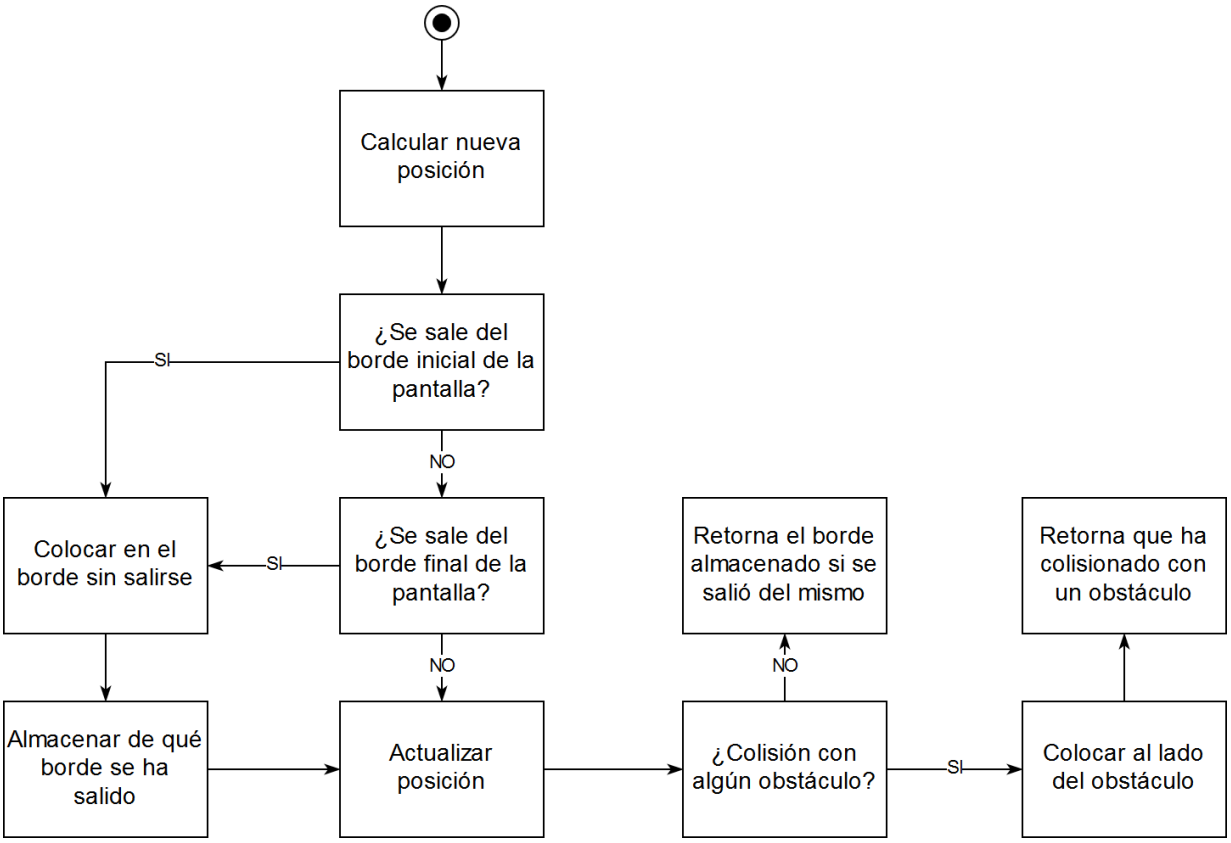

Figura 6.18.: Segunda versión del algoritmo de movimiento.

sincronización vertical en esa iteración del bucle.

Para tener este doble buffer se tienen que reservar 32KB lo que supone la mitad de memoria de la máquina, este motor de juego no esta pensado para usar bancos de memoria con los que ampliar el espacio disponible pero una opción que quizás se podría contemplar para siguientes iteraciones es la compresión de imágenes ya que gran parte del espacio se lo llevan los sprites y el tileset.

Estos dos bloques de memoria de vídeo se sitúan en la posición por defecto, 0xC000, y la posición 0x8000, juntos forman 2 bloques adyacentes de 16KB cada uno, reservando desde la posición 0x8000 hasta la 0xFFFF. Esto puede dar problemas ya que en un principio la pila se sitúa en la posición 0xC000, si usamos estos 2 buffers en estas posiciones de memoria es muy probable que el buffer en 0x8000 sobrescriba la pila. La solución a este problema no es otro que cambiar el puntero a pila a 0x8000 evitando que se sobrescriba, sin embargo, es importante tener en cuenta que solo podemos cambiar los valores del puntero si se ha deshabilitado el firmware de la máquina previamente.

Este cambio no sólo supone intercambiar los buffers y saber en cuál hay que dibujar sino que también se necesita borrar una entidad en función de su anterior posición ya que está hace referencia al buffer que ya no se está mostrando en ese momento, como se vio

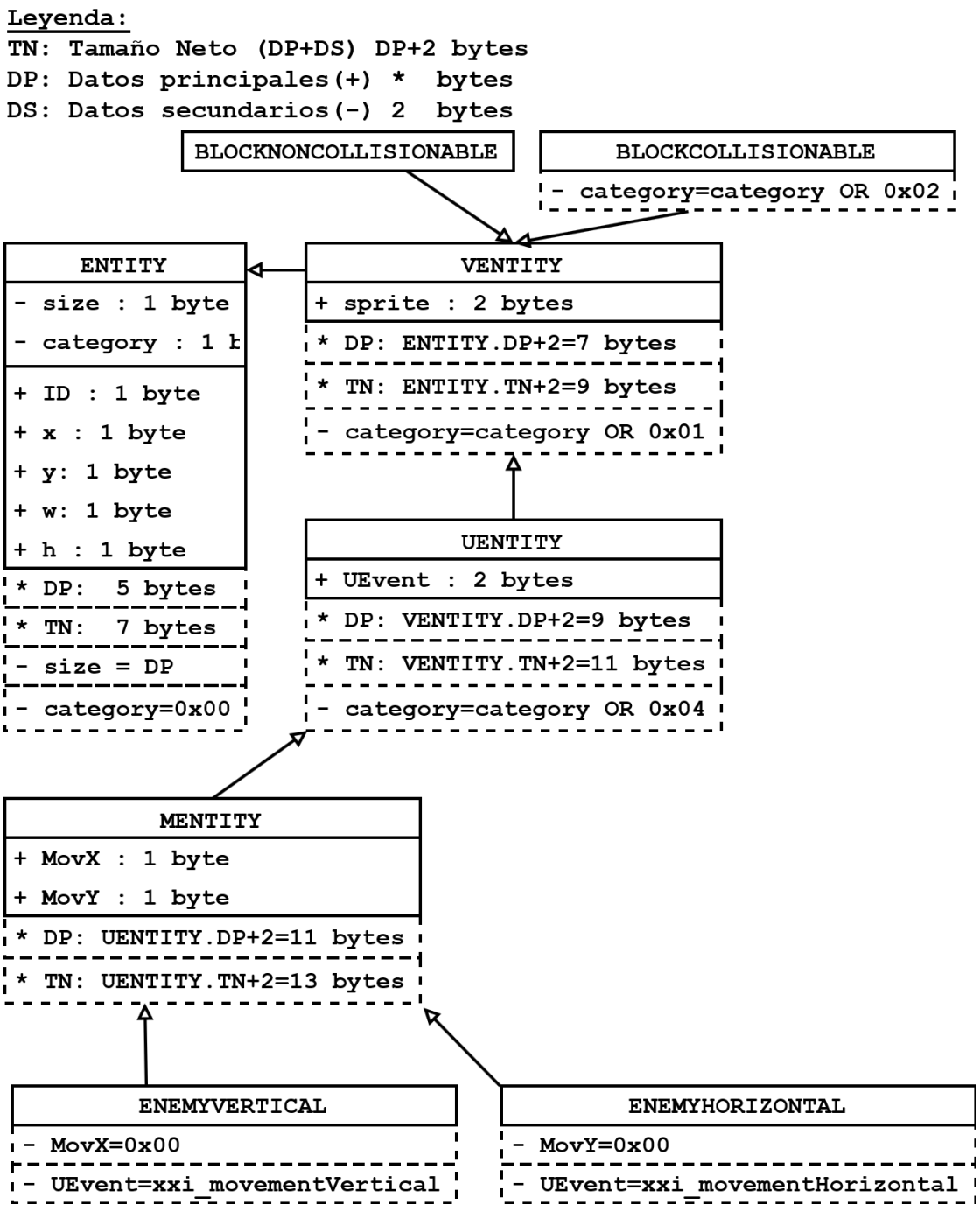

Figura 6.19.: Diseño de las entidades con el añadido de "Moveable Entity (MENTITY)" con los enemigos.

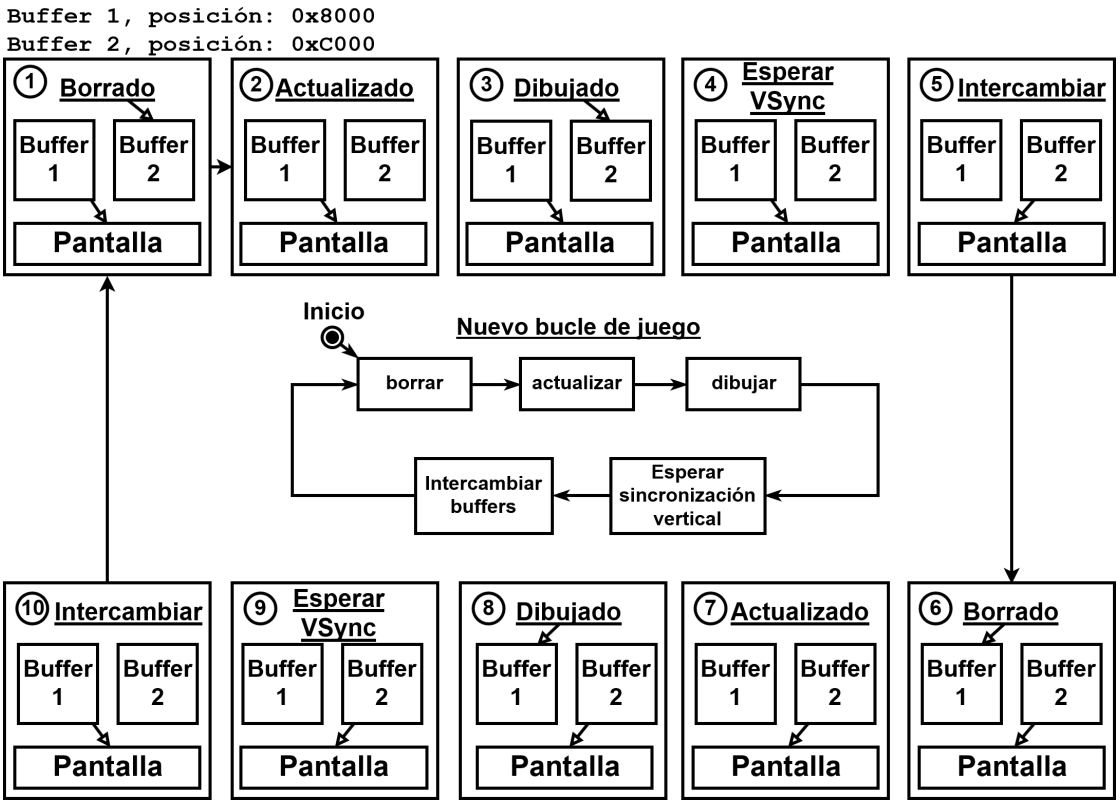

Figura 6.20.: Diseño del bucle de juego con doble buffer y un ejemplo de su funcionamiento.

en la Figura 6.20. Todo ello requiere que añadamos dos campos en una entidad, la última posición en los dos ejes del espacio, la pregunta es en cuál entidad, por una parte la más acorde será la entidad visible por otra la entidad básica también es un buen candidato, esto es así debido a que cada vez que una entidad se mueve se tiene que actualizar la última posición en ese momento, sin embargo, el algoritmo de movimiento está pensado para ser usado desde las entidades básicas hasta las entidades móviles. Hay dos opciones, la primera es usar la entidad visible relegando el algoritmo a entidades movibles que a su vez también son visibles, con el problema de no poder usarlo para otras entidades o tener que duplicar código con pequeños cambios, y la segunda que es usar la entidad básica, la cuál, simplifica la solución del problema y el algoritmo queda relegado a todas las entidades pero con un precio, el espacio, ya que si hay muchas entidades que no se visualicen tener dos bytes por entidad almacenando la última posición es cuanto menos inútil y un desperdicio de memoria, es por esta desventaja que se escoge la primera opción, ya que si no necesita movimiento para entidades básicas no ocupa tanta memoria.

Si cada vez que se mueve la entidad se actualiza su última posición eso quiere decir que cuando se pare no será actualizada lo que provoca que al volver a moverse queden rastros de la entidad cuando aún estaba parada, para solucionar esto solamente hay que

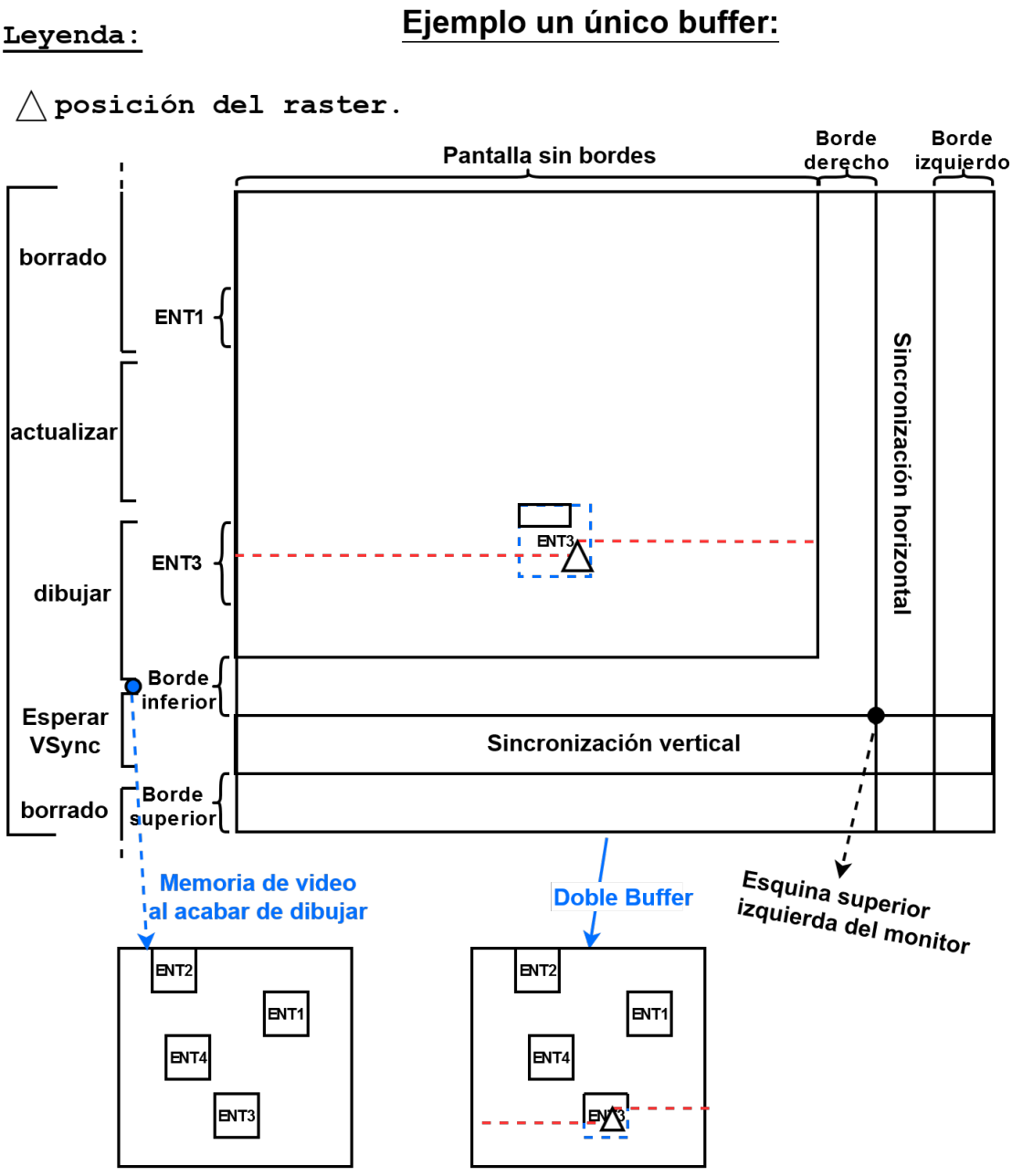

Figura 6.21.: Ejemplo de la ventaja del doble buffer.

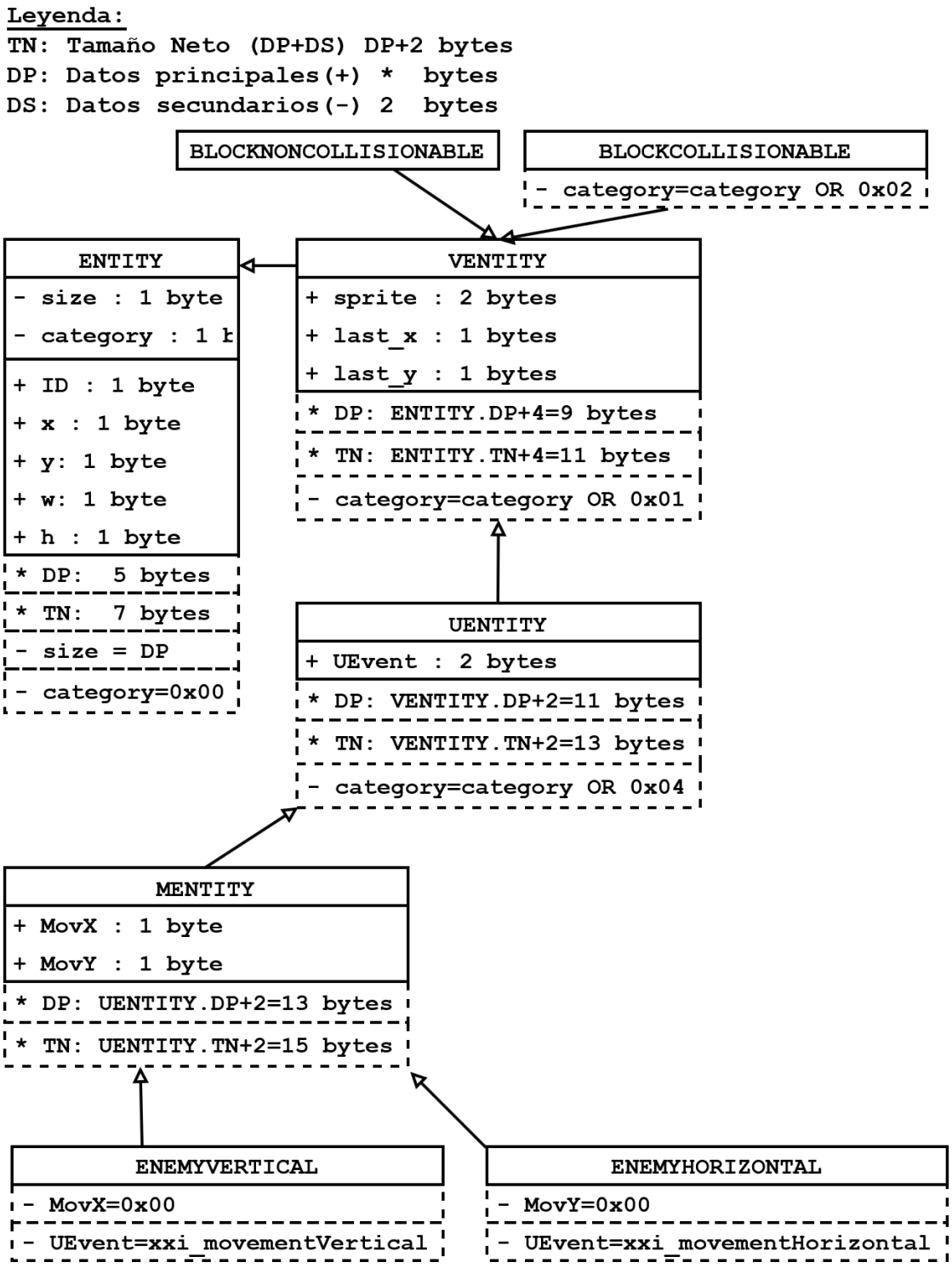

Figura 6.22.: Diseño de las entidades con la modificación en las entidades visuales.

actualizar la última posición con la actual cuando deja de ser útil, es decir, justo después del borrado, además este diseño no contempla que la entidad cambie de tamaño porque puede generar problemas.

6.2.3. Objetos

Los objetos son elementos que pertenecen al jugador que además se pueden llegar a coleccionar, usar y eliminar de un inventario pudiendo haber hasta objetos consumibles de una única unidad o varias unidades y permanentes que se mantienen en el inventario desde la primera vez que los recoges. Aplicada esta idea al motor los objetos son entidades que al colisionar con el jugador aumentan la cantidad de ese objeto a un inventario predefinido. Los objetos solo se podrán recoger y almacenar una cantidad aunque se espera poder dar las herramientas al usuario para poder usarlos.

Inventario

El inventario es el lugar donde se almacenan la cantidad de los objetos que recoge el jugador, independientemente de que sean consumibles o permanentes, porque en el inventario no se almacenan las entidades sino la cantidad de ese objeto ya que si fuera el caso supondría un gasto grande de memoria, por defecto habrá una cantidad máxima de 10 objetos y cada objeto puede tener como máximo 255 de cantidad porque cada hueco del inventario es un byte, por lo tanto el inventario ocupará un máximo de 10 bytes. En principio todos los objetos pertenecerán únicamente al jugador pero esto puede cambiarlo el usuario que puede asignar huecos a enemigos mediante código si lo necesita.

Inventario				
Array de Cantidades	OBJ1_Cantidad	OBJ2_Cantidad	$\cdots$	OBJ10_Cantidad

Tabla 6.4.: Ejemplo del Inventario

Objetos recogibles

Para realizar objetos recogibles se necesita la nueva categoría que mencionamos anteriormente, los colisionables, el pertenecer a esta categoría implica que cada vez que se colisionen dos entidades se accionarán dos eventos pertenecientes a cada entidad, de esta manera si se quiere tener mucha más lógica al colisionarse solo hay que añadir eventos y asociarlos a las entidades correspondientes.

Las entidades móviles se modifican para contener un evento, porque al moverse son las entidades capaces de accionarlos, entonces por ahora los objetos recogibles formarán parte de las entidades móviles que no son capaces de moverse pero en el futuro puede que el usuario tenga la opción de agregarle una inteligencia artificial primitiva para que el jugador lo tenga que atrapar.

Los objetos recogibles además tienen tanto la cantidad que aumenta o disminuye al inventario como el lugar donde está el hueco a modificar del inventario. Al ser capaz de

aumentar o disminuir con el byte de cantidad se usa el bit más significativo del mismo como valor de signo dando la posibilidad de modificarlo con valores entre -128 y 127. El usuario introducirá el hueco al que pertenece con un número del 1 al 10 y se precalculará su posición al compilar el proyecto, esto se ha hecho así para no tener que calcular constantemente la posición del inventario cuando se está en el bucle de juego en pos del rendimiento. Una decisión de diseño que se planteó es que si se comprueba que tienen eventos vacíos antes de ejecutarlos, se tarda más que si directamente se llama a un evento en que lo único que se hace es retornar, por lo tanto ese evento de retorno es puesto por defecto para las entidades móviles.

Por último, se comprueba si colisionan las entidades entre todas. No se pudo plantear como los obstáculos porque se accionaban cuando se movía una entidad, si dos entidades se mueven la una hacia la otra se podrían accionar los dos eventos dos veces al chocar y si solamente se accionaba un evento de las dos entidades si uno de ellos no se mueve no afecta a la otra. Aún así hay mejores maneras que comprobar las colisiones todos con todos, por ejemplo, la malla de colisiones que consiste en dividir la pantalla en una malla para que cada entidad pertenezca a una celda de la malla, al moverse puede salirse de esa celda y pasar a otra, y al comprobar las colisiones solamente se comprueban las entidades en sus celdas actuales y en las vecinas. Este algoritmo que parece muy atractivo con sus promesas de gran rendimiento pero tiene sus limitaciones, el tamaño de un celda debe ser aproximadamente de la entidad de mayor tamaño para evitar que puedas estar en 3 celdas en un eje al mismo tiempo y si en el juego las entidades suelen estar cerca entre sí puede empeorar el rendimiento en lugar de mejorarlo.

6.2.4. Mapa de durezas

Un mapa de durezas consiste en una copia del mapa de tiles que en lugar de tener identificadores de tiles para cada hueco de 4x4 píxeles tiene almacenadas las durezas del mapa de tiles, siendo las durezas tiles que provocan una acción determinada en las entidades, en este caso solamente habrán colisiones pero se podrían hacer otros tipos. Las durezas del motor son muros intraspasables como las entidades que están presentes en el array de la categoría obstáculos por lo tanto tienen un comportamiento parecido.

Al tener un comportamiento parecido para añadirlo como característica del motor hay que situarlo en el mismo lugar que los obstáculos, el algoritmo de movimiento como se puede observar en la Figura 6.25.

Los tiles tienen un tamaño de 4x4 píxeles y 2x4 bytes. Esto provoca que si una entidad tiene capacidad de moverse lo suficientemente rápido pueda saltar varios tiles de durezas fácilmente. Para evitarlo se calcula el rastro que la entidad deja tras de sí en su movimiento con su última posición usándolo como parte de la propia entidad, de esta manera se podrá ver qué tiles ocupa y ha ocupado la entidad para entonces comprobar si esos tiles eran colisionables o no y colocar la entidad junto a esos tiles si es necesario.

Un mapa de durezas ocupa lo mismo que un mapa de tiles, es decir, el fondo de una habitación. Por lo tanto tener un mapa de durezas almacenado por habitación aunque rápido es un gasto de memoria grande que se podría usar en meter el doble de habitaciones. Es por ello que se optó por un mapa de durezas auxiliar que se rellenase

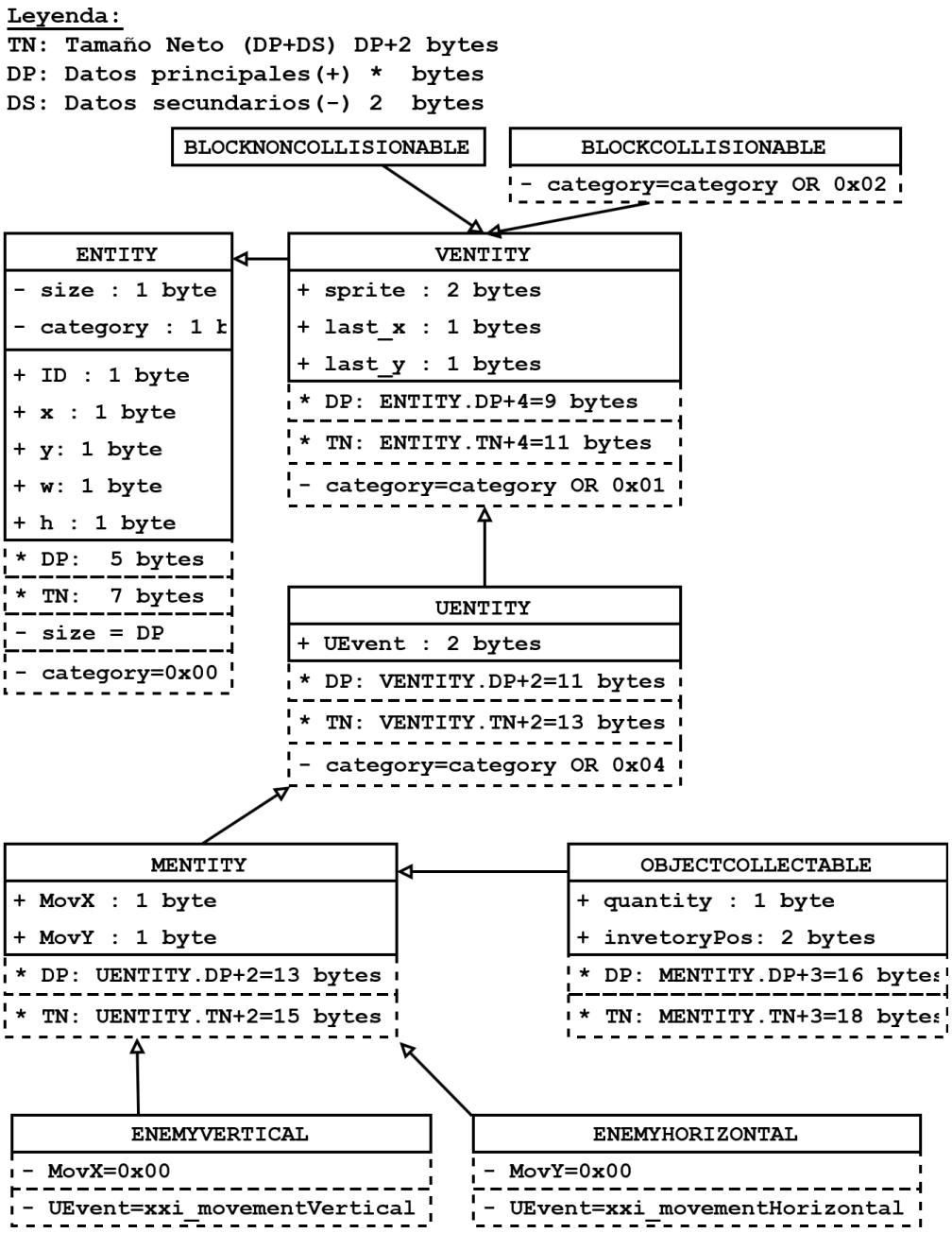

Figura 6.23.: Diseño de las entidades con la modificación en las entidades móviles y el añadido de los objetos.

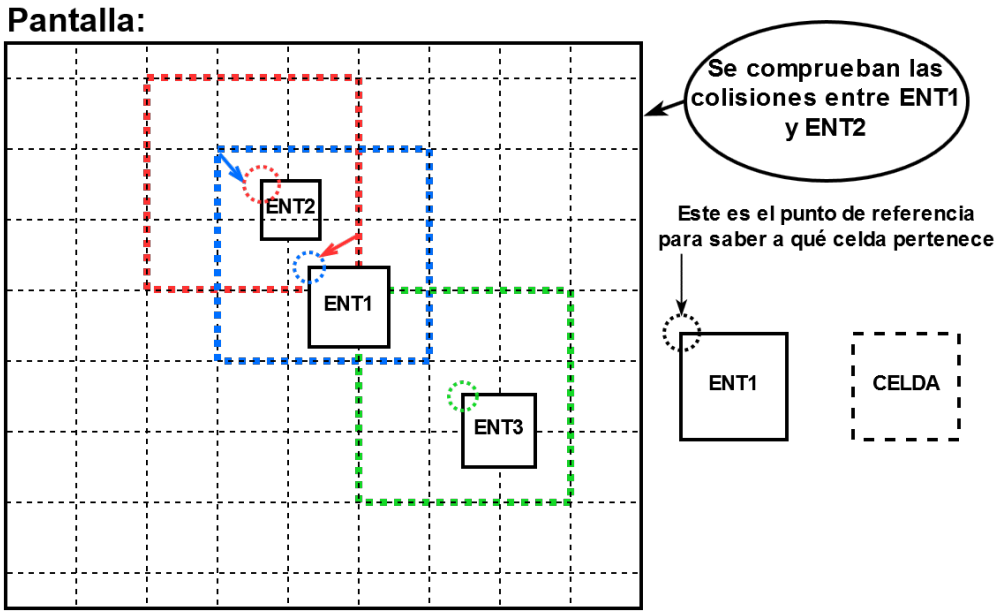

Figura 6.24.: Ejemplo de malla de colisiones.

al cargarse la habitación. Es una opción que ralentiza el proceso de cargar habitaciones pero que una vez cargada puedes comprobar los tipos de dureza a la misma velocidad que si hubiera un mapa de durezas por habitación. Existe la posibilidad de directamente usar el mapa de tiles como mapa de durezas pero esto es bastante ineficiente ya que si, por ejemplo, el usuario decide que 80 tiles tienen el mismo tipo de dureza hay que comprobar que coincida cada uno de esos 80 tiles con el tilemap provocando que se ralentice el rendimiento general del juego pero no la carga de habitaciones.

El algoritmo de movimiento funciona para cada eje de manera independiente, por lo tanto también nuestro algoritmo de colisión con durezas, a raíz de ello, hay una optimización que se puede hacer si te mueves por solo un eje. Como se puede ver en la Figura 6.27 al moverse en un eje si colisionas para un tile de una fila/columna no necesitas volver a revisar el siguiente tile de la misma fila/columna porque al colocar la entidad a su lado hace imposible que colisione con alguno de ellos.

Otra optimización que se podría realizar es empezar por el lugar más cercano a la última posición, de esa manera en el momento que colisionase se podría parar la comprobación pero por falta de tiempo se planteará para próximas iteraciones.

6.2.5. Exportación

La exportación no es una característica del motor como tal sino un programa externo que junta todos los elementos que ha creado el usuario, enemigos, niveles, mapas... etc. y se le da al motor con toda la información y parámetros necesarios para ejecutar el juego, además de dar error en caso de que se haya introducido algún dato no esperado. Todo

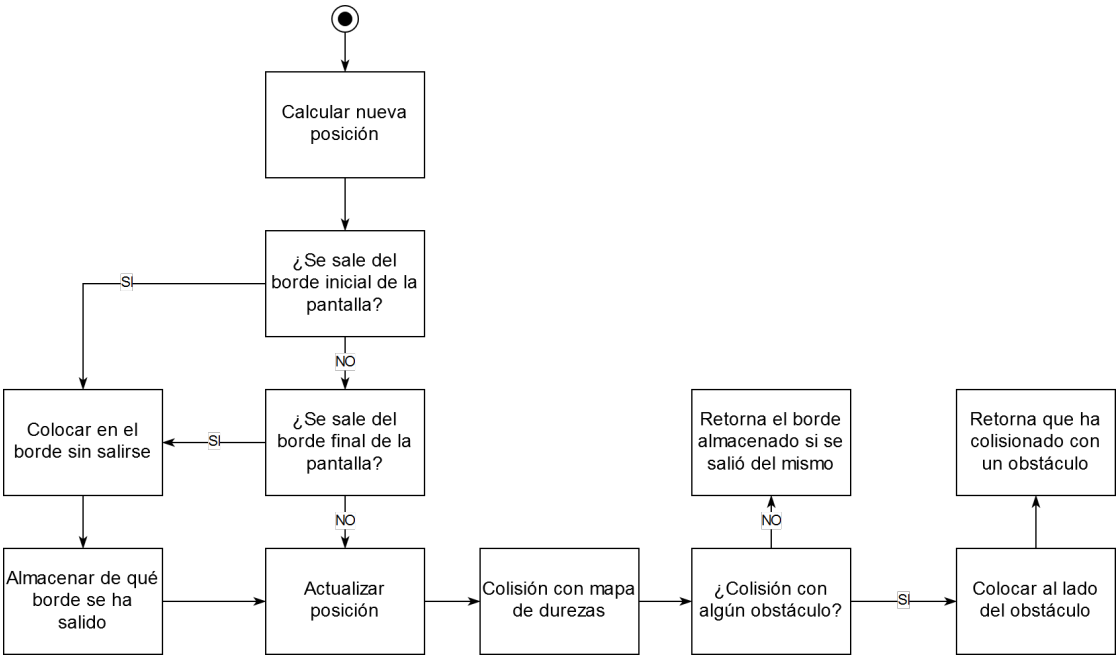

Figura 6.25.: Cambios realizados en el algoritmo de movimiento.

ello se debe hacer con unas herramientas predefinidas y sacando el máximo partido a las mismas para favorecer al usuario.

Usuario

El usuario es la persona que va a usar el motor para desarrollar videojuegos de visión cenital en el Amstrad CPC 464. Su motivación principal es evitar tener que implementar toda la lógica inherente a este tipo de juegos y tener que saber todos los entresijos del motor para hacerlo funcionar, solamente dar datos al exportador y si no hay ningún problema en la exportación y compilación que empiece a funcionar su juego. El usuario va a realizar todas estas tareas a través de un editor con el que podrá crear gráficos y añadirles propiedades que equivalgan comportamientos en el juego, por ejemplo, crear un sprite y asociarlo a un tipo de entidad o añadirle eventos y variables propias.

Tiled

Tiled es un editor de niveles que ayuda en el desarrollo de videojuegos. Su principal característica es la capacidad de editar mapas de tiles o tilemaps, colocar imágenes en cualquier posición de la mapa, añadir atributos auxiliares a cualquier elemento y crear objetos predefinidos. Permite trabajar fácilmente con mapas de tiles rectangulares, isométricos e incluso hexagonales, y también permite un conjunto de tiles de una imagen con todos los tiles o varias imágenes individuales. El propósito de este editor es ser flexible e intuitivo es por ello que aprenderlo es más sencillo y ahorra mucho

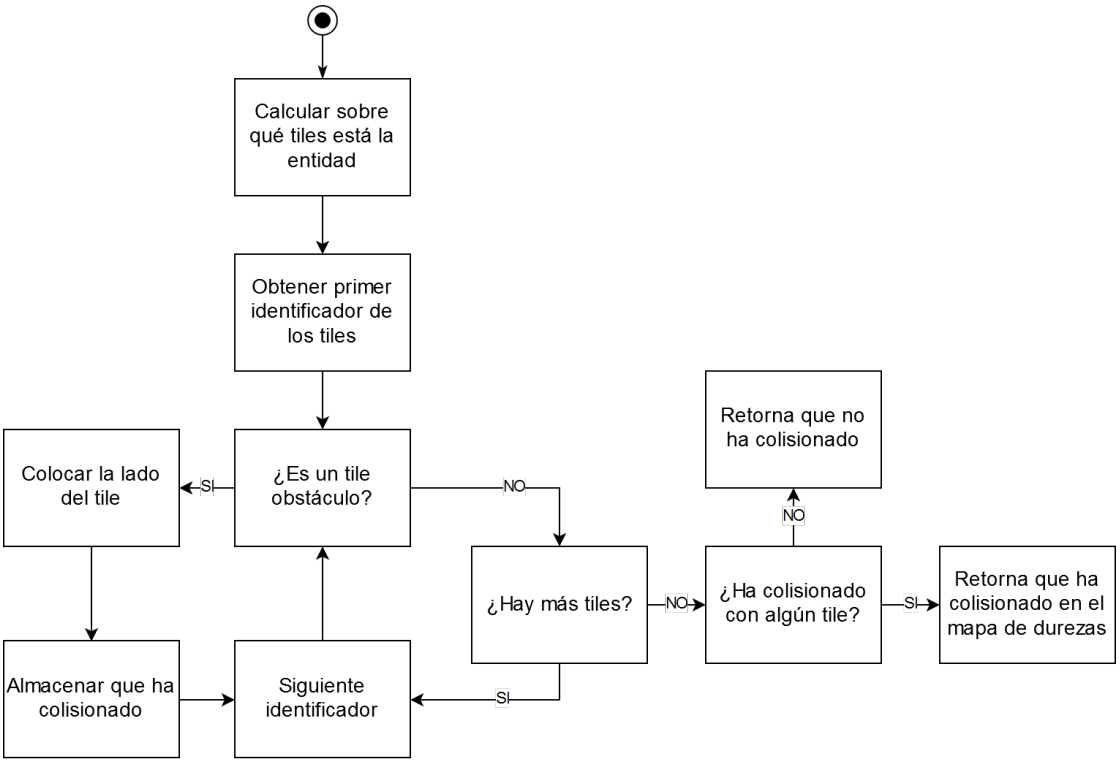

Figura 6.26.: Algoritmo de colisión con durezas.

trabajo.[Tiled Documentation Writers, 2018] Lo ideal sería realizar un editor específico para el motor pero se planteará para próximas iteraciones.

El usuario tendrá una plantilla con la mayoría de los datos configurados y conociendo su propósito sabrá donde se debe modificar si es pertinente, sin embargo nunca se deben modificar los nombres de los atributos de los tipos de objetos predefinidos ya que esos mismos serán los que se busquen para encontrar los valores en la exportación.

Para tener dos o más elementos iguales uno de ellos será el principal y el resto tendrán el mismo nombre precedido por *copyof_* aunque los únicos datos que se usen de la copia sea la posición en X e Y. Esto se hace de esta manera para ahorrar memoria y no tener que repetir elementos sin necesidad. Si dos elementos tienen el mismo nombre y no están precedidos de *copyof_* el exportador dará error, como también si no existe el elemento al que hace referencia después del *copyof_*.

Los eventos son los nombres de las rutinas integradas en el motor o que haya hecho el propio usuario. Todos los eventos definidos por el usuario deben estar en un archivo llamado “user.s” que deben ser globales y que podrán ser usados en Tiled introduciendo su nombre. Si se requiere una serie de parámetros a una rutina definida en el motor, el usuario deberá crear un evento con esos parámetros introducidos.

En el Anexo A.1 se profundiza más en Tiled, sus similitudes a los componentes del motor y en cómo crear tu juego desde el mismo Tiled.

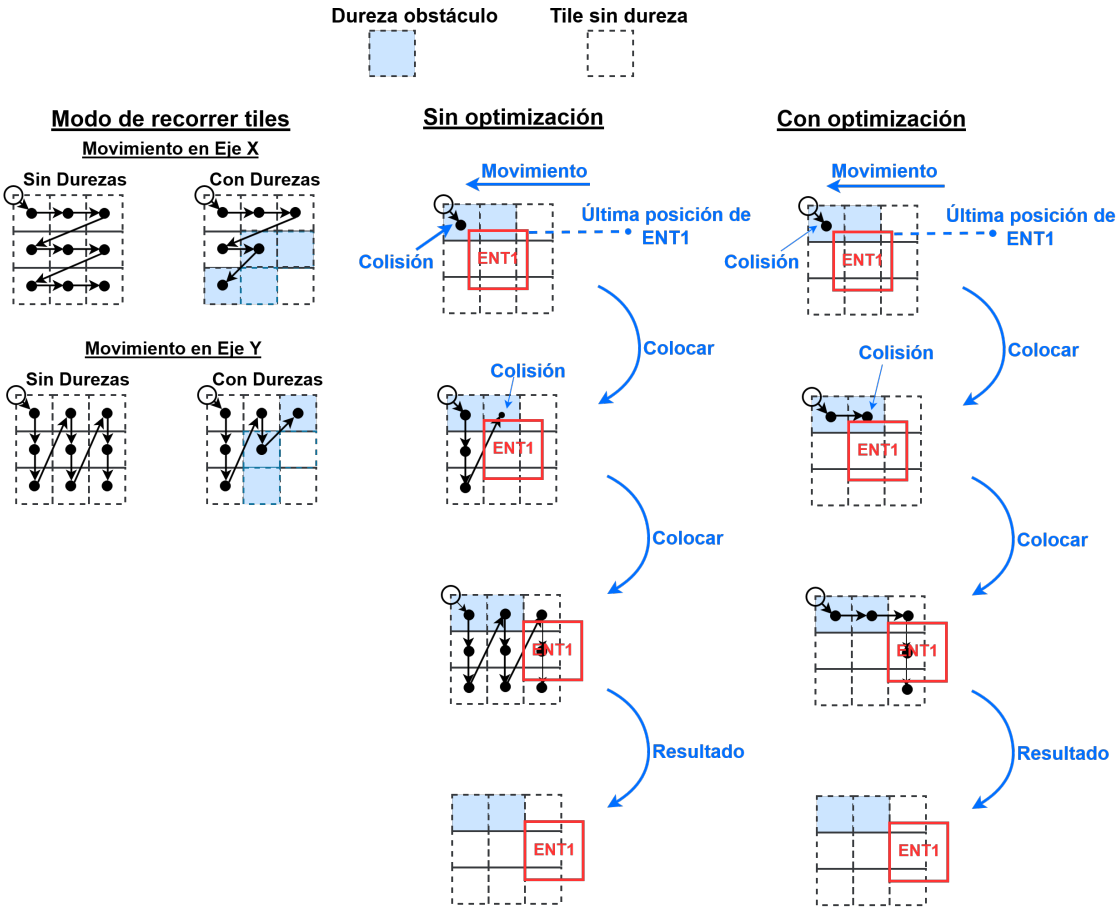

Figura 6.27.: Algoritmo de colisión con tiles y ejemplo de optimización al colisionar.

Restricciones

Ahora hay que definir y listar todas las restricciones que se han hecho a lo largo de todo el proyecto e introducirlos en el exportador para que el usuario pueda ver sus errores como para que el motor no reciba valores que lleven a comportamientos indefinidos.

- **Paleta:** Se tienen únicamente 16 colores de 24 a escoger por el modo 0, una vez elegidos en la paleta, no se pueden cambiar. Se tienen que hacer los gráficos acordes a esta elección.
- **Entidades en memoria:** Las entidades nunca pueden estar guardadas entre 0x0000 y 0x00FF además de 0xFF00 y 0xFFFF.
- **Fuera de la pantalla:** Nunca salirse del tilemap definido, puede llevar a modificar el propio código del programa.
- **Límite de habitaciones:** Si todos los mapas juntos generan más de 127 habita-

ciones el exportador tiene que generar un error en el caso de que no haya dado error antes por memoria.

- **Nombres de eventos:** No introducir el prefijo *xxi_* ya que es el que usa el motor para sus rutinas globales y puede llegar a haber coincidencias.
- **Tileset:** Solamente debe haber un tileset.
- **Arrays de IDS:** Generar IDs para mapas, habitaciones y entidades.
- **Límite máximo:** El número máximo de un identificador por tipo (entidad/habitación/mapa) que puede haber es 127, es decir, no puede haber más de 127 de cada uno de esos tipos.
- **Movimiento:** El movimiento en X y en Y en las entidades móviles tiene que tener valores entre 0 y 127 ya que son dos bytes que tienen en cuenta su signo para desplazarse en cualquier dirección.
- **Inventario:** No se puede escoger un hueco en el inventario menor que 1 o mayor que 10, el inventario solo tiene 10 huecos. Y la cantidad que puede tener un objeto
- **Tilemap:** Cada tile debe ser de 4x4.

6.3. Iteración 3. Exportador

En la iteración anterior no se pudo terminar la exportación y no se empezó la aceleración de entidad ni las animaciones. Esta iteración se centrará principalmente en el desarrollo del exportador y después de comprobar su correcto funcionamiento si el tiempo de esta iteración lo permite se implementará o mínimo diseñará la inserción de música en el motor y las dos características que no se pudieron diseñar previamente.

6.3.1. Diagrama de carpetas

El exportador necesita un entorno en el que ejecutarse, archivos específicos y el proyecto de CPCtelera que va a modificar para ello se tendrá que ejecutar en una carpeta que cumpla todas estas condiciones de una estructura de carpetas definida.

En este diagrama de carpetas se tienen todas las imágenes como es de esperar en la carpeta *img/*, todos los archivos de formato TMX de Tiled cada uno representando un mapa estarán en la carpeta *map/* el exportador solo captara los archivos con la extensión del formato, *.tmx*; los archivos en ensamblador que se exporten se almacenarán en *exp/* enlazándose al código fuente del proyecto de CPCtelera en su carpeta *src/*, los tilemaps serán creados por el exportador en *src/tiles* mientras que las imágenes serán creadas en *src/sprites* por CPCtelera lo que implica exportar un archivo de configuración con todas las imágenes a convertir y enlazar este archivo al encargado de trabajar con imágenes en CPCtelera. Toda esta configuración se ve reflejada en la Figura 6.29. El exportador es ejecutado en la carpeta principal del proyecto por lo que la manera de acceder a

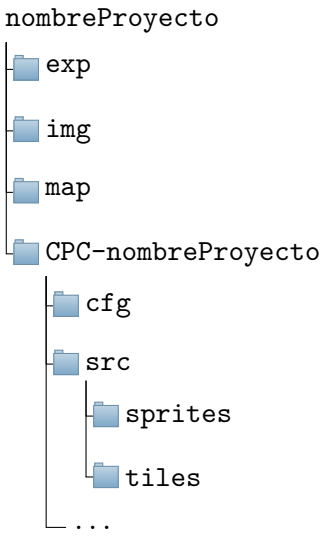

Figura 6.28.: Diagrama inicial de carpetas del proyecto

la carpeta de CPCtelera puede cambiar dependiendo del nombre que haya escogido el usuario, para evitar este problema al acceder a la misma al exportador se le tiene que proporcionar el nombre del proyecto por parámetro.

6.3.2. Formato Tiled

Para poder leer los archivos del Tiled en formato TMX, se usa una librería externa encargada de dar acceso a todos datos como las capas, objetos, rutas a las imágenes y atributos de los mismos guardados en ese archivo y así poder trabajar con ellos.

La librería es *tmxlite* [Marchant, 2018] creada por Matt Marchant que a diferencia de otras librerías es muy completa pudiendo leer cualquier información que se presente en los archivos del formato. Además *tmxlite* incluye una serie de ejemplos muy explicativos. Todo ello bajo una licencia que obliga entre otros términos a mencionar si se le realizan modificaciones a la librería y es un dato a tener en cuenta ya que se necesita hacer un cambio en la librería en lo que los atributos se refiere.

La librería guarda cada una de los atributos en un vector STL[cplusplus.com, 2018b], es decir, guarda un elemento tras de otro por lo que si se quiere buscar el nombre de un atributo se tiene que recorrer elemento por elemento lo que puede consumir tiempo de la exportación, es por ello que se optó por un mapa desordenado STL que es más rápido que el vector[cplusplus.com, 2018a], pudiendo ahorrar tiempo y desarrollo. Por esta razón siempre que se necesite buscar en un conjunto numeroso de objetos se usará el mapa desordenado.

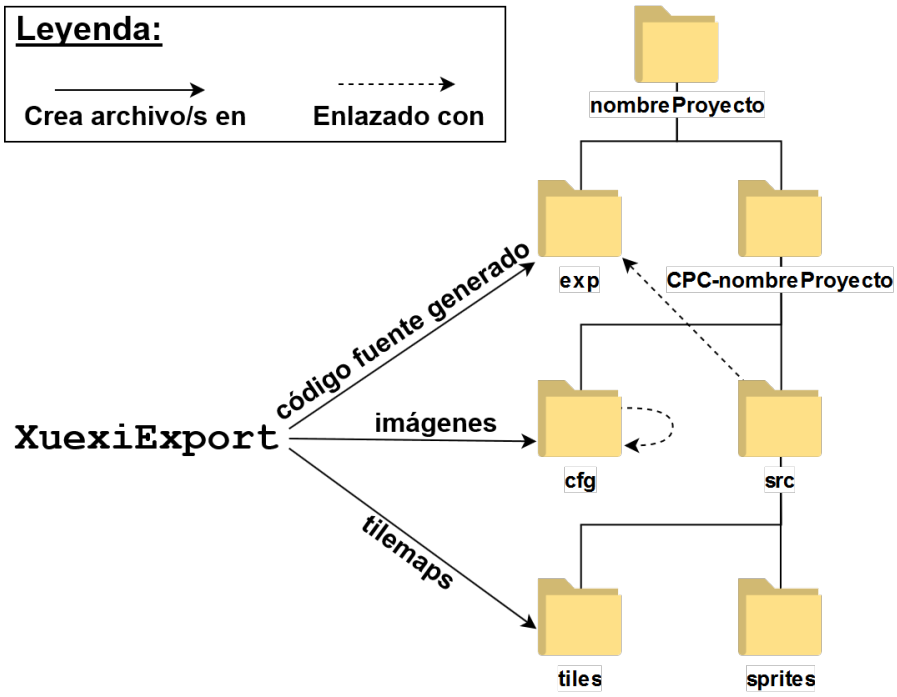

Figura 6.29.: Diagrama de exportación.

6.3.3. Componentes del flujo de trabajo

En primer lugar se analizarán los componentes que contribuyen al exportador y el propósito que poseen, ya que son gran parte de cómo funciona el flujo de trabajo, no se puede entender uno sin el otro. Cada componente se traduce a un código fuente capaz de ser leído por el compilador o a un archivo de configuración. Estos componentes se pueden ver reflejados en la Figura 6.30.

El componente entidad o *Entity* y sus derivados contienen también los mismos datos que sus respectivos componentes en el código fuente, manteniendo la herencia que poseían como en la Figura 6.30 traduciéndose en su definición respectiva.

El componente de mapa o *Map* contiene los mismos datos que los mapas en el código fuente como en la Figura 6.12 traduciéndose en la definición de un mapa y el array que representa el conjunto de habitaciones.

El componente de habitación o *Room* como el mapa contiene los mismos datos que las habitaciones en el código fuente como en la Figura 6.12 traduciéndose en la definición de una habitación sin su tilemap ya que muchas habitaciones pueden tener el mismo.

El componente de Tilemap, como su mismo nombre indica, tiene toda la información visual necesaria para mostrar el fondo de la habitación. En pos de ahorrar memoria nunca habrán tilemaps repetidos siempre que el usuario duplique o haga el mismo tilemap con Tiled, ya sea con el mismo o diferente nombre, el exportador lo comprueba y le proporciona a las habitaciones un mismo nombre de tilemap que se verá reflejado en el

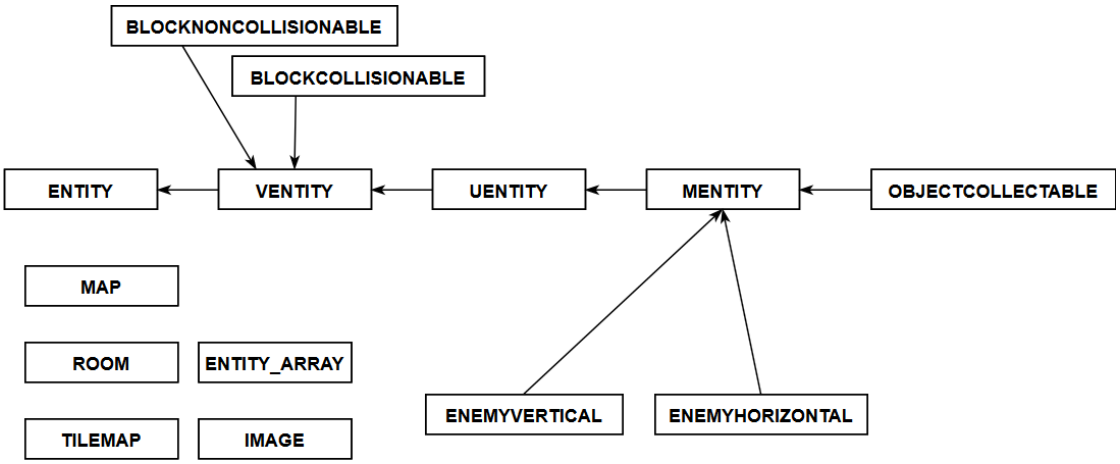

Figura 6.30.: Diagrama de exportación.

código fuente generado.

Es a través del diseño de este componente que se decide retirar el proceso de que el usuario haga un *copyof_* cada vez que quiera duplicar algún elemento que tenga el mismo nombre debido a que todos los elementos pueden ser diferenciados sin ambigüedad a excepción de las capas de objetos. Además el usuario no debe encargarse de la optimización de memoria, el exportador se encarga de ello y es por ello que cada vez que se agrega un elemento si ya existe no se añade usándose el ya existente.

El componente array de entidades o *ENTITY_ARRAY* es creado a través de una capa de objetos. Este componente contiene cada uno de los identificadores de las entidades existentes. La capa de objetos es ambigua porque no tiene ninguna propiedad única como un identificador por lo que si hay dos capas diferentes pero con el mismo nombre no se puede saber a cuál se refiere en las habitaciones como se puede ver en el Anexo A.1.

El componente imagen o *Image* contiene el nombre de una imagen que es extraído del fichero TMX quitando la ruta dejando únicamente el nombre de la imagen. Se espera siempre que esa imagen esté en la ruta antes mencionada, *img/*, de esta manera se evitan problemas con las rutas en los ficheros TMX porque Tiled guarda las rutas en posición relativa.

6.3.4. Diseño del flujo de trabajo

El exportador comienza con un bucle que comprueba si existe el directorio *map/*, si no es el caso se para la ejecución y se manda un mensaje de error pero si existe se obtienen todos los archivos que hay en el directorio y cuando alguno de ellos tenga la extensión *.tmx* se procede a procesar el mapa que una vez terminado pasa al siguiente así hasta que recorre todos los archivos del directorio. Se comprueban errores como que haya un mapa inicial, que haya al menos un mapa, una habitación y una entidad, además por último comprobando si no existe alguna habitación asociada a un array de entidades inexistente. Este exportador acumula todos los errores posibles para entonces mostrarlos al final de

la exportación para no hacerle perder el tiempo al usuario re-ejecutando constantemente como se ve en la Figura 6.31.

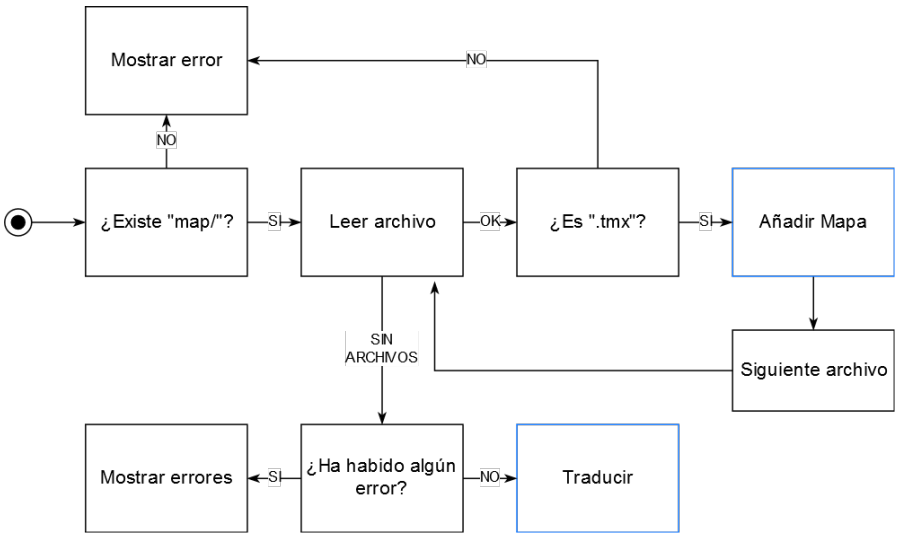

Figura 6.31.: Diseño del trabajo principal del exportador.

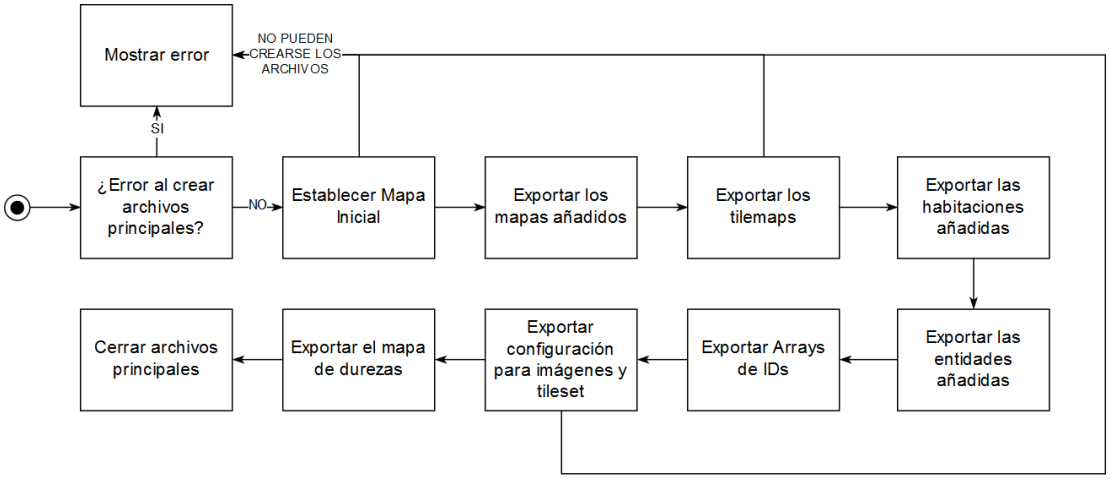

Figura 6.32.: Diseño del proceso de traducción del exportador.

Si no se ha producido ningún error se empieza el proceso de traducción visible en la Figura 6.32 y consiste en convertir todos los datos obtenidos y procesados en código fuente y archivos de configuración. El único error que puede dar en esta parte es no poder crear los archivos de la exportación por no tener permisos suficientes o que no existan los directorios necesarios y si no ha habido ningún problema se muestra al usuario. Todo ello se puede observar en con el trabajo principal y en la traducción.

Cuando se añade un mapa hay una serie de pasos a seguir y en este texto se profundiza lo visto en la Figura 6.3.4. Al extraerse los atributos del mapa se le proporciona un identificador como al resto de componentes y si supera el límite permitido de mapas, habitaciones o entidades salta un error y se continua con el procesamiento del fichero TMX aunque si no da error también a la hora de extraer si falta algún atributo obligatorio también saltará error, aunque la mayoría de atributos tienen valores por defecto. Si no salta error se leen cada uno de los tilesets que representan o el tileset porque solo puede haber uno o las imágenes, la manera de diferenciarlos en la cantidad de tiles, ya que las imágenes solamente tienen un tile. Esto provoca que en el remoto caso de que suceda, no se pueda usar un tileset con solo un tile. Si es imagen se añade, si no existe ya, y pasa al siguiente tileset si lo hay. Si es *el* tileset hay que comprobar si no esta ocupada esa posición o si en la misma se encuentra el mismo tileset ya que varios mapas usarán el mismo, en caso contrario se mostrará error, si no da error se pregunta si es el mapa inicial en caso de que lo sea se obtienen las durezas de cada tile y se añaden al contenedor para posteriormente exportarlos, si no lo es continua el proyecto. Posteriormente se añaden habitaciones, entidades y arrays de entidades que consisten en buscar duplicados y evitarlos si los hay, para entonces calcular la posición de las habitaciones del mapa consistiendo en únicamente en colocarlas en una matriz que representa el mapa. Para acabar se añade el mapa al conjunto a traducir si no ha ocurrido ningún error.

La mayoría de errores surgen de las restricciones de la iteración anterior es por ello que no se ha profundizado en todos los detalles de los mismos. Si se desea ver que podría dar más errores se recomiendo mirar el Anexo A.1.

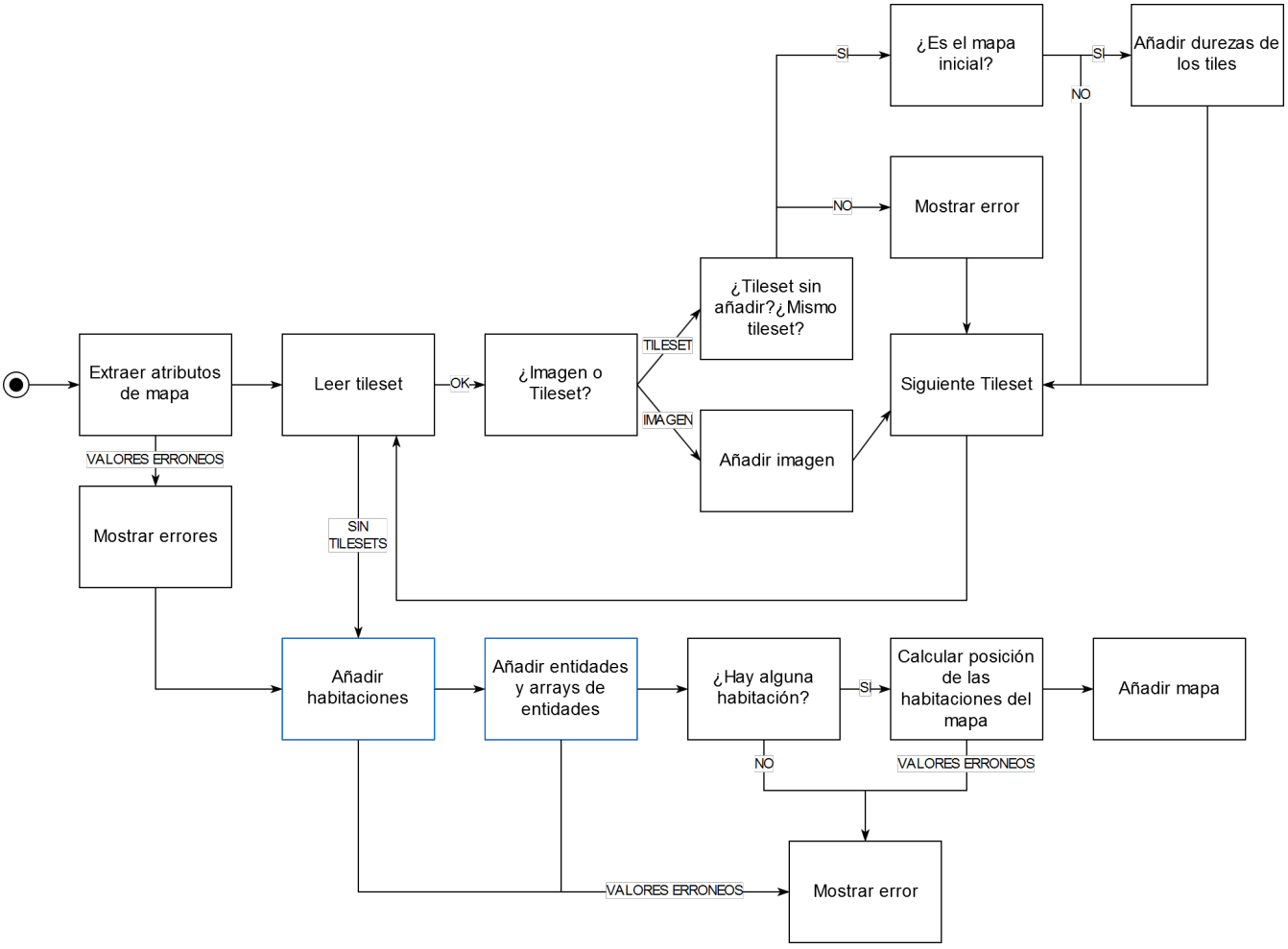

Figura 6.33.: Diseño del proceso añadir un mapa del exportador.

7. Conclusiones

En todo este proyecto se han realizado una serie de características las cuales han contribuido a tener un motor y exportador con funcionalidades básicas aunque aún se puede y debería introducir muchas más características y optimizaciones.

Características como poder crear un mapa en Tiled, añadirlo a un directorio y que automáticamente se exporte en forma de código fuente y archivos de configuración con todas las habitaciones y entidades en el mapa, para entonces que se pueda ver reflejado en la ejecución del juego. Además hay un abanico de entidades por las que elegir. Se espera el uso por parte del usuario de la entidad móvil, los bloques no colisionables, los bloque sí colisionables, enemigos con patrones de movimiento vertical-horizontal y objetos coleccionables. Si el usuario quiere profundizar puede usar la entidad básica, visible y actualizable en el editor aunque para aprovecharlas se debe usar código adicional.

Los mapas tienen una habitación de inicio, donde se aparece al cargar el mapa, y una de destino cuyo comportamiento dependerá del usuario, además el traspaso de una habitación a otra es hecho por el motor automáticamente.

Los objetos recogibles que cada vez que son tocados por el jugador añaden o quitan una cierta cantidad definida por el usuario a uno de los 10 huecos del inventario.

En la parte más técnica se consiguió el doble buffer consiguiendo que se borrarán las entidades visibles en el buffer adecuado con los datos adecuados y el mapa de durezas.

Por último el propio exportador que se encarga de cubrir todas las restricciones y transformar nuestros mapas de Tiled en el juego deseado a través de *Makefile*.

Aún así faltan muchas características que se quisieron implementar pero no se llegó a poder como aceleración del movimiento de las entidades, animaciones, música e invertir tiempo en optimizaciones, distando de un motor tan completo como AGD, sin embargo, como ya se ha reiterado varias veces se planteará en futuras iteraciones.

A través de este proyecto se han aprendido muchos conceptos y conocimientos que no se habrían asimilado de no ser por haberse embarcado en este proyecto. El Amstrad CPC sin ir más lejos se ha investigado, debido a que para programar un motor completamente en lenguaje ensamblador se necesita conocer la máquina al nivel más profundo lo que lleva a conocer su procesador, el Zilog Z80, y su código máquina ya que es la herramienta con la que se comunica al resto de componentes. Al tener que conocer el ensamblador en profundidad se han diseñado muchas optimizaciones de memoria y rendimiento que no hubieran sido posibles en otro lenguaje de mayor nivel ni en otra máquina al no tener que enfrentarse a estos problemas de rendimiento y memoria.

Además en cada iteración se ha realizado una planificación que normalmente caía en subestimar la cantidad de trabajo que podía suponer una tarea u otra, o también errores que gastan más tiempo del necesario pero al hacer un registro de todas estas tareas, iteración a iteración se ha ido comprendiendo las implicaciones y situaciones que

conlleva el desarrollo software de un producto.

Uno de los apartados que más se necesita mejorar es ponerse en el papel del usuario, qué es lo que quiere, cómo lo quiere y por qué, aún se puede mejorar mucho los diseños con este enfoque.

El motor se llama “Xuexi” en chino significa aprender/estudiar, porque eso es lo que se buscaba con este proyecto aprender, crecer como desarrollador y llevar este producto hasta el final. Aún queda mucho por mejorar, diseñar e implementar pero lo más importante es la experiencia ganada de este proyecto.

Bibliografía

[Amstrad International, 1984] Amstrad International (1984). Amstrad PLC Company Profile. Web. http://www.amstrad.com/about/profile.html.

[ClrHome, 2018] ClrHome (2018). *The Z-80 Instruction Set*. http://clrhome.org/table/.

[cplusplus.com, 2018a] cplusplus.com (2018a). *Unordered Map STL*. http://www.cplusplus.com/reference/unordered_map/unordered_map/.

[cplusplus.com, 2018b] cplusplus.com (2018b). *Vector STL Reference*. http://www.cplusplus.com/reference/vector/vector/.

[Godden, 1984] Godden, B. (1984). *CPC464 Firmware*. AMSOFT. https://archive.org/details/CPC464_Firmware_1984_AMSOFT.

[Joseph C. Nichols, 1984] Joseph C. Nichols, Elizabeth A. Nichols, P. R. R. (1984). *Microprocesador Z-80, Programación e interfaces*. Publicaciones Marcombo, S.A.

[Landley, 1998] Landley, R. (1998). *The Z80 Microprocessor*. http://landley.net/history/mirror/cpm/z80.html.

[Marchant, 2018] Marchant, M. (2018). *tmxlite*. https://github.com/fallahn/tmxlite.

[Smith, 2014] Smith, T. (2014). You're NOT fired: The story of Amstrad's amazing CPC 464. *The Register*, pages 1,2. https://www.theregister.co.uk/2014/02/12/archaeologic_amstrad_cpc_464/.

[Sommerville, 2005] Sommerville, I. (2005). *INGENIERÍA DEL SOFTWARE*. PEARSON EDUCACIÓN, S.A., Madrid, seventh edition.

[The Cambridge Centre for Computing History, 1978] The Cambridge Centre for Computing History (1978). Zilog Z-80 Microcomputer System. *Centre for Computing History*. http://www.computinghistory.org.uk/det/12157/Zilog-Z-80-Microcomputer-System/.

[The Cambridge Centre for Computing History, 1984] The Cambridge Centre for Computing History (1984). Amstrad CPC 464. *Centre for Computing History*. http://www.computinghistory.org.uk/det/2805/Amstrad-CPC-464/.

[Tiled Documentation Writers, 2018] Tiled Documentation Writers (2018). *Introduction About Tiled*. http://doc.mapeditor.org/en/stable/manual/introduction/.

[Zilog Inc., 2006] Zilog Inc. (2006). *Z80180 Microprocessor Unit Product Specification*. IXYS company. `http://www.zilog.com/docs/z180/ps0140.pdf`.

A. Anexo I

A.1. Tiled

Tiled ofrece muchas posibilidades y aunque no se van a repasar todas, sí las que se vean con utilidad para el motor. Si se ejecuta el programa y se obtiene esta pantalla inicial de la Figura A.1.

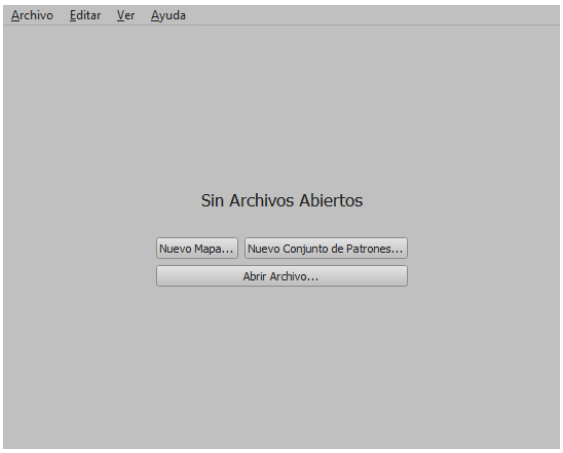

Figura A.1.: Pantalla inicial de Tiled.

Tiled trabaja tanto con mapas de tiles como con conjuntos de patrones también llamados tilesets, en el momento que se tiene inicialmente un nuevo mapa hay que ajustar una serie de parámetros, en concreto los que se pueden apreciar en la Figura A.2.

La orientación es el parámetro que nos indica con qué tipo de tiles se va a trabajar, ortogonal, es decir, tiles rectangulares, isométrico o hexagonal, en este caso se escogen Ortogonal ya que el motor trabaja en entornos 2D en visión cenital.

El formato de la capa de patrones, es decir, el formato de la capa del mapa de tiles, puede escoger la codificación en la que se guardará el mapa, se puede incluso comprimir, el CSV es la codificación que guarda toda la información del tilemap en un fichero de texto, se escoge este porque el mapa se postprocesa en el momento de la exportación haciendo inútil comprimirlo o codificarlo de otra manera.

El orden en el que se guardarán capa uno de los identificadores del mapa, en nuestro caso Derecha Abajo porque el monitor del Amstrad recorre la pantalla de esa misma manera.

Por último nuestros tiles serán de 4x4 píxeles porque los hemos usado antes así con CPCtelera, por ello el mapa será de 40 tiles de ancho y 50 de alto porque el modo 0 es

de 160x200 de resolución y al dividirse en ese tamaño de tiles resulta en el ancho y alto de mapa antes mencionado.

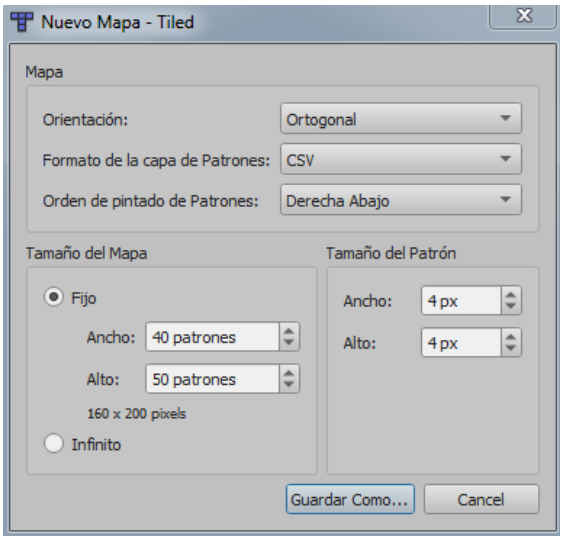

Figura A.2.: Pantalla inicial de Tiled.

Una vez creado el nuevo mapa tenemos acceso a la pantalla que hay en la Figura A.3, que se divide y subdivide en varios bloques mencionados a continuación.

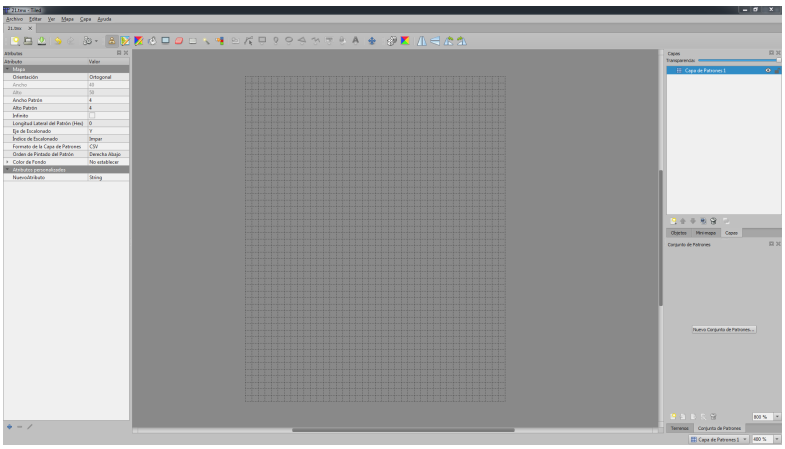

Figura A.3.: Pantalla principal de tiled.

La pantalla central que es nuestro mapa, ahí podemos introducir tiles, objetos e imágenes en una capa de patrones y ver cómo resultarían visualmente en pantalla.

La pantalla lateral izquierda que tiene los atributos del elemento que se está seleccionando en ese momento, aquí se pueden añadir atributos personalizados con los que podemos dar valores a las entidades con sus eventos, movimientos, también a los mapas con quién es el mapa inicial, su punto de inicio y final, y las habitaciones.

La pantalla lateral inferior tiene para añadir tilesets de diferentes tamaños, da varias opciones para hacer el tileset a nuestro gusto sin embargo al añadir un nuevo tileset como se ve en la Figura A.4, los únicos valores que modificará el usuario serán el nombre y la ruta del tileset, además se restringió anteriormente que en el motor solamente hay un tileset por lo que en ese tileset debe estar todos los tiles que necesite el usuario.

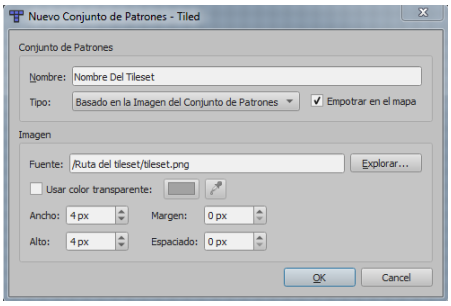

Figura A.4.: Creación de un nuevo tileset.

La pantalla lateral derecha en la parte superior tiene 3 pestañas, objetos, Mini-Mapa y Capas. La primera pestaña de tiene las capas de objeto en modo árbol con sus respectivos objetos que se explicarán posteriormente, la segunda pestaña muestra una imagen con todo lo dibujado como vista general y con el que el usuario se puede mover rápidamente a través del mapa, y la última pestaña contiene cada una de las capas posibles.

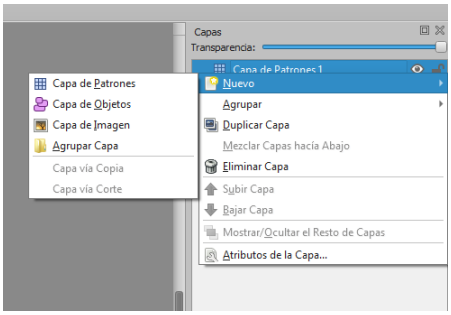

Figura A.5.: Tipos de capa.

Tiled funciona por capas, se pueden agrupar poner una encima de otra, quitar y añadir a gusto del usuario, cada tipo capa tiene un propósito diferente. Las capas de patrones son tilemaps sobre los que se insertan los tiles en pos de editarlo, las capas de objetos ofrecen elementos fuera del tilemap a los que se les puede añadir atributos personalizados y tipos predefinidos y por último la capa de imagen que, valga la redundancia, sirve para introducir imágenes aunque al motor no le sirva por ahora el usuario puede usarlo si tiene en su mano bocetos y desea insertarlos como imágenes para tener una base sobre la que trabajar.

Al ser de 40x50 tiles cada capa de patrones representa una habitación, para tener habitaciones adyacentes solamente habría que coger 2 capas y ponerlas adyacentes ajustándo-

lo a la rejilla final de la otra habitación como se puede observar en la Figura A.6.

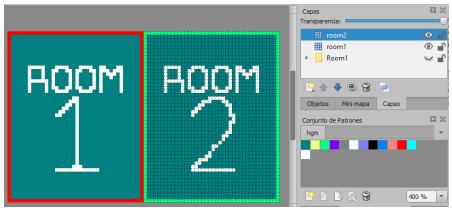

Figura A.6.: Como se colocan 2 habitaciones adyacentes para que en el mapa también se encuentren así.

Para hacer que se ajuste a la rejilla de la capa de patrones la colocación de elementos hay que seleccionar la opción de vista en la Figura A.7, si no se selecciona esta opción el exportador tendría que aproximar el valor de la posición y el tamaño a lo que más se acerque a la resolución que soporta el motor porque Tiled guarda las posiciones y tamaños como valores decimales, provocando que no se muestre una visión fiel de lo que se vería en el motor, por ello se tiene que ajustar a rejilla obligatoriamente y colocar los elementos dentro de la rejilla de la capa de patrones para tener posiciones y tamaños válidos para el motor y la exportación.

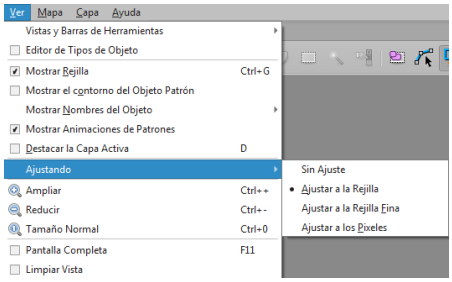

Figura A.7.: Ajustar los elementos a la rejilla.

Los objetos son elementos que tienen posición X e Y, ancho y alto, tipo, e identificador, además cada tipo que le asignes a un objeto tienen una serie de valores específicos a los que se les debe dar un valor si no tienen ya uno por defecto coincidiendo a nivel práctico con las entidades. En principio cuando se genera el primer objeto tienes una serie de atributos predefinidos pero ninguno personalizado, al crear los tipos y asignarlos se crearán automáticamente esos atributos sin necesidad de estar creando los mismos atributos para cada nuevo objeto.

Para poder crear tipos hay que activar el “Editor de Tipos de Objeto” como se ve en la Figura A.9, estos nuevos tipos harían referencia a cada una de las entidades que se han creado y agilizaría el proceso de crear objetos.

En la figura A.10 se puede observar como se crearía un tipo pudiendo hasta usar un código de colores y en la Figura A.11 también se vislumbra como afecta el añadirle un tipo a sus atributos personalizados los atributos se no pueden eliminar a no ser que dejarás de ser ese tipo pero en cambio si que se pueden añadir.

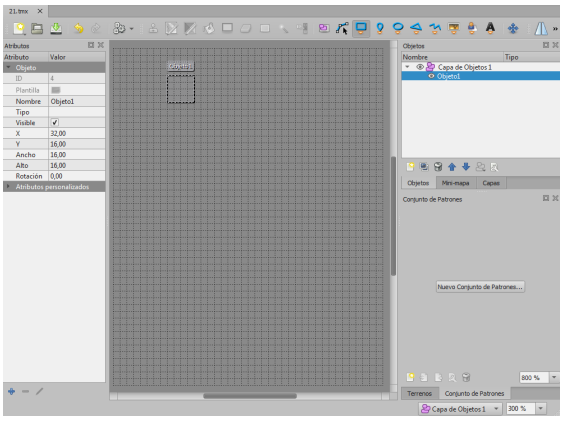

Figura A.8.: Objeto recién creado.

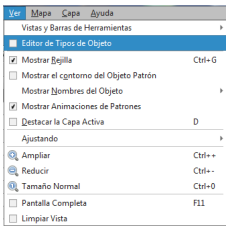

Figura A.9.: Menú para activar el Editor de Tipos de Objeto.

Sin embargo gran parte de las entidades son visibles, por lo que la pregunta que se genera es cómo se añadiría un sprite o si se tendría que pasar la ruta por atributo, pues resulta haber un tipo de objeto que visualmente puedes añadirle una imagen siempre que sea parte de un patrón, en este caso se añadiría un nuevo conjunto de patrones que sean exactamente del mismo tamaño de la imagen y si se selecciona ese tile con este tipo de objetos se pueden añadir esos mismos objetos.

Todos los elementos de Tiled pueden tener atributos e incluso los tiles, es por ello que cuando se desea que añadir un tile al mapa de durezas se debería acceder al .[Ed]itar el conjunto de patrones.[a]l lado del icono de la papelera porque si no se está en este entorno no se permite añadir ni quitar atributos. Una vez seleccionado un tile se le puede agregar un tipo, si es un mapa de durezas seguramente en el tipo haya un valor que indique si son obstáculos o no.

Los mapas en Tiled son un mapa por archivo en el formato de tiled (.tmx) y en el motor tienen definidos una serie de datos que pueden ser calculados a través de Tiled, uno de ellos es la estructura del mapa ya que saber qué habitaciones hay en cada hueco del mapa se contemplan a base de colocar conjuntos de patrones adyacentes el uno al otro, sin embargo otros tendrá que introducirlos el usuario a su gusto:

- **EventFinalRoom(Evento Final):** Un evento accionado por llegar a la posición final.

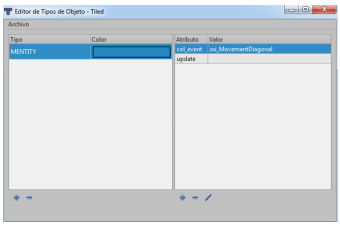

Figura A.10.: Editor de Tipos de Objeto con un tipo creado.

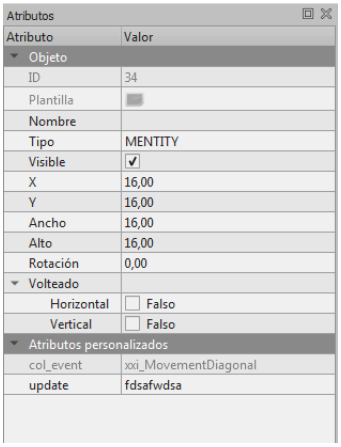

Figura A.11.: Objeto recién creado.

- **InitMap(Mapa Inicial):** Un atributo de booleano que indica si es el mapa por el que empezará el juego, si más de un mapa tiene esta variable activada se mostrará un error en la exportación.

- **Palette(Paleta):** Si es el mapa inicial se tiene adicionalmente un string con toda la paleta que se usará en todo el juego en este formato $X\ X\ X\ \ldots\ X$ con 16 valores de color del firmware en lugar de las X.

Las habitaciones siendo una capa de patrones aunque solo tiene estos atributos adicionales es importante definirlos:

- **ObjectLayerName(Capa de Objetos):** Un string que contenga el nombre de la capa de objetos que usará la habitación siendo una capa de objetos el conjuntos de entidades que habrá en esa habitación. El usuario puede hacer una capa de objetos para cada habitación, sin embargo, si se desea ahorrar en memoria se puede poner la misma capa de objetos a dos habitaciones y el exportador se encarga de que se use la misma, también funciona al revés y es que 2 habitaciones iguales, es decir, una copia de la otra pueden tener diferentes capas de objetos. Hacer un *copyof_* con una capa de objetos solamente sirve para que el usuario pueda visualizar las habitaciones en Tiled.

- **InitRoom:** Un booleano que indica si es la habitación inicial.
- **EndRoom:** Un booleano que indica si es la habitación final.

  Si una capa de objetos tiene el mismo nombre que otra puede llevar a comportamientos indefinidos.

Las entidades tienen almacenadas enteramente sus datos en los objetos que a su vez están en la capa del mismo nombre con cada uno de sus tipos se necesitan diferentes atributos, aquí se repasan todos los tipos con sus respectivos atributos vistos en la Figura 6.23:

- **ENTITY:** La entidad básica, solamente aquí se repasarán los atributos comunes que tendrán todas las entidades y uno específico ya sean dados por defecto en Tiled o no en siguientes entidades solo se recalcarán los atributos personalizados.
  - **Nombre(ID):** Llamado en el diseño ID y en Tiled, nombre, es el identificador que le da el jugador a la entidad.
  - **X:** Posición en el eje de coordenadas, dado por Tiled.
  - **Y:** Posición en el eje de coordenadas, dado por Tiled.
  - **Ancho(w):** Anchura en el eje de coordenadas, dado por Tiled.
  - **Alto(h):** Altura en el eje de coordenadas, dado por Tiled.
  - **Category(Categoría):** Categoría asociada a la entidad puede no ser necesario en todas las entidades, atributo personalizado. Se recomienda usarlo solamente si se desea personalizar en profundidad.
  - **isPJ:** Booleano que indica si la entidad es el personaje principal. Solo puede haber uno por capa de objetos.
- **VENTITY:** La entidad visible, este tipo y el resto de tipos deben de ser construidos con un objeto hecho a partir de una imagen como se ve en el Anexo A.1.
  - **Category(Categoría):** Categorías a las que se añade la entidad. Se recomienda usarlo solamente si se desea personalizar en profundidad.
- **UENTITY:** La entidad actualizable, se le asocia un evento que a cada iteración del bucle de juego se activa en el bloque de actualización. Puede ser necesario para tareas rutinarias de la entidad.
  - **UEvent:** Evento de actualización.
  - **Category(Categoría):** Categorías a las que se añade la entidad. Se recomienda usarlo solamente si se desea personalizar en profundidad.
- **MENTITY:** La entidad móvil, este tipo se asocia a entidades que se van a mover por pantalla libremente.
  - **UEvent:** Evento de actualización.
  - **MovX:** Cantidad de píxeles de 2 en 2 que se puede mover en el eje X.

- **MovY:** Cantidad de píxeles que se puede mover en el eje Y.
- **CollEvent(Evento de colisión):** Evento accionado al colisionar con otra entidad móvil.
- **Category(Categoría):** Categorías a las que se añade la entidad. Se recomienda usarlo solamente si se desea personalizar en profundidad.

- **BLOCKCOLLISIONABLE:** Bloque colisionable, al chocarte con él se para el movimiento y posicionándote a su lado sin seguir colisionando. No tiene atributos personalizados.
- **BLOCKNONCOLLISIONABLE:** Bloque no colisionable, no ofrece ningún tipo de lógica, solamente es decoración.
- **ENEMYVERTICAL:** Enemigo que se mueve de arriba hacia abajo y de abajo a arriba.
  - **MovY:** Cantidad de píxeles que se puede mover en el eje Y.
  - **CollEvent(Evento de colisión):** Evento accionado al colisionar con otra entidad móvil.
- **ENEMYHORIZONTAL:**
  - **MovX:** Cantidad de píxeles de 2 en 2 que se puede mover en el eje X.
  - **CollEvent(Evento de colisión):** Evento accionado al colisionar con otra entidad móvil.
- **OBJECTCOLLECTABLE:** Objeto que al colisionar con él añade o quita la cantidad en su posición del inventario.
  - **Quantity (Cantidad):** Cantidad que añade o quita al inventario.
  - **InventoryPos (Hueco de inventario):** Número del 1 al 10 que indica a qué hueco de los 10 del inventario afecta.

Los tiles cada tile puede ser un tipo de dureza aunque por ahora solo hay para colisiones de tipo obstáculo:

- **Obstacle(Obstáculo):** Atributo booleano que cuando se activa al exportar se tiene en cuenta el identificador del tile como dureza.