Endless Story AmstradCpc Game

Grado en Ingeniería Informática

Trabajo Fin de Grado

Autor:
Robert Esclapez García

Tutor/es:
Francisco José Gallego Duran

Septiembre 2018

1. Justificación y Objetivos

El juego que se pretende realizar es un tributo a una gran saga de juegos titulados *Zelda*. El juego se va a desarrollar para la plataforma Amstrad CPC (versión 64kBytes de memoria RAM). La CPU que tendremos que utilizar es un Z80, la cual tiene una potencia de 4MHz. Dada la carencia de potencia, en comparación con los procesadores actuales, el desarrollo se llevará a cabo en C y ensamblador, buscando siempre la optimización. Se llevará a cabo un estudio de necesidades y de soluciones posibles aplicables, además de una justificación.

El juego que se plantea desde un principio es un juego con las siguientes características:

- Sistema de mapas basado en un sistema de *tiles*.
- Renderizado mediante doble *buffer* de pintado.
- Entidades móviles y entidades objeto.
- Diferentes entidades móviles.
- Sistema de *scroll* regulado al tamaño del mapa actual.
- Desacoplamiento total de la visualización del juego del personaje principal.
- Capacidad de cambiar de mapa simulando un mapa gigante seccionado.
- Posibilidad de alternar entre mapas de diferentes dimensiones.
- Sistema de compresión que permita incrementar exponencialmente el contenido del juego.
- Sistema de animación.
- Sistema de actuación por estados. MEF
- Capacidad de añadir *checkpoints*, para reaparecer desde un punto dado.
- Sistema sencillo de creación de mapas.
- Sistema de colisionado entre entidades y haciendo uso del mapa.

2. Agradecimientos

A Jose Francisco Gallego Duran, mi tutor del trabajo de fin de grado, por solventar cualquiera duda que haya podido tener.

Y a Julia García Martínez por el desarrollo artístico de los mapas que se pueden encontrar en el juego.

3. Índices

3.1 Índice de contenido

3.2 Índice de ilustraciones

4. Cuerpo del documento

4.1. Introducción

Endless Story es un juego que se ha llevado a cabo bajo unas grandes restricciones, tanto computacionales como espaciales, y por este motivo se han utilizado técnicas para mejorar tanto el tiempo de cálculo, como el ahorro de memoria. Es un tributo a una gran saga de una gran compañía, que empezando a producir juegos para la consola NES ha vendido un total aproximado de 80 millones de unidades de esta saga.

El objetivo del juego es rescatar a tu princesa, para ello el héroe tendrá que abrirse paso a través de obstáculos y de muchos enemigos que se va a encontrar en su camino. El paso del héroe se verá obstruido por unas puertas selladas, que tendrá que abrir con la llave correspondiente. Las llaves las encontrará dentro de ciertos enemigos de las mazmorras. Por esta razón, tendrá que explorar y derrotar a todos los enemigos para encontrar las llaves y llegar hasta la princesa.

El juego está desarrollado en C casi totalmente con algunas partes en ASM. Hace uso de software ya desarrollado como es la librería CPCTelera y la rutina de compresión/descompresión Exomizer. Se ha diseñado teniendo en mente siempre la combinación de de eficiencia y generalización. Por eso muchas de las estructuras de datos que se pueden encontrar en el juego parten de una misma base aprovechando su posición en la estructura para realizar funciones generales.

Para el desarrollo del juego se han contemplado y usado técnicas bastantemente extendidas como son el desplazamiento de pantalla y un mapa basado en *tiles*. Se podrá encontrar un gestor de entidades y de mapa. Al mismo tiempo que un sistema de animación pintado y cámara.

4.2. Marco teórico

Vamos a subdividir el estado del arte en 3 aspectos:

- A la plataforma la cual el juego va destinado.
- El uso de herramientas disponibles para desarrollar el juego.
- Un análisis de posibilidad de desarrollo de videojuego.

4.2.1 Amstrad CPC

La plataforma Amstrad CPC y en nuestro caso la versión 464, dotada con 64KB de memoria RAM, está compuesta de muchos componentes. En nuestro caso nos interesaremos por el procesador y por el CRTC.

4.2.1.1 Procesador

En cuanto al procesador, trabajaremos con un Zilog Z80 con una potencia de 4 Mhz. El procesador, es un procesador de 8 bits, por lo que únicamente podemos referenciar datos de un byte de grandes. Si quisiéramos referenciar datos de más de un byte tendríamos que hacer más lecturas. El procesador, incorpora una rutina de interrupciones, que ocurrirán unas 300 cada segundo y funciona de la siguiente manera. Un procesador lee una instrucción la decodifica y la ejecuta, así con todas de una manera lineal y secuencial. Aquí es donde entran las interrupciones. Incorporan un mecanismo en el cual, cuando ocurre una interrupción se salta a una dirección previamente asignada y ejecuta ese código, para luego volver al lugar desde donde saltó, incorporando así una mecánica de pseudoparalelismo.

4.2.1.2 CRTC

Por otro lado encontramos el CRTC, que es el que nos permitirá pintar en pantalla. Las pantallas originales estaban basadas en sistemas de tubos de rayos catódicos. El CRTC será

el encargado de leer los datos de la zona destinada de la memoria RAM, que denominaremos, memoria de vídeo y pintar su valor asociado a la pantalla. El CRTC funciona de una manera asíncrona al procesador, lo que se significa que no hay espera entre los dos componentes, cada uno funciona de forma independiente, lo que puede llevar a ciertos errores que se tratarán más adelante. La tasa de refresco de la pantalla de es de 50fps, es decir cada segundo el *raster* (componente que pinta la pantalla) pasa por la pantalla pintando lo que encuentre en la memoria de video unas 50 veces por segundo. Al haber 300 interrupciones por segundo, hablamos que por cada pasada del *raster* para pintar la pantalla ocurren 6 interrupciones.

Amstrad CPC 464 posé una memoria total de 64KBytes de memoria RAM, la memoria de video destinada a la pantalla es de 16Kbytes. Cuando se va a diseñar un juego se tiene que tener en cuenta que los colores que se van a usar va a definir la resolución de pantalla que tengamos. Amstrad nos ofrece tres modos de video, y cada modo presentar unos retos y una manera de codificar los bytes de la pantalla diferente para mostrar los colores:

Modo 2:

El modo 2 nos permite tener una resolución de 640×200px con una combinación de dos colores, ya que se utiliza un bit para codificar el color. Dos posibles estados, dos posibles colores.

Modo 1:

El modo 1 nos permite tener cuatro colores, por lo que en vez de usar un bit para la codificación de colores usa dos bits ($2^2 = 4$).

Por esta misma razón, al usar el doble de bits que el modo 0 la resolución se reduce a la mitad, por lo que podemos tener una resolución de 320x200px con cuatro colores.

Modo 0:

El modo 0 nos permite una hacer uso de una paleta de 16 colores a costa de usar 4 bytes para la codificación de todos estos colores. Por lo tanto volvemos a reducir la resolución de nuestro juego a unos 160x200px.

Veamos un ejemplo de codificación en modo 1:

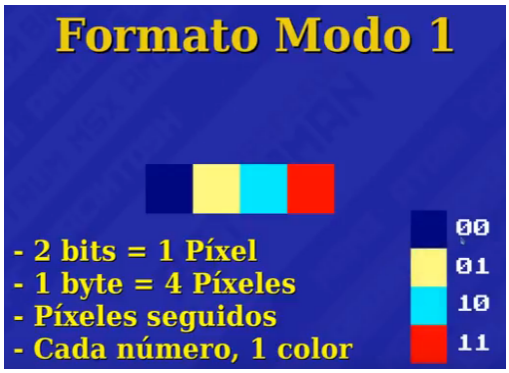

Ilustración 1: Formato Modo 1 : en Youtube ProfesorRetroman

Dado que con un byte podemos codificar 4 pixeles (8bits/2bits cada color), si queremos pintar esa secuencia de colores en pantalla, deberíamos escribir en la memoria el valor :

Ilustración 2: Codificación bits : en Youtube ProfesorRetroman

La codificación se mantiene, únicamente que se lee de derecha a izquierda. El valor azul oscuro, como se ha indicado en la imagen anterior, es el 00, y es el valor que se muestra en la posición 3 y 7 (00). El valor amarillo es 01 y es el valor correspondiente a los bit 2 y 6 (01). Así funciona el formato de bits.

El CRTC a parte de ser el encargado de de pintar en pantalla los datos descritos en la memoria de video, también se presta a la modificación de sus características. Por ejemplo, es posible modificar el ancho y alto de la pantalla, las señales de sincronización verticales y

horizontales, o el origen de la memoria de vídeo, todo gracias a la modificación que se puede realizar mediante los registros que podemos manipular desde el procesador. Podemos obtener una serie de efectos visuales, como puede ser el *overscan* (ampliar la pantalla visible usando los bordes laterales de esta) o incluso prologar la cantidad de tiempo de cálculos antes de actualizar la pantalla (doble *buffer* de pintado).

4.2.2 Juego

4.2.2.1 Bucle de juego

Todo juego se podría resumir en dos, o tres rutinas como máximo, y un bucle infinito que las ejecuta constantemente. Son nada más ni nada menos:

- **Rutina de pintado:** Pintar en pantalla todas las entidades que se crean necesarias, y si es requerido el mapa.
- **Rutina de actualización:** Actualizar el juego, ya sea con el *input* del usuario o en tiempo real de forma independiente del *input* del usuario.
- **Rutina de borrado (Opcional):** Se borra de forma selectiva ciertos elementos de la pantalla, y si es requerido se vuelven a rellenar con el fondo.

4.2.2.2 Contenido visual

En un juego 2D como fueron el 99% de los juegos desarrollados para Amstrad CPC, está compuesto por diferentes componentes visuales, como pueden los *sprites* o los mapas.

Un sprite, es la codificación de una imagen o figura en memoria. Nosotros tendríamos según el modo de video que estemos usando, una cantidad de colores, que tendrán un valor numérico asociado.

Si quisiéramos codificar el siguiente *sprite* deberíamos tener en memoria los datos correspondiente de cada píxel, para que luego sea posible pintarlo en la pantalla del juego.

Ilustración 3: Sprite de Megaman

Las animaciones o efectos visuales, se realizan de la misma forma, son un conjunto de *sprites* que pintados uno detrás de otro dan una sensación de movimiento y de realidad. En el ejemplo siguiente del juego MegaMan se puede apreciar la sensación de movimiento al pintar un *sprite* seguido de otro.

Ilustración 4: Animanción Sprite

Todo *sprite* es rectangular, por lo que para conseguir el efecto de transparencia, se suele sacrificar un color para tomarlo como transparente, así cuando se vaya a pintar y se encuentre ese valor, no se pintará nada, dejando visible el fondo.

Por otro lado encontramos el mapa o *background*. Hay varias maneras de diseñar un mapa, y según que técnica se utilice, puede tener un uso mayor a simplemente el decorativo. Encontramos lo mapas de píxeles, y los mapas basados en mapas de *tiles*.

Lo mapas de píxeles son mapas que están codificados en memoria tal cual se quieren pintar en la pantalla. Los puntos a favor es que permiten una gran complejidad de artística, ya que puedes diseñarlos con todo detalle. Por contra, al estar en memoria tal cual se van a pintar, tienen un consumo mayor de memoria.

Los mapas basados en mapas de *tiles*, es una técnica más compleja que nos puede aportar varias utilidades. La primera y más importante es el ahorro considerable de memoria por el uso repetitivo del mismo *tile*, la otra utilidad que tiene se verá más adelante y nos facilita el

sistema de colisionado de nuestro juego. Como bien indica el concepto, un mapa de *tiles* está formado de *tiles*. La técnica consiste en el desarrollo de una serie de *tiles*, que no son más que un conjunto de píxeles, ya sea de 4x4, 8x8 o como el juego lo requiera. Un *tile* es un patrón de píxeles que representará algo, a lo mejor una hierba, una caja o a lo mejor formará parte de un conjunto mayor. El conjunto de estos *tiles* diseñados se le denomina ***tileset***.

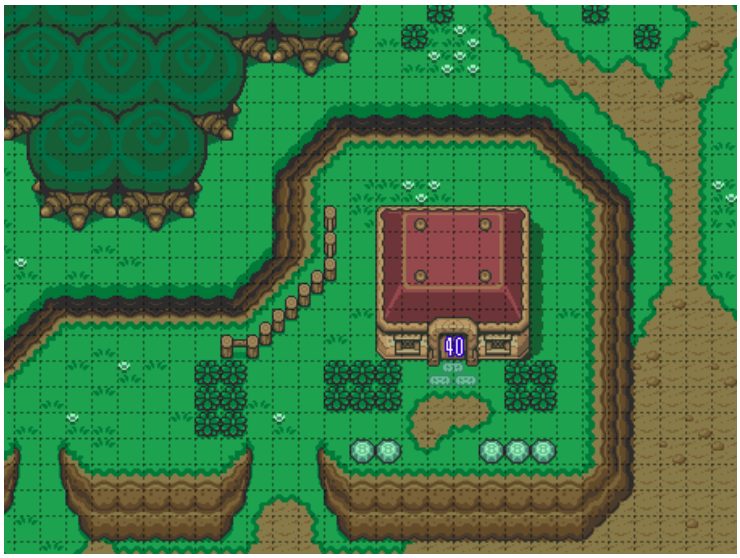

Ilustración 5: Pantalla dividida en un grid mostrando los tiles

En la imagen, el árbol grande está formado por diferentes *tiles*, mientras que las hierbas de enfrente de la casa están formados por un único *tile*.

Una vez se tienen definido los *tiles* en memoria, se trata como si fueran *sprites*. Por si por algún causal quisiéramos pintar toda la pantalla con el mismo *tile*, únicamente tendríamos ese fragmento en memoria y lo pintaríamos repetidas veces en la pantalla hasta llenarla, consiguiendo así un ahorro sustancial de memoria, ya que en vez de tener todo el mapa en memoria tenemos un *tile* de dimensiones muchas más reducidas que usamos para pintar por toda la pantalla. La idea es realizar un *tileset*, con el cual podamos realizar todos los mapas de nuestro videojuego.

Dentro del *tileset*, podemos identificar cada *tile* de forma individual por la posición en el *tileset*, siendo el primer *tile* el número 0, el segundo el 1 y así. Para poder tener nuestro *tiles* en funcionamiento necesitamos colocarlos en un *grid*, también llamado mapa. Sin usar ninguna técnica de desplazamiento de pantalla, que se comentará más adelante, el mapa será

tan grande como *tiles* podamos tener simultáneamente en nuestra pantalla, y por lo tanto dependerá de las dimensiones de ésta. Así pues, si tenemos una pantalla de 200x200 píxeles, y nuestros *tiles* son de 8x8, en el caso que no usemos una técnica denominada *scroll*, tendremos como máximo un mapa de 25x25 *tiles*. Por lo tanto, si nos disponemos a diseñar gran cantidad de mapas, cada mapa nos ocuparía 625 bytes, en el caso de que podamos codificar todos los *tiles* del *tileset* con un byte (máximo 256 *tiles*, ya que un byte transcurre entre 0 y 255), mientras que si diseñáramos los mapas a nivel de pixel, sin usar *tiles* nos ocuparía 40.000 bytes, suponiendo que nuestra pantalla sea de 200x200. El proceso de pintado es tan simple como leer de nuestra matriz de *tiles*, es decir, de nuestro mapa, leer valor a valor y pintar en su posición de la pantalla el *tile* correspondiente.

Una vez explicado cómo funcionan los juegos basados en sistemas de *tiles*, vamos a hacer un repaso de las técnicas existentes de *scroll*, los pros y los contras de cada una.

Primero, hagamos una definición de *scroll*. ***¿Qué es el scroll?***. Su traducción directa al español es *trasladar o desplazar*, pero ¿el qué?. Se hace uso de *scroll* cuando queremos realizar un mapa más grande que nuestra pantalla.

Existen juegos con *scroll*, y juego sin él. Por ejemplo, el mítico Pacman, es un juego que no incorpora *scroll*, es decir la pantalla es estática y nosotros vemos a nuestro personajillo moverse por ella. Un ejemplo de un juego con *scroll*, podría ser El Gost'n Goblins donde nuestro personaje conforme se mueve en una de las cuatro direcciones disponibles, el mapa cambia conforme a esa dirección que se ha movido, si me muevo hacia la derecha, el mapa se desplazará a la izquierda dando una sensación de avance.

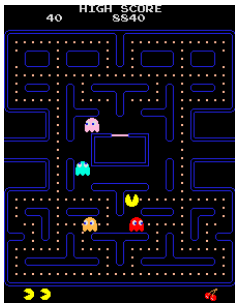

Ilustración 6: Juego sin Scroll

Ilustración 7: Juego con Scroll

Para entender mejor lo que es el ***scroll***, antes de adentrarnos en explicar los diferentes tipos que hay, vamos a ver y explicar un término que está completamente asociado con el *scroll*.

Hablamos del ***viewport***. Si nos encontramos en un mapa el cual es más grande nuestra pantalla, nosotros únicamente podremos ver e interactuar con las cosas que tenemos en pantalla. Esta pantalla que nosotros visualizamos se denomina ***viewport***. En la siguiente imagen se puede visualizar mejor:

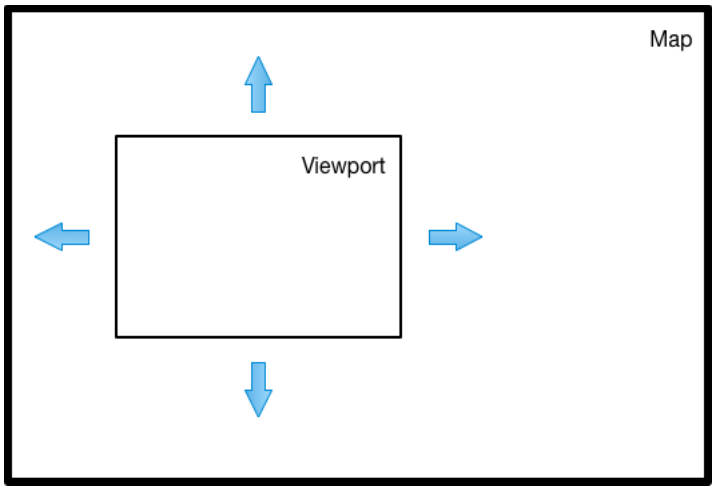

Ilustración 8: Viewport dentro de un mapa

Si tenemos *scroll*, tenemos un *viewport* que programar, van siempre unidos.

Volviendo a los tipos de *scroll*, encontramos dos tipos:

Scroll por software:

Es el tipo de *scroll* más común, ya que es el más fácil de implementar. Se realiza por software (código), redibujando **toda** la pantalla. Es una alternativa costosa, ya que por cada iteración del bucle de juego tenemos que repintar todo, se haya movido algo o no en pantalla. El *scroll* por software va estrechamente de la mano del doble *buffer* de pintado, ya que repintar toda la pantalla y los *sprites* necesarios ya ocupan más de una pantalla de cálculos. Se suele combinar con un sistemas de *tiles* y una implementanción acorde del *viewport*. Al dibujar toda la pantalla en cada pasada, no es necesario una rutina de borrado.

Scroll por hardware:

Esta técnica es mucho más compleja por la cantidad de combinaciones existentes para realizarla. La manera más popular es la siguiente:

Cada vez que nos movamos el *viewport* por la pantalla, ya sea en cualquiera de las cuatro direcciones, hay que modificar el registro 13 del CRTC. Este registro indica el inicio de la memoria de video, por lo tanto, si nos desplazamos a la derecha incrementamos en uno la dirección de memoria contenida en el registro 13. Lo contrario para la izquierda, mientras que para desplazarnos hacia arriba o hacia abajo abría que incrementar o decrementar 80. El desplazamiento viene dado por la voluntad del desarrollador y las necesidades del juego, no tiene porqué ser 1 byte o 80 bytes, ya que es posible incluso realizar *scroll* por hardware al pixel, aunque su implementación sea mucho más compleja. La gran ventaja del *scroll* por hardware, es el poco consumo computacional que requiere ya que únicamente se pinta una mínima parte de la pantalla.

En la siguiente imagen se ve visualmente cómo funciona.

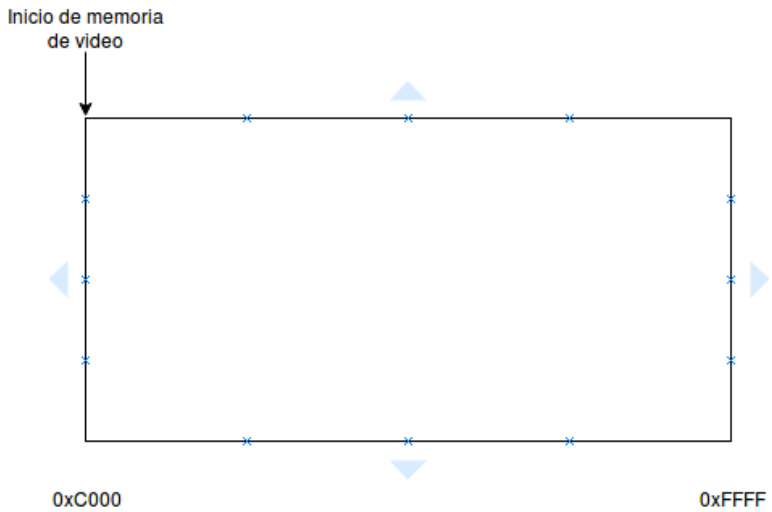

Ilustración 9: Ilustración inicio puntero a memoria de video

Inicialmente tenemos nuestro registro apuntando al inicio de la memoria de video, pero si hacemos *scroll* hacia la derecha hay que aumentar el *offset* del puntero, lo que nos daría el siguiente resultado:

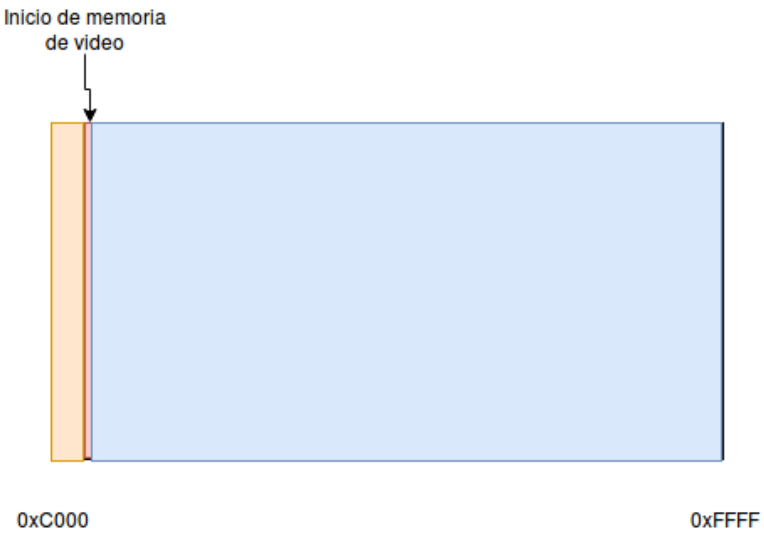

Ilustración 10: Memoria desplazada hacia la derecha

En este punto, el registro está adelantado, por lo que el nuevo inicio de nuestra memoria sería *0xC000* más el *scroll* que hayamos hecho. Al ser la memoria de video circular, el CRTC empezaría a dibujar de izquierda a derecha en la pantalla, el rectángulo azul, acabando por último pintando el rectángulo naranja. El resultado visual en la pantalla de nuestro monitor sería el siguiente:

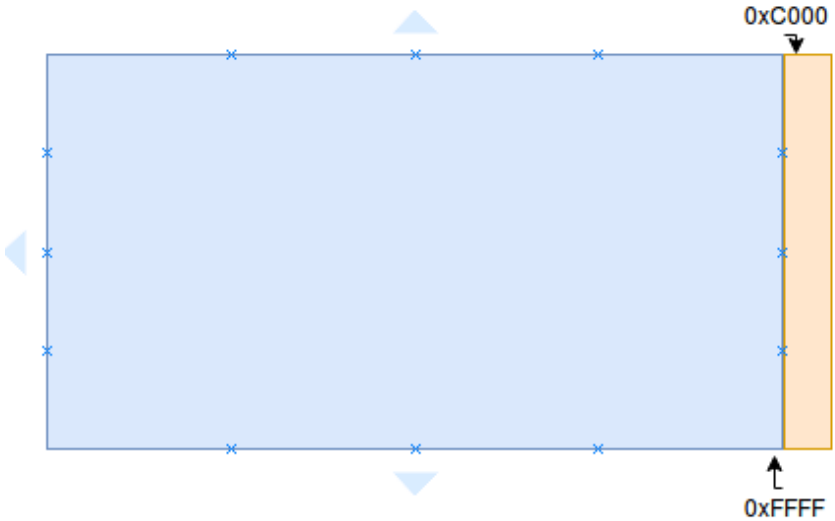

Ilustración 11: Reinterpretación de la memoria

Por lo tanto, a efectos prácticos, el resultado que obtenemos es un corrimiento de la pantalla en la dirección que nosotros deseemos. Esta técnica, es altamente eficiente, ya que únicamente tendríamos que pintar la zona desplazada, en este caso, la zona naranja. A

diferencia del *scroll* por software, no se pinta toda la pantalla, sino que se pinta exactamente la nueva zona del mapa.

Ambas técnicas tienen sus pros y sus contras. El *scroll* por hardware, nos ofrece una gran versatilidad. Primeramente nos permite incorporar a un juego tanto un *scroll* como que pueda funcionar a 50fps, cosa que con un *scroll* por software es imposible. La complejidad del *scroll* por hardware radica en cómo de fino se quiera hacer, ya que se puede hacer a nivel de *tile*, o directamente a nivel de píxel. Hay que tener en cuenta que la memoria es circular, y eso implica que en algún momento del *scroll* por hardware da una vuelta, ya que estamos modificando el registro. Las implicaciones que tiene es que si pintamos un *sprite* justo en la zona donde está dando la vuelta, este *sprite* se mostrará partido en dos, por lo que a la hora de realizar las rutinas de dibujado y borrado, se ha de contemplar este caso, añadiendo aún más complejidad, cosa que no encontramos en el *scroll* por software. El hecho de tenernos que preocupar por borrar las entidades hacen que tengamos que tener unas variables auxiliares en cada entidad. Estas variables son las posiciones previas de esa entidad, por lo que tendría (x,y) y además (px, py), lo que nos permite borrar la entidad en su posición anterior, para que en el siguiente paso del bucle del juego se vuelvan a pintar de forma correcta.

Lo que realmente nos ofrece el *scroll* por harwdare es la gran eficiencia de pintado, ya que únicamente se pinta una mínima parte de la pantalla, mientras que la alternativa, pinta toda la pantalla. Si decidieramos implentar ambos con un sistema de *tiles*, el *scroll* por hardware tienes que preocuparte por repintar las zonas que has modificador, por ejemplo si nuestro personaje pega con una espada o directamente nos movemos por el *viewport*, mientras que en el software no, ya que a cada pasada se repinta todo.

Una vez vistos los posibles contenidos visuales, vamos a tratar el renderizado de éste. Antes de empezar hay que entender cómo funciona el pintado de datos en pantalla.

El *raster* pasa por la pantalla unas 50 veces por segundo, por lo que podemos conseguir como máximo que el juego funcione a 50 *fps*, pero no suele ser lo normal. Si queremos que nuestro juego funcione a 50 *fps*, tendremos que tener nuestra rutina de pintado, nuestra rutina de actualización y nuestra rutina de borrado ejecutada y terminada antes de que el *raster* vuelva a pasar de nuevo por la pantalla. Desde este punto entra un nuevo concepto, es

“una pantalla de cálculos”, y hace referencia al tiempo que tarda el *raster* en dar una vuelta para volver a pintar desde el principio.

El CRT es un componente asíncrono, es decir, funciona de una forma paralela a la CPU, por lo que al mismo tiempo que nosotros estamos ejecutando código, el CRT está actualizando la pantalla. Esto nos aporta agilidad en cuanto al desarrollo y posibilidades, pero al mismo tiempo puede ocasionarnos problemas si no intentamos sincronizar ciertas partes de nuestro código con el dibujado del CRT o le damos algún otro tipo de solución. Al ejecutarse de forma paralela podemos definir este tipo de medida. Así pues, un paso completo del *raster* por la pantalla será para nosotros una pantalla completa de cálculo, ya que al mismo tiempo que pasa, estamos ejecutando código.

Por lo tanto, únicamente tendríamos una pantalla de cálculos de tiempo para ejecutar una iteración de nuestro bucle de juego, o lo que es lo mismo, uno de los 50 *frames* que puede pintar el *raster* por segundo, si lo que queremos es que nuestro juego funcione a 50fps.

En la siguiente imagen se muestra la pantalla del Amstrad, corriendo en un emulador, teniendo el borde pintado de un color gris, y el centro, en amarillo, donde irá nuestro juego y nosotros podemos pintar.

Ilustración 12: Pantalla Amstrad con fondo amarillo y bordes grises

Como se ha mencionado con anterioridad la CPU sufre unas 300 interrupciones por segundo, como tenemos unos 50 *frames* por segundo, ocurren en total unas 6 interrupciones

por *frame*. Estas interrupciones son el medio por el cual podemos sincronizar la CPU y el CRTC.

Cuando el CRT pinta, recorre toda la pantalla, de arriba a abajo, no recorre únicamente la zona amarilla comentada anteriormente, sino que empieza desde el principio del borde superior y acaba en el final del borde inferior. La siguiente imagen muestra dónde empieza el CRT a trabajar y dónde acaba (acaba un poco más abajo debido a que la memoria de video tiene una zona especial de memoria denominada *spare*).

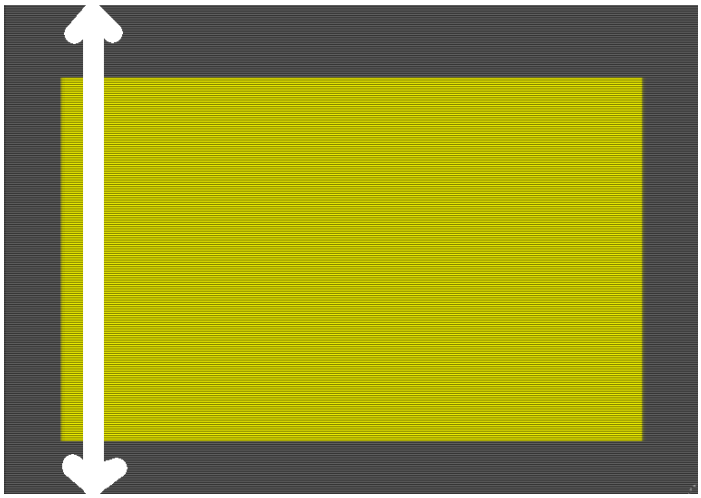

Ilustración 13: Indicación de cálculos

Como la zona de pintado y borrado de nuestros sprites es exclusivamente la zona amarilla, debemos sincronizarnos con las interrupciones para pintar y borrar exclusivamente en el borde inferior y el borde superior. De no ser así, estaríamos cambiando la memoria de video, al mismo tiempo que se está pintando lo que ocasionaría un efecto no deseados llamado *flickering* que se explicará más adelante.

Así pues, si queremos no tener efectos indeseados, tenemos que destinar las zonas no visibles de la pantalla a las rutinas de pintado y borrado y las zonas visibles para la actualización del juego, además de mantenernos siempre sincronizados y empezar las rutinas siempre desde el mismo punto.

¿Qué hacer si la actualización o pintado y borrado ocupan más de lo disponible en una pantalla de cálculos? Hay dos alternativas o reducimos los *fps* o usamos un doble *buffer* de pintado. La primera alternativa es dividir las rutinas en diferentes *frames*, porque en uno no caben, reduciendo así el refresco del juego por segundo (*fps*). La otra alternativa es más

costosa en términos de memoria, pero más eficiente de cara al usuario, esta técnica se explicará más adelante.

Veamos ahora los problemas que pueden surgir en al renderizar nuestro juego en pantalla.

- ***Flickering***: Como se ha mencionado con anterioridad consiste en un efecto de parpadeo no deseado en los *sprites* que se estrán mostrando en pantalla. Y... ¿por qué ocurre?, bien, por el simple motivo de que el CRT funciona de una forma paralela a la CPU. Por ejemplo, si mientras nuestra rutina de dibujado está siendo ejecutada y el CRT pasa por la zona donde está siendo modificada la memoria de video asociada a esa posición que queremos dibujar el *sprite*, el CRT dibujará los datos que ya hayan sido trasladados a la memoria de video, mientras que el resto del *sprite* que aún no ha sido copiado no se verá, dando un efecto de parpadeo y no mostrando lo esperado. También ocurre en la rutina de borrado y puede ocasionar, no que se vea medio *sprite*, sino que no se vea nada. Como es obvio, hay que evitar que esto ocurra si queremos dar una buena experiencia al usuario final.

  Hay varias soluciones:

  - ***Doble buffer de pintado***: La técnica consiste en doblar la memoria de video, con lo cual nos deja una memoria RAM de 32Kbytes. Para poder llevar a cabo esta técnica hace falta modificar los registro del controlador de nuestro CTR. Trata de los siguiente, dado que nuestros cálculos llevan más tiempo que una pantalla completa , “necesitamos otra pantalla”. Al doblar la memoria de video, por poner un ejemplo tendríamos nuestra primera pantalla en la posición `0xC000` hasta la `0xFFFF` (la pantalla por defecto), y nuestra segunda pantalla en la `0x8000` hasta la `0xBFFF`. La memoria quedaría lago así:

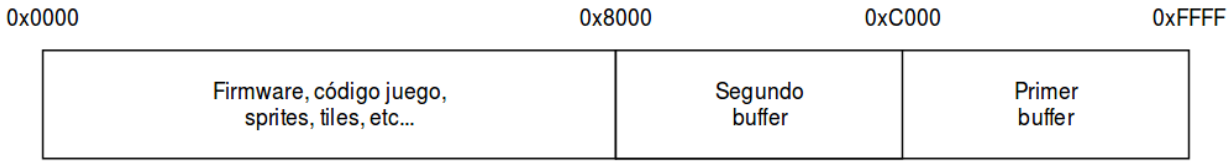

Ilustración 14: Esquema memoria con doble buffer

Por defecto, en el registro 13, el que indica la localización de la memoria de video, está en la 0xC000. Bien, la memoria, como se mencionó anteriormente, es circular, eso significa que cuando termina de pintar 16Kbytes vuelve a empezar. El uso que se le da al nuevo *buffer* es el siguiente, se pinta en pantalla lo haya en el primer *buffer*, entonces aplicamos nuestras rutinas de dibujado y borrado en el segundo *buffer*. Una vez tienes toda la iteración acabada, cambias el registro 13 al segundo *buffer*, es decir, al 0x8000, así el CRT pintará en pantalla los datos que estén en el segundo *buffer*. De esta manera nos aseguraremos que el CRT nunca nos pillará a medio pintar o a medio borrrar. De nuevo se vuelve aplicar el mismo proceso pero para el primer *buffer* y así continuamente. De esta manera conseguimos un tiempo ilimitado para los cálculos, dibujado y borrado; y por tanto, podernos despreocupar del ***flickering.*** Pero cuidado, puede dar como resultado un juego muy lento y dar una mala sensación al jugador.

A continuación se muestra un pequeño y sencillo diagrama explicativo.

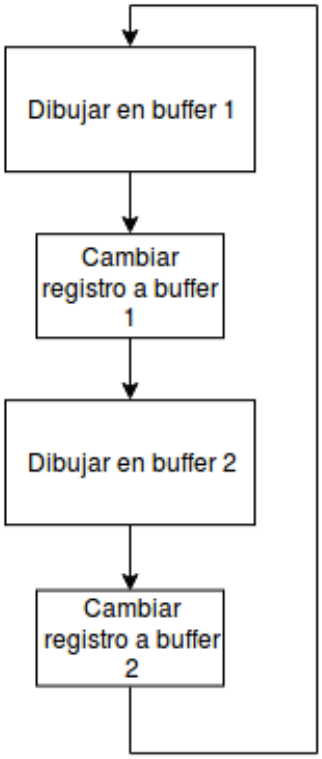

Ilustración 15: Diagrama de ejecución con doble buffer de pintado

- Sincronización con interrupciones: Para realizar esta técnica hace falta estar familiarizado con las interrupciones de la CPU. Cada segundo ocurren 300 interrupciones de CPU y cada segundo el CRT pasa unas 50 veces por la pantalla, lo que es un equivalente a que por cada vez que pasa el CRT para dibujar en pantalla han ocurrido 6 interrupciones de CPU.

  Como la zona de pintado y borrado de nuestros *sprites* es exclusivamente la zona amarilla, debemos sincronizarnos con las interrupciones para pintar y borrar exclusivamente en el borde inferior y el borde superior.

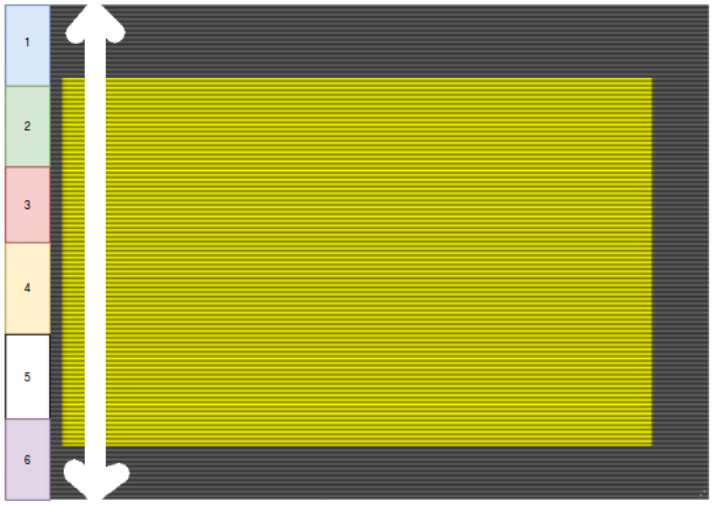

Ilustración 16: Interrupciones sufridas por la CPU

Para hacerlo más visual, observemos la siguiente imagen. Las interrupciones mostradas no son exactamente las reales ya que algunas son más grandes que otras, pero son más o menos similares. Los rectángulos numerados que están en la zona izquierda hacen referencia a las 6 interrupciones que ocurren a cada pasada del CRTC. Si yo sitúo mi rutina de borrado en la zona 6, estaría borrando mis *sprites* mientras el CRT **no** está modificando nada en pantalla, y al mismo tiempo si coloco la rutina de dibujado en la primer interrupción, estaría pintando mientras el CRT **no** está modificando nada en pantalla de cara al jugador.

- ***Desincronización***:

  Ocurre cuando comenzamos nuestras rutinas sincronizadas a las interrupciones, pero debido a que alguna rutina ocupa más tiempo del que debería hace retrasarse las siguientes. Si las rutinas ocupan más de una interrupción, podrían pasar varias cosas. Si por ejemplo mi rutina de borrado, que está en la interrupción seis, tarda más que el tamaño de la interrupción seis en completarse, comenzaría a ocupar tiempo de la interrupción primera, lo que significa que atrasaría la ejecución de la rutina de pintado que estaba situada en la interrupción uno y podría ocurrir que al desplazarla la rutina de pintado terminase de ejecutarse más allá de la interrupción uno, interfiriendo en la zona de la interrupción dos, y por lo tanto pintando en zona de juego, lo que nos podría ocasionar ***flickering***.

Por esta razón debemos medir los tiempos de dibujado y borrado, optimizar y calcular estrategicamente los ciclos que nos van a consumir, si se diera en caso de que a pesar de optimizar lo máximo no logramos realizar todas las rutinas de borrado y dibujado, podemos subdivir el problema en subproblemas. Por ejemplo, no dibujando, pintando y actualizando **todo** en el mismo *frame*, sino que particionar y priorizar qué pintar, qué actualizar, qué borrar en cada *frame*. Un ejemplo práctico sería el siguiente:

Correr todas la rutinas siempre sobre nuestro personaje principal, y sobre **un tercio** de los enemigos en pantalla. Por lo tanto nuestro héroes siempre será accesible y sólo un tercio de los enemigos harán sus rutinas de movimiento, dibujado y borrado en cada *frame*. Hay que tener en cuenta, que si por ejemplo el usuario golpea un enemigo, aunque este enemigo se haya decidido no aplicarle nada en ese *frame*, el golpe se ha de efectuar.

Por lo tanto si queremos mantener los 50 *fps* debemos asegurarnos de que siempre nuestras rutinas de borrado y dibujado no ocupan más de la interrupción seis y una juntas, y que nuestra rutina de actualización (detección de colisiones entre entidades y *tiles*, cambio de *sprite*, cambio de direcciones, etc) no supere el tamaño de las interrupciones dos, tres, cuatro y cinco juntas.

Dado que las interrupciones ocurren durante todo momento, haya un juego cargado en el Amstrad CPC o no, cuando carguemos nuestro juego no sepamos en qué interrupción nos encontramos. Esto nos inhabilita la posibilidad de localizar en las interrupciones que queramos nuestras rutinas. Por ello es necesario buscar una manera de poder conocer en qué interrupción nos encontramos. Afortunadamente, cuando el CRT completa una pantalla, se envía una seña **VSYNC** indicándolo. Así pues, una vez iniciamos el juego, esperaríamos a la señal y una vez la hemos detectado, contamos 5 interrupciones y podemos empezar nuestro ciclo de borrado, pintado y actualizar.

Recalcar que aunque nuestro sistema sea más rápido que una pantalla de cálculo, siempre que acabemos de actualizar, esperar a la interrupción seis para volver a iniciar el ciclo de nuevo y no desincronizarnos. Esto se puede lograr fácilmente usando una variable que simplemente se pondrá a **true** cuando estemos en la interrupción seis, por lo tanto en nuestro bucle de juego si nuestras rutinas acaban antes de tiempo, no podría iniciar otra iteración de juego ya que únicamente está a **true** en la interrupción seis, y tendría que esperar.

4.2.2.3 Colisiones

No todos los juegos, pero sí la mayoría hacen un uso de un sistema de colisiones. Veamos algunas técnicas:

4.2.2.3.1 AABB por fuerza bruta:

La manera más básica de implementación de un sistema de colisiones se conoce como aabb por fuerza bruta. Esto es, por cada entidad que tengamos tenemos que comprobar mediante geometría si esa entidad colisiona con **todas** las demás. El método es el más sencillo de implementar, pero el mismo tiempo si se da el caso de que tenemos muchas entidades en pantallas es el más costoso computacionalmente, ya que estaríamos hablando de una complejidad temporal de $O(n^2)$.

4.2.2.3.2 Sistemas de particionado de espacio:

Existen unas técnicas basadas en el particionado del espacio. Estas técnicas intentan no tener un todo, sino subdividir el espacio, para que a la hora de hacer las comprobaciones de las colisiones, únicamente se efectúen con los elementos cercanos. Cuánto más se particione

el espacio, menos comprobaciones geométricas se harán, pero pero más memoria se consumirá, siendo en Amstrad CPC un bien escaso.

Por ejemplo, podemos encontrar los árboles BSP, o los octree. La implementación de estos mecanismos es costosa y antes de llevarlas a cabo es muy necesario hacer un estudio de necesidades para ver si realmente merece la pena el esfuerzo para el resultado obtenido. También hay que tener en cuenta que cuando se programa para amstrad, se haga en ensamblador Z80 como tal o desde un compilador en C, la memoria dinámica “no existe”. No hay un mecanismo de reserva de memoria, por lo que si se quiere hacer uso de ésta, hace falta implementar un sistema de gestión de memoria dinámica.

4.2.2.3.3 Colisiones a nivel de *tile*

También se debe de tener en cuenta las colisiones no sólo con las entidades, sino también con los obstáculos que pueda presentar el juego, ya sea el suelo, muros, etc. Una manera de implementar esto es tratando los obstáculos como otras entidades especiales, que su única función es la decorativa e impedir el paso. Ésta implementación aumentaría el número de entidades y como hemos mencionado anteriormente, el calculo computacional aumenta de manera exponencial.

La alternativa a esta aproximación se hace mediante el uso de un mapa o *grid*. Cuando hemos mencionado anteriormente el diseño del mapa mediante el uso de *tiles* también hemos mencionado que se le podía dar un uso mayor al decorativo y nos referíamos a este.

Al dividir el mapa en un *grid*, siendo cada casilla del tamaño de un *tile*, podemos dividir el mapa de manera uniforme. Al mismo tiempo, al tener el *tileset*, que recordemos, es el conjunto de *tiles*, numerado desde el primero al último podemos, al mover una entidad, ver en qué posición se encuentra respecto al mapa de *tiles* y ver si ese *tile* en el que se encuentra corresponde a un *tile* obstáculo o no.

La ventaja que presenta este método, es la eficiencia temporal, ya que es un simple lookup en una tabla y por tanto hablamos de una complejidad de O(1).

En este ejemplo tan básico se puede apreciar. Tenemos un *tileset* de dos *tiles*, el negro y el blanco. Tenemos nuestra entidad en un *tile* blanco actualmente, cuando decidamos moverlo, simplemente tendremos que comprobar si al *tile* o *tiles* que va a acceder son accesibles (blancos) o por el contrario son de obstáculo (negro) y por tanto impidiendo el paso.

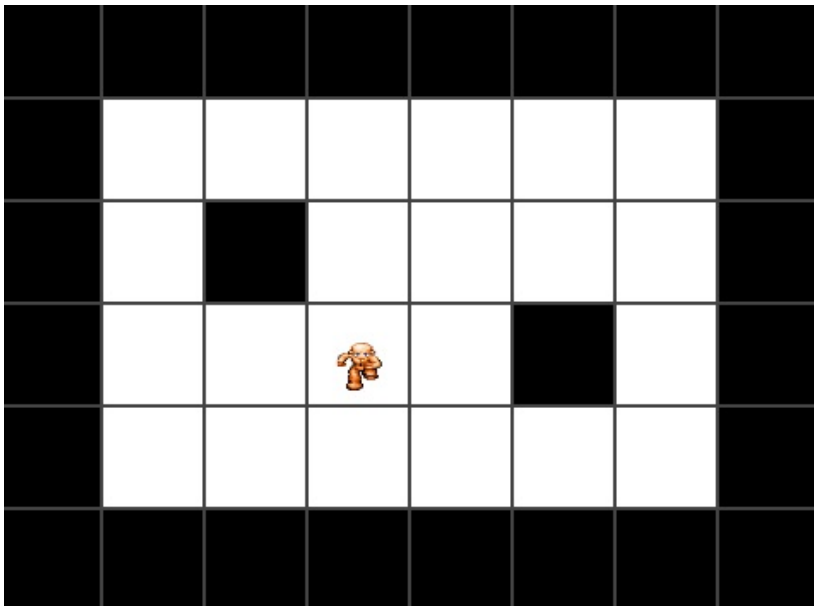

Ilustración 17: Ilustración de tile map con tiles transpasables y tiles que no

4.2.2.4 IA

Cuando hablamos de IA, podemos referirnos a muchas cosas, como pueden ser el tratamiento del movimiento a seguir de un enemigo, si este enemigo tiene un rango de acción, si el enemigo se comporta según el estado en el que se encuentra, etc.

Todos los enemigos que podamos tener en un juego, normalmente se mueven o realizan un desplazamiento. Existen técnicas tan sencillas como complejas para realizar tal simple tarea. Uno de los mecanismos más sencillos de implementar es, si mi objetivo está a la izquierda intento moverme a la izquierda, si mi enemigo está arriba me desplazo hacia arriba, y así en las dos direcciones restantes. Como resultado obtenemos un mecanismo sencillo pero no realista de cara al jugador, ya que si hay un obstáculo entre una entidad y otra no la esquivará ya que se quedará atorado en ella. A este mecanismo se le pueden añadir modificaciones tales como si en una dirección no se pueden mover, moverse en otra hasta, y así hasta que lo logren su objetivo.

Otro mecanismo que se emplea en la industria actual es el denominado *pathfinding*. Es idóneo en juego que usen un mapa de *tiles* ya que simplifica mucho la labor porque usan los propios *tiles* del mapa como nodos para el algoritmo. Si no existe tal mapa de *tiles* se suele hacer uso de mallas de navegación.

Se basa en la búsqueda del camino más óptimo o solución subóptima si lo que se desea es rapidez en los cálculos. Este mecanismo requiere una gran carga computacional y está basado en la búsqueda del camino más corto de Dijkstra. En los videojuegos se suele usar la variante A* que permite el uso de una heurística inicial para eliminar de forma probabilista caminos más largos.

Pero no todo se basa en en movimientos de enemigos, también existen los comportamientos o fases. Se pueden diseñar enemigos de muchas maneras, desde que hagan una única cosa aunque sea compleja, como puede ser, seguir al enemigo y dispararle mientras mantiene la distancia con él, hasta un comportamiento por fases que se activan si se cumplen ciertos eventos. Pero lo normal no es esto, normalmente en el diseño de enemigos se hace por su tipo asociado, es decir cada enemigo se puede entender como un tipo (gusano, pistolero, etc) y su estado correspondiente (andando, cubriéndose, etc).

Cuando se hace el diseño de enemigos y se quiere realizar de la forma más maleable posible, se hacen uso de estados, o lo que es lo mismo Máquinas de Estado Finito (MEF). En resumen, una máquina de estados finitos, en un complejo de estados posibles en los que se puede encontrar un enemigo. La navegación o el cambio de estado se realiza mediante eventos, ya sean que el enemigo ha bajado de cierto % de vida, o que un temporizador ha accionado el cambio de estado.

La máquina de estados el la manera más fácil de realizar que una entidad pueda realizar muchas acciones y se comporte de manera muy diferente sin complicar el código, aparte de permitiendo la modularización del código, pudiendo llevar cada acción a una función y ser reutilizada por otra MEF.

Aquí un ejemplo sencillo:

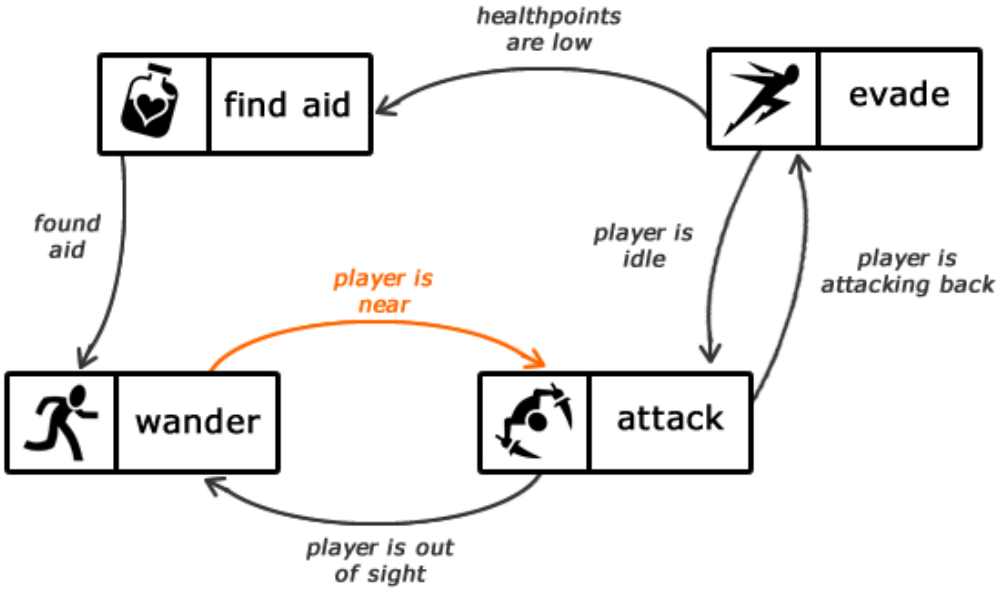

Ilustración 18: Ejemplo de máquina de estados finitos

Los estados sería buscar vida, evadir, patrullar y atacar, mientras que los accionadores del cambio de estado pueden ser vida baja, que el jugador esté cerca, etc.

Otra técnica es el uso de patrones. Es una técnica mucho menos maleable y más costosa en término de memoria. Consiste en la ejecución programada de una serie de acciones independientemente de lo que curra alrededor. No son más que un tipo de máquinas de estado finito menos generalistas.

4.3 Metodología

La metodología que se ha llevado a cabo para la realización del proyecto ha sido en espiral . ¿Porqué he decidido esta metodología y no otra? Bien, este es mi primer proyecto serio que realizo para esta plataforma, aunque antes he realizado otros trabajos, y las experiencias previas me han hecho ver que una vez que formas la idea de tu juego en mente es complicado tener que rechazar ciertas características que tenías planeadas desde un principio, e incluso muchas otras características que pueden surgir durante el desarrollo. No siempre se pueden desarrollar todas la características planeadas ya sea porque llevan más tiempo del esperado, a veces para poder desarrollarlas haca falta remodelar cierta parte del videojuego, o simplemente porque la *deadline* del proyecto está muy próximo.

Por poner un ejemplo, inicialmente se planteó un juego que ofreciera tener por mazmorra un enemigo especial, pero por motivos de espacio de memoria no se han podido desarrollar, ya que tocaría remodelar el mapa de juego para deducir costes de memoria y eso lleva mucho tiempo. También se pensó en un principio poder darle el héroe principal diversas armas y que pudiera cambiar entre ellas, pero al final sólo tiene una espada.

A la hora de realizar la asignación de característica a incorporar a la iteración, una previa experiencia ayuda en gran medida a saber cuánto tiempo puede llevar cierta tarea, y poder organizarlas en iteraciones más prematuras o tardías. Teniendo en cuenta que un producto es un conjunto de característica, unas con más impacto que otras, y que el tiempo siempre es un factor determinante, hacer uso del desarrollo en espiral te permite en cualquier instancia del desarrollo tener un producto acabado y funcional.

En la primera iteración seleccioné las características más fundamentales que permitieran tener el juego en una versión muy básica. Éstas eran: estructura del juego, movimiento del héroe y primer y único enemigo; todo muy básico, y funcionaba dentro de la pantalla. Al final de esta primera iteración el resultado fue 7 rectángulos moviéndose por la pantalla, pero con el proceso de pintado, borrado y actualización actuando correctamente.

La siguiente iteración incorporaba las siguientes características: los *sprites* para el héroe y el enemigo, la capacidad para atacar el héroe, con el *sprite* para la espada, y la capacidad de matar e los enemigos.

Lo siguiente que decidí añadir ya era la capacidad de desplazarse por un mapa más grande de la pantalla, por lo que tendría que implementar un doble *buffer* de pintado, un *viewscroll* y todas las funciones de comprobación de pintado necesarias.

Cuando tuve el *scroll*, vi necesario añadir la capacidad de cambiar de mapa, ya sea por tocar un borde o por tocar un portal, así mismo como también la posibilidad de permitir la teletransportación del héroe por el mapa. Por este último motivo implementé la abstracción del *viewport* en una cámara.

Esta iteración fue la última iteración que me dediqué a añadir nuevas mecánicas. Fueron me añadidas las puertas y las generaciones de vida y llaves. También se diseño el menú y la interfaz.

Los últimas iteraciones se basaron únicamente en añadir nuevos enemigos, y nuevos mapas.

De esta manera, fuera cual fuera la iteración en la que tuviera que detenerme tendría un producto listo para entregar, mejor o pero, pero el producto estaría listo.

4.4 Cuerpo del trabajo - Endless story

Cuando uno se adentra en diseñar un proyecto software, ya sea un videojuego o un proyecto cualquiera, la estructura de ficheros que va adoptar el proyecto es totalmente dependiente de las herramientas de desarrollo que se vayan a utilizar.

En el lenguaje de programación C o en ensamblador, la existencia de clases como las podemos encontrar en Java o en C++ es impensable, por lo que para una mayor organización de los ficheros y de la responsabilidades de cada “entidad” los distribuí acorde a su papel en el juego.

4.4.1 Estructura del proyecto

Conforme el juego iba tomando forma e iban apareciendo nuevas entidades con nuevas responsabilidades tuve que ir dividiéndolas y separándolas de manera que su función fuese clara con ver el nombre de la carpeta contenedora. Muchas de estas clasificaciones también se hicieron acorde al papel tomado en el juego y por tanto la estructura de datos que contiene cada entidad. Por ejemplo, dado que todas las entidades del juego, ya sea un enemigo, el héroe, o una llave comparten unas características comunes, como son una posición x, una posición y un *sprite* asociado, agrupan todos dentro de la carpeta entidad.

Así pues el proyecto está dividido en diferentes carpetas que tratarán diversas funcionalidades en el proyecto, la estructura es la siguiente:

- **main.c:** se encargará de iniciar todo el entorno, tanto el doble *buffer* como la rutina de interrupciones y por último iniciar el juego.
- **State:** Dividido en dos ficheros, el menú y el juego. Definen los dos posibles estados en el que se puede encontrar el juego.
- **Entities:** Contiene todos aquellos elementos que se van a tratar como “todo aquello que se va a pintar en la pantalla y se puede interactuar con”.

- **Animation:** En él se definirán todas las estructuras de datos necesarias para poder plasmar todas las animaciones de las entidades.
- **Descompresor:** Contendrá la rutina de descompresión de mapas
- **Map:** Contendrá la matriz que definirá el mapa y todas las funciones asociadas.
- **Sprites:** Contendrá todos los *sprites* del juego. Simples archivos con lo datos en forma de vector.
- **Tilemap:** Contrendrá los *tiles* codificados. También en forma de vector.
- **Video:** Contendrá todo lo asociado al video, desde las funciones para hacer funcionar el doble *buffer* de pintado, las funciones de pintados de interfaz de usuario, hasta la cámara que nos servirá para manejar el *viewport*.
- **Fichero constants.h:** Fichero donde centralizaremos todos los valores constantes de nuestro juego.

El siguiente diagrama muestra lo comentario con anterioridad, pero con la peculiaridad que se ha dividido en vídeo, cámara y interfaz de usuario.

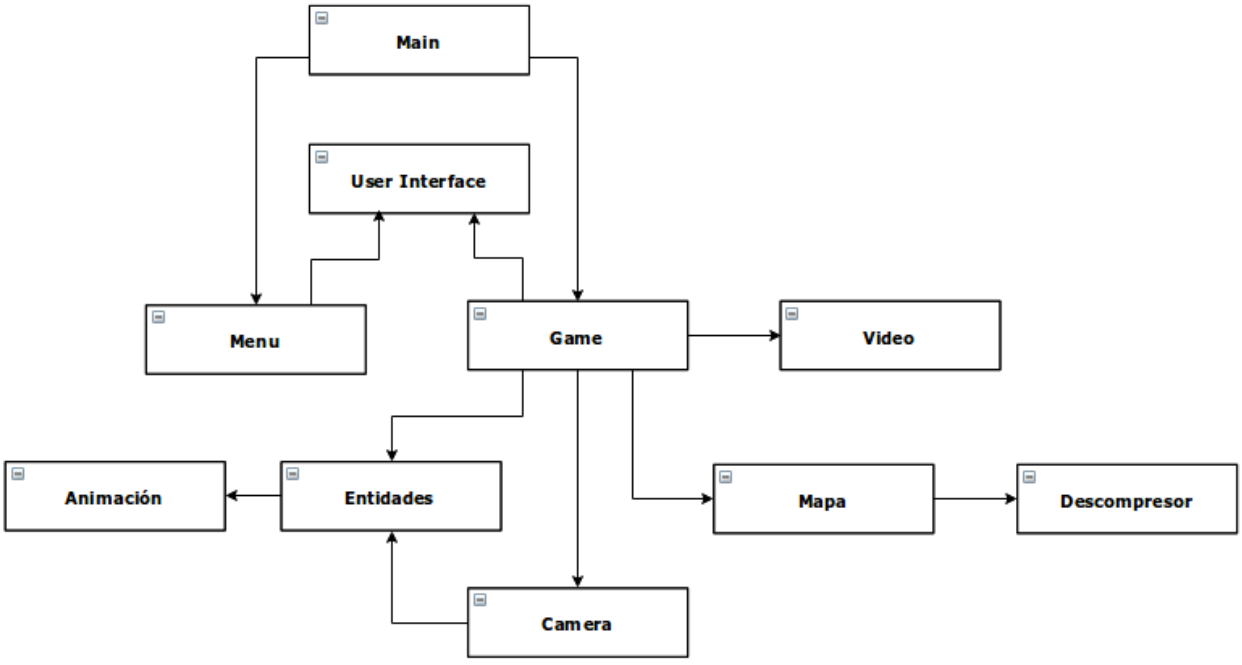

Ilustración 19: Diagrama de funciones y sus asociaciones

Antes de comenzar es necesario explicar cómo funciona el juego de una manera general, ya que todas las funcionalidades se concentran en dos funciones.

El juego inicia desde el main.c donde se llevan a cabo tareas de inicialización completamente necesarias, ya que como se explicará más adelante el juego hace uso de un doble *buffer* de pintado y del servicio de interrupciones. Así pues una vez inicializado el juego, se realizan las siguiente tareas, se cambia la pila de posición en la memoria, se inicializa el doble *buffer*, se configura el servicio de interrupciones, se configura la paleta, se indican que *tiles* se van a hacer uso y por último se indica el tamaño del mapa a la rutina de pintado.

Una vez acabada la inicialización se muestra el menú, pasado el menú es donde empieza el bucle del juego, que se ejecutará continuamente hasta que el jugador complete el juego, que volverá al menú. El dicho bucle llamarán a dos rutinas únicamente, la rutina de pintado y la rutina de actualización.

4.4.2 Entidades

Empecemos por las entidades. Una entidad es un elemento el cual va ser posible dibujar en pantalla. Es un elemento que lejos de ser un mero obstáculo, tendrá acciones a realizar dentro del juego o influirá de alguna manera.

Las entidades tienen unos atributos básicos y comunes, que son:

- Posición x
- Posición y
- *Sprite*

Los dos primeros atributos nos permiten ubicar la entidad en la pantalla y el último, cuando vayamos a pintarlo en pantalla poder representarlo visualmente como es debido.

Encontramos diferentes entidades, las agruparemos en móviles y en no móviles.

4.4.2.1 Entidades móviles

Todas las entidades que pertenecen a este grupo son usadas por las dos rutinas principales del juego. Son pintadas y son actualizadas.

- Héroe: Será el personaje con el que el usuario podrá jugar. Comparte todas las características con la estructura de un enemigo, pero funciona diferente. Es la entidad que modificaremos su posiciones x e y mediante únicamente la interacción del usuario con el teclado. Esta entidad tiene característica que no tienen los enemigos, como la posibilidad de cambiar de mapa o la posibilidad de interactuar con las entidades no móviles. En similitud con los enemigos, a ambos se le aplican las restricciones de movimiento que vienen dadas por el mapa (no pueden atravesar paredes).
- Enemigos: Son los encargados de ralentizar el avance del héroe hacia la victoria. Se diferencian del héroe en que posen ciertos atributos que el héroe no posee, como pueden ser variables auxiliares indicado el estado de la animación del enemigo, el tipo de enemigo, la dirección que se han de desplazar al ser golpeados, un contador individual y por último un puntero a función asociado que indica qué acción se ha de realizar una vez muera el enemigo.
- Bala: Entidad diminuta que se mueve a una gran velocidad por la pantalla, puede golpear tanto a héroe si es disparada por el enemigo, o a los propios enemigos si el héroe la ha devuelto.

4.4.2.2 Entidades inmóviles:

Entidades que no tienen la capacidad de moverse por la pantalla, su función es aportar jugabilidad bloqueando pasos, permitiendo pasos o curando al héroe. Estas entidades únicamente se dibujan, es decir sobre ellas únicamente se aplican las rutinas de dibujado, las entidades móviles se encargan de detectar si han colisionado con alguno de estos elementos inmóviles. Estas entidades son:

- Llave: Son generadas por ciertos enemigos al morir, existen cuatro tipos y cada una tiene la capacidad de abrir una puerta diferente.
- Puerta: Puerta que bloquea el paso del héroe, para poder abrirla ha de conseguir la llave del tipo correspondiente y colisionar con la puerta.
- Medio corazón: Los enemigos tienen 1/5 de probabilidad de generar esta entidad, su función es curar al héroe, si no tiene la vida al completo ya.
- Portal Local: Permite la teletransportación del héroe por el mapa que se encuentra.
- Portal Exterior: Permite al héroe teletransporarse a un mapa diferente al actual.

4.4.5 Funcionamiento entidades

Bien hasta ahora hemos conocido cómo las entidades son tratadas y cómo se muestran al usuario, pero aún no se ha explicado cómo funcionan.

Dado que las entidades no móviles únicamente se pintan y no tienen efecto en la rutina de actualización del juego, ya que son las entidades móviles las encargadas de accionar sus funciones asociadas, se explicará el funcionamiento de las móviles y en derivación, las no móviles.

Héroe:

La rutina de actualización del héroe consiste en una máquina de estados finita. Se muestra en la siguiente imagen:

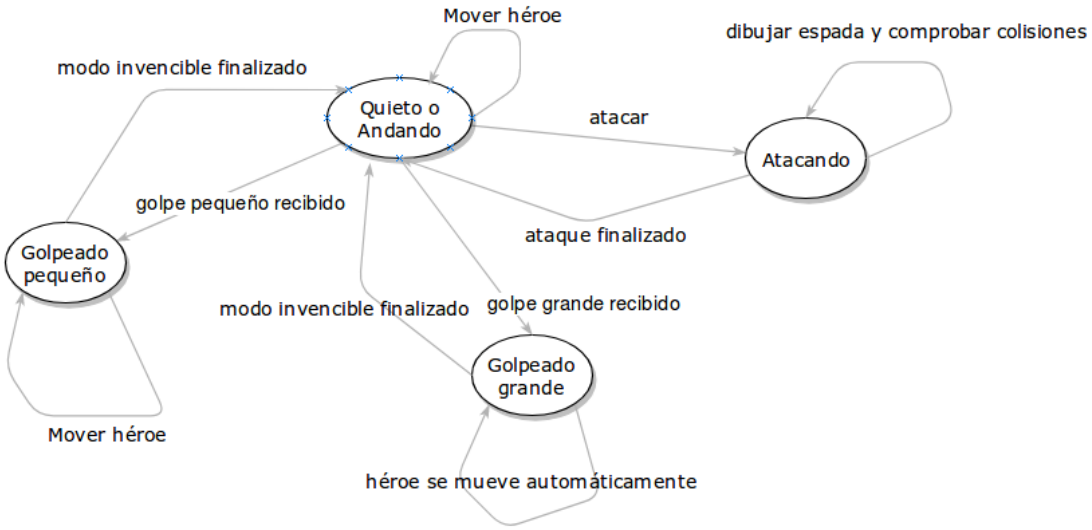

Ilustración 20: MEF Héroe

Cuando entra en la función de actualización, comprueba el estado en el que se encuentra el héroe, y acorde a ese estado se realizan varias acciones.

- Si el héroe, en la anterior iteración se movió o no, se encontrará en el estado andando o quieto. En este estado podrá hacer dos acciones: podrá atacar, y cambiará al estado atacando, o podrá moverse y volverá al mismo estado que se encontraba. Si estando en este estado una entidad enemiga colisiona con él, dependerá del tipo de daño realizado al héroe. Pueden ser dos tipos de daños, grande o pequeño. Sea el que sea

cambian al estado correspondiente, además de reducir la vida del héroe acorde al daño recibido.

- Si el héroe está atacando, el jugador tendrá que esperar que el ataque (pintar la espada en pantalla y comprobar si ha colisionado con alguna entidad) termine. Esta espera son unos 4 ciclos de juego, y una vez terminada la espera el personaje vuelve al estado de quieto.
- Si el héroe está en golpeado pequeño, a cada ciclo se reducirá un contador especial que indica el estado de invencibilidad del héroe y para pintarlo de forma intermitente. Mientras este contador sea diferente de 0 el héroe permanecerá en golpeado pequeño permitiendo al héroe moverse, pero no atacar. Una vez llegue a 0 el contador volverá al estado quieto.
- Si el héroe está en golpeado grande, a cada ciclo se reducirá un contador especial que a parte de indicar el estado de invencibilidad del héroe, se usará una variable auxiliar especial para indicar y almacenar la dirección de movimiento del agresor cuando el héroe fue golpeado. Esta variable es utilizada para este estado en concreto. El uso que tiene es mover al héroe de esta dirección durante unos ciclos, para simular el efecto de un golpe fuerte que hace retroceder al héroe. Mientras esté en este estado no podrá realizar ninguna acción. Cuando la variable llega a 0 vuelve al estado quieto.

El movimiento del héroe tiene ciertas características y limitaciones:

- El movimiento del héroe se efectúa con los *inputs* de teclado.
- El movimiento del héroe está limitado a partir de ciertas posiciones. El héroe tiene bloqueado movimientos menores de 0 y mayores que (*max_w – HERO_WIDTH*) para la x. Esta limitación superior viene dada por el mapa en cuestión, ya que según el mapa tendrá una anchura u otra. Por otro lado la variable *HERO_WIDTH* es una constante que vale la anchura del héroe. Para la y el rango viene limitado por una manera similar, para movimientos menores que *OFFSET_SCREEN* y mayores (*max_h – HERO_HEIGHT*). Esta nueva variable *OFFSET_SCREEN* entra en acción debido a las características distributivas de la interfaz del juego, se explicará más

adelante. El motivo de estas limitaciones es para evitar efectos no deseados. Para la x, si se superara el límite inferior el héroe aparecería justo por el otro lado de la pantalla, por la derecha. Mientras que si se superara el límite superior de la x o cualquiera de la y el héroe entraría en contacto con la interfaz de usuario dando un efecto no deseado, ya que esa zona no está pensada para que el héroe pueda caminar por ella.

- Si el héroe toca algún borde, se efectuará un cambio de mapa.
- Si el héroe toca un *tile* no designado como andable no se efectuará el movimiento.
- Si el héroe toca un *tile* especial designado, en este caso un *tile* de color negro por completo que hace referencia a portales externos efectuará un cambio de mapa.
- Si el héroe toca un *tile* especial designado que hace referencia a portales locales efectuará un cambio de posición en el mapa actual.

Enemigos:

Sea el enemigo que sea, todos realizan los mismo pasos, y por tanto la misma comprobación. Antes de ejecutar su siguiente movimiento, se comprueba si el enemigo tiene el una variable activada. Si es afirmativo significa que ha sido golpeado por el héroe y por lo tanto debe realizar un movimiento similar a si el héroe ha sido golpeado por un enemigo fuerte, es decir, debe realizar un retroceso simulando que el golpe del héroe lo ha lanzado hacia atrás. Si no tiene la variable activada se procede a actualizar el enemigo acorde a su tipo. Según el tipo que posean se comportarán de una manera muy distinta.

Tenemos 4 tipos diferentes: el pato, el girador, el disparador y por último el gusano.

Tanto el primero como el segundo tiene el mismo MEF:

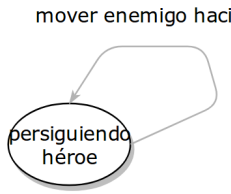

Ilustración 21: MEF Pato y Girador

Básicamente se limitan a perseguir al héroe con la única diferencia de que el pato efectúa un golpe pequeño mientras el girador realiza un golpe grande.

Por otro lado encontramos al disparador, el disparador realizar un un MEF un poco más elaborado:

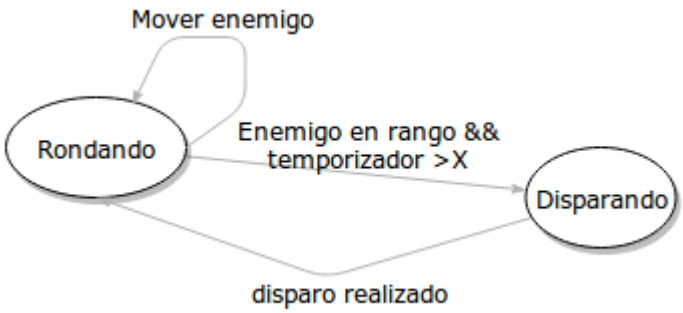

Ilustración 22: MEF Disparador

El enemigo está rondando la zona, esto es, a cada iteración se obtiene un número aleatorio y mediante la operación binaria `&3` se obtienen los 2 último bits pudiendo obtener desde 0 hasta el 3 que son las 4 posibles direcciones. De esta manera se le indica al enemigo que se desplace a una de esas 4 direcciones y mientras ocurre esto, un contador va incrementando su valor cada iteración de juego. Cuando supera 20 puede disparar, pero no significa que vaya a hacerlo ya que requiere que el héroe “esté en rango”. El calculo de este rango se realiza mediante la proyección de un rectángulo hacia una de las 4 direcciones, depende de la posición del héroe respecto al enemigo.

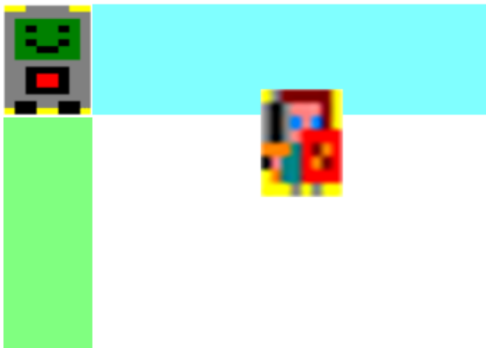

Ilustración 23: Detección del héroe por parte del disparador

Si el héroe está a la izquierda del enemigo el rectángulo azul se proyectaría hacia la izquierda, sino se proyectará hacia la derecha. En cuanto el rectángulo verde se proyectará hacia donde el héroe esté o arriba o abajo. Si este alguno de los rectángulos proyectados (únicamente 2 al mismo tiempo) colisiona con el héroe, se procesa una bala hacia la dirección del rectángulo colisionado.

Por último encontramos al gusano, el gusano es el único enemigo que en uno de sus estados no puede golpear al héroe. Éste enemigo también posee una MEF:

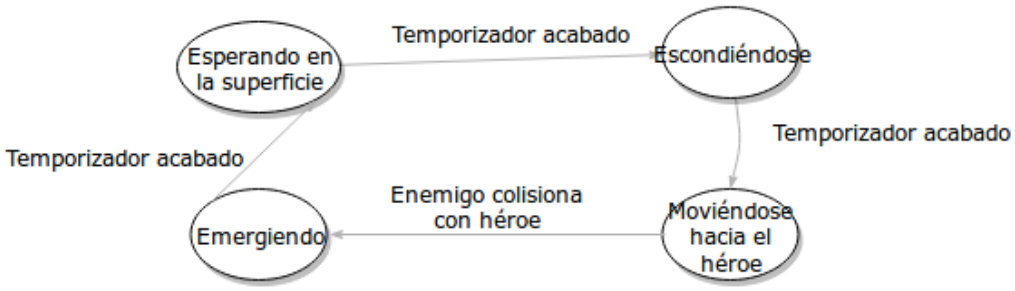

Ilustración 24: MEF Gusano

Las fases por las que está compuesta el comportamiento del gusano son la siguientes:

- Tanto la fase de espera en la superficie, la fase de escondiéndose y la fase de emergiendo son fases de animación ya que no realizan otra acción que cambiar de *sprite*.
- La fase moviéndose hacia el héroe está compuesta de dos acciones. La primera es la comprobación de la colisión del gusano y héroe, si esta comprobación no se cumple, se realiza un movimiento hacia el héroe. Si se diera el caso de que ha colisionado, se activa un contador para darle tiempo al héroe huir del gusano emergente, ya que si está oculto bajo la tierra no le puede hacer daño al héroe. Una vez termina este temporizador cambia de estado a emergiendo y ya aquí le puede hacer daño.

Función *on_dead*

Cuando un enemigo es golpeado su vida baja en uno. Cuando la vida de un enemigo llega a cero se llama a una función que tiene asociada. Dicha función depende del mapa a cargar.

En un principio todos los enemigos tienen una función denominada *generic_drop*, el trabajo que realiza es la generación de un número aleatorio. Si ese número es menor que 50 entonces genera una entidad no móvil, en este caso un corazón. El número 50 es porque se usa una variable de tipo *u8* (*unsigned char*) por lo que los valores van de 0 a 255, lo que significa que la probabilidad de generar un corazón es de un quinto. Hay otros enemigos especiales, que generar un objeto diferente, como pueden ser todos los tipos de llaves existentes. Se ha realizado de esta manera para "obligar" al jugador a matar los enemigos para encontrar la llave que abre la puerta a la siguiente zona. La llave siempre se genera con una probabilidad de 100%, pero una vez cogida no se vuelve a generar, debido que al adquirir la llave con el héroe, se modifica la función del enemigo que la ha generado.

No todos los enemigos son golpeables por el héroe, del mismo modo que no todos los enemigos en todas sus fases pueden golpear al héroe. Cuando se va a realizar la detección de colisiones de la espada con el enemigo, primeramente se comprueba si dicho enemigo es golpeable. Para las comprobaciones se ha usado los estados de los enemigos. Dado que todos los enemigos menos el gusano tienen un estado de *walking*, que es el que usan primordialmente, si el enemigo se encuentra en este estado es golpeable. De esta manera cubrimos el enemigo pato y el enemigo girador, pero faltaría por cubrir el disparador y el gusano. Por motivos de diseño no se ha querido que el disparador pueda ser golpeado por el héroe, sino que si se quiere matar se le tenga que devolver la bala que el enemigo dispara. Para el gusano se ha añadido los estados donde se encuentra fuera, está saliendo o escondiéndose, logrando así, que si está bajo tierra sea ingolpeable.

```
bool can_be_hit(u8 state)
{
    switch(state)
    {
        case enemy_walking: case gusano_waiting_above: case gusano_hiding: case gusano_emerging:
            return true;
        default:
            return false;
    }
}
```

Ilustración 25: Código para conocer si un enemigo puede ser golpeado

Del mismo modo que dadas ciertas circunstancias el héroe no puede golpear a los enemigos, también hay ciertas circunstancias donde los enemigos no pueden golpear al héroe. Lo

podemos encontrar cuando el gusano está bajo tierra, y por tanto no tiene sentido que pueda golpear al héroe.

Del mismo modo se muestra el extracto del código:

```
bool can_hit_hero()
{
    switch(enemies[update_index].state)
    {
        case enemy_walking: case gusano_waiting_above: case gusano_hiding: case gusano_emerging:
        case disparador_walking: case disparador_shotting:
            return true;
        default:
            return false;
    }
}
```

Ilustración 26: Código para conocer si el héroe puede ser golpeado

4.4.6 Animaciones

En cuanto al sistema de animaciones de *sprites* para la entidades encontrados dos maneras:

- Mediante la variable de posición y la variable de posición de la iteración anterior.
- Mediante el modelo de MEF.

El primer punto se aplica para el héroe en su movimiento normal y para el pato y el girador. En este caso las animaciones consisten en dos *sprites*, al alternarlos da una sensación de vivacidad de las entidades. El mecanismo, es el siguiente. Se realiza una comprobación, si la variable de posición y la variable de posición de la iteración anterior son iguales significa que la entidad se está moviendo en la misma dirección. Si lo hace durante un número determinado de iteraciones se activa un *flag*, que en vez de cargar por ejemplo, el *sprite* derecha 1, cargaría el *spirte* derecha 2. Del mismo modo, si sigue desplazándose para la misma dirección volvería a 0. Si por algún casual cambia de dirección la *flag* volvería a 0 automáticamente. De esta manera se van alternado un *sprite* u otro para una misma dirección. Pero, ¿qué pasa si lo queremos hacer de manera independiente a la dirección?. Usamos la técnica de las MEF.

Las animaciones por MEF son sencillas, tenemos estados y acciones que hacen que se activen las transiciones. Bien, lo que se puede hacer es asignar un *sprite*, o un grupo de

sprites a cada estado y mediante un contador cambiar el *sprite* asociado al estado. Pudiendo tener en cada estado un conjunto de *sprites*. Por ejemplo, el gusano en el estado de *hidding* lo que realiza es el cambio de *sprites* de fuera del todo hasta que está oculto de todo en la tierra.

4.4.7 Mapas

La implementación de un sistema de mapas, puede ser sencilla, pero si se se quiere añadir versatilidad puede complicarse bastante. Si se quieren unir diversos mapas para dar la sensación de uno más grande, se tiene que crear un sistema, una codificación de estos y programar un *viewport*, que nos permita la tarea. Un mapa es mucho más que un *grid* indicado el valor de cada *tile*.

Dado que tenemos el objetivo de desarrollar un *scroll* por software haciendo uso de la técnica de un mapa basado en *tiles*, empecemos con la explicación del desarrollo del *scroll* que nos permite tener un mapa más grande que el *viewport* de nuestro juego.

4.4.7.1 Scroll

El desarrollo de *scroll* por software requiere de forma imprescindible para la plataforma Amstrad, que es el caso que nos ocupa, de un doble buffer de pintado debido a la baja capacidad computacional que disponemos. De no ser así caeríamos en el effecto no deseado de *flickering*.

Bien, el doble *buffer* de pintado requiere una configuración previa que consta de dos fases:

- Reservar memoria para el doble *buffer* (16kbytes).
- Cambiar la posición de la pilla de la *0xC000* a la *0x8000*.

Para la primera fase realmente no tendríamos que realizar ninguna tarea, simplemente mantener en mente dónde tenemos el doble *buffer* para no usar esa zona de memoria.

Para la segunda fase, sí que tenemos que realizar trabajo de forma explícita. La pila está en la posición *0xC000* por defecto debido a que la memoria de video ocupa de la posición *0xC000* a la *0xFFFF*, por lo que el código y la pila iría de la posición *0x0000* a la posición *0xC000*. Al añadir un nuevo *buffer* de pintado necesitamos cambiar la posición de la pila para que no se sobreescriba ni ocurran efectos no deseados. Una vez cambiamos el *sp* (*Stack pointer*) a la posición *0x8000* ya estaría listo.

4.4.7.2 Viewport

Tenemos en este momento lo básico. El proceso de renderizado se explicará en su propio apartado, ahora tratemos el efecto de *scroll* o desplazamiento. Para que haya un desplazamiento tiene que haber un *viewport*, y para haber un *viewport* tienen que ocurrir dos cosas imprescindibles. La primera es que el mapa sea más grande que el *viewport*, es decir no va a ocurrir un desplazamiento de la pantalla, o no debería, si el mapa es de 15x15 y el *viewport* de 15x15. Lo segundo es el control de los *offset* de los ejes. Estos *offset* nos indicarán cuánto nos hemos desplazado en el mapa en cualquiera de las cuatro direcciones. Veamos un ejemplo:

Si poseemos un mapa de estas características, 10x10 , como se muestra en la imagen:

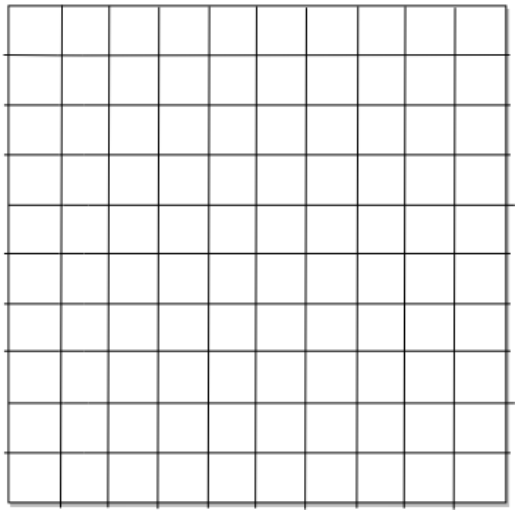

Ilustración 27: Grid de 10x10

Y tenemos un *viewport* de 5x5 que supongamos que es el máximo de nuestra pantalla (monitor) quedaría el esquema de la siguiente manera:

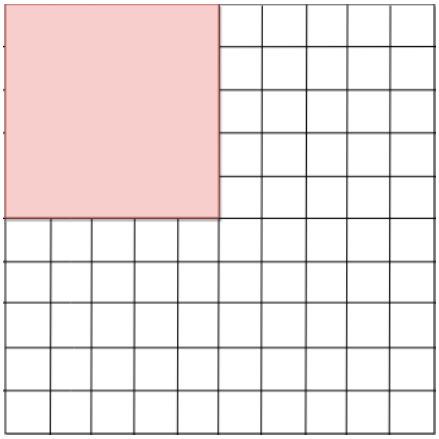

Ilustración 28: Viewport de 5x5 sobre grid *de 10x10*

Es decir, de todo el mapa sólo estamos visualizando el rectángulo rojo, así pues necesitamos poder desplazar ese rectángulo rojo, y es aquí donde entran las variables de *offset*. Se han utilizado 4 varibles, aunque dos de ellas se han usado como valor precalculado. Los dos *offset* principales se distribuyen en los dos ejes. Si por ejemplo el *offset_x* vale uno significa que se ha de desplazar la pantalla un valor a la derecha y lo mismo para el *offset_y*, pero en dirección hacia abajo. Para visualizarlo mejor, un *offset_x* de 1 y un *offset_y* de 2 sería lo siguiente:

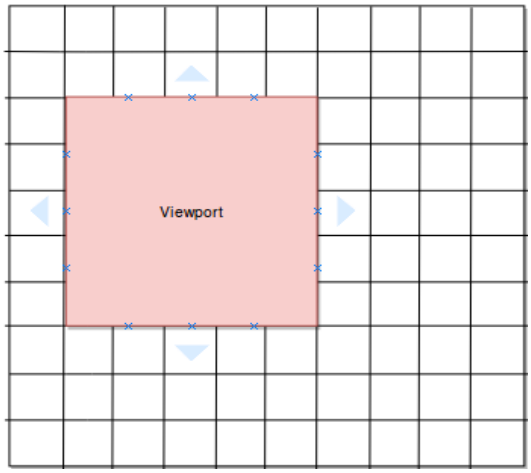

Ilustración 29: Viewport con valores offset_x=1 y offset_y=2.

Estas dos variables se usarán en el renderizado para que se pinte correctamente el mapa. Por ello es tan importante actualizar estos valores adecuadamente. Hay varias maneras de actualizar este valor. En una primera instancia se asoció al movimiento del héroe este actualizado, pero debido al acoplamiento innecesario, ya que son cosas totalmente distintas aunque muy asociadas, decidí abstraerlo a una cámara.

4.4.7.3 Cámara como abstracción del viewport.

Este componente hay que entenderlo como una cámara que se posiciona sobre el mapa y el cono de proyección es el *viewport*. Esta cámara, se asociará con una entidad y la seguirá por todo el mapa. Esta asociación se hace mediante dos punteros que se les asignara la posición de memoria de los atributos x e y de la entidad que deben seguir. Así pues, a cada iteración del juego accederán a estos valores y harán el calculo correspondiente del *offset_x* y del *offset_y*.

El cálculo de *offsets* se realiza de la siguiente manera:

- Si el valor de x es menor que 40 el *offset* es 0.
- Si el valor de x es mayor que la anchura del mapa – 40 el *offset* es el *offset* máximo.
- Si no se cumplen la dos anteriores condiciones se comprueba que esté dentro de un rango relativo. Este rango relativo es relativo al *viewport*. Es decir, se calcula la posición de la entidad dentro del *viewport*, no de forma global. Los datos del ejemplo siguiente son falsos, pero para hacer una representación conceptual :

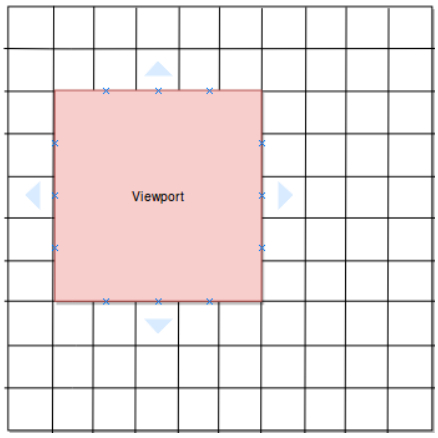

Ilustración 30: Héroe sobre mapa dentro del viewport

La esquina superior izquierda del mapa indica la posición (0,0) absoluta y la esquina inferior derecha la posición (100,100). Al mismo tiempo, la esquina superior izquierda del *viewport* se encuentra en la posición (10,20) absoluta y la esquina inferior derecha la posición (60,70). El héroe se sitúa dentro del *viewport* y se

encuentra en una posición absoluta de (42, 33), pero al mismo tiempo está en una posición relativa dentro del *viewport* de (32, 13), es decir la posición absoluta del héroe menos la posición inicial del *viewport*. De esta manera podemos situarlo dentro del *viewport* de una manera relativa.

Bien, pues si las dos condiciones no se han cumplido y si el héroe se encuentra en una posición menor que 15 o mayor que 45, ambos valores relativos, entonces se actualiza *offset_x*. La fórmula es la siguiente:

La posición x actual del héroe dividida por el resultado de la anchura total del mapa divido por la cantidad máxima de *offset* para la x posible. La cantidad máxima de offset para un eje viene dado por la anchura del mapa menos la anchura del *viewport*.

De esta manera se consigue el *offset* para la x. El proceso para la y es igual.

4.4.7.4 Metadatos de un mapa

Sabemos cómo funciona el *scroll* y el *viewport* sobre la matriz del mapa general, pero no hemos tratado cómo se gestiona un mapa como entidad y estructura de datos.

Un mapa contiene la siguiente información:

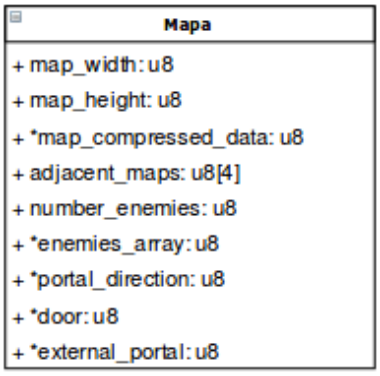

Ilustración 31: Estructura datos del un metamapa

Esto es:

- *map_width*: Ancho del mapa en *tiles*.
- *map_height*: Altura del mapa en tiles.
- *map_compressed_data*: Dirección de memoria del inicio del mapa comprimido.

- *adjacent_map*s: Ids de los mapas adjacentes a este. El valor *0xFF* indica que no existe mapa adjacente en esa dirección.
- *number_enemies*: Cantidad de enemigos en este mapa.
- *enemies_array*: Puntero a los enemigos.
- *portal_direction*: Puntero a los datos del portal local de este mapa.
- *door*: Puntero a los datos de la puerta del mapa.
- *external_portal*: Puntero hacia los datos del mapa externo.

Todos los mapas son contenidos en un *array*. La posición del mapa en el *array* es su id. De esta manera con un id se puede acceder a la posición del *array* que contiene la información del mapa.

4.4.7.5 Mecanismo de cambio de mapa

Como bien sabemos, hay diferentes maneras de cargar un nuevo mapa y para ello debe haber un algoritmo que se encargue. El siguiente sistema es el que se emplea en este juego.

Se hace uso de una variable llamada *change_map*. Cuando se colisiona con un portal externo o un borde de la pantalla, esta variable se cambia su valor a 1. Si ha sido por colisionado por mapa externo el id del mapa a cargar se obtiene de los datos del portal externo. Si por el contrario ha sido por el contacto con un borde, el id se obtiene accediendo a un vector de cuatro elementos, que son los cuatro ids asociados a las cuatro direcciones posibles (arriba, abajo, izquierda, derecha). Este vector se obtiene al hacer el cargado de un mapa. Una vez cambiado la variable change_map a 1 pasará lo que pasará en la siguiente iteración es esto, en el bucle de juego se llaman a las rutinas de pintado y actualización, y acto seguido se realiza una comprobación. Esta comprobación testea el valor de la variable, y si es true comienza el cambio de mapa. Dependiendo de si ha sido por un mapa o si ha sido por tocar el borde podemos encontrar dos caminos diferentes pero con un mismo puerto.

Si ha sido por portal:

- Se realiza una copia de la información del portal, que previamente se cargó del cambio de mapa anterior, en la variables x e y del héroe (para hacerlo aparecer donde deseemos dentro del mapa siguiente), y acto seguidos se realizan las acciones de un cambio de mapa por contacto por borde .

En el cambio de mapa por contacto por borde se realizan las siguientes acciones:

- Se cambia el valor de la variable *change_map* a 0.
- Se eliminan todos los enemigos. Se itera sobre el *array* estático de enemigos y se modifica sus variables *lives* a 0. Esta acción se realiza debido a la naturaleza estática de los *arrays* de las entidades. Esta característica puede ocasionar que al cargar los nuevos enemigos, que a lo mejor hay tres en este mapa dado, en el mapa anterior hubieran cinco y por lo tanto se sobre escribirían los tres primero, pero los otros dos seguirían vivos.
- Se eliminan todos los objetos. Se marcan todos los objetos como cogidos modificando su variable *picked* a uno.
- Se modifica el *array* de mapas adyacentes para colocar los nuevos ids asociados a los bordes.
- Se carga las dimensiones del mapa. El ancho y el alto en dos variables.
- Se cargan los enemigos nuevos. Esto se logra iterando desde 0 hasta la cantidad que indique la variable *number_enemies*. Dado que los enemigos tienen muchas variables, únicamente se cargan los datos que se creen totalmente necesarios para realizar tal cambio. Aquí une ejemplo de cómo se codifica el enemigo:

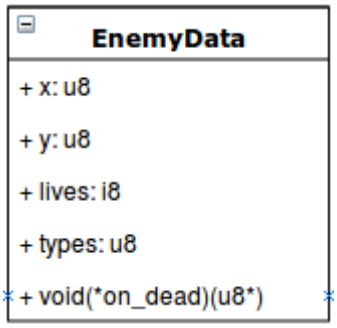

Ilustración 32: Estructura de datos de un paquete de datos de un enemigo que cargará el mapa

Es decir, la posición x e y en *tiles* que deseemos que ese enemigo tenga al cargar el mapa, la cantidad de vida que tenga, el tipo de enemigo y por último, la función que tiene que realizar una vez muera.

- Se cargan el portal local si lo hubiera (si vale 0x0000 el puntero es que no hay).
- Se caga el portal exterior si lo hubiera.
- Se carga la puerta si la hubiera.
- Se llama a la función, con el parámetro del nuevo ancho del mapa, que se encarga de configurar las variables para el pintado del mapa.
- Por último, se llama a la rutina de descompresión para obtener la información de los *tiles* del nuevo mapa descomprimido.

Estos son los pasos que se han tomado para la cargar un nuevo mapa. Como vemos todo el proceso requiere, metadatos de los mapas, los datos reales y un algoritmo que efectúe tales cambios.

4.4.7.6 Compresión de los mapas

Como se ha mencionado con anterioridad, los datos del mapa están comprimidos. Los datos a los que nos referimos son los *tiles*, los datos que se van a copiar en la matriz del mapa. La rutina de compresión no se menciona porque no se encuentra en el código del juego debido a que la compresión del mapa se realiza en el proceso de desarrollo.

Actualmente hay 67 mapas en el juego. Todos los mapas unidos suman una cantidad de 29.672 bytes. Sabiendo que disponemos de 32.000 bytes después del doble *buffer* de pintado, nos dejaría con una cantidad de 2.328 bytes para el código, *sprites* y el resto de la información de los mapas, es decir, una cantidad tan pequeña que sería imposible diseñar un juego. Una vez comprimidos todos los mapas pasamos de una cantidad de 29.672 bytes a 4.608 bytes más los 319 bytes que ocupa el algoritmo y la tabla auxiliar. Es decir hemos logrado reducir el tamaño total de los mapas en un factor de 6 aproximadamente, o lo que es lo mismo, gracias a comprimir los mapas hemos conseguido un aumento de contenido unas 6 veces mayor que sin el algoritmo compresor.

El proceso que se ha llevado para la compresión de los mapas ha sido la siguiente:

- La librería de desarrollo CPCTelera incorpora un algoritmo de traducción para los archivos **.tmx**, que son los archivos generados por el programa utilizado para crear los mapas, **Tiled**. Del archivo .tmx nos los convierte de forma automática al un vector en .c con los valores de los *tiles* del mapa diseñado. Colocamos los ficheros .tmx dentro de nuestro proyecto y mediante el archivo de configuración *tilemap_conversion.mk* indicamos cuales son los *tilemaps* que queremos que nos lo convierta a un vector en c. Aquí un ejemplo para el *tilemap* 12:

  $(eval $(call TMX2C,img/maps/Maps_game/12.tmx,map_,src/tilemap/,))

- En el paso siguiente nos hemos creado un miniprograma en C para convertir todos los vectores a ficheros c_mapX.bin, siendo X el número del mapa. Una vez tenemos los arrays en .c copiamos sus valores a nuestro programa a su array correspondiente, un ejemplo:

```
const u8 map_66[] = {
21,21,21,21,21,21,8,8,8,21,21,21,21,21,21,
21,21,21,21,21,21,8,8,8,21,21,21,21,21,21,
21,21,21,21,21,21,8,8,8,21,21,21,21,21,21,
21,21,21,21,21,21,1,1,1,21,21,21,21,21,21,
21,21,21,21,21,21,1,1,1,21,21,21,21,21,21,
21,21,1,1,1,1,1,1,1,1,1,1,1,21,21,
21,21,1,1,1,1,1,1,1,1,1,1,1,21,21,
21,21,1,1,1,1,1,1,1,1,1,1,1,21,21,
21,21,1,1,1,1,1,1,1,1,1,1,1,21,21,
21,21,1,1,1,1,1,1,1,1,1,1,1,21,21,
21,21,1,1,1,1,1,1,1,1,1,1,1,21,21,
21,21,1,1,1,1,1,1,1,1,1,1,1,21,21,
21,21,1,1,1,1,1,1,1,1,1,1,1,21,21,
21,21,1,1,1,1,1,1,1,1,1,1,1,21,21,
21,21,1,1,1,1,1,1,1,1,1,1,1,21,21,
21,21,1,1,1,1,1,1,1,1,1,1,1,21,21,
21,21,1,1,1,1,1,1,1,1,1,1,1,21,21,
21,21,1,1,1,1,1,1,1,1,1,1,1,21,21,
21,21,21,21,21,21,21,21,21,21,21,21,21,21,21,
21,21,21,21,21,21,21,21,21,21,21,21,21,21,21,
};
```

Ilustración 33: Ejemplo datos precompresión en programa de compresión

- Así con todos los valores del los mapas. Una vez esté todos los mapas, al ejecutar el programa se generar 67 ficheros correspondientes a los 67 mapas.
- Una vez tenemos todos los ficheros c_mapX.bin generados tenemos que ejecutar el ejecutable que podemos obtener compilando desde el source de exomizer. Debido que el proceso de creación de mapas fue incremental se generó un *script* que hiciera todo el proceso siguiente de forma automática.

```
rm -rf c_*

gcc maps.c -o maps -O3

./maps

for file in $(ls | grep map_)
do
    ./exomizer raw $file -P0 -o c_$file
done

rm -rf map_*
```

Ilustración 34: Script de compresión

El proceso es el siguiente:

  - Se borran los mapas comprimidos generados por la ejecución del script previamente.
  - Se compila el programa de generación de ficheros por si se ha añadido o modificado nuevos mapas.
  - Se ejecuta el programa para generar los ficheros con los mapas.
  - Se recorren todos los ficheros que comiencen por map_, es decir todos los ficheros generados por el programa que los pasa a archivos.
  - Por cada uno se ejecuta el compresor, exomizer.
  - Una vez termina borra todos los ficheros generados por nuestro miniprograma de traspaso ficheros binarios.

De esta manera sencilla comprimimos todos los ficheros a un formato .bin, que de nuevo CPCTelera nos los transforma de forma automática si están en nuestro

directorio /src en archivos .c y .h con los datos pasados a vector. A partir de este punto únicamente nos falta referencia los vectores desde nuestros metadatos de los mapas. De esta manera sencilla podemos pasar de tener los mapas en ficheros .tmx a tenerlos comprimidos y listos para descomprimir en el juego.

4.4.7.7 Problemas con la descompresión de los mapas al ser irregulares.

La descompresión de los datos se realizar en nuestra matriz de 32*27 que hemos denominado mapa. Las matrices en memoria se son distribuidas como si de un vector se tratara. Cuando vamos a descomprimir un mapa de 15*20 por ejemplo, que son las dimensiones más pequeñas que hemos definido para un mapa, la descompresión se hace de forma linear y sin tener en cuenta el número de filas ni de columnas del mapa a descomprimir. A continuación se muestra de forma visual y el por qué del problema que esto genera.

Vamos a suponer que queremos descomprimir un mapa de 5x5 en una matriz de 10x10 para simplificar el ejemplo:

Lo lógico que al descomprimir el supuesto mapa de 5x5 en en nuestra matriz de 10x10 consiguiéramos el siguiente resultado:

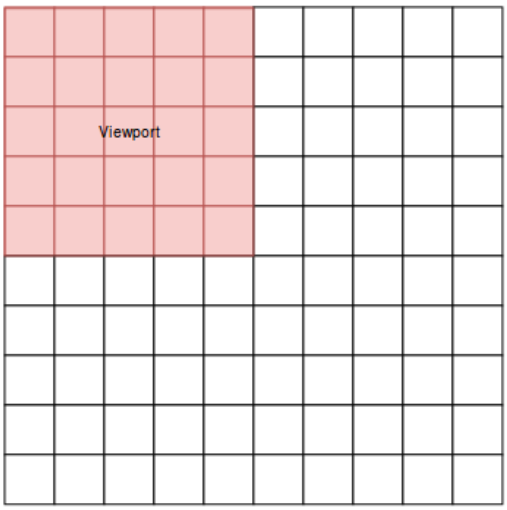

Ilustración 35: Supuesta descompresión deseada de un mapa

Es decir, que el mapa se descomprima de una forma perfecta en memoria respetando las dimensiones del mapa original comprimido. Pero la realidad es que se descomprime de la siguiente manera:

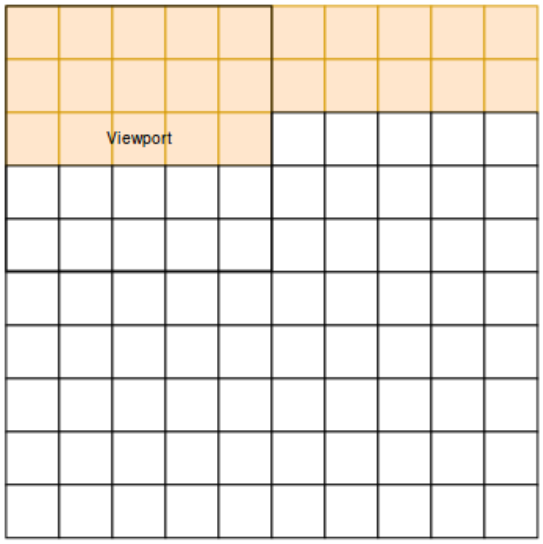

Ilustración 36: Descompresión real de un mapa

Como vemos, la descompresión se hace sin tener en cuenta las dimensiones y por tanto obtenemos un resultado nada favorable. Estos nos va a ocasionar tanto error en el pintado, como errores en la detecciones de colisiones con los obstáculos del mapa.

Tenemos varias alternativas para solucionar el problema:

- Modificar el algoritmo de descompresión para que cada X datos descomprimidos salte de fila en la matriz del mapa. De esta manera lograremos el resultado del primer ejemplo.
- Adaptar el acceso a la información del mapa de la matriz al tamaño de la del mapa local. Esto se puede conseguir realizando el calculo de dirección de forma manual.

En el ejemplo siguiente se muestra en el acceso de información de las colisiones.

El pintado hace uso de las variables *offset_x* e *offset_y*:

```
* ( (u8*)map + (index >> 8 & 0xFF)*width_map + (index & 0xFF) )
```

Ilustración 37: Acceso al mapa dependiente de las dimensiones del mapa

El desplazamiento aritmético y la operación binaria & son indiferentes, se realizan para la extracción de una variable u16 dos varibles de u8 que contienen el índice y e el índice x de la posición del mapa a comprobar. Se podría traducir a :

```
* ( (u8*)map + index_y*width_map + index_x )
```

Ilustración 38: Fórmula de acceso simplificada

De esta manera evitamos realizar map[index_y][index_x] ya que este tipo de acceso no tiene en cuenta las dimensiones del mapa que estamos tratando. Multiplicamos el índice en y por el ancho del mapa actual y le sumamos el *índice_x* para obtener la valor correspondiente. Este resultado se sumará a la posición de memoria map que está casteada a un u8, por lo que el resultado será unos desplazamientos a partir de map hacia delante como resultado tengamos. Por último con el operador * obtenemos el resultado de la posición del mapa correspondiente.

De forma visual este es resultado que obtendríamos:

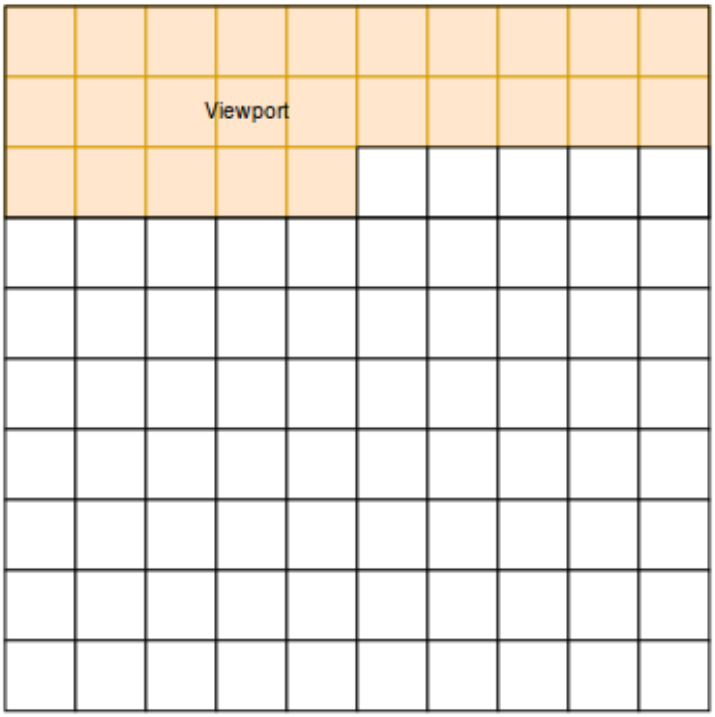

Ilustración 39: Resultado de adaptación del viewport a los datos

Como vemos conseguimos adaptar el *viewport* y el acceso de la información al tamaño del mapa. Para el usuario, la pantalla se vería completamente igual, esto es, de forma rectangular.

Los problemas asociados a la no corrección de la problemática sería un pintado en pantalla impredecible y las entidades no colisionaría donde deben. El resultado sería **catastrófico.**

4.4.8 *Checkpoints*

El juego se compone de 2 fases completamente diferenciadas. Una que es el menú, que una vez pasado se inicia el juego. Para volver al menú hace falta reiniciar el juego o pasárselo por completo, lo que significa que hay que buscar un mecanismo para cuando el héroe se quede sin vidas. Aquí es donde entran los *checkpoints* o los puntos de salvado.

Cuando el jugador cambia de mapa se comprueba si ese mapa es un mapa designado como punto de salvado. Si no lo es no ocurre nada, pero si lo es se copia la información de ese punto de salvado en una estructura especial designada para los puntos de salvado. De esta manera, cuando el héroe muere ocurre el siguiente procedimiento:

- Se aumenta una variable que se mostrará al final del juego indicando cuantas veces ha muerto el héroe.
- Se carga el mapa indicado en la variable del *checkpoint*.
- Se mueve el héroe a la posición que indica el *checkpoint*.
- Se le recarga la vida del héroe hasta 4 puntos de vida, es decir 4 corazones mediante una animación mostrando los corazones aumentar poco a poco.
- Se le devuelve el control del juego al jugador.

4.4.9 Sistema de colisiones.

El sistema de colisionado nos permite una gran versatilidad. Desde realizar acciones al colisionar con ciertas entidades hasta bloquear pasos. En el juego el principal actor que realiza estas comprobaciones es el héroe. Hagamos un repaso de todas las colisiones entre entidades existentes y sus consecuencias.

4.4.9.1 *Detección de colisiones a nivel de entidades.*

Empecemos por las colisiones de héroe y enemigos:

- Colisión con Pato, Gusano, Disparador y la bala: El héroe pierde medio corazón. Se actualiza la interfaz de usuario, cambiando los corazones mostrados.
- Colisión con Girador: El héroe pierde un corazón y sale disparado hacia la dirección que el girador está mirando. Se actualiza la interfaz de usuario, cambiando los corazones mostrados.

Héroe y entidades no móviles:

- Cualquier tipo de llave: La llave desaparece marcando su variable picked a uno. Acto seguido se actualiza la interfaz de usuario incrementando el contador de la llave correspondiente.
- Medio corazón: El corazón desaparece marcando su variable picked a uno. El héroe aumenta su vida en uno y acto seguido se actualiza la interfaz de usuario incrementando el contador de corazones
- Puerta: Si el héroe colisiona con una puerta pero no tiene la llave correspondiente para abrirla la puerta lo empujará hacia atrás denegándole el paso. Por el contrario, si tiene la llave correspondiente, la puerta se abrirá. El proceso es el siguiente:
    - Se modifica el valor del *array* de puertas a 1 indicando que está abierta.

- Se decrementa la llave correspondiente.
- Se actualiza la interfaz de usuario para la llave correspondiente.

Colisiones de enemigos con enemigos:

Cuando un enemigo detecta la colisión con otro enemigo simplemente se bloquean el paso. El bloqueo del paso se hace de la siguiente manera. El enemigo que se está actualizando se desplaza, después del desplazamiento se comprueba si ha colisionado con el resto de enemigos vivos, si ha colisionado se deshace el movimiento. El motivo de esta idea de desarrollo es para evitar el solapamiento de los enemigos ya que es algo irreal.

Colisiones de enemigos con entidades no móviles:

Las detección de colisiones entre cualquier enemigo y entidad no móvil no es efectuado, por lo que, para explicarlo de alguna manera, para los enemigos no existen las entidades no móviles.

Espada y demás entidades:

La espada es una entidad temporal que se genera cuando el héroe acciona la tecla de atacar. Esta entidad dura 4 ciclos de juego en pantalla, y durante estos 4 ciclos se detectan las colisiones con los enemigos y la bala.

- Enemigos: El enemigo decrementará su variable de vidas en uno además de retroceder hacia la dirección de que el héroe está mirando.
- Bala: Al golpear la bala, esta cambiará de dirección hacia donde está mirando el héroe. Al mismo tiempo se cambiará la variable *hit_by_hero* a uno, así conseguirmos que la bala pueda impactar en enemigos.
- Entidades no móviles: No ocurrirá nada ya que no se intentará detectar su colisión.

Ambos portales se explicarán en la siguiente sección, ya que son entidades, pero forman parte de la codificación del mapa.

4.4.9.2 *Detección de colisiones a nivel de mapa*

Las detecciones entre entidades móviles y el mapa es una técnica bastamente extendida y es altamente eficiente, ya que la comprobaciones que se realizan tienen un coste de O(1). Para que esta técnica funcione unicamente es necesario que tengamos nuestros mapas basados en un sistema de *tiles*.

Hay que tener en cuenta que nuestras entidades móviles, ocupan un *tile* de ancho y dos *tiles* de altos, cosa que como veremos nos complicará la detección. La detección de una entidad con un elemento se hace la siguiente manera:

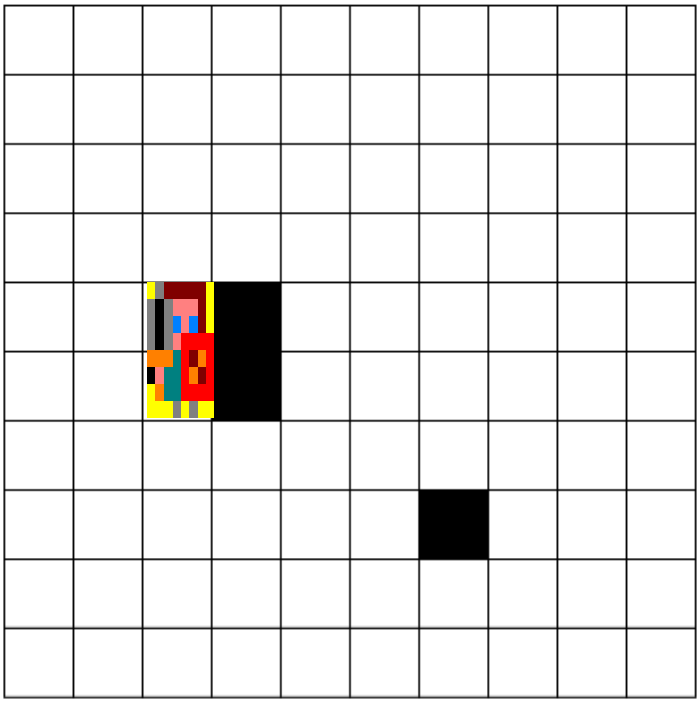

Ilustración 40: Héroe colisionando por la derecha con tiles obstáculos

En el ejemplo de arriba, todos los *tiles* blancos son transitables, mientras que los tiles negros son obstáculos no transitables. Como vemos nuestro personaje tiene bloqueado el paso a la derecha. Una aproximación inicial, dado que toda entidad móvil es rectangular, es comprobar las colisiones en las esquinas de la entidad para comprobar si colisiona en alguna de ellas. Pero caeríamos en el problema siguiente:

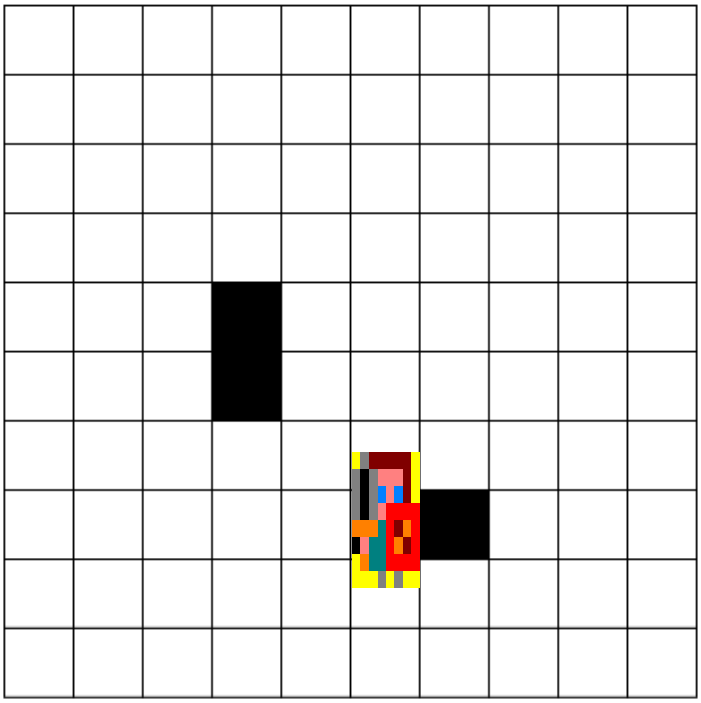

Ilustración 41: Héroe en posición que le permite atravesar un obstáculo

Dado que nuestra entidad es dos *tiles* de alto podría darse el caso de cruzarnos con un obstáculo justo por el centro de nuestra entidad, y como los puntos de detección están en las esquinas, podríamos atravesar el obstáculo cuando no deberíamos poder.

La solución a este problema, es añadir dos puntos extras a cada lado de la entidad en la zona media del personaje. Estas comprobaciones se han implementado mediante 3 funciones para el eje x y dos funciones para el eje y. La comprobación de estas colisiones dependen totalmente de la dirección que la entidad vaya a tomar.

Para los cuatro movimientos posibles así se distribuyen los puntos de colisión. Se explicarán el cálculo de las posiciones de cada punto desde la esquina izquierda superior hasta la esquina derecha inferior:

- Derecha:

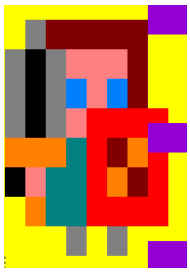

Ilustración 42: Puntos de colisión derecha

  - 1º Punto: Posición x del hero más su ancho. Posición y del héroe.
  - 2º Punto: Posición x del hero más su ancho. Posición y más la mitad del alto del héroe.
  - 3º Punto: Posición x del hero más su ancho. Posición y más la altura del héroe.

- Izquierda:

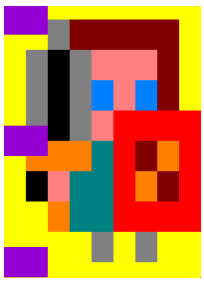

Ilustración 43: Puntos de colisión izquierda

  - 1º Punto: Posición x del héroe. Posición y del héroe.
  - 2º Punto: Posición x del héroe. Posición y más la mitad del alto del héroe.
  - 3º Punto: Posición x del héroe. Posición y más la altura del héroe.

- Arriba:

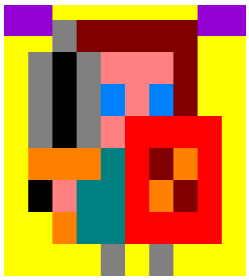

Ilustración 44: Puntos de colisión arriba

  - 1º Punto: Posición x del héroe. Posición y del héroe.
  - 2º Punto: Posición x más el ancho del héroe. Posición y del héroe.

- Abajo:

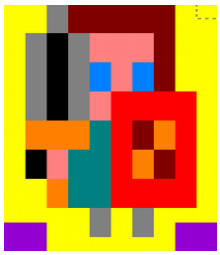

Ilustración 45: Puntos de colisión abajo

  - 1º Punto: Posición x del héroe. Posición y más la altura del héroe.
  - 2º Punto: Posición x más el ancho del héroe. Posición y más la altura del héroe.

Cuando se hace el calculo de las posiciones de los puntos hace falta transformar ambos datos, x a y, a su análogo en tiles. Para saber en qué posición x e y nos encontramos en el mapa tenemos que dividir x por el ancho del mapa en tiles e y por la altura del mapa en tiles. Los cocientes obtenidos de las divisiones son las posiciones x e y correspondientes en el mapa. Teniendo ambas coordenadas podemos acceder al valor asociado en la posición del mapa. El valor obtenido es el índice del tile en el tileset y en base a esta valor podemos decidir la acción que se va a tomar.

La mayoría de los tile corresponden a tiles no transitables o tiles transitables, pero también los hay con funciones especiales. El qué define qué tiles son transitables o cuales no lo son es el desarrollador. Es una decisión de diseño. Por ejemplo, los tiles transitables son hierba, cesped, tierra, etc; mientras que los tiles no transitables son muros, rocas, agua, etc.

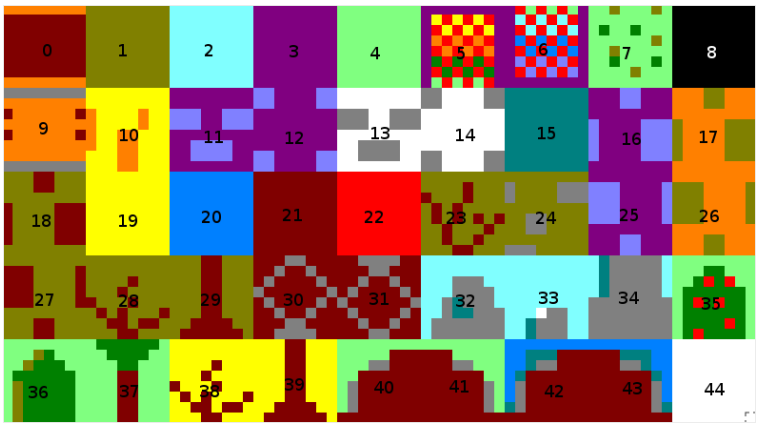

Ilustración 46: Tileset numerado utilizado en el juego

Esta imagen que se muestra es el tileset usado para desarrollar el juego con los índices colados encima. La manera que se ha predispuesto los tiles transitables y los tiles no transitables ha sido la siguiente:

- Transitables: 0-4, 6-7, 9-10
- No transitables: 11-44

Los *tiles* 5 y 8 son dos *tiles* especiales diseñados para actuar como portales locales y externos.

El tipo de acción a tomar al detectar la colisión funciona dependiendo con qué tipo de *tile* hayamos colisionado además de un prioridad. Dado que puede darse el caso que al desplazarse colisione con más de un tipo de *tile*, como puede ser un teletransportador local, un *tile* transitable y un *tile* no transitable debemos realizar una arquitectura de prioridad para saber qué acción tomar.

La prioridad de la acción es la siguiente, por orden de importancia:

1. Teletransportadores locales/externos.
2. Bloque de paso.

3. Permitir paso.

Por lo tanto si nos movemos a la izquierda por ejemplo y colisionamos con un tile no transpasable y con dos sí transpasable, el movimiento no se efectuará porque el no transpasable tiene prioridad.

La función que lleva a cabo la detección de los *tiles* es la siguiente:

```
bool entity_can_move(u16 index, u8 who){

    //true: not obstacle, false: obstacle

    switch( * ( (u8*)map + (index >> 8 & 0xFF)*width_map + (index & 0xFF) ) )
    {
            case 5:
                if(who) portal_touched(index);
                return false;
            case 8:
                if(who) external_portal_touched();
                return false;
            case 0:case 1:case 2:case 3:case 4:case 6:case 7:case 9:case 10:
                return true;
            default:
                return false;
    }
}
```

Ilustración 47: Código de la detección del tile colisionado

Como vemos el *tile* 5 y el 8 que son los teletransportadores tienen su función especial, y los demás se limitan a indicar si puede moverse o no. Esta función es llamada desde las funciones de movimiento de los enemigos y héroe. Todos los puntos de detección de colisión se anidan en un if de forma encadenada con &&, de forma que si hay uno punto que toca un *tile* no transpasable se evalúa a false y no se produce el movimiento.

Los teletransportadores únicamente puede ser accionados por le héroe, por ese motivo en la imagen anterior se filtra quién puede activar la función mediante un if.

Cuando se acciona un portal externo se carga la dirección x e y de la variable ExternalPortal en el héroe, y se carga el mapa que indice que se indique en la variable del portal exterior.

En cuanto al teletransportador local una vez se detecta que ha colisionado con él, mediante el índice en el mapa se comprueba en qué teletransportador nos ubicamos para efectuar el teletransporte. Recordar que la variable del teletransportador local tiene dos x y dos y en *tiles*, por lo tanto con nuestra ubicación actual comprobamos si estamos en las primeras x e y. Si estamos, nos teletransportamos donde ponga en la segunda coordenada x e y, sino es igual que las primera coordenada, nos teletransportaremos a esta. De esta manera podemos

hacer portales que se comuniquen entre ellos, o únicamente un portal que nos teletransporte a donde nosotros indiquemos dentro del mismo mapa.

4.4.10 Renderizado

El renderizado del mapa y todas las entidades se tienen que hacer en un orden dado para lograr el efecto deseado, además en el renderizado de las entidades hay que tener ciertas precauciones para que no se pinten dónde no deben.

El orden de pintado es el siguiente:

1. Mapa
2. Objetos
3. Enemigos
4. Balas
5. Héroe
6. Puertas.

El motivo de establecer un order. El pintado del mapa se debe entender como un pintado por capas. Si se pintan a los enemigo y luego se pintan los objetos, si hay un objeto en la misma posición que un enemigo el objeto aparecerá por encima del enemigo, pero si invertimos el orden, el objeto no se verá porque el enemigo está por encima. Por ello el mapa es lo primero en pintarse ya que sino todo lo anterior pintado no se mostraría.

Hay que tener en cuenta a la hora de pintar todas nuestras entidades que no siempre hay que pintarlas. La decisión de pintarlas o no puede venir de dos direcciones:

- El enemigo no está dentro del *viewport*.
- La entidad no debe pintarse por algún motivo.

4.4.10.1 *Saber si una entidad está dentro del viewport*

Si una entidad no está dentro del *viewport*, significa que no la podemos ver en nuestra pantalla, y por lo tanto no debe pintarse. De pintarse posiblemente se pinte en nuestra interfaz de usuario (donde están las llaves, corazones, etc) y daría un efecto visual completamente erróneo. Puede darse el caso de que una entidad esté dentro de nuestro *viewport* y al mismo tiempo fuera. Esto es que se sitúe media entidad dentro y media fuera. Para esto hay dos alternativas, o diseñar una rutina de pintado que permita pintar por secciones nuestros *sprites* o directamente, sino está completamente dentro de nuestro *viewport*, no pintarlo. Esta última alternativa es la solución dada al problema.

Para comprobar si una entidad está dentro del *viewport* se hace uso de una función modificada especial de colisiones. Únicamente habrá colisión si la entidad está completamente dentro de la pantalla. Para la comprobación se utilizará la posición del *viewport* y de la entidad que estemos comprobando. La formula es la siguiente:

> La x del la entidad debe ser mayor o igual que la x del *viewport*. La x más su ancho de la entidad debe ser menos o igual que la x más el ancho del *viewport*.
>
> Además, la y del la entidad debe ser mayor o igual que la y del *viewport*. Y por último, la y más el alto de la entidad debe ser menos o igual que la y más el alto del *viewport*.

De esta manera sabemos si una entidad está completamente dentro del *viewport*.

4.4.10.2 Condiciones extra para el renderizado

Aparte de que las entidades deban estar dentro del *viewport* es necesario también otra condición. Veamos por entidades:

- Héroe: Siempre se pintará. Se da el caso que cuando ha sido golpeado se pinte de forma intermitente.

- Enemigos: Únicamente se pintarán si tienen la variable *lives* con un valor mayor que cero.
- Llaves y corazones: Se pintarán su variable *picked* está con el valor cero.
- Puertas: Se pintarán si en la posición del vector global que indica el id de la puerta está a cero.
- Balas: Se pintarán si tiene en la variable x un valor diferente a 0xFF.
- Espada: Se dibujará en pantalla únicamente cuando el héroe realice un ataque y se encuentre a una distancia prudencial de los bordes para que la espada no sea dibujada en la interfaz de usuario.

4.4.11 Interfaz de usuario

La interfaz de usuario es una zona o zonas destinada a mostrar información de la partida o del juego al jugador. La interfaz nunca forma parte del *gameplay* en sí. Su única utilidad es mostrar información. Puede ir desde el menú del juego hasta otros aspectos del juego, veamos un ejemplo:

Ilustración 48: Menú principal del juego

Este es el menú principal del juego, su única función es presentar los enemigos del juego, el héroe, la princesa a salvar y por último el título del juego. Es decir, una función meramente informativa y visual.

En el juego tenemos la siguiente interfaz:

Ilustración 49: Pantalla ingame mostrando la interfaz ala derecha

En la interfaz está compuesta por los elementos de la zona marrón. Estos son, el título del juego en la esquina superior derecha y la información de las vidas que le queda al jugador y por último la cantidad de llaves que tiene de cada tipo.

Conforme el héroe recibe un golpe y decrementa su vida o recoge un corazón del suelo los corazones se actualizan acorde. Cada corazón representa dos golpes, por lo tanto, tenemos como máximo 8 de vida.

En cuanto las llaves ocurre lo mismo, el recoger una llave o gastar una llave con una puerta hará que se actualicen mostrando el número disponible de llaves.

La interfaz se pinta en momentos precisos. El menú se pinta cuando se acaba el juego o cuando se inicia, mientras que el la interfaz del juego únicamente cuando inicial el *gameplay*.

4.4.12 Herramientas utilizadas

Para el desarrollo de este proyecto se han usado varias herramientas para facilitar y hacer posible este trabajo. Existen muchas herramientas para el desarrollo de un videojuego, ya sea a nivel artístico o a nivel de código. Estas siguientes herramientas han sido las elegidas:

4.4.12.1 *Desarrollo software*

Para el desarrollo del código se han utilizado dos herramientas fundamentales: una librería, y un compilador.

Librería utilizada

Actualmente existen muchas librerías para el desarrollo de en Amstrad CPC. Algunas de ellas son Cpcrslib, 8 bits de poder, CPCTelera, etc. La librería que se ha elegido para el desarrollo ha sido CPCTelera. Es una librería desarrollada completamente en ensamblador bastante bien optimizada y está pensada para desarrolladores de C y ensamblador. Esta librería incorpora una grandísima cantidad de herramientas desarolladas. Desde la facilidad de generar toda la estructura del proyecto, makefiles preparados para compilar el proyecto y la generación de archivos asociados a contenido visual. Estos es, transformar todas las imágenes que tengamos a sus respectivos archivos en .c y .h con los datos en vectores, la transformación de una imagen en *tiles*, la transformación de los mapas en formato .tmx en archivos .c con los datos en un vector, la transformación de todo archivo .bin que haya bajo la carpeta /src en archivos .c, y muchas herramientas más que facilitan el desarrollo.

Compilador

Existen muchos compiladores pero nos quedaremos con los dos principales y más conocidos, estos son:

- SDCC

- Z88DK

La elección final será SDCC debido a las altas prestaciones que presenta en comparación con Z88DK. El resumen sería el siguiente:

- Estabilidad:
  - Z88DK presenta problemas al iniciar los programas desarrollados en el sistema de fimware de Amstrad.
  - Z88DK falla, deja congelado Amstrad, en algunos casos de uso de variables de 32 bits.
  - Para SDCC no existen problemas conocidos.
- Velocidad:
  - SDCC genera código más rápido en la mayoría de casos, hablando de factores de 6 o 7 veces más rápido.
- Tamaño:
  - SDCC genera en la gran mayoría de casos código mucho menor.

4.4.12.2 *Desarrollo artístico*

Gimp:

Herramienta de libre uso que nos facilitará el proceso de diseño de sprites y tiles. Para la tarea que nos ocupa es mucho más que suficiente.

Tiled:

Herramienta para el diseño de mapas. Es de uso gratuito que permite diseñar mapas de tiles de una manera muy sencilla e importando nuestro tileset para ello.

4.4.12.3 *Compresor*

Existen muchos compresores para procesadores de 8 bits, estos son lo más conocidos:

- Exomizer
- Dan3
- Dan4
- Dan1
- Dan2
- PuCrunch
- MegaLZ
- Pletter, BitBuster
- ZX7
- ApLib aka Apack
- RLE

Después de hacer unas búsquedas el compresor utilizado para el desarrollo de nuestro juego ha sido Exomizer, ya que como he podido comprobar mirando un estudio de benchmarks es el algoritmo que mejor ratio de compresión, aunque sea unos de los más lentos, sino el más lento. Debido a la poca cantidad de datos que tiene que descomprimir este tiempo no es mayor que 0.1 segundos por lo que no es inconveniente. El algoritmo de descompresión consume en código unos 163 bytes aproximadamente y además un plus de 156 bytes para una tabla de descompresión.

4.4.12.4 *Emulador:*

WinApe:

Es el emulador que haremos uso ya que es el que incorpora CPCTelera. El emulador incorpora herramientas de visualización de memoria y la posibilidad de poner *breakpoints* para permitir un debuggeo sencillo.

5. Conclusiones

El estudio y desarrollo de un videojuego me ha hecho ver la grandísima cantidad de técnicas que hay para abordar un mismo problema. No hay una solución general óptima, ya que siempre depende del problema que estemos tratando y de la dimensionalidad del problema que tengamos entre manos. No todo es cuestión de datos y algoritmos, conocer la máquina en la que estamos trabajando es casi tan importante como averiguar cómo darle solución a cierto problema.

El conocimiento de la máquina que he tenido que trabajar con, en este caso, Amstrac CPC, es fundamental. Conocer el entorno es igual de importante tanto para desarrollar, como para poder encontrar errores, como para poder encontrar soluciones. Un ejemplo de muchos es la memoria de video, sin el conocimiento previo de cómo se se estructura, los modos que hay, y las implicaciones de cada uno resulta imposible hacer siquiera funcionar el juego.

Es por ello, que si se quiere realizar un buen proyecto, es totalmente necesario sentarse, estudiar el entorno, averiguar el funcionamiento al 100% del sistema y diseñar lo máximo posible antes de ponerse a teclear, que al fin y al cabo debería ser la parte que menos tiempo debamos dedicar de un proyecto.

El desarrollo de este proyecto me ha parecido muy divertido y un reto, ya que he descubierto nuevas técnicas que no conocía, he tenido que buscar caminos alternativos cuando veía que no había salida y he tenido que descubrir nuevas técnicas que han hecho que pueda incrementar el contenido del videojuego de forma exponencial. El mundo de los videojuegos desde la silla y un mando es divertido, pero animo a cualquier persona que se interese por este mundo a que deje el mando y coja el teclado y ratón, de seguro que encuentra un camino muy interesante en el aprendizaje y aún más sabiendo la gran variedad de herramientas que hay hoy en día para desarrollar un videojuego.

7. Bibliografía:

1. Modelo Espiral: http://www.ojovisual.net/galofarino/modeloespiral.pdf

2. Exomizer: https://bitbucket.org/magli143/exomizer/wiki/Home

3. Benchmark compresores: http://atariage.com/forums/topic/268244-data-compression-benchmark-dan-pletter-zx7-exomizer-pucrunch-megalz/

4. Mejora código exomizer: http://www.amstrad.es/forum/viewtopic.php?t=2608

5. Benchmark SDCC & Z88DK: http://www.cpcmania.com/Docs/Programming/SDCC_vs_z88dk_Comparing_size_and_speed.htm

6. Canal Profesor Retroman: https://www.youtube.com/user/ronaldoCheesetea

7. Documentación CPCTelera: http://lronaldo.github.io/cpctelera/files/readme-txt.html

8. Github CPCTelera: https://github.com/lronaldo/cpctelera

8. Anexos

8.1 Repositorio proyecto

Código del juego: https://github.com/RollitoDelicioso/tfg

8.2 Codificación de entidades

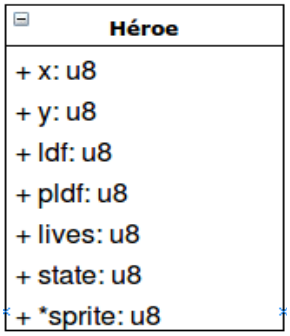

Ilustración 50: Estructuración datos héroe

Existe 1 máximo en memoria.

Como vemos posee las siguientes características:

- *x*: posición x del héroe en el mapa.
- *y*: posición y del héroe en el mapa.
- *ldf* (*Last direction faced*): Dirección a la cual el héroe está mirando.
- *pldf* (*Previous Last direction faced*): Dirección a la cual el héroe miró en la iteración anterior.
- *sprite*: *sprite* que se ha de pintar cuando se invoque la rutina de pintado.
- *lives*: Vida que tiene el héroe en el momento dado, el máximo será 8.
- *state*: Estado que se encuentra el héroe.

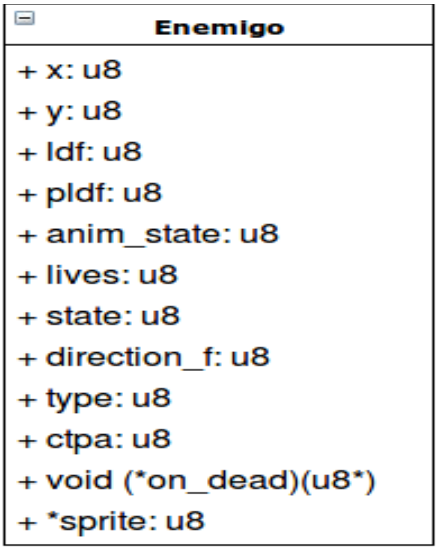

Ilustración 51: Estructuración datos enemigo

Existe 6 máximo en memoria.

Como vemos posee los siguientes atributos:

- *x*: posición x del enemigo en el mapa.
- *y*: posición y del enemigo en el mapa.
- *ldf* (*Last direction faced*): Dirección a la cual el héroe está mirando.
- *pldf* (*Previous Last direction faced*): Dirección a la cual el héroe miró en la iteración anterior.
- *anim_state*: Estado de la animación que se encuentra el enemigo, siempre es 0 o 1, para poder alternar y dar sensación de movimiento.
- *lives*: Vida que tiene el enemigo en el momento dado, el máximo será variable, se puede ajustar como se quiera.
- *state*: Estado que se encuentra el enemigo.
- *direction_f*: Flag que se usará para indicar el movimiento que tiene que efectuar el enemigo cuando sea golpeado por el héroe.
- *type*: Tipo del enemigo. Del 0 al 4.
- *ctpa (Count to perform action)*: Contador contabilizar las iteraciones.

- *on_dead*: Puntero a función, indicará qué acción realizar una vez *lives* llegue a 0.
- *sprite*: *sprite* que se ha de pintar cuando se invoque la rutina de pintado.

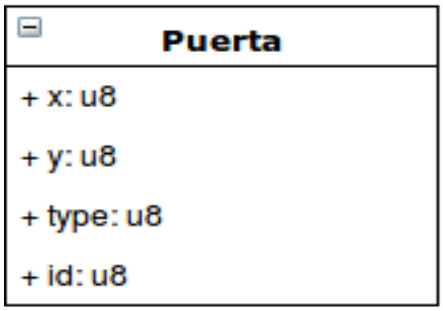

Ilustración 52: Estructuración datos puerta

Existe 1 máximo en memoria.

- *x*: posición x del enemigo en el mapa.
- *y*: posición y del enemigo en el mapa.
- *type*: Tipo de puerta. (0-3)
- *id*: Identificador único para poder mantener un estado global en la aplicación donde todas las puertas que se hayan abierto, nunca se muestren de nuevo cerradas.

  Esto se logra mediante un *array* global, usando el id como como llave para acceder a un valor, 0 cerrada, 1 abierta.

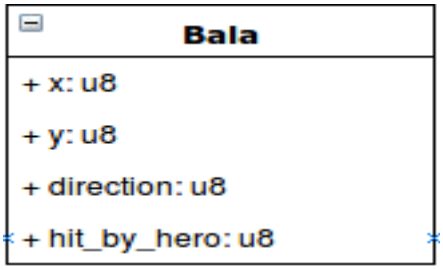

Ilustración 53: Estructuración datos bala

Existe 6 máximo en memoria.

- *x*: posición x del enemigo en el mapa. Si x vale *0xFF* (256) significa que ha impactado.
- *y*: posición y del enemigo en el mapa.
- *direction*: Dirección hacia donde la bala va a realizar el desplazamiento.
- *hit_by_hero*: Variable auxiliar para mejorar la jugabilidad. Indica si la bala ha sido golpeada por el héroe poder devolverla y golpear al enemigo.

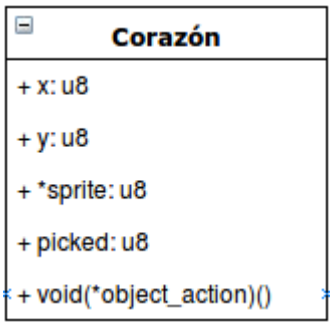

Ilustración 54: Estructuración datos corazón

Existe 4 máximo en memoria.

- *x*: posición x del corazon en el mapa.
- *y*: posición y del corazon en el mapa.
- *sprite*: *sprite* asociado al corazon.
- *picked*: Indica si el objeto ha sido pillado por el héroe.
- *object_action*: Acción asociada a pillar un corazón.

Se hizo dinámico tanto el *sprite* como la acción porque en un principio se planteó la posibilidad de generar medios corazones y corazones enteros, pero se descartó.

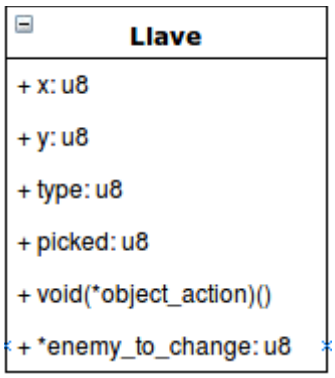

Ilustración 55: Estructuración datos llaves

Existe 1 máximo en memoria.

- *x*: posición x de llave en el mapa.
- *y*: posición y de la llave en el mapa.
- *type*: Tipo de puerta (0-3).
- *picked*: Indica si el objeto ha sido pillado por el héroe.
- *object_action*: Incrementará el contador del tipo de llaves correspondiente.
- *enemy_to_change*: Puntero a enemigo que ha soltado la llave, para que si se adquiere la llave, no vuelve a soltarla.

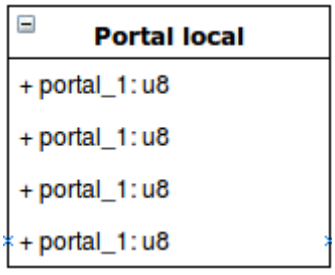

Ilustración 56: Estructuración datos portal local

Existe 1 máximo en memoria.

- *portal_1_x*: posición x del primer portal o destino.
- *portal_1_y*: posición y del primer portal o destino.

- *portal_2_x*: posición x del segundo portal o destino.
- *portal_2_y*: posición y del segundo portal o destino.

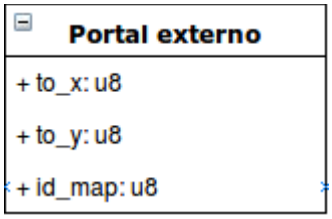

Ilustración 57: Estructuración datos portal exterior

Existe 1 máximo en memoria.

- *to_y*: posición x a la que se va a teletransportar en el nuevo mapa al héroe.
- *to_x*: posición y a la que se va a teletransportar en el nuevo mapa al héroe.
- *id_map*: mapa al cual se va a teletransportar.

Ambos portales funcionan de una manera diferente a las demás ya que el pintado se realiza mediante el mapa de *tiles* que se explicará más adelante, además de que interacción entre el héroe y el el portal se hace mediante el colisionado de *tiles*. Queda una entidad más pero que se verá en su propio apartado, y es la estructura de los metadatos que permiten el cambio de mapa, enemigos, portales, etc.

8.3 Sprites

Una vez que hemos visto cómo se codifican en el juego cada entidad, pongámosle cara a estos elementos:

Héroe:

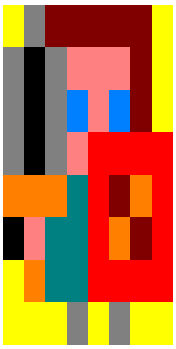

Ilustración 59: Héroe frontal 1

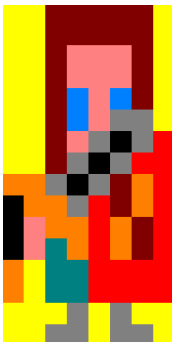

Ilustración 58: Héroe frontal 2

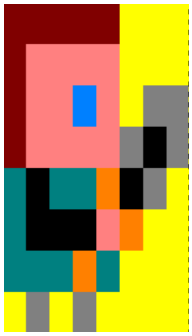

Ilustración 61: Héroe derecha 1

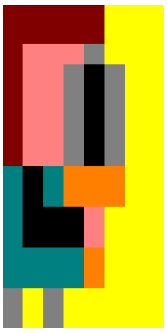

Ilustración 60: Héroe derecha 2

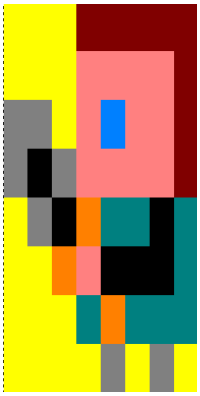

Ilustración 63: Héroe izquierda 1

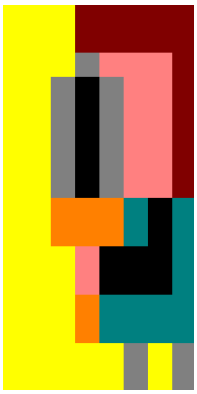

Ilustración 62: Héroe izquierda 2

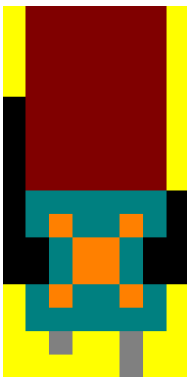

Ilustración 65: Héroe espalda 1

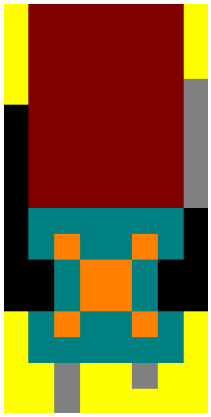

Ilustración 64: Héroe espalda 2

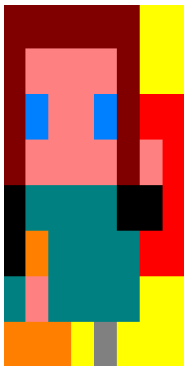

Ilustración 66: Héroe atacando frontal

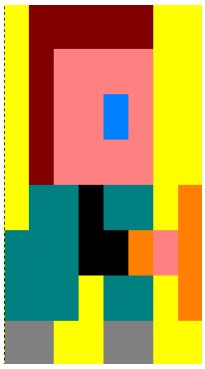

Ilustración 67: Héroe atacando derecha

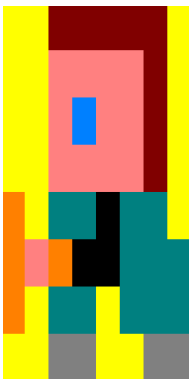

Ilustración 69: Héroe atacando izquierda

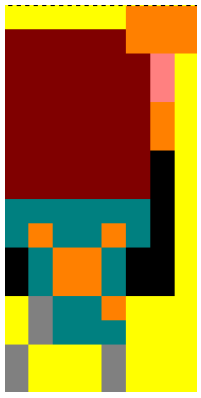

Ilustración 68: Héroe atacando espalda

Enemigos:

Pato:

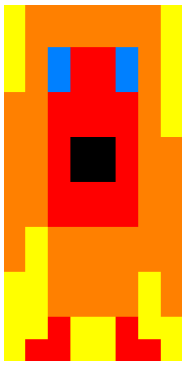

Ilustración 70: Pato frontal 1

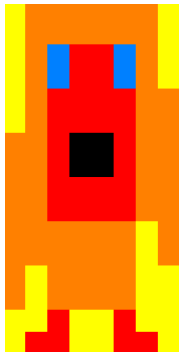

Ilustración 71: Pato frontal 2

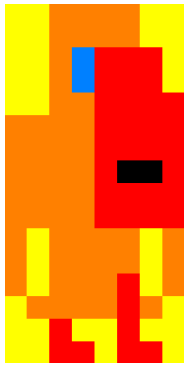

Ilustración 73: Pato derecha 1

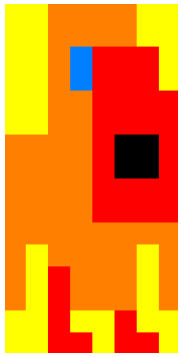

Ilustración 72: Pato derecha 2

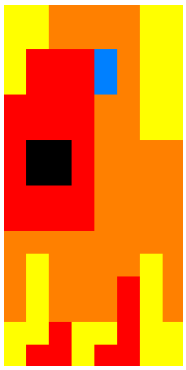

Ilustración 75: Pato izquierda 1

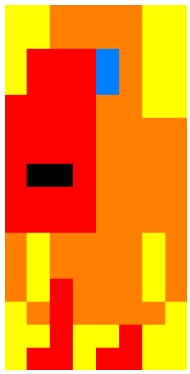

Ilustración 74: Pato izquierda 2

Ilustración 77: Pato espalda 1

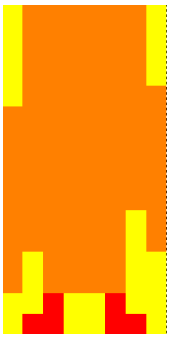

Ilustración 76: Pato espalda 2

Gusano:

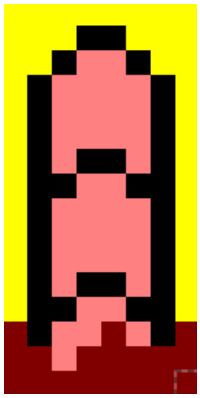

Ilustración 79: Gusano fuera

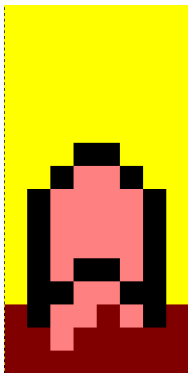

Ilustración 78: Gusano escondiéndose 1

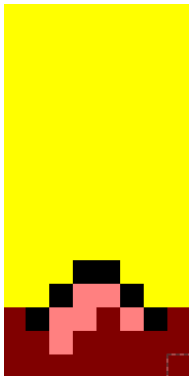

Ilustración 81: Gusano escondiéndose 2

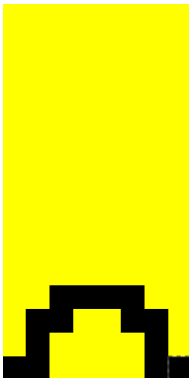

Ilustración 80: Gusano escondido

Girador:

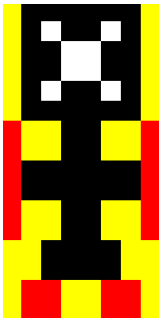

Ilustración 82: Girador 1

Ilustración 83: Girador 2

Disparador:

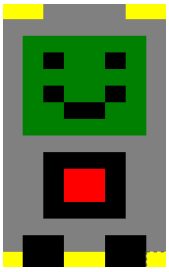

Ilustración 84: Disparador frontal

Ilustración 85: Disparador espalda

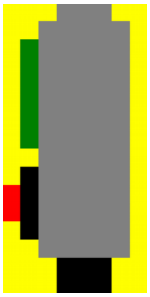

Ilustración 87: Disparador izquierda

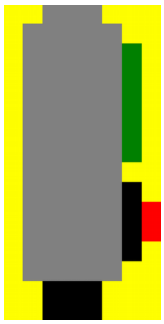

Ilustración 86: Disparador derecha

Ilustración 88: Disparador atacando derecha

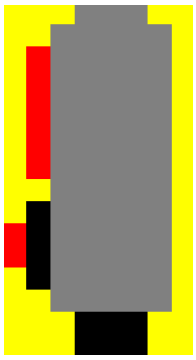

Ilustración 89: Disparador atacando izquierda

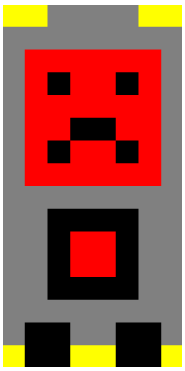

Ilustración 91: Disparador atacando frontal

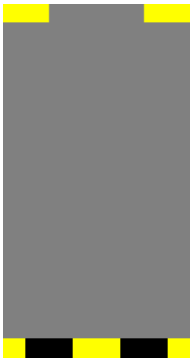

Ilustración 90: Disparador atacando espalda

Princesa:

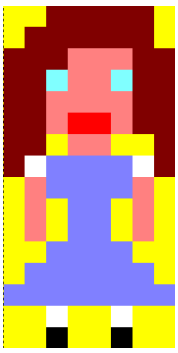

Ilustración 92: Princesa

Espada:

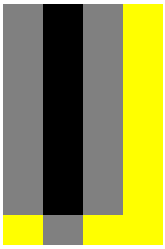

Ilustración 93: Espada abajo 1

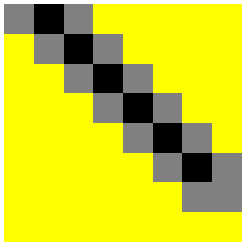

Ilustración 94: Espada abajo 2

Ilustración 96: Espada izquierda 1

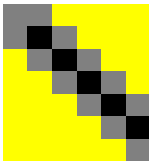

Ilustración 95: Espada izquierda 2

Ilustración 97: Espada derecha 1

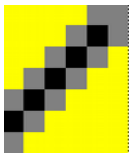

Ilustración 98: Espada derecha 2

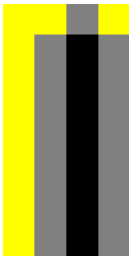

Ilustración 99: Espada arriba 1

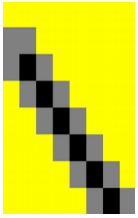

Ilustración 100: Espada arriba 2

Llaves:

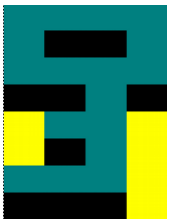

Ilustración 102: Llave azul

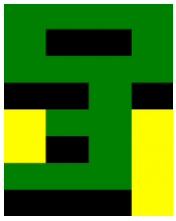

Ilustración 101: Llave verde

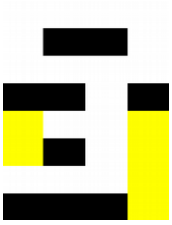

Ilustración 103: Llave blanca

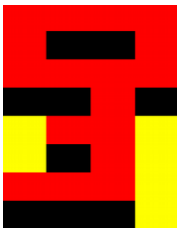

Ilustración 104: Llave roja

Puertas:

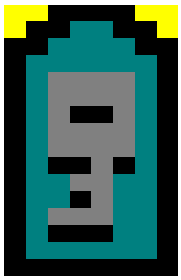

Ilustración 105: Puerta azul

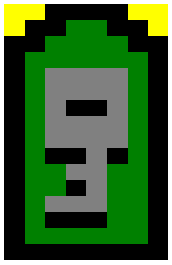

Ilustración 106: Puerta verde

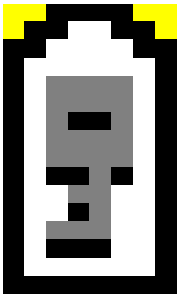

Ilustración 107: Puerta blanca

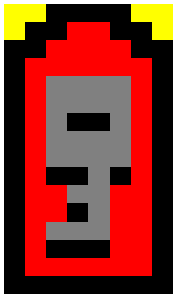

Ilustración 108: Puerta roja

Corazón:

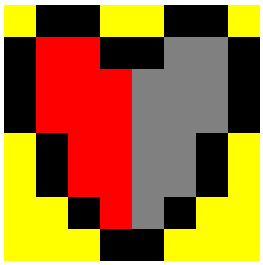

Ilustración 109:
Corazón vida

A excepción de la espada que aparece cuando el héroe ataca, todos estos elementos se pueden encontrar en el juego.

8.3 Cómo jugar

Para poder jugar al videojuego hay dos maneras, mediante un emulador y mediante un AmstradCPC. Dado que el acceso al emulador es mucho más sencillo se explicará este método.

Lo primero de todo es descargar el emulador. Se puede encontrar aquí: www.winape.net/download/WinAPE20B2.zip

Una vez descargado, lo descomprimimos en la carpeta deseada y lo ejecutamos el binario WinApe.exe.

Se iniciará el emulador. Iremos a la pestaña superior izquierda de la ventana llamada “File” → “Tape” → “Insert Tape Image...”. Una vez se nos abra el explorador, buscaremos el fichero “tfg.cdt” y pulsamos en “Abrir”. Una vez hemos introducido el juego, escribimos “***run”tfg***“ en el emulador. El juego cargará y se mostrará.