

PTFC 'Knight Lore 2006'

ÍNDICE

1 INTRODUCCIÓN.....	06
1.1 INTRODUCCIÓN.....	07
1.2 OBJETIVO.....	07
1.3 HERRAMIENTAS.....	08
1.4 TEMPORALIDAD.....	09
1.5 ESTRUCTURA DEL DOCUMENTO.....	09
2 EL Z80 Y EL SPECTRUM.....	11
2.1 EL MICROPROCESADOR Z80.....	12
2.2 EL SINCLAIR ZX SPECTRUM.....	15
3 EMULACIÓN.....	18
3.1 QUE ES UN EMULADOR.....	19
3.2 COMO FUNCIONA UN EMULADOR.....	19
3.3 EL EMULADOR ASPECTRUM.....	26
3.3.1 FUNCIONAMIENTO INTERNO.....	26
3.4 COMPILACIÓN.....	27
4 INGENIERÍA INVERSA.....	29
4.1 CONCEPTOS BÁSICOS DEL JUEGO.....	30
4.2 TÉCNICA FILMATION.....	31
4.3 INGENIERÍA INVERSA.....	32
4.4 INFORMACIÓN DE REFERENCIA.....	33
4.5 HABITACIONES EN LA MEMORIA DE TRABAJO.....	36
4.5.1 TRATAMIENTO DE LOS FONDOS.....	38
4.5.2 TRATAMIENTO DE LOS OBJETOS.....	40
4.6 ELEMENTOS DE LA PANTALLA.....	44
4.7 MUNDO 3D HIGH Y LOW.....	46
4.8 EL PERSONAJE PRINCIPAL.....	48
4.9 LOS GRÁFICOS DE LOS FONDOS Y OBJETOS.....	50
5 IMPLEMENTACIÓN DE KNIGHT LORE 2006.....	53
5.1 PUNTO DE ENTRADA.....	54
5.2 ADQUISICIÓN DE LOS DATOS.....	56
5.3 INICIALIZANDO OPENGL.....	58
5.4 DIBUJANDO LOS FONDOS.....	59
5.5 DIBUJANDO LOS OBJETOS.....	60
5.6 DIBUJANDO EL PERSONAJE PRINCIPAL.....	62
6 RESULTADOS, CONCLUSIONES Y TRABAJOS FUTUROS.....	65
6.1 RESULTADOS.....	66
6.2 CONCLUSIONES.....	68
6.3 TRABAJOS FUTUROS.....	68
7 BIBLIOGRAFÍA.....	70
APÉNDICES.....	73
A. LOS FONDOS.....	74
B. LOS OBJETOS.....	76

1. INTRODUCCIÓN

1.1 INTRODUCCIÓN

Actualmente los videojuegos pasan por un momento de esplendor con la introducción de nuevas herramientas como son las tarjetas gráficas aceleradoras 2D/3D y poderosos algoritmos de estructuración.

Pero siempre se dice que los viejos juegos, principalmente los desarrollados durante los años 80, tienen un encanto del cual los nuevos carecen. Recientemente ha surgido un movimiento consistente en crear remakes de los antiguos juegos, versiones actualizadas de estos juegos que utilizan a fondo las posibilidades que ofrece el hardware actual.

1.2 OBJETIVO

El objetivo de este PTFC es desarrollar el remake de un juego clásico del ZX-Spectrum, el Knight Lore.

Para conseguir unos resultados lo más cercanos al juego original, se ha optado por utilizar un nuevo enfoque donde, en lugar de reconstruir de nuevo el juego desde cero, queremos aprovechar al máximo el original. Dada que este funciona sobre los desaparecidos Spectrum, nos vemos forzados a trabajar sobre un emulador.

La idea principal es realizar la ingeniería inversa del juego para posteriormente desviar el flujo de datos en el momento de generar las imágenes, utilizando nuestros propios algoritmos 3D para crear la escena a partir de los datos generados por el propio juego. De esta forma la jugabilidad permanecerá intacta, sólo actualizaremos los gráficos. El motivo principal para escoger el juego Knight Lore es que trabajo con unas técnicas pseudo 3D (llamada técnica Filmation), que permitirán una reconstrucción parcial de la escena.

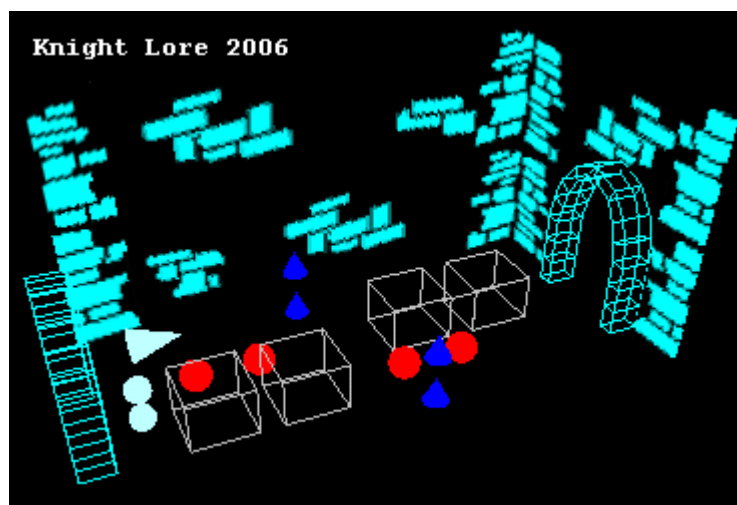


Figura 1.1 – Ejemplo del objetivo del PTFC

1.3 HERRAMIENTAS

La primera herramienta que utilizamos fue el emulador ZX-Spin para el proceso de ingeniería inversa. Es un completo emulador a la par que sencillo, con el cual la tarea de estudiar el funcionamiento del Knight Lore se realizó de manera intuitiva y rápida. Lamentablemente es un programa propietario y su código fuente no está disponible. Existe otro gran emulador, FUSE, que también podríamos haber utilizado para estos menesteres (y como base para el proyecto), está diseñado para trabajar bajo Linux, es un excelente emulador, ampliamente usado por la comunidad de usuarios de los emuladores, pero actualmente presenta complicaciones en su instalación y cuenta con la limitación de trabajar exclusivamente bajo Linux. El emulador escogido como base del proyecto fue el Aspectrum, primeramente por disponer de su código fuente y por sus compatibilidad con muchas de las plataformas existentes.

Como compilador nos decantamos por el MinGW, un popular port de las herramientas de programación GNU para el sistema operativo Windows. Junto al compilador utilizamos también la biblioteca Allegro, que es la que utiliza el emulador Aspectrum bajo Windows. Es una biblioteca para programadores de C/C++ orientada al desarrollo de videojuegos, distribuida libremente, y que funciona en las siguientes plataformas: DOS, Unix (Linux, FreeBSD, Irix, Solaris), Windows, QNX, BeOS y MacOS X. Tiene muchas funciones de gráficos, sonidos, entrada del usuario (teclado, ratón y joystick) y temporizadores. También tiene funciones matemáticas en punto fijo y coma flotante, funciones 3D, funciones para manejar ficheros, ficheros de datos comprimidos y una interfáz gráfica.

Finalmente la representación gráfica 3D en paralelo fue posible mediante el uso de la biblioteca OpenGL que es una especificación estándar que define una API multi-lenguaje multi-plataforma para escribir aplicaciones que producen gráficos 3D, desarrollada originalmente por Silicon Graphics Incorporated. Su nombre significa Open Graphics Library, cuya traducción es biblioteca de gráficos abierta. Entre sus características podemos destacar que es multiplataforma (habiendo incluso un OpenGL ES para móviles), y su gestión de la generación de gráficos 2D y 3D por hardware ofreciendo al programador una API sencilla, estable y compacta. Además su escalabilidad ha permitido que no se haya estancado su desarrollo, permitiendo la creación de extensiones, una serie de añadidos sobre las funcionalidades básicas, en aras de aprovechar las crecientes evoluciones tecnológicas.

1.4 TEMPORALIDAD

La primera tarea a realizar fue la de estudiar el juego, esta se retomó más tarde para contrastar algunos resultados. Posteriormente se estudió el ordenador ZX Spectrum, especialmente su código ensamblador. La siguiente tarea trato sobre la ingeniería inversa, que fue una de las que requirió de más tiempo, pues resultó la más creativa para descubrir vacíos conceptuales. Posteriormente se realizó un estudio del emulador y de su adaptabilidad a nuestras intenciones. Finalmente se realizó la tarea de la implementación del proyecto, que fue la que requirió de más tiempo, pues consistió en su mayoría en poner a la práctico muchos de los conocimientos adquiridos en anteriores etapas.

	2005												2006							
	4	5	6	7	8	9	10	11	12	1	2	3	4	5	6	7	8			
Knight Lore	■			■																
ZX Spectrum		■	■																	
Ingeniería inversa				■	■	■	■													
Emulador								■	■											■
Implementación										■	■	■	■	■	■	■	■			

1.5 ESTRUCTURA DEL DOCUMENTO

Este documento está estructurado siguiendo el camino recorrido, conceptualmente, que ha llevado a la finalización del proyecto:

- Una introducción al proyecto.
- Una capítulo dedicado al microprocesador Z80 y al ordenador ZX Spectrum, su historia, capacidades y posibilidades.
- Toda la información necesaria para entender y comprender la emulación y el emulador Aspectrum.
- Un extenso capítulo dedicado a la ingeniería inversa y como se aplicó en el proyecto para descifrar el funcionamiento a nivel gráfico del juego.
- Implementación del nuevo entorno enumerando los pasos para conseguir el resultado final.
- Un capítulo final comentando los resultados, conclusiones y trabajos futuros.

2. EL Z80 Y SPECTRUM

2.1 EL MICROPROCESADOR Z80

El Zilog Z80 (Z80) es un microprocesador de 8 bits que fue lanzado al mercado en el año 1976 por la compañía Zilog. Su arquitectura se caracteriza por estar a medio camino entre la organización de acumulador y de registros de propósito general. Se popularizó en los años 80 través de los ordenadores personales como el Sinclair ZX-Spectrum, el Amstrad CPC o los ordenadores de sistema MSX. Es uno de los procesadores de más éxito del mercado, del cual se han producido infinidad de versiones clónicas, y sigue siendo usado de forma extensiva en la actualidad en multitud de dispositivos empotrados.



Figura 2.1 - Pastilla del Z80

El Z80 fue diseñado principalmente por Federico Faggin, que estuvo trabajando en Intel como diseñador jefe del Intel 4004 y del Intel 8080. En 1974 dejó la compañía para fundar Zilog y comenzó a trabajar en el diseño de Z80 basándose en su experiencia con el Intel 8080.

El Z80 estaba diseñado para ser compatible a nivel de código con el Intel 8080, de forma que la mayoría de los programas para el 8080 pudieran funcionar en él, especialmente el sistema operativo CP/M.

El Z80 tenía ocho mejoras fundamentales respecto al Intel 8080:

- Un conjunto de instrucciones mejorado, incluyendo los nuevos registros índice IX e IY y las instrucciones necesarias para manejarlos.
- Dos bancos de registros que podían ser cambiados de forma rápida para acelerar la respuesta a interrupciones.
- Instrucciones de movimiento de bloques, E/S de bloques y búsqueda de bytes.
- Instrucciones de manipulación de bits.
- Un contador de direcciones para el refresco de la DRAM integrado, que en el 8080 tenía que ser proporcionado por el conjunto de circuitos de soporte.
- Alimentación única de 5 voltios.
- Necesidad de menos circuitos auxiliares, tanto para la generación de la señal de reloj como para el enlace con la memoria y la E/S.
- Más barato que el Intel 8080.

El Z80 eliminó rápidamente al Intel 8080 del mercado y se convirtió en uno de los procesadores de 8 bits más populares. Las primeras versiones funcionaban a 2,5 MHz, pero su velocidad ha aumentado hasta los 20 MHz. Así, la versión más utilizada, el Z80A funcionaba a 4 MHz.

A comienzos de los años 80 el Z80, o versiones clónicas del mismo, fue usado en multitud de ordenadores domésticos. Posteriormente, en los 90, el Z80 ha sido usado en las consolas Master System, Game Gear y Mega Drive de Sega. Las consolas de Game Boy y Game Boy Color de Nintendo utilizan una variante del Z80 fabricada por Sharp. En la actualidad parte de la gama de calculadoras gráficas programables de Texas Instruments emplean una versión clónica del Z80 fabricada por NEC como procesador principal. Además el Z80 también es un microprocesador popular para ser usado en sistemas empujados, campo donde se emplea de manera extensiva.

A pesar de ser un microprocesador de 8 bits, el Z80 puede manejar instrucciones de 16 bits y puede direccionar hasta 64 Kb de RAM. Una de las características más reseñables es que tiene las instrucciones del Intel 8080 como subconjunto, de modo que algunos ordenadores basados en Z80 podían ejecutar programas diseñados para el CP/M. Esto ha hecho que los formatos de instrucción del Z80 sean bastante complejos, ya que tienen que mantener su compatibilidad con el 8080. Sin embargo el Z80 ha conseguido mejorar al microprocesador de Intel en velocidad, ha añadido nuevos modos de direccionamiento y contiene un juego de instrucciones más amplio.

La estructura de registros del Z80 esta compuesto por un banco principal, otro alternativo y por último un banco compuesto por registros especiales. La existencia del banco alternativo mejora la velocidad ante la presencia de las interrupciones ya que permite cambiar desde el banco principal al alternativo.

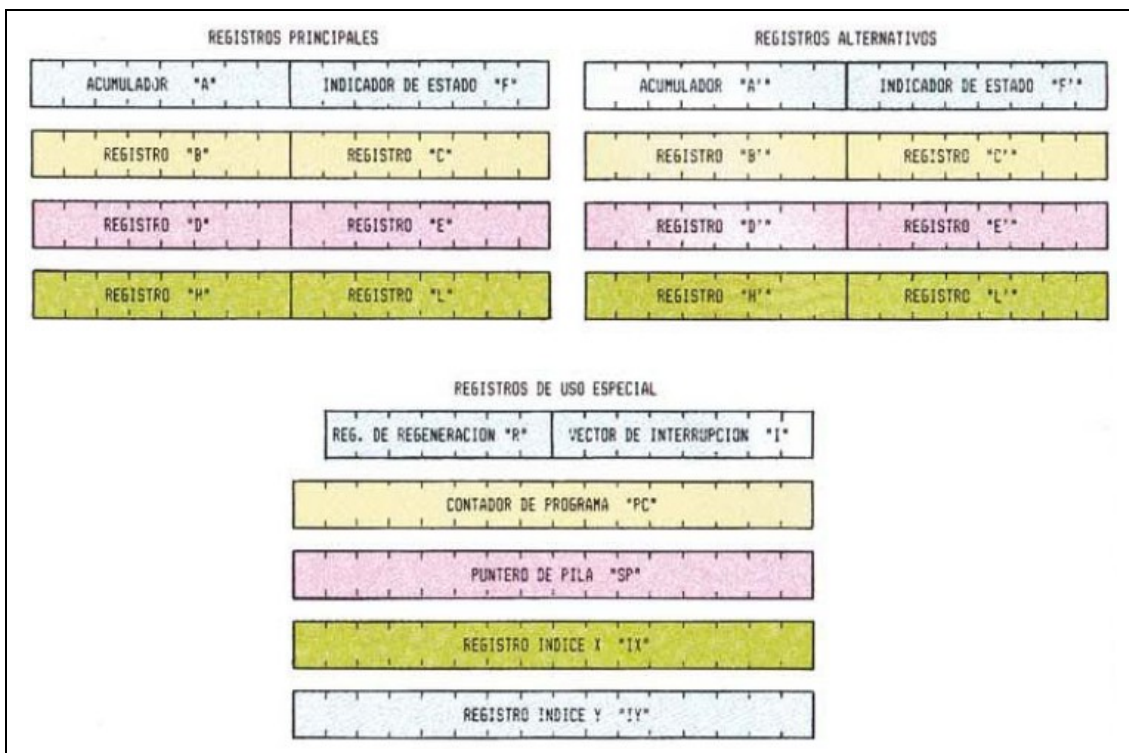


Figura 2.2 – Registros del Z80

Los registros del banco principal son generales y de 8 bits. Se pueden tomar por parejas, siendo entonces IX e IY los registros índices. El registro A sirve de acumulador. El R almacena el bloque de memoria a cuyo refresco se va a proceder. El SP es el puntero de cima de pila. El PC es el contador de programa. El F contiene los flags o también llamados bits de condición.

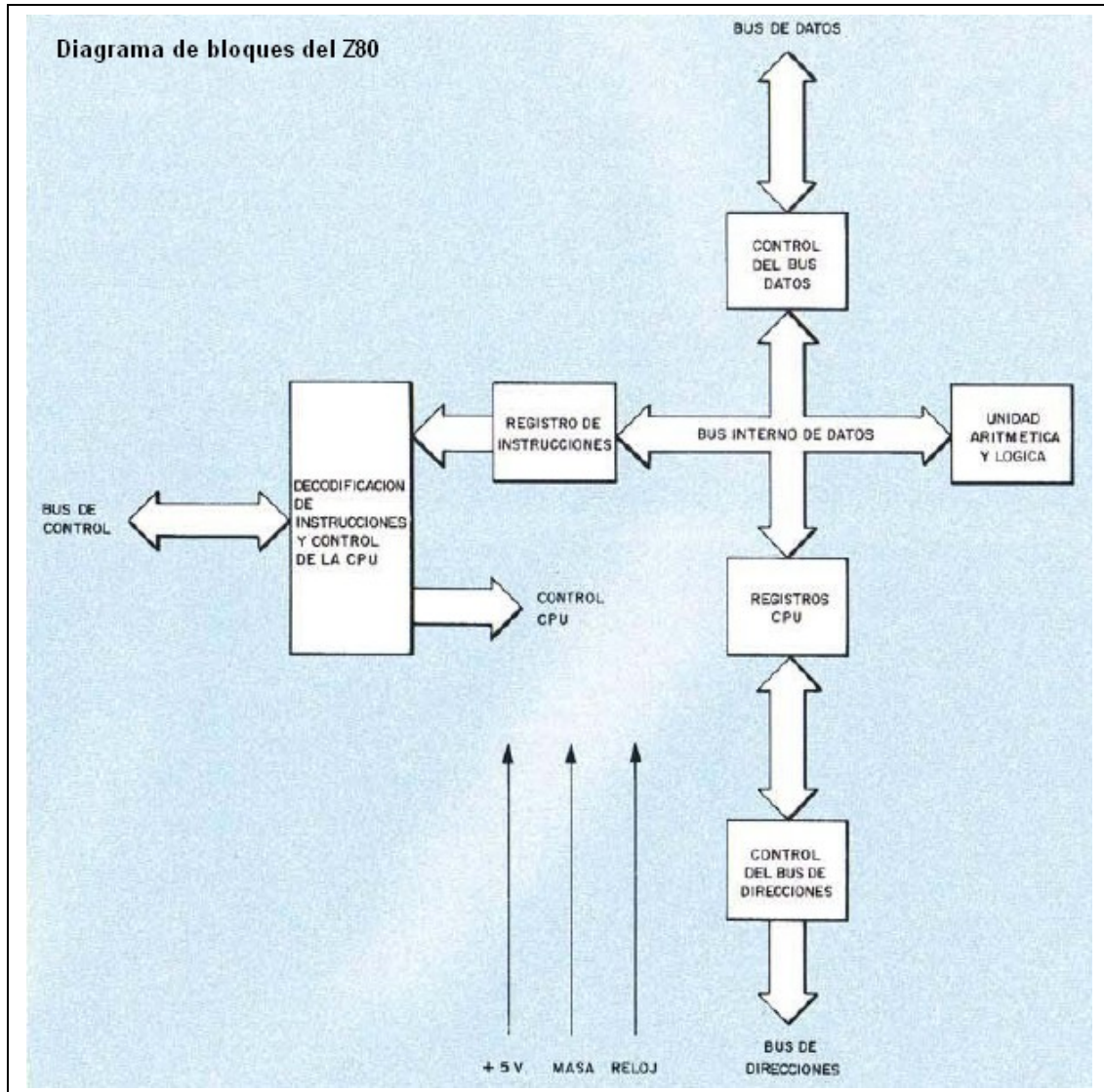


Figura 2.3 – Diagrama de bloques del Z80

2.2 EL SINCLAIR ZX SPECTRUM

El Sinclair ZX Spectrum fue un microordenador de 8 bits basado en el microprocesador Z80 de Zilog, fabricado por la compañía británica Sinclair Research y lanzado al mercado europeo en el año 1982. El hardware fue diseñado por Richard Altwasser y el software por Steve Vickers. El Sinclair ZX Spectrum fue el microordenador doméstico más popular de la década de los 80.

Sinclair ZX Spectrum de 48K



Figura 2.4 – El Sinclair ZX Spectrum 48K

Sus características principales eran:

- Microprocesador Zilog Z80 a 3.5 Mhz (bus de datos de 8 bits y bus de direcciones de 16 bits). Contenía también un chip denominado ULA (Uncommitted Logia Array) que reducía el número de chips necesarios.
- Resolución gráfica máxima (y única) de 256x192 pixels. Tenía una ingeniosa manera de implementar el video con 16 colores usando sólo 8 KB de memoria RAM. La señal de vídeo era modulada para su visualización en un televisor normal y corriente.
- Diversas configuraciones de RAM con 16K ó 48K.
- 16 KB de ROM (incluía un intérprete del lenguaje BASIC SINCLAIR desarrollado por la compañía Nine Tiles Ltd. para Sinclair y que era una evolución del que ya desarrollaran para dos anteriores máquinas comerciales de la marca, el ZX-80 y el ZX-81, y de las que el Spectrum es continuador).
- Teclado de caucho integrado en el ordenador (en el modelo de 16K y en la primera versión de 48k. Un modelo que incorporaba un teclado mejorado llevó el nombre de ZX Spectrum Plus).
- Sistema de almacenamiento por cinta de casete de audio común a 1.200 baudios (la velocidad soportada por el sistema operativo en ROM, pero había juegos que usaban su propio sistema de carga "turbo" a mayor velocidad, aunque algo más propensos a producir errores de carga).

Todas estas características convertían al ZX Spectrum en un equipo muy asequible, lo que acercó la microinformática a un elevado número de personas. Con el paso de los años fueron apareciendo diversos periféricos como por ejemplo Interface 1, disqueteras (Discovery, Microdrive), lápices ópticos, impresoras o mandos de juego.

El Interface 1 era un puerto RS232 que le daba al Spectrum características de red, ya que permitía interconectar hasta 64 de ellas.

El Microdrive era una unidad permitía leer unos cartuchitos de una cinta magnética sin fin. Cada cartucho almacenaba 85 Kb. Su coste era mucho menor que el de una disquetera de cualquier otra computadora de la época. Los cartuchos se basaban en una idea sencilla y a la vez ingeniosa: una pequeña cinta magnética sin fin enrollada, que se desplazaba a gran velocidad por la acción del rodillo que incorporaban las unidades lectoras, siendo leída o grabada la información por un cabezal. Los cartuchos microdrives contenían una cinta de unas 200 pulgadas, que era desplazada a 28 pulgadas por segundo. En menos de 8 segundos, la cinta había dado una vuelta completa, lo cuál significaba que cargaba alrededor de 15 Kb. por segundo. Sin embargo, la alta velocidad unida al roce con la cabeza lectora/grabadora, provocaban un desgaste muy rápido de la cinta que no leía bien y se cortaba bastante seguido.



Figura 2.5 – ZX Spectrum con Interface 1 y Microdrive

El Spectrum, a pesar de sus obvias limitaciones, fue un boom. Se vendió en más de 30 países y se convirtió el líder en Europa superando en ventas al Commodore 64, que era superior técnicamente.

3. EMULACIÓN

3.1 QUE ES UN EMULADOR

Un emulador es un programa destinado a recrear internamente el funcionamiento de una arquitectura diferente a aquella en que se ejecuta. El emulador no es más que un programa, sin partes hardware, que utilizando los recursos de la máquina donde se ejecuta, simula el comportamiento de la CPU, memoria y demás elementos de una máquina determinada.

Por ejemplo, ZXSpin es un emulador de Spectrum, ya que es un programa diseñado para emular un micro Z80 (que es el micro del Spectrum), una ULA (que es el "chip gráfico" del Spectrum), una unidad de cinta o de disco, etc... Del mismo modo que existe ZXSpin, NESTicle es un emulador de NES que emula el micro de la NES, los chips de la NES, es capaz de ejecutar juegos de cartucho de NES, etc.

Normalmente los emuladores son programas que simulan una única arquitectura, aunque en algunos casos existen "macroemuladores" capaces de emular varios sistemas, como por ejemplo MAME (Multi Arcade Machine Emulator) que emula gran variedad de microprocesadores diferentes, lo cual le permite recrear gran cantidad de máquinas recreativas. Nótese un detalle muy importante: un emulador no es más que un programa como cualquier otro instalado en la máquina. La diferencia está en que es vez de editar un texto, hacer cálculos, dibujar, o jugar (como el resto de programas), lo que hace es comportarse tal y como lo haría el sistema emulado.

Actualmente hay muchísimos sistemas emulados:

- Consolas: NES, SuperNES, Game Boy, Nintendo 64, Master System, Megadrive, Game Gear, Saturn, Atari 2600, Atari 7800, Lynx, Neo Geo, TurboGrafx, ...
- Máquinas recreativas: Los emuladores MAME y RAINE.
- Ordenadores: Spectrum, Amstrad, Commodore, MSX, Atari ST, Amiga, PC, etc.

3.2 COMO FUNCIONA UN EMULADOR

Un emulador, a grandes rasgos, simula un microprocesador. Por ejemplo, es posible hacer un programa que lea instrucciones del microprocesador Z80, las comprenda, las ejecute, y guarde los resultados de las ejecuciones.

Un emulador de Z80 puede estar implementado en diversos lenguajes como C, Visual BASIC, PASCAL o en ensamblador (por citar algunos lenguajes comunes) que entienda las instrucciones en código máquina del Z80 proporcionadas al emulador, las ejecute y cambie los registros emulados del microprocesador, quedando todos los registros emulados igual que quedarían en un Z80 real si ejecutara el mismo código. Así, se puede realizar un programa en código máquina de Z80, y al ejecutarlo en un emulador,

o en un Z80 real, se obtendrían exactamente los mismos resultados en los diferentes registros del microprocesador. Un emulador recrear el hardware del sistema.

Pero no es suficiente con tan sólo recrear al hardware de una máquina (así sólo emularíamos un microprocesador o un chip gráfico, y estos últimos por si mismos no son capaces de hacer nada), para que el emulador realmente pueda hacer algo, necesita un Sistema Operativo (más o menos complejo). Este sistema solía ser almacenado en un chip ROM (memoria de sólo lectura) con las funciones básicas grabadas en él. Antiguamente, y muy especialmente en el caso del spectrum, los conceptos de sistema operativo y el intérprete Basic estaban mezclados por lo que se hace difícil hacer una distinción.

Al arrancar un Spectrum, por ejemplo, lo que se tiene es una ROM de 16KB (16384 bytes) que contiene el arranque de la máquina, el intérprete de BASIC, y todas las funciones necesarias para cargar y ejecutar el software de Spectrum. Esta ROM en realidad es un chip de memoria con el Sistema Operativo del Spectrum (un programa escrito en código máquina de microprocesador Z80) grabado en él. Este programa no tiene diferencias prácticas con un juego o un programa en cinta o disco de Spectrum, simplemente que su función no es jugar, sino dotarnos de un interfaz de gestión del Spectrum (el BASIC) y que en lugar de estar grabado en una cinta, lo está en una memoria de sólo lectura. Es decir, hubo una persona (o varias) que programó (como si fuera un juego) el arranque de la máquina, el menú, el BASIC, etc., y lo ensambló con un "programa ensamblador" para obtener el código máquina almacenado en esa memoria ROM.

El emulador replica un micro Z80 (y los demás componentes del Spectrum), y lo primero que hace es cargar desde fichero el contenido de la ROM del Spectrum (rom48k.rom), de forma que el emulador, al igual que el Spectrum real, pueda arrancar, ejecutando las instrucciones de la ROM del Spectrum mediante su "microprocesador emulado". Para obtener este fichero se coge un chip con la ROM del Spectrum, se inserta en un lector/grabador de memorias, y se lee, grabando el contenido de los 16384 bytes en un fichero. Cada byte de este fichero se podría decir que es una instrucción del programa de la ROM del Spectrum.

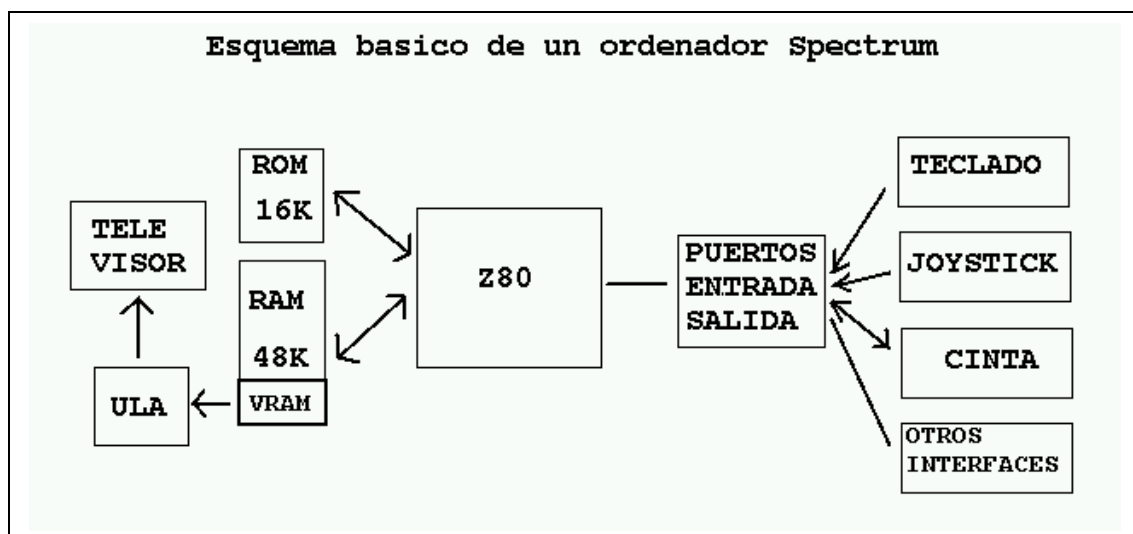


Figura 3.1 – Esquema básico de un ordenador Spectrum

La parte central es un micro Z80 el cual ve la ROM y la RAM de forma continuada como la totalidad de su memoria. Es decir, ve 64KB de memoria de los cuales los primeros 16K son el contenido de chip de ROM, y los siguientes 48K los del chip de RAM. Al encender el Spectrum éste se inicializa y muestra el BASIC porque comienza a ejecutar instrucciones desde la dirección de memoria 0, donde está el principio de la ROM de 16K, es decir, el intérprete BASIC.

Es por eso que, cuando encendemos el ordenador, se "ejecuta" la ROM. Al encender un Spectrum (que perdió en su apagado toda alimentación eléctrica) todos los registros del microprocesador Z80 valen 0, incluido uno que se llama PC (Program Counter o Contador de Programa), que es el que apunta a la siguiente instrucción que el Z80 debe leer y ejecutar. Un microprocesador funciona a grandes rasgos de la siguiente forma:

- Leer instrucción apuntada por el registro PC.
- Incrementar PC para apuntar a la siguiente instrucción.
- Ejecutar la instrucción recién leída.
- Repetir continuadamente los 3 pasos anteriores.

Así pues, al encender el ordenador, PC vale 0. Al estar la ROM mapeada en la posición de memoria 0 (mediante cableado hardware de los chips de memoria en la placa del Spectrum), lo que pasa al encender el ordenador es que ese contador de programa (PC) está apuntando al principio de la ROM, y es por eso que se ejecuta la ROM paso a paso, instrucción a instrucción, cada vez que lo encendemos. Para el Spectrum todos los chips de memoria de su interior son como si fuera un gran baúl de 64KB continuados, algo que se consigue mediante cableado de los diferentes chips a las patillas correctas del microprocesador. A grandes rasgos, las patillas de datos y de direcciones del microprocesador están conectadas a los diferentes chips de memoria de forma que cuando el micro lee datos de la memoria, lo ve todo como si fuera un sólo chip de memoria de 64KB. Esto se consigue con un sencillo proceso de diseño (al hacer el esquema del ordenador antes de fabricarlo) conocido como "mapeado de memoria".

En el mapa de memoria del Spectrum, los primeros 16KB son la ROM (que está en un chip aparte, pero que como acabamos de ver es algo que el Spectrum no distingue, ya que la visualiza como una sección de memoria continua desde la posición 0 hasta la 16383 de su "baúl total" de 64KB) y luego viene la RAM, a partir de la posición 16384. Ahí es donde se almacenan los programas, los gráficos de la pantalla (en un trozo determinado de esa memoria), etc. En esta RAM es donde el intérprete de BASIC introduce los programas para su ejecución.

Estos programas pueden entrar desde los diferentes dispositivos de entrada/salida (gestionados por el Z80) como el teclado, la cinta o disco, etc.

Cabe hacer una mención especial a que una parte de la memoria RAM (desde el byte 16384 hasta el 23296) está conectada con la ULA, el chip "gráfico" del Spectrum, y encargado de convertir el contenido de esta "videoram" o VRAM a señales de vídeo para la televisión. Cuando los juegos dibujan gráficos, sprites o cualquier otra cosa en pantalla, en realidad están escribiendo bytes en estas posiciones de memoria, que la ULA muestra en la TV en el siguiente refresco de la pantalla.

Visto de una manera simple: al escribir un valor numérico (por ejemplo un 1) en alguna dirección de esta parte de la RAM, de forma inmediata aparece un punto en nuestro monitor, ya que la ULA está continuamente "escaneando" la videoram (de forma independiente del Z80) para reflejar en el monitor o televisión todos los valores numéricos que introduzcamos en ella. En los PCs ocurre igual: al escribir un valor numérico en una determinada posición de memoria, aparecen puntos en pantalla. Según en qué dirección escribamos aparecen en un lugar u otro de la pantalla. En algunos modos de vídeo (320x200, por ejemplo), escribir en la posición `A000h` hace aparecer un punto de color en la posición (0,0) del monitor, hacerlo en `A001h`, lo hace aparecer en (1,0), y así un pixel tras otro.

Con esto obtenemos un software que emula un micro (el Z80) y que gracias a la ROM del mismo arranca y nos muestra y permite usar el Spectrum en sí mismo, tal y como se podría usar un Spectrum recién arrancado. El emulador lo que hace, en resumen, es leer los eventos de teclado y comunicarlos al micro emulado de la misma forma en que lo hacía el Spectrum real al pulsar una tecla. Al mismo tiempo, lee de la memoria del Spectrum el contenido de la pantalla, y lo muestra en nuestro monitor tal y como lo "leería" la Televisión. El microprocesador virtual, mientras tanto, se dedica simple y únicamente a ejecutar instrucciones, ya sea de la ROM o de un juego que carguemos o ejecutemos en él.

La ROM es muy importante, ya que indica cómo debe de comportarse el microprocesador en todo momento, cómo debe atender las interrupciones recibidas por el teclado y los mandos, etc. Por ejemplo, un ordenador ZX Spectrum y un Amstrad CPC tienen el mismo microprocesador, un Z80, pero sin embargo son sustancialmente diferentes. ¿Por qué? Pues porque aparte de que los circuitos que acompañan al micro son diferentes, la ROM es totalmente diferente, de forma que cambia el Sistema Operativo, las direcciones de memoria donde se guardan los datos de pantalla, etc.

El mero hecho de poder utilizar una arquitectura determinada y su sistema operativo ya podría ser un gran aliciente, pero por si esto fuera poco, en la mayoría de los casos podemos además usar todo el software original del sistema físico en el emulador.

Ya sea un cartucho, una cinta o un disco, el objetivo es obtener una copia en formato digital (en un fichero) de sus datos, de forma que se puedan cargar en el emulador. Veamos los diferentes tipos de software a emular:

ROMs del sistema: como ya hemos visto, las ROMs de las máquinas se obtienen volcando el contenido de los chips de memoria de las mismas, donde están almacenados los programas en Código Máquina, mediante lectores de memorias o similares.



Figura 3.2 – Circuitería de un ZX Spectrum

ROMs de cartuchos: los cartuchos de consola no suelen ser más que soportes de plástico de una forma determinada, que en su interior contienen chips de memoria (iguales que los chips de la ROM) conectados eléctricamente a los contactos metálicos o pines que sobresalen del plástico del cartucho. En realidad los juegos de consola no son más que ROMs del sistema, ya que realmente las consolas no suelen tener ROM (si las encendemos sin juego dentro, no hacen nada), y somos nosotros los que introducimos una ROM (que en realidad es el juego) al introducir el cartucho. Esa ROM, en lugar de ser un Sistema Operativo, es un juego. Recordemos que el microprocesador lo único que sabe hacer es ejecutar instrucciones en código máquina, sea un juego, o sea un intérprete de BASIC. En algunos sistemas sí que tenemos una ROM pregrabada en la máquina capaz de realizar tareas cuando no introducimos ningún juego. De esta forma, las consolas clónicas de NES podían llevar cientos de juegos grabados en un chip ROM interno, de forma que cuando la encendemos sin introducir un cartucho, ese chip se activa y se ejecuta su contenido. Cuando insertamos un juego, por contra, el chip que realmente se convierte en la memoria de la máquina (y que por tanto se ejecuta) es el ubicado en el interior del cartucho. Para volcar los juegos a ficheros de disco de forma que se puedan usar en los emuladores, basta con desmontar el cartucho, sacar el chip de memoria con el código del juego grabado, y al igual que en el caso de las ROMs del sistema, volcarlo a fichero con un lector de chips de memoria. Otra opción sería utilizar alguno de los lectores de cartuchos que se insertaban en la máquina y permitían grabar a disquete los contenidos de las ROMs. Estos aparatos fueron principalmente utilizados como "copiones". En general, en el mundo de la emulación, se le llama ROM a todo fichero volcado a disco desde una memoria. Los ficheros de ROM tendrán diferentes formatos según la arquitectura de la máquina. Por ejemplo, el fichero con extensión smc será una ROM de cartucho de SuperNES, la extensión smd de Sega MegaDrive, la extensión nes un juego de NES, la extensión gb de Gameboy, etc. Así mismo, existen ROMs en el Spectrum, obtenidas de los cartuchos del Interface 2 (un periférico de Spectrum que permitía cargar juegos en cartucho, realizando exactamente la misma función que la inserción del cartucho en una consola). Las máquinas recreativas son placas con microprocesadores y circuitos propios (al estilo de una consola) donde los juegos suelen estar grabados también en chips de ROM, o bien se introducen como si fueran cartuchos. El procedimiento para extraer las ROMs de las recreativas es similar al de los cartuchos en la mayoría de los casos.



Figura 3.3 – Circuito de un cartucho de NES

Snapshots: son volcados de la memoria del Spectrum con los juegos ya cargados desde Cinta o Disco. Es decir, supongamos que en un Spectrum con 48KB de memoria (ZX Spectrum 48K) cargamos desde cinta un juego determinado. Este juego (que sin duda ocupará menos de 48KB) se almacena en la RAM del Spectrum listo para ser jugado (por ejemplo, en el menú del juego). Si en este momento grabamos el contenido de la RAM en un fichero, y al mismo tiempo almacenamos el estado completo de la CPU, tenemos una "copia" del estado de la máquina. Al cargar este fichero en otra máquina igual, o en un emulador, pondremos al sistema destino en el mismo estado exacto que estaba la máquina original. Es decir, delante del menú, o del juego, en el punto en que lo grabamos.

Los Snapshots (que tampoco son ROMs) se suelen obtener cargando cintas en los emuladores y grabando el contenido de la memoria a disco con un formato especial según el tipo de Snapshot. Tenemos ficheros .z80 (que empezó a utilizar el emulador Z80 para MS-DOS), ficheros con extensión sp y sna (usados por emuladores antiguos), y ficheros con extensión szx, entre otros. Este formato de fichero no es el ideal para preservar los juegos, ya que no permiten recrear las cintas, tan sólo jugar de una forma rápida y directa a los juegos. En general podemos grabar y cargar snapshots en los emuladores de una forma muy rápida (como ficheros) y en cualquier momento, de forma que pueden ser usados como método para "grabar las partidas" y continuar posteriormente.

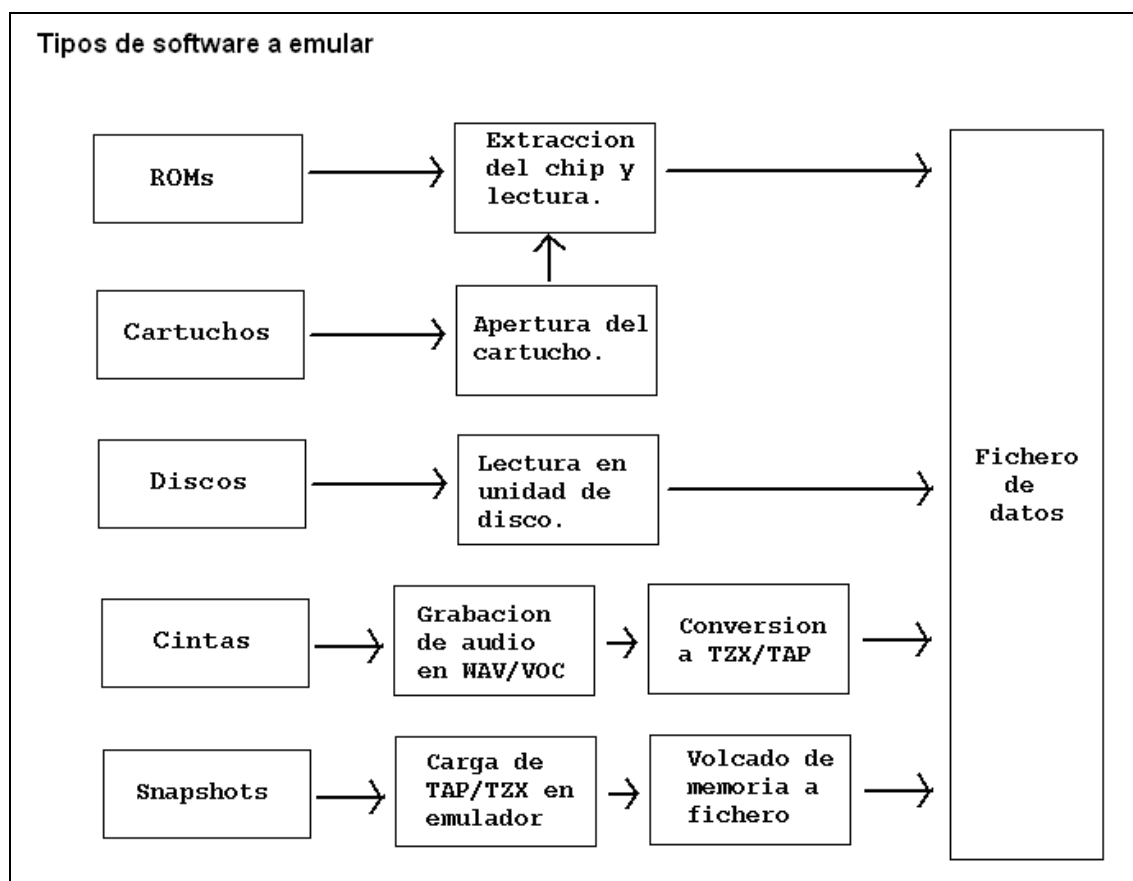


Figura 3.5 – Diagrama de los tipos de software a emular

3.3 EL EMULADOR ASPECTRUM

Aspectrum es un emulador desarrollado por Álvaro Alea y Santiago Romero, compatible con cualquier arquitectura y sistema operativo, especialmente diseñado para que sea capaz de sobrevivir con el tiempo. Actualmente, aunque no se ha terminado su desarrollo, es funcional y se ha probado con más de 450 juegos. Además es un emulador con licencia GPL: se puede copiar, modificar, añadir nuevas funcionalidades sin tener que dar explicaciones a nadie.

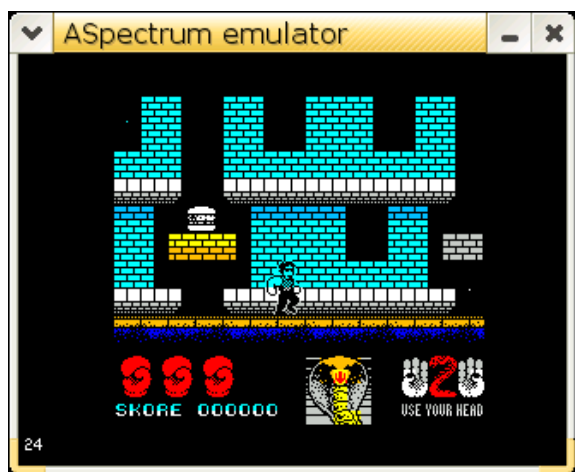


Figura 3.6 – Aspectrum bajo Linux

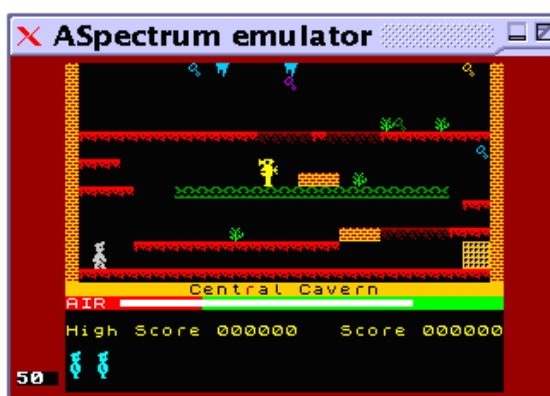


Figura 3.7 Aspectrum bajo Mac OS

Está escrito totalmente en lenguaje C (sin una sola línea de ensamblador), puede compilarse perfectamente en cualquier plataforma existente que soporte alguna de las librerías gráficas que puede utilizar: Windows, Linux, Beos, diversas consolas, etc. Esto es importante porque hay plataformas que no tienen emulador de Spectrum (o al menos que sea licencia GPL) y mediante Aspectrum este problema puede ser subsanado.

Aspectrum requiere de una de las dos librerías gráficas más populares para implementar sus gráficos, SDL o Allegro. Si bien es cierto que no todas las plataformas soportan estas librerías dado que el código gráfico sólo supone un 2% del código del emulador (que está escrito en C), resultaría sencillo portarlo a móviles, PDAs u otros dispositivos, reescribiendo tan sólo la parte gráfica. Actualmente se ha compilado el emulador en MS Dos, Windows, Linux y Mac OS sin problemas y con un rendimiento óptimo.

El hecho de no depender de una única arquitectura le concede una ventaja respecto a muchos de emuladores, que basan parte de su código escrito en ensamblador.

3.3.1 FUNCIONAMIENTO INTERNO

El funcionamiento interno del emulador, que a nosotros nos interesaba, era la zona donde se realizaba la iteración principal. Esta consiste en un bucle que se repite hasta que finaliza su ejecución. En la figura 3.8 se muestra esquemáticamente su funcionamiento.

```

int main (int argc, char *argv[])
{
    ...

    // while (!done) // MAIN LOOP
    // {
        emuMainLoop();
    // }

    return (1);
}

```

Figura 3-8 – Bucle principal del emulador

La función `emuMainLoop()` representa todas las sentencias que hacen posible una iteración del emulador.

3.4 COMPILACIÓN

Una vez realizados los cambios sólo es necesario compilar. Para ello modificamos el fichero `Makefile` que trae el emulador, se deben añadir los enlaces necesarios a las bibliotecas OpenGL, GLU y GLUT.

```

# windos (mingw32)

# agup is a cosmetic mod of allegro gui, if you [ don't want | can't ] use
# eliminate from CFLAGS in Makefile.architecture
AGUPDIR=agup-0.99.7
AGUPLIB=$(AGUPDIR)/lib/release/libagup.a

LFLAGS = -lalleg -mwindows -lopengl32 -lglu32 -lglut32
CFLAGS = -W -Wall -O2 -mwindows -DVERSION=\"${VERSION}\" -DNO_GETOPTLONG \
        -DSOUND_BY_STREAM -DENABLE_LOGS -DI_HAVE_AGUP -I$(AGUPDIR) \
        -DDOSSEP=\"\\"'\"
RM=del
EXT=.exe
MAKE=mingw32-make

# objetos para windows las fuentes no hacen falta por que se usa makedeps.bat
objects=sound.o v_alleg.o snaps.o graphics.o menu.o debugger.o \
main.o z80.o disasm.o mem.o langs.o contrib/getopt.o

#dos dep
dep:
    makedeps.bat

```

Figura 3.9 – Contenido del fichero `Makefile`

Los cambios realizados en el fichero `Makefile`, observar figura 3.9, consisten en añadir en la sección `LFLAGS` los argumentos necesarios, mediante la opción `-l`, para especificar las bibliotecas necesarias para compilar correctamente el código. El argumento para incluir la biblioteca OpenGL, que es la que permite dibujar en 3D, es `-lopengl32`.

Mediante los argumentos `-lglu32` y `-lglut32` especificamos que incluiremos las bibliotecas GLU y GLUT respectivamente. Tales bibliotecas nos permiten añadir nuevas funcionalidades de dibujo, de control de eventos y definición de ventanas.

4. INGENIERÍA INVERSA

4.1 CONCEPTOS BÁSICOS DEL JUEGO

En el juego controlamos a un personaje que debe recorrer diferentes localizaciones en busca de una serie de objetos y depositarlos en una habitación en particular, con el objetivo de romper una maldición que lo transforma en licántropo.

El escenario está dividido en más de 200 habitaciones las cuales están formadas por fondos y objetos.

Los fondos y objetos están representados en el código del programa en forma de sprites, con sus dimensiones y propiedades gráficas. Los fondos suelen ser las paredes y las puertas. Los objetos son muy numerosos: bloques, fuego, púas, guardas, etc. Como ejemplo podemos observar la habitación de la figura 4.1 que contiene como fondos las paredes y dos puertas, y como objetos un grupo de bloques organizados en una malla regular de 3x4.

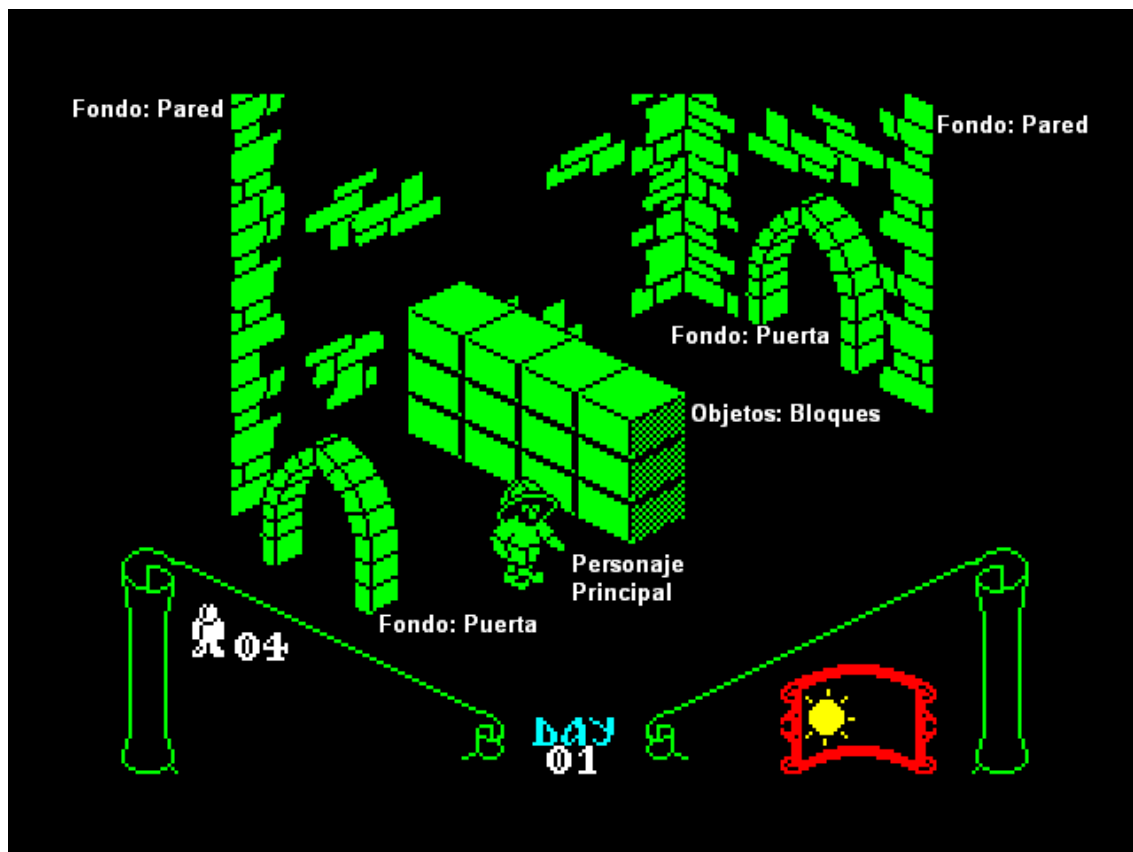


Figura 4.1 – Pantalla del juego con sus elementos identificados

4.2 – TÉCNICA FILMATION

La técnica Filmation es un sistema especial de gestión de los gráficos y de la localización en la que se desarrolla el juego, la cual se halla volcada en pantalla en una perspectiva isométrica y mediante técnica de sprites. Esta técnica produce una visión tridimensional del entorno. No hay que confundir la técnica Filmation (cuyo nombre viene de la similitud con “filmar” una película) con la gestión de sprites. Un defecto del sistema Filmation es la necesidad de operar con el mismo en un sistema bicolor, es decir, tinta de un color y papel de otro distinto. Esto es así debido a que el muñeco o sprite del personaje que manejemos no se mueve horizontal o verticalmente por la pantalla sino que, debido a que la misma es una proyección isométrica, se moverá por vectores 30° inclinados a la horizontal de la pantalla. Esto hace prácticamente imposible en nuestro caso, el manejo de colores debido a la organización por separado de la memoria de atributos con la de pantalla en el Spectrum.

Los conceptos básicos que se emplean en la técnica son:

- Gráfico: Es el dibujo de un muñeco, coche, pelota, ..., situado en la memoria del ordenador y que pretendemos mover o pintar en la pantalla.
- Sprite: Gráfico definido en la memoria de una forma especial y que, gestionado con una rutina también especial, se puede mover por la pantalla sin perjudicar lo que en la misma hubiera. Asimismo, al moverlo, se crea la ilusión de que el sprite está en un plano anterior a lo que tengamos en la pantalla. Digamos que pasa “por encima” de la misma. No hay que confundir gráfico con sprite, ya que son dos cosas distintas.
- Sombra o máscara del sprite: El sprite está almacenado en memoria de una forma especial, por una parte como si de un gráfico vulgar se tratase y por otra se encuentra definida también la “sombra” o máscara necesaria para manejar el sprite. En esta máscara está codificada la información de la opacidad o transparencia de ciertas zonas del gráfico.

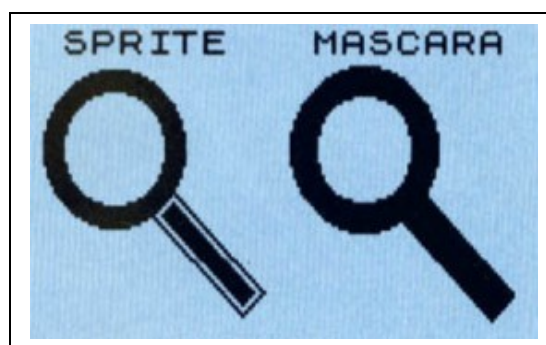


Figura 4.2 – Elementos de técnica Filmation

Como se aprecia en la figura 4.2, el gráfico que corresponde a una lupa está compuesto por su sprite y su máscara. La máscara permite definir que parte del sprite queremos que sea opaca y la que queremos que sea transparente. A nivel de código esto se realiza, como podemos observar en la figura 4.3, efectuando operaciones de bits entre la máscara y el sprite para luego aplicarlas a la pantalla.

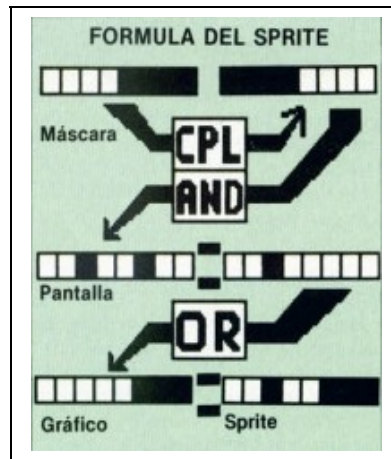


Figura 4.3 – Operaciones Filmation

El resultado de estas operaciones permite situar el sprite en la pantalla sin perjudicar el resto. Para ello:

- Realiza el complemento a 1 de la máscara, reenplazando 1 por 0 y viceversa.
- Al resultado anterior se le aplica la operación lógica AND con la porción de la pantalla donde se quiere trabajar.
- Este último resultado recibe la operación lógica OR con el gráfico para obtener el sprite final.

Cabe destacar que como se puede observar en la figura 4.2, el tamaño de la máscara es ligeramente superior al del sprite. El motivo es obtener al realizar su intersección de el sprite con una silueta negra que facilita su visión y a la vez la embellece.

4.3 INGENIERÍA INVERSA

El objetivo de la ingeniería inversa es obtener información técnica a partir de un producto accesible al público, con el fin de determinar de qué está hecho, qué lo hace funcionar y cómo fue fabricado. Los productos más comunes que son sometidos a la ingeniería inversa son los programas de ordenadores y los componentes electrónicos.

Este método es denominado ingeniería inversa porque avanza en dirección opuesta a las tareas habituales de ingeniería, que consisten en utilizar datos técnicos para elaborar un producto determinado. En general si el producto u otro material que fue sometido a la ingeniería inversa fue obtenido en forma apropiada, entonces el proceso es legítimo y legal.

La ingeniería inversa es un método de resolución. Aplicar ingeniería inversa a algo supone profundizar en el estudio de su funcionamiento, hasta el punto de que podemos llegar a entender, modificar, y mejorar dicho modo de funcionamiento.

En particular en este proyecto hemos utilizado las siguientes técnicas de ingeniería inversa:

- Estudio de la documentación previa, especialmente la documentación sobre las estructuras de datos estáticas del juego. Ver sección 4.4.
- Estudio del código fuente del programa para entender el funcionamiento de las rutinas que se encargaban de la gestión gráfica. Ver sección 4.5
- Modificación de los datos del programa en tiempo real para ver los cambios que surgen, y deducir el por qué de los efectos provocados. Ver sección 4.8.

4.4 INFORMACIÓN DE REFERENCIA

El proceso de ingeniería inversa comenzó a partir del desensamblado del binario del juego y de dos documentos, *On filmation* de Neil Walker y *Knight Lore data format* de Christopher Jon Wild. Dichos escritos contienen la información básica de como dispone el juego la información estática en memoria.

Nada más iniciar el juego, disponemos en memoria de los datos estáticos de todas las habitaciones ordenados por localización, de manera compactada, que representan el estado inicial de cada habitación. La información que las representa es:

- Identificador de la localización
- Desplazamiento a la siguiente dirección
- Dimensiones y color
- Fondos
- Objetos

Un ejemplo de una habitación tal como se almacena en memoria sería el siguiente (habitación 00Eh):

00Eh, 00Bh, 015h 001h, 003h, 00Dh, 0FFh, 053h, 012h, 01Dh, 02Ch, 023h
--

Figura 4.4 – Datos estáticos de la habitación 0Eh

La primera línea contiene el identificar, el color y las dimensiones de la habitación y una referencia a la siguiente habitación mientras, que la segunda contiene información sobre los fondos y los objetos.

- El primer byte corresponde al identificador, que es único para toda habitación, en este caso es el 00Eh.
- El siguiente byte corresponde al desplazamiento que hay que aplicar al puntero de programa para situarnos en la siguiente habitación, que en este ejemplo es 00Bh (11 bytes).
- En el tercer byte se nos presenta un caso en el cual compactamos varios datos en un solo byte. En concreto 015h representa 00010101b del cual, aplicando el patrón de la figura 4.5, obtendremos la información referente al color y dimensiones.

			Size		Colour		
8	7	6	5	4	3	2	1

Figura 4.5 – Patrón atributos habitación

El color es 101b que corresponde a los tres primeros bits, el tamaño será 10b que se obtiene de los dos siguientes bits (los diferentes tipos de habitación se encuentran presentados en la figura 4.6) y el resto de los bits pueden ser ignorados ya que no significan nada. Como podemos comprobar, en un solo byte podemos almacenar mucha información y, como veremos más adelante, era algo muy corriente a causa de las limitaciones técnicas de la época. A nivel práctico, su manipulación se consigue de manera sencilla con operaciones de bits.

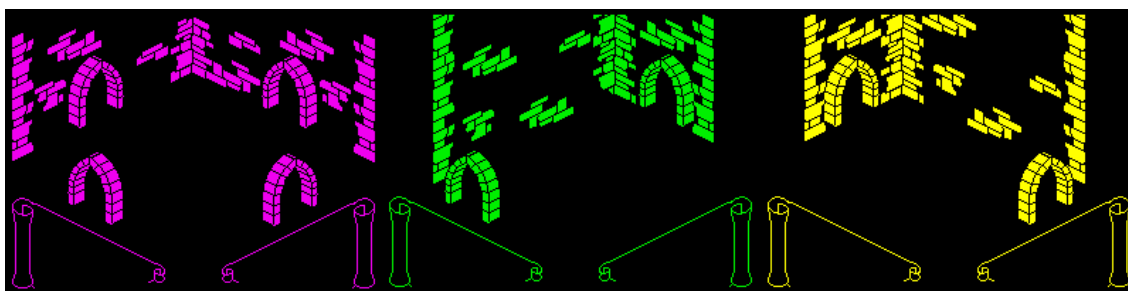


Figura 4.6 – Los diferentes tipos de habitaciones

Una vez tratados los anteriores datos, se pasa a leer los datos de los fondos y objetos, que son los restantes bytes hasta el inicio de la siguiente localización. Primeramente aparecen los fondos, donde cada byte representa el identificador de un fondo hasta que aparezca uno con valor FFh. El byte con ese valor es ignorado y simplemente determina que disponemos de los fondos y que empieza la sección de los objetos.

En el ejemplo de la figura 4.4, observamos que los siguientes bytes hasta el marcador de inicio de los objetos son:

001h, 003h, 00Dh, 0FFh

Estos identificadores representan el arco este (001h), el arco oeste (003h) y un pasillo (00Dh) con fondo, como puede verse en la figura 4.7:



Figura 4.7 – Fondos de la habitación 0Eh

Llegado al punto de la lectura de los objetos, el proceso vuelve a requerir de la división del byte. El método consiste en obtener el primer byte, desglosado, el cual nos indicará el identificador de objeto y su cantidad (una localización puede contener varios objetos del mismo tipo). La cantidad n de un descriptor indicará que los siguientes n + 1

bytes del descriptor contienen las coordenadas de los objetos de ese tipo que contiene esa determinada habitación. Como se puede deducir el número máximo de réplicas de un objeto es 8, pues se disponen de 3 bits para indicar el número. Eso no impide situar más de 8 objetos en una habitación: se puede conseguir añadiendo tantos descriptors de objetos como sean necesarios para llegar al número de objetos total requerido. El patrón está especificado en la figura 4.8.

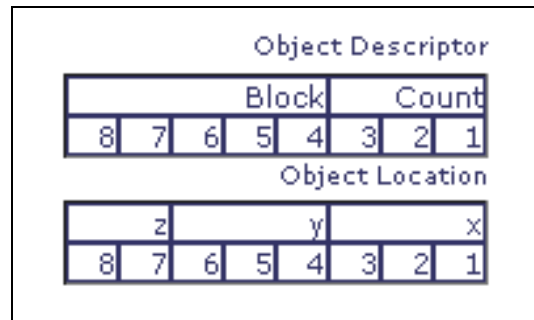


Figura 4.8 – Patrón de los objetos

En el ejemplo que hemos utilizado hasta ahora (figura 4.4) vemos que los bytes después de la marca de fin de fondos son:

053h, 012h, 01Dh, 02Ch, 023h

El byte 053h representa al descriptor de objeto, desglosado en bits obtenemos el valor 01010011b. Los cinco últimos bits representan el identificador de objeto, en este caso será el objeto con el identificador 01010b = 0Ah que es el fuego (ver figura 4.16). Los tres primeros bits contienen el valor 011b, que corresponde al número de repeticiones del objeto. Eso significará que tendremos cuatro objetos ($n = 3, 3 + 1 = 4$), que serán de tipo fuego.

Los cuatro siguientes bytes contienen las coordenadas de los objeto de tipo fuegos:

012h	= 00010010b	→	(x,y,z) = (2,2,0)
01Dh	= 00011101b	→	(x,y,z) = (5,3,0)
02Ch	= 00101100b	→	(x,y,z) = (4,5,0)
023h	= 00100011b	→	(x,y,z) = (3,4,0)

En este ejemplo ya no hay más objetos, pero podría suceder que sí existieran. En ese caso seguiríamos leyendo bytes con la pauta antes establecida, byte descriptor y bytes consecutivos especificando las coordenadas. Llegado a este punto ya disponemos de toda la información estática que representa una habitación, en concreto el aspecto gráfico del ejemplo sería como muestra la figura 4.9

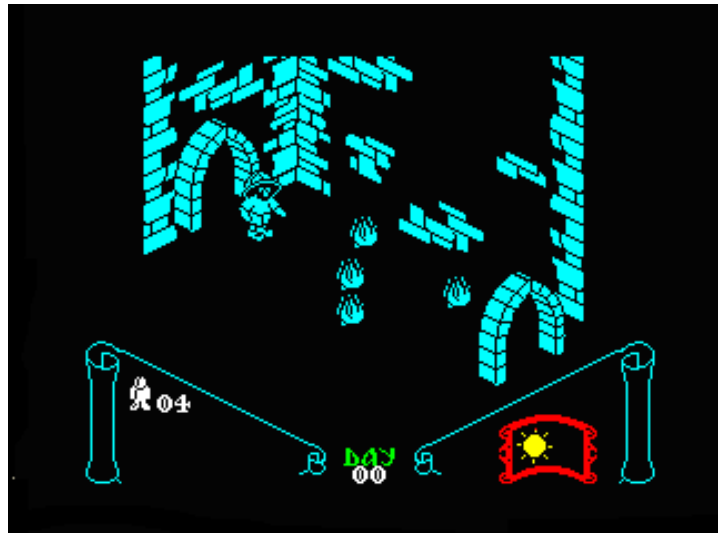


Figura 4.9 – Habitación 00Eh

Pero con esto no es suficiente para saber el estado en todo momento de una localización, pues hemos obtenido los datos iniciales, que son estáticos. Los elementos del fondo no cambiarán de posición, pero muchos objetos sí (pueden moverse, el personaje principal los puede coger, hay bloques que desaparecen al situarse encima el personaje principal, etc). La figura 4.10 muestra la misma habitación pero vista unos instantes más tarde, con una disposición diferente de los objetos.

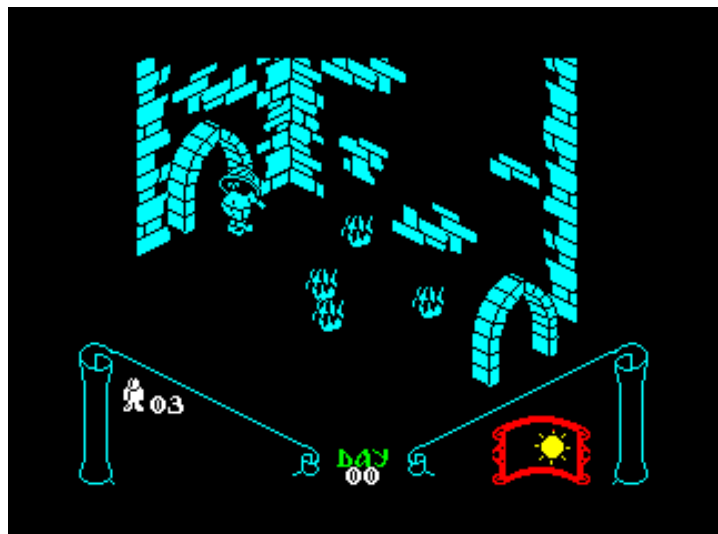


Figura 4.10 – Habitación 00Eh

4.5 HABITACIONES EN LA MEMORIA DE TRABAJO

Con la información referente a las localizaciones que inicializa el juego no es suficiente para plasmar gráficamente las habitaciones, pues esos datos no contienen las variaciones que sufren los objetos con el paso del tiempo o las interacciones con otros objetos. Sólo se refieren a la situación inicial. Además, disponemos del binario desensamblado del juego y con ciertas funciones y mapas de memoria especificados. Una de esas zonas de memoria era la memoria de trabajo (scratch mem) y fue la que

dedujimos que reflejaría dinámicamente los datos de las localizaciones. Una comprobación tan sencilla como confirmar que surgieran cambios en ella durante el transcurso de una partida aclaró nuestras suposiciones. Esto lo realizamos a través de la función debug del emulador ZXSPin. Sabíamos donde se producían los cambios, en la memoria de trabajo (que está situada entre las direcciones de memoria 5BA0h y 6107h) pero no sabíamos como los hacía. Tras muchas horas de estudio, y observando cómo se trata la información del código fuente del juego desensamblado, nos llamó la atención una función llamada *RetrieveScreen*. Así pues, decidimos examinarla para descubrir si era la que almacenaba los datos de cada localización en memoria, y cómo lo hacía.

Traducida a pseudocódigo sería así:

```
Situar el puntero de trabajo en el id de la primera habitación;

Repetir {
    Saltar al id de la siguiente habitación;
} hasta que el puntero de trabajo apunte a la habitación actual

Obtener los atributos de la habitación
Almacenar los atributos de la localización;
```

El anterior fragmento recorre los datos iniciales estáticos de las habitaciones, en busca de los de la habitación donde entra el personaje principal. Una vez localizada obtiene los atributos de la habitación (ver figura 4.5).

El siguiente fragmento en pseudocódigo es:

```
Por cada fondo hacer {
    Obtener su sprite y la información sobre éste;
    Obtener sus dimensiones;
    Obtener su posición;
    Calcular posición 3D de alta definición;
    Almacenar la información obtenida;
}

Por cada objeto hacer {
    Obtener su sprite y la información sobre éste;
    Obtener sus dimensiones;
    Obtener su posición;
    Calcular posición 3D de alta definición;
    Almacenar la información obtenida;
}
```

Los dos anteriores fragmentos, se encargan de adquirir, por cada fondo y, después, por cada objeto, todas sus propiedades y atributos, para luego almacenar la información obtenida.

4.5.1 TRATAMIENTO DE LOS FONDOS

RetriveScreen, por cada fondo a través de su valor, consulta la tabla de fondos, almacenada de forma estática a partir de la dirección 6CE2h (ver figura 4.11).

6CE2	DW	06D12h	;0	Arch north
6CE4	DW	06D23h	;1	Arch east
6CE6	DW	06D45h	;2	Arch south
6CE8	DW	06D67h	;3	Arch west
6CEA	DW	06D78h	;4	Tree arch north
6CEC	DW	06D89h	;5	Tree arch east
6CEE	DW	06D9Ah	;6	Tree arch south
6CFO	DW	06DABh	;7	Tree arch west
6CF2	DW	06DBCCh	;8	Gate
6CF4	DW	06DC5h	;9	Gate
6CF6	DW	06DCEh	;A	Gate
6CF8	DW	06DD7h	;B	Gate
6CFA	DW	06DE0h	;C	walls size 0
6CFC	DW	06E49h	;D	walls size 1
6CFE	DW	06EBAh	;E	walls size 2
6D00	DW	06F2Bh	;F	trees size 0
6D02	DW	06F8Ch	;10	trees size 1
6D04	DW	06F9Dh	;11	trees size 2
6D06	DW	06FAEh	;12	Wizard
6D08	DW	06FBFh	;13	Pot
6DOA	DW	06D34h	;14	High Arch north
6DOC	DW	06D56h	;15	High Arch east
6DOE	DW	06FDOh	;16	High Arch south
6D10	DW	06F1Eh	;17	High Arch west

Figura 4.11 – Tabla de punteros de los fondos

Esta tabla permite obtener la dirección de los datos de un fondo simplemente añadiendo a una dirección base el valor de cada fondo. Por ejemplo, para saber la dirección donde encontrar los datos del fondo arco este (ver figura 4.7), bastará con añadir a la dirección base de la tabla de la figura 4.11, que es 6CE2h, el identificador del arco este multiplicado por 02h (pues cada dirección requiere de dos bytes para almacenarla) que es 001h, el resultado será 6CE4h. En esta última dirección encontraremos los datos que definirán el arco este. Los datos que definen los fondos se almacenan siguiendo el formato mostrado en la figura 4.12.

address 0x6d12 background type data																	
type	offset	description															
BYTE	00	sprite															
	01	x															
	02	y															
	03	z															
	04	width (x)															
	05	depth (y)															
	06	height (z)															
	07	<table><tr><td>v</td><td>h</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr><tr><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr></table>	v	h	5	4	3	2	1	0	7	6	5	4	3	2	1
v	h	5	4	3	2	1	0										
7	6	5	4	3	2	1	0										

v = flip vertical
h = flip horizontal
4 = draw - this is internal but seems to always be set

keep reading an entry until the sprite is zero. all these sprites make one entity

Figura 4.12 – Formato de los datos de un sprite de fondos

El resultado será uno o más grupos de 8 bytes, cada grupo sigue el formato de la figura 4.12. Este formato define:

- El byte 00 contiene el id de sprite.
- Los bytes 01, 02 y 03 contienen la posición X, Y y Z del sprite.
- Los bytes 04, 05 y 06 definen las dimensiones del sprite.
- El byte 07 contiene una serie de flags: el v y h si están activos indicarán una rotación horizontal o vertical del sprite, el resto son bits que no nos interesan pues son para uso interno del juego.

Finalmente los datos del ejemplo, el arco este (figura 4.7) serán:

002h, 0c4h, 073h, 080h, 005h, 003h, 028h, 010h, 003h, 0c4h, 080h, 080h, 005h, 003h, 028h, 010h, 00h

Observamos en la anterior línea que el arco este está formado por dos sprites, el 002h y el 003h, lo que pude apreciarse en la figura 4.13.

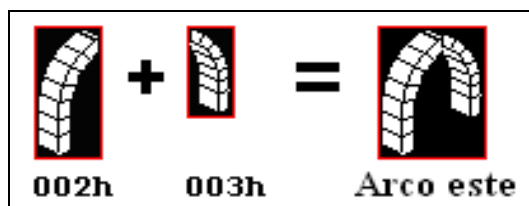


Figura 4.13 – Sprites que forman el sprite del arco este

El resultado final en la memoria de trabajo será el presentado en la figura 4.14.

\$5C80:	00	00	00	00	00	00	00	00	00	02	C4	73	80	05	03	28	00
\$5C90:	0E	C4	80	80	00	00	00	00	00	00	00	F9	FD	00	00	00	00
\$5CA0:	03	34	B0	2C	03	34	B0	2C	00	03	C4	8D	80	05	03	28	00

Arco este

Figura 4.14 – Datos del objeto arco este en la memoria de trabajo

Podemos observar que hay más bytes aparte de los encuadrados en la figura 4.14, son datos que utiliza el juego para otros aspectos de los objetos que a nosotros no nos interesan para su visualización 3D.

4.5.2 – TRATAMIENTO DE LOS OBJETOS

RetriveScreen, por cada objeto a través de su valor consulta la tabla de objetos presentada en la figura 4.15.

6BD1	DW	06C0Bh	;0	block 07 block
	DW	06C3Ch	;1	sprite B0 (Fire)
	DW	06C43h	;2	sprite B2 (Ball) [up/down]
	DW	06C66h	;3	block 06 Rock
	DW	06C6Dh	;4	block 16 Gargoyle
	DW	06C74h	;5	block 17 Spike
	DW	06C90h	;6	sprite 55 (Chest)
	DW	06C97h	;7	sprite 54 (table)
	DW	06C9Eh	;8	sprite 96/90 (guard) [west/east]
	DW	06CB8h	;9	sprite 52 (ghost)
	DW	06CBFh	;A	sprite B5 (fire) [north/south]
	DW	06C12h	;B	block 07 block high
	DW	06C4Ah	;C	sprite B2 (ball) [up/down]
	DW	06CABh	;D	sprite 1e/90 (Guard) [square circuit]
	DW	06C19h	;E	block 36 [west/east]
	DW	06C20h	;F	block 37 [north/south]
	DW	06C27h	;10	block 3E
	DW	06C7Bh	;11	block Spike - high!!!
	DW	06C82h	;12	sprite 3F (Spike Ball)
	DW	06C89h	;13	sprite 3F (Spike Ball) [falling]
	DW	06CC6h	;14	sprite 56 Fire [west/east]
	DW	06C2Eh	;15	Block 5B
	DW	06C35h	;16	block 8F [Collapse]
	DW	06C5Fh	;17	sprite B6 (Ball)
	DW	06C51h	;18	sprite B2 (Ball)
	DW	06CCDh	;19	sprite A4 (Spell) [repel player]
	DW	06CD4h	;1A	sprite 8 (Gate) [up/down]
	DW	06CDBh	;1B	sprite 8 (Gate) [up/down]
	DW	06C58h	;1C	sprite B2 (Ball)

Figura 4.15 – Tabla de punteros de los objetos

Esta tabla permite obtener la dirección de los datos de un objeto simplemente añadiendo a su dirección base el valor de cada objeto.

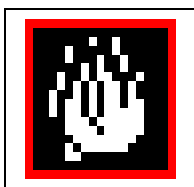


Figura 4.16 – Sprite del objeto fuego

Por ejemplo para saber la dirección donde encontrar los datos del objeto fuego (figura 4.16) bastará con añadir a la dirección base de la tabla de la figura 4.15, que es `6BD1h` el identificador del objeto tipo fuego que es `0Ah` multiplicado por `02h` (una dirección requiere de dos bytes), el resultado será `6BE9h`. En esta última dirección encontraremos los datos que definirán el objeto tipo fuego. Los datos que definen los objetos se almacenan siguiendo el formato mostrado en la figura 4.17.

address 0x6c0b block type data																		
type	offset	description																
BYTE	00	sprite																
	01	width (x)																
	02	depth (y)																
	03	height (z)																
		<table><tr><td>v</td><td>h</td><td>5</td><td>4</td><td>3</td><td>m</td><td>1</td><td>0</td></tr><tr><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr></table>	v	h	5	4	3	m	1	0	7	6	5	4	3	2	1	0
v	h	5	4	3	m	1	0											
7	6	5	4	3	2	1	0											
04		v = flip vertical h = flip horizontal 4 = draw - this is internal but seems to always be set m = movable/moving 0-3 - seem to be block type dependent																
		<table><tr><td colspan="5">z1</td><td colspan="2">y1</td><td>x1</td></tr><tr><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr></table>	z1					y1		x1	7	6	5	4	3	2	1	0
z1					y1		x1											
7	6	5	4	3	2	1	0											
05		these values are added in the block positioning calculation. keep reading an entry until the sprite is zero. all these sprites make one entity																

Figura 4.17 – Formato de los datos de un sprite de objetos

El resultado será uno a más grupos de 6 bytes tal como, cada grupo sigue el formato de la figura 4.17, este formato define:

- El byte 00 contiene el id de sprite.
- Los bytes 01, 02 y 03 contienen la posición X, Y y Z del sprite.
- El byte 04 contiene los flags para determinar si el sprite requiere de una rotación horizontal o vertical, si es un objeto con movimiento y su dependencia respecto otro objeto.
- El byte 05 contiene los datos para las correcciones de posición (ver sección 4.7).

Hay un serie de objetos (los que se mueven de manera regular, como por ejemplo los guardias) que requieren de una corrección de posición en función del movimiento, esto se especifica en el byte 05 de la figura 4.17. Contiene si alguna de las coordenadas requiere de la corrección.

Finalmente los datos del ejemplo, el fuego (figura 4.16) serán:

0B5h, 006h, 006h, 00Ch, 010h, 000h, 000h

El resultado final en la memoria de trabajo será:

\$5ECO:	03	12	B8	84	03	12	B8	84	B4	68	79	80	06	06	0C	30	Fuego
\$5EDO:	0E	00	FE	00	00	A0	00	00	00	00	F8	FC	00	00	00	00	
\$5EEO:	03	0F	59	5C	03	10	5B	5D	B4	98	89	80	06	06	0C	30	Fuego
\$5EFO:	0E	00	FE	00	00	A0	00	00	00	00	F8	FC	00	00	00	00	
\$5F00:	03	0F	99	4C	03	10	9B	4D	B4	88	8B	80	06	06	0C	30	Fuego
\$5F10:	0E	00	02	00	00	A2	00	00	00	00	F8	FC	00	00	00	00	
\$5F20:	03	0F	8B	55	03	10	89	54	B4	78	99	80	06	06	0C	30	Fuego
\$5F30:	0E	00	00	00	02	A0	00	00	00	00	F8	FC	00	00	00	00	

Figura 4.18 – Datos de los objetos tipo fuego en la memoria de trabajo

Podemos observar que hay más bytes aparte de los encuadrados en la figura 4.18, correspondientes a las cuatro réplicas del objeto tipo fuego. Estos son datos que utiliza el juego para otros aspectos de los objetos que a nosotros no nos interesan para su visualización.

La función *RetriveScreen* sitúa a partir de la dirección de memoria 5C88h (ver figura 4.20) toda la información de la habitación y, desde entonces, es en ese lugar donde se realizarán todos los cambios que se producen en la localización mientras el jugador está en ella.

Por cada fondo u objeto de la localización (ver figura 4.20), disponemos de un mínimo de dos líneas de grupos de 8 bytes. Un fondo u objeto puede estar representado por más de un sprite. Cada una de ellas representa un sprite, con su posición, dimensiones e información de su representación (ver figura 4.19).

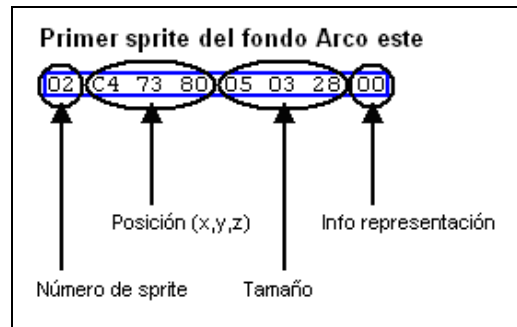


Figura 4.19 – Información de un sprite

\$5C80:	00 00 00 00 00 00 00 00	02 C4 73 80 05 03 28 00	Arco este
\$5C90:	0E C4 80 80 00 00 00 00	00 00 F9 FD 00 00 00 00	
\$5CA0:	03 34 B0 2C 03 34 B0 2C	03 C4 8D 80 05 03 28 00	
\$5CB0:	0E 00 00 00 00 00 00 00	00 00 F7 FD 00 00 00 00	
\$5CC0:	02 23 C8 39 02 23 C8 39	02 3B 73 80 05 03 28 10	Arco oeste
\$5CD0:	0E 3B 80 80 00 00 00 00	00 00 F9 FD 00 00 00 00	
\$5CE0:	04 34 27 71 04 34 27 71	03 3B 8D 80 05 03 28 00	
\$5CF0:	0E 00 00 00 00 00 00 00	00 00 F7 FD 00 00 00 00	
\$5D00:	03 23 3F 7E 03 23 3F 7E	0D 3F 98 80 00 08 28 00	Paredes
\$5D10:	0E 00 00 00 00 00 00 00	00 00 F8 FC 00 00 00 00	
\$5D20:	03 30 4F 80 03 30 4F 80	0E 47 A0 80 08 00 28 00	
\$5D30:	0E 00 00 00 00 00 00 00	00 00 F8 FC 00 00 00 00	
\$5D40:	03 30 5F 80 03 30 5F 80	0F 3F 63 80 00 08 2C 00	
\$5D50:	0E 00 00 00 00 00 00 00	00 00 F8 FC 00 00 00 00	
\$5D60:	03 30 1A 66 03 30 1A 66	0F B8 A0 80 08 00 2C 40	
\$5D70:	0E 00 00 00 00 00 00 00	00 00 F8 FC 00 00 00 00	
\$5D80:	02 30 D0 48 02 30 D0 48	0F 3F 63 AC 00 08 2C 00	Fuego
\$5D90:	0E 00 00 00 00 00 00 00	00 00 F8 FC 00 00 00 00	
\$5DA0:	03 2E 1A 92 03 2E 1A 92	0F B8 A0 AC 08 00 2C 40	
\$5DB0:	0E 00 00 00 00 00 00 00	00 00 F8 FC 00 00 00 00	
\$5DC0:	02 30 D0 74 02 30 D0 74	0D 3F 98 A8 00 08 28 00	
\$5DD0:	0E 00 00 00 00 00 00 00	00 00 F8 FC 00 00 00 00	
\$5DE0:	03 18 4F A8 03 18 4F A8	0E 47 A0 A8 08 00 28 00	
\$5DF0:	0E 00 00 00 00 00 00 00	00 00 F8 FC 00 00 00 00	
\$5E00:	03 18 5F A8 03 18 5F A8	0F B8 A0 D0 08 00 2C 40	Fuego
\$5E10:	0E 00 00 00 00 00 00 00	00 00 F8 FC 00 00 00 00	
\$5E20:	02 28 D0 98 02 28 D0 98	0A 80 A0 80 14 00 14 50	
\$5E30:	0E 00 00 00 00 00 00 00	00 00 EC FF 00 00 00 00	
\$5E40:	06 19 8C 67 06 19 8C 67	0A 3F 7E B0 00 14 14 00	Fuego
\$5E50:	0E 00 00 00 00 00 00 00	00 00 EC FF 00 00 00 00	
\$5E60:	06 19 29 A6 06 19 29 A6	0B 60 A0 90 0C 00 14 40	
\$5E70:	0E 00 00 00 00 00 00 00	00 00 F4 FE 00 00 00 00	
\$5E80:	04 18 74 86 04 18 74 86	0A 60 A0 B8 14 00 14 40	Fuego
\$5E90:	0E 00 00 00 00 00 00 00	00 00 EC FF 00 00 00 00	
\$5EA0:	06 11 6C AF 06 11 6C AF	0C A0 A0 B0 0C 00 0C 40	
\$5EB0:	0E 00 00 00 00 00 00 00	00 00 F8 FC 00 00 00 00	
\$5EC0:	03 12 B8 84 03 12 B8 84	B4 68 79 80 06 06 0C 30	Fuego
\$5ED0:	0E 00 FE 00 00 A0 00 00	00 00 F8 FC 00 00 00 00	
\$5EE0:	03 0F 59 5C 03 10 5B 5D	B4 98 89 80 06 06 0C 30	
\$5EF0:	0E 00 FE 00 00 A0 00 00	00 00 F8 FC 00 00 00 00	
\$5F00:	03 0F 99 4C 03 10 9B 4D	B4 88 8B 80 06 06 0C 30	Fuego
\$5F10:	0E 00 02 00 00 A2 00 00	00 00 F8 FC 00 00 00 00	
\$5F20:	03 0F 8B 55 03 10 89 54	B4 78 99 80 06 06 0C 30	
\$5F30:	0E 00 00 00 02 A0 00 00	00 00 F8 FC 00 00 00 00	
\$5F40:	03 0F 89 64 03 10 89 64	00 00 00 00 00 00 00 00	

Figura 4.20 – Información dinámica total de la habitación

Podemos observar también, en la figura 4.20, que finalmente lo que se almacena en memoria corresponde al conjunto de sprites que conformaran tanto el fondo como los

objetos, con la información referente al empleo de la técnica filmation. Como podemos apreciar aquí, la información ya no está comprimida. En cambio, en la memoria estática el objeto tipo fuego está definido una vez y seguidamente se enumeraban sus diferentes posiciones. Ahora, en cambio, en la memoria dinámica, el objeto fuego dispone de cuatro representaciones con sus respectivos atributos especificados separadamente.

4.6 ELEMENTOS DE LA PANTALLA

Una vez situados todos los elementos de la pantalla en la memoria de trabajo, ya se puede dibujar la pantalla. Por ejemplo, el primer sprite de la habitación 0Eh es el 01h (observar figura 4.7), que corresponde al sprite 02h (figura 4.21). Esta información ha sido depositada en la memoria de trabajo mediante la consulta de la información estática. Continuamente el programa comprueba el estado de todos los objetos de la habitación y, si es necesario, actualiza la información referente a la posición, estado, etc. Pero esta actualización se realiza solamente sobre la información dinámica almacenada en la memoria de trabajo. Por ello es sumamente importante para este proyecto entenderla completamente.

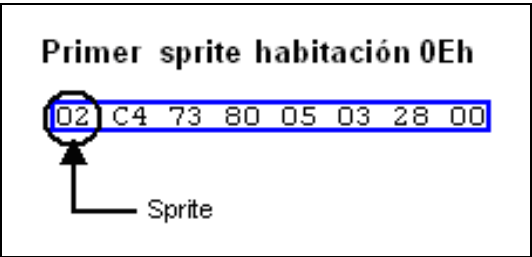


Figura 4.21 – Primer sprite habitación 00Eh

Para obtener la información que representa gráficamente el sprite, el programa consulta la tabla situada a partir de la posición a 7112h que contiene los punteros a los datos de cada sprite (figura 4.22):

@	Puntero
7112	728A
7114	728A
7116	9ACA
...	...
7282	840E
7284	837C
7286	7348

Figura 4.22 – Tabla de punteros de datos de los sprites

El identificador de cada sprite (en el de la figura 4.21 es 02h) multiplicado por 02h (pues una dirección de memoria requiere de dos bytes para su almacenamiento) sirve para sumarlo a la dirección de inicio de la tabla de la figura 4.22 y obtener un puntero hacia la dirección donde se almacenan los texturas que representan un sprite, 7112h +

04h = 7116h. En esta dirección dispondremos de los datos para representar gráficamente un sprite.

Los datos que almacena un sprite siguen el formato presentado en la figura 4.23.

address	0x728a	sprite data
data for sprites		
type	offset	description
BYTE	00	width
BYTE	01	height
BYTE	03...	image data
(size = (width* height)*2)		
[0 ... height]		
[0 ... width]		
BYTE	00	image
BYTE	01	mask

Figura 4.23 – Almacenamiento de los datos gráficos de un sprite

Que se define como:

- El byte 00 indica el ancho del gráfico.
- El byte 01 indica el alto del gráfico.
- A partir del byte 01 tenemos la información gráfica.

El producto de los valores ancho y alto nos indica el número total de bytes que formarán el gráfico. A partir de entonces, el programa simplemente va cogiendo los bytes sucesivos de dos en dos, el primero corresponderá a la textura de la imagen y el segundo a la textura opaca que representa la máscara. Se va leyendo hasta obtener el número total de bytes que forman el gráfico. Sólo faltará dibujar esta información teniendo en cuenta el tamaño y posición (figura 4.19).

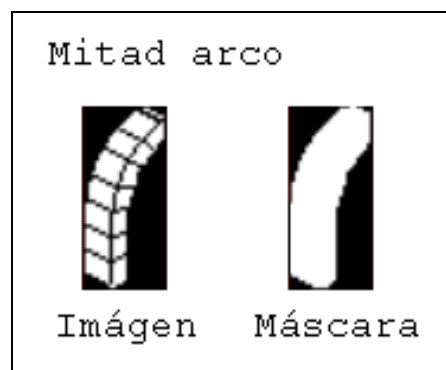


Figura 4.24 – Imagen y máscara

El apartado 4.9 nos muestra un ejemplo completo de los datos de un sprite.

4.7 MUNDO 3D HIGH Y 3D LOW

Una de las particularidades que había que tener en cuenta era que, aunque las coordenadas de los fondos, objetos y el personaje principal inicialmente estuvieran en 3D de baja resolución (low) en la memoria de trabajo se almacenaban en 3D de alta resolución (high). Eso era así para poder graficarlas mediante la técnica filmation. La solución estaba en el documento de *Knight Lore data format* de Christopher Jon Wild. Utilizando el siguiente sistema de ecuaciones, podemos pasar de 3D Low a 3D High y viceversa, que era lo que nos interesaba, pues la información dinámica trabajaba en 3D de alta resolución. El sistema es el siguiente:

$$\begin{aligned}X_H &= (X_L \cdot 16) + (X_1 \cdot 8) + 72 \\Y_H &= (Y_L \cdot 16) + (Y_1 \cdot 8) + 72 \\Z_H &= (Z_L \cdot 12) + (Z_1 \cdot 4) + ScreenZ\end{aligned}$$

X_L , Y_L y Z_L son las coordenadas en 3D de baja resolución (leídas de la información estática del juego). X_1 , Y_1 y Z_1 son valores que forman parte de la información que define los sprites y que permiten trabajar con los objetos que se desplazan solos (ver figura 4.17). En función de cada objeto tienen un valor u otro. Finalmente *ScreenZ* es una constante que siempre vale 128.

Gracias a este sistema podíamos movernos de $3D_L$ a $3D_H$ y viceversa fácilmente.

Un detalle del cual no disponíamos de información era el punto de referencia de la pantalla. Fue resuelto de una manera muy sencilla, dado que sólo era necesario ir modificando los valores de una posición inicial de un objeto de alguna pantalla e ir visualizando los cambios.

Tomando la habitación 255 como ejemplo (figura 4.25) se modificó su definición para cambiar la posición de un objeto de 02Eh => (6,5,0) a 000h => (0,0,0):

```
0FFh,00Bh,006h
002h,003h,00Ch,0FFh,02Bh,02Eh,035h,037h,03Eh
```

por

```
0FFh,00Bh,006h
002h,003h,00Ch,0FFh,02Bh,000h,035h,037h,03Eh
```

El resultado puede verse en la figura 4.25.



Figura 4.25 – Modificación de los ejes X e Y

La siguiente modificación nos ilustra sobre el eje Z, modificando el mismo byte de 02Eh => (6,5,0) a 0FFh => (7,7,3):

```
0FFh, 00Bh, 006h
002h, 003h, 00Ch, 0FFh, 02Bh, 02Eh, 035h, 037h, 03Eh
```

por

```
0FFh, 00Bh, 006h
002h, 003h, 00Ch, 0FFh, 02Bh, 0FFh, 035h, 037h, 03Eh
```

El resultado se aprecia en la figura 4.26.



Figura 4.26 – Modificación eje Z

Una vez hecho todo esto, ya teníamos claro como el sistema de coordenadas estaba orientado, tal como muestra la figura 4.27.



Figura 4.27 – Orientación eje coordenadas

4.8 EL PERSONAJE PRINCIPAL

Del personaje principal no disponíamos de información sobre cómo se almacenaba durante el juego, así que nos situamos en una localización sin objetos y, tras parar el tiempo (anulando la llamada a la función que se encarga del transcurso del día y la noche, la cual también guarda información en la memoria de trabajo), capturamos completamente la memoria de trabajo. Desplazamos el personaje para volver a capturar la memoria de trabajo y así sucesivamente. Tras contrastar las diversas capturas de la memoria de trabajo encontramos las coordenadas del personaje:

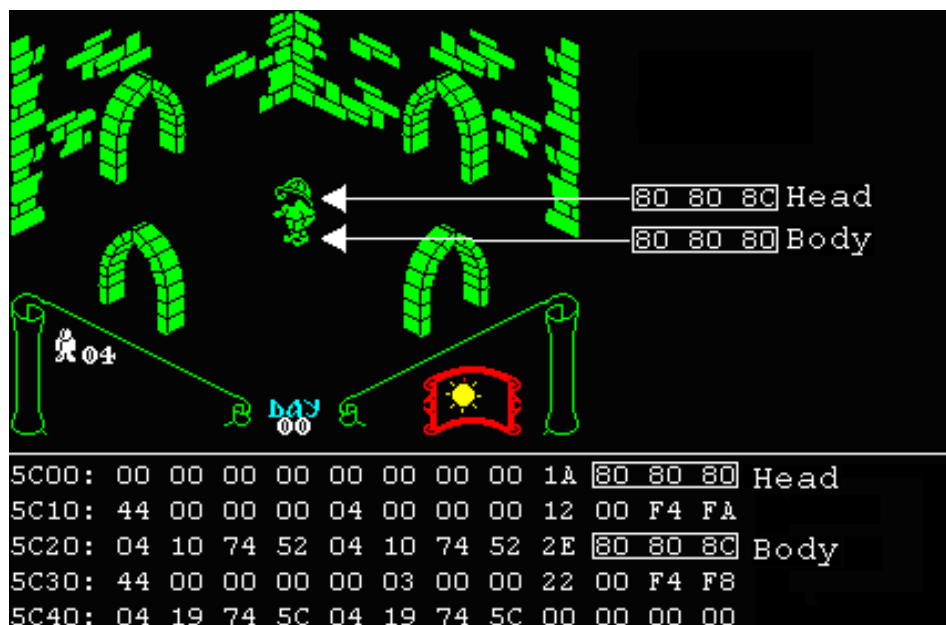


Figura 4.28 – Coordenadas del personaje

Las coordenadas 80 80 80, situadas en las posiciones de memoria 5C09h, 5C0Ah y 5C0Bh, representan la posición del cuerpo del personaje principal y 80 80 8C, situadas en las posiciones de memoria 5C20h, 5C2Ah y 5C2Bh, representa la posición de la cabeza. La representación de los personajes en el juego está formada por dos bloques (figura 4.29), para así poder aplicar diferentes animaciones en el mismo sprite de manera separada.



Figura 4.29 – Sprites que forman el personaje principal

Pero con sus coordenadas no era suficiente, ya que necesitábamos de algún indicador que comunicase la dirección en la cual estaba orientado el personaje. Eso se deduce controlando los siguientes bytes marcados en la figura 4.30.

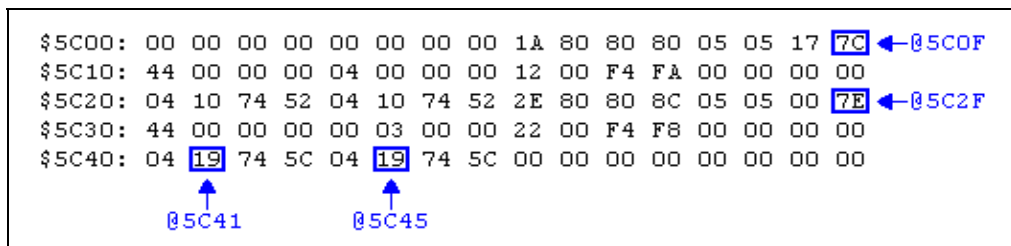



Figura 4.30 – Datos orientación personaje

Los bytes situados en la dirección 5C41h y 5C45h (figura 4.30) indican el sprite del personaje (cabeza y resto del cuerpo). Esto no es suficiente, dado que el sprite con

4.9 LOS GRÁFICOS DE LOS FONDOS Y OBJETOS



La información que define gráficamente el sprite del bloque se muestra en la figura 4.32.

```

04 1C
00 00 01 00 80 00 00 00 00 00 07 01 E0 80 00 00
00 00 1F 06 F8 A0 00 00 00 00 7F 1E FE 50 00 00
01 00 FF 7E FF AA 80 00 07 01 FF FE FF 55 E0 00
1F 07 FF FE FF AA F8 80 7F 1F FF FE FF 55 FE 50
FF 7F FF FE FF AA FF AA 7F 7F FF FE FF 55 FF 54
FF 7F FF FE FF AA FF AA FF 7F FF FE FF 55 FF 54
FF 7F FF F9 FF 8A FF AA FF 7F FF E7 FF E5 FF 54
FF 7F FF 9F FF F8 FF AA FF 7E FF 7F FF FE FF 54
FF 79 FF FF FF FF FF 8A FF 67 FF FF FF FF FF E4
FF 5F FF FF FF FF FF FA FF 7F FF FF FF FF FF FE
7F 1F FF FF FF FF FE F8 1F 07 FF FF FF FF F8 E0
07 01 FF FF FF FF E0 80 01 00 FF 7F FF FE 80 00
00 00 7F 1F FE F8 00 00 00 00 1F 07 F8 E0 00 00
00 00 07 01 E0 80 00 00 00 00 01 00 80 00 00 00

```

50

sprite data		
data for sprites		
type	offset	description
BYTE	00	width
BYTE	01	height
BYTE	03...	image data (size = (width* height)*2) [0 ... height] [0 ... width]
BYTE	00	image
BYTE	01	mask

Figura 4.33 – Formato de la información gráfica de los sprites

El programa, para adquirir la información gráfica de un sprite utiliza el patrón de la información gráfica del sprite de la figura 4.33 y aplica los siguientes pasos:

- Obtiene el primer byte corresponde a la anchura del sprite. Si observamos la figura 4.32 veremos que en el caso del bloque, tiene un valor de 04h. Esto indica que este sprite “medirá” 32 pixels de ancho, eso es así porque cada byte representa 8 pixels: $4 \times 8 = 32$.
- El segundo byte corresponde a la altura del sprite. En el caso de la figura 4.32 es 1Ch, 28 decimal. Esto indica que el sprite del bloque tendrá una altura de 28 pixeles (aquí no se multiplica por 8 pues un byte sólo representa una fila, no una columna).
- Del producto de los dos anteriores bytes obtendremos el tamaño total del sprite, 4 bytes x bytes 28 = 112 bytes. Este resultado nos indica que la información gráfica del sprite constará de 224 bytes: como podemos observar en la figura 4.33 la información seguirá la pauta de imagen – máscara, por lo tanto la imagen constará de 112 bytes y la máscara de 112 bytes.
- Ahora el proceso consiste en ir adquiriendo los bytes sucesivamente hasta llegar al número total (224). En nuestro proyecto hemos descartado la máscara pues no nos servía para nada. Los primeros bytes de la imagen descompondrían de la manera ilustrada en la siguiente tabla:

BYTE IMAGEN	BINARIO	PIXELS
00h	00000000b	
01h	00000001b	
80h	10000000b	
...
07h	00000111b	
...

Una vez adquiridos todos los bytes, almacenados en la estructura correcta, se puede utilizar para generar texturas. Todos los gráficos de los fondos y objetos siguen el mismo proceso para su adquisición.

5. IMPLEMENTACIÓN DE KNIGHT LORE 2006

5.1 PUNTO DE ENTRADA

Una vez delante del código fuente del emulador, nos dispusimos a modificarlo para poder permitir la coexistencia de las dos ventanas (la del emulador con el juego clásico y la del entorno 3D).

Para crear el punto de entrada del nuevo entorno 3D se tuvo que encapsular gran parte del bucle principal del emulador en una función específica, para que pudiéramos controlar el flujo de llamadas y así multiplexar su ejecución. Creamos una función que realizaba una iteración del emulador y después añadimos una función que se encargaba de actualizar la ventana del nuevo entorno 3D.

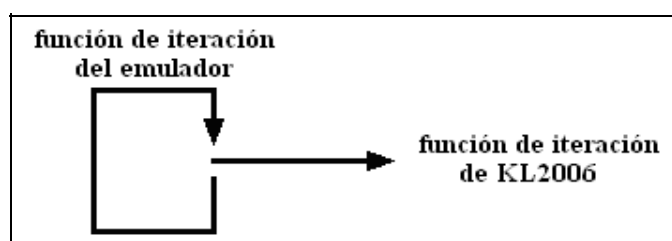


Figura 5.1 – Diagrama de entrada inicial

Llegado este punto nos surgió una problemática al pasar el control al nuevo entorno 3D. Por definición, OpenGL es una máquina de estados, y una vez se le ha pasado el control, itera indefinidamente hasta que se termina el programa o se le obliga a terminar. Esto hacía que el emulador se quedara “estancado” ejecutando el entorno 3D. La situación comportó un cambio de planteamiento de las modificaciones del emulador. En lugar de ejecutar la función que realiza una iteración del emulador, y luego ejecutar la que genera el entorno 3D, se optó conceder el control absoluto al entorno 3D y mediante funciones propias de OpenGL que permiten especificar qué hacer en periodos de inactividad, ir concediendo iteraciones a la función de emulación.

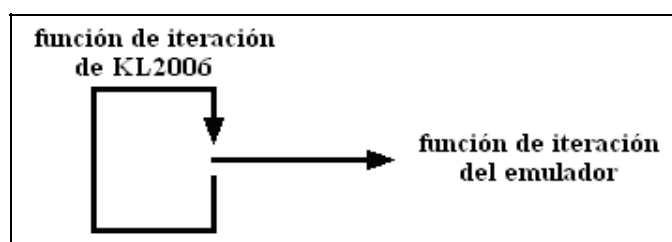


Figura 5.2 – Diagrama de entrada actual

En la figura 5.3 podemos observar como se implementó el esquema de la figura 5.2 para permitir la simultaneidad de los dos entornos. La función principal del emulador es la que lleva por nombre main (o emuMain, dependiendo del compilador). Podemos ver el código comentado de cómo funcionaba el emulador originalmente: disponía de un bucle que iba realizando sucesivas operaciones, las que ahora están encapsuladas en la función emuMainLoop, lo que permite llamarla cuando se considere necesario ejecutar un “ciclo” del emulador.

El código continúa con una serie de funciones de GLUT (llevan el prefijo glut) que se encargan de definir el entorno OpenGL, de las cuales dos se encargan del flujo de ejecución: `glutDisplayFunc()` y `glutIdleFunc()`.

La función `glutDisplayFunc()` se ocupa de indicar que función realiza el dibujado, que será la función `display()`. Ahora cada vez que el usuario modifique la ventana, o genere un evento se ejecutará la función `display()` que es la que se encarga de llamar a la función de adquisición de los datos y, una vez hecho eso, los plasmarlos gráficamente. El hecho de que la función `display()` esté asociada a los cambios que puedan producirse no significa que también pueda ser llamada explícitamente.

La función `glutIdleFunc()` se encarga de especificar que se hará en los periodos de inactividad, en este caso se llamará a la función `emuMainLoop()`.

Vemos que estas dos disposiciones hacen que se vaya dibujando el nuevo entorno cuando se necesaria, y a la vez se ejecuta el código de emulador que nos presenta la figura 5.3. Originalmente, la función de la figura 5.3, sólo constaba del bucle, y dentro de él todas las sentencias de las que consta actualmente la función `emuMainLoop()`, estas sentencias son las que hacen posible el funcionamiento del emulador.

```
#ifndef ZXDEBUG_MFC
int main (int argc, char *argv[])
#else
int emuMain (int argc, char *argv[])
#endif
{
    ...

    // MAIN LOOP
    // while (!done)
    // {

        emuMainLoop();

    // }

    glutInit(&argc,argv);
    glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGBA | GLUT_DEPTH);
    glutInitWindowPosition(0, 0);
    glutInitWindowSize(500, 500);
    glutCreateWindow("Knight Lore 06");
    init();
    crear_menu();
    glutDisplayFunc(display);
    glutIdleFunc(emuMainLoop);
    glutMainLoop();

    return (1);
}
```

Figura 5.3 – Implementación del punto de entrada

Después de implementar la pauta, se pudieron visualizar los dos entornos a la vez, como se muestra en la figura 5.4.

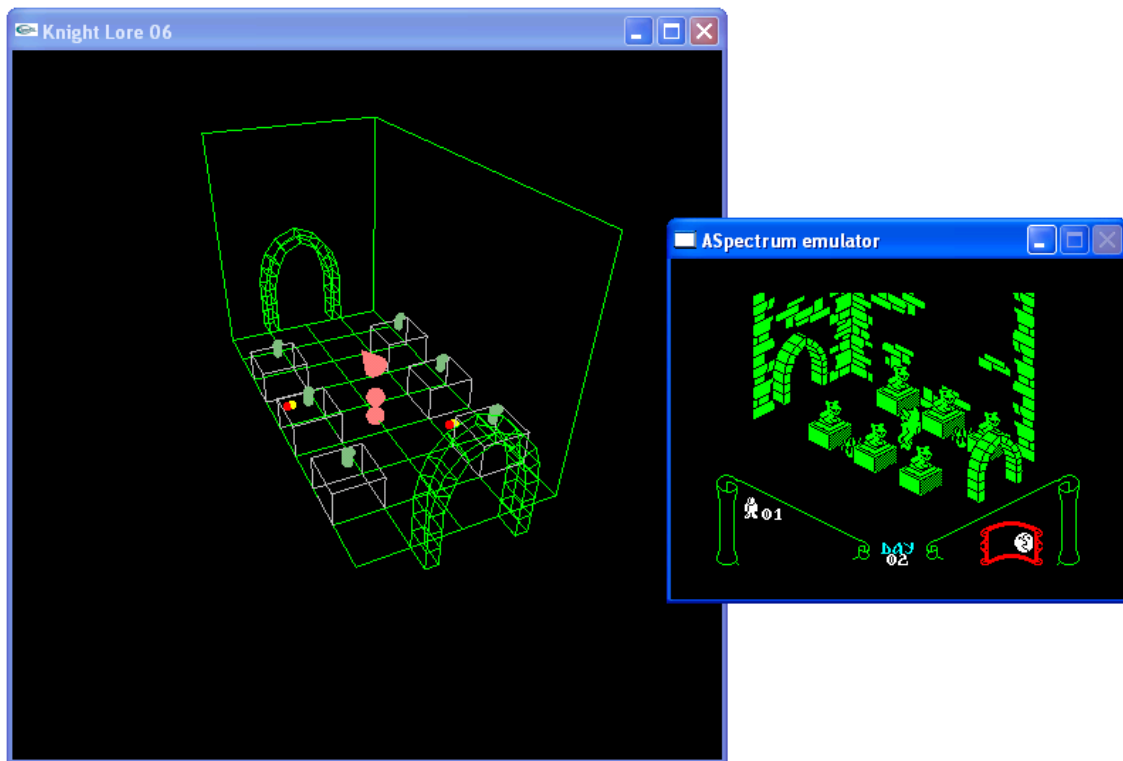


Figura 5.4 – Los dos entornos funcionando simultáneamente

5.2 ADQUISICIÓN DE LOS DATOS

Primeramente se hizo necesario crear una función que adquiriera todos los gráficos que se requirieran para la nueva recreación del juego. La función se denomina `cargar_datos_habitación()` y como indicia su nombre, se encarga de adquirir la información del juego, concretamente las fuentes que consulta son:

- Información inicial estática que define las localizaciones
- Memoria de trabajo
- Tabla de los datos de los fondos
- Tabla de los datos de los objetos

Antes que nada se añadió al código del emulador una estructura para almacenar los datos adquiridos, cuyo pseudocódigo es el presentado en la figura 5.5:

```
Definir estructura habitación {  
    atributos  
    fondos  
    objetos  
}
```

Figura 5.5 – Estructura de datos de la habitación

El pseudocódigo de la función `cargar_datos_habitación()` es el que se encuentra en la figura 5.6:

```
repetir {  
    atributos_habitación -> habitación.atributos  
  
    datos_estáticos.fondos -> habitación.fondos  
  
    datos_estáticos.objetos -> habitación.objetos  
    memoria_trabajo.objetos -> habitación.objetos  
}
```

Figura 5.6 – Pseudocódigo función `cargar_datos_habitación()`

Los atributos de la habitación se adquieren de las posiciones de memoria que los contienen:

- El identificador de la habitación (con un valor situado entre 000h y 0FFh) está situado en la dirección 5C10h.
- El color de la habitación está en la dirección 5BADh.
- El tamaño (x,y,z) lo podemos encontrar en las direcciones 5BABh, 5BACH y 5BAEh respectivamente.

Los fondos de la habitación se adquieren de la información estática. El motivo por el que se adquiere únicamente de este lugar es porque sus datos nunca variarán. Las paredes o las puertas nunca se moverán ni desaparecerán. Se muestra la porción de código que realiza esto en la figura 5.7, la cual lee tantas veces como fondos hay, los identificadores de los fondos. Para entender completamente el código de la figura 5.7 puede consultarse la sección 4.5.

```
// Obtener backgrounds  
t_room.num_backgrounds = 0;  
  
p_fin_hab_act = pos_mem_num_hab_tmp + t_room.offset_hab;  
pos_mem_backg_tmp = pos_mem_num_hab_tmp + 3;  
  
do {  
    t_room.backgrounds[t_room.num_backgrounds] = readmem(pos_mem_backg_tmp);  
    t_room.num_backgrounds++;  
    pos_mem_backg_tmp++;  
} while ((readmem(pos_mem_backg_tmp) != 0x00FF) && (pos_mem_backg_tmp <= p_fin_hab_act));
```

Figura 5.7 – Obtención de los datos de los fondos de la información estática

Los datos que se obtienen de los objetos deben adquirirse no sólo de la información estática, sino que también de la memoria de trabajo y de la tabla de datos de los objetos. De la información estática, obtenemos de qué objetos dispone la habitación y el número de sus representaciones (ver figura 5.8).

De la memoria de trabajo, adquirimos sus coordenadas (los objetos pueden moverse o desaparecer) y otros datos (ver figura 5.9). Finalmente de la tabla de datos del objeto obtenemos los valores para hacer, si son necesarias, correcciones de posicionamiento (consultar apartado 4.7). Para entender completamente el código de la figura 5.7 y 5.8 pueden consultarse la sección 4.5.2.

```

// Obtener objects
t_room.num_objects = 0;

pos_mem_obj_tmp = pos_mem_backg_tmp + 0x0001;

while (pos_mem_obj_tmp < p_fin_hab_act) { // Comprueba que hay objetos
    tip_obj = (readmem(pos_mem_obj_tmp) & 0x00f8) >> 3; // Objeto
    cant_obj = (readmem(pos_mem_obj_tmp) & 0x0007) + 1; // Cantidad del objeto

    for (i = 1; i <= cant_obj; i++) {
        pos_mem_obj_tmp++;
        t_room.tobjects[t_room.num_objects].num_obj = tip_obj;
        t_room.num_objects++;
    }

    pos_mem_obj_tmp++;
}

```

Figura 5.8 – Obtención de los datos de los objetos de la información estática

```

if (t_room.num_objects > 0) {

    //Obtener coordenadas objetos de la memoria de trabajo
    pos_mem_obj_tmp = 0x5c89;

    despla_hasta_objs = 0; // Contendrá el número a sumar a pos_mem_obj_tmp

    for (i = 0; i < t_room.num_backgrounds; i++) {
        despla_hasta_objs += (long_datos_backs(t_room.backgrounds[i]) * 0x0020);
    }

    pos_mem_obj_tmp += despla_hasta_objs;

    for (i = 0; i < t_room.num_objects; i++) {
        t_room.tobjects[i].sprite = readmem(pos_mem_obj_tmp-1);
        t_room.tobjects[i].coord_x = readmem(pos_mem_obj_tmp);
        t_room.tobjects[i].coord_y = readmem(pos_mem_obj_tmp+1);
        t_room.tobjects[i].coord_z = readmem(pos_mem_obj_tmp+2);
        pos_mem_obj_tmp += long_datos_objs(t_room.tobjects[i].num_obj) * 0x0020;
    }
}

```

Figura 5.9 – Obtención de los datos de los objetos de la memoria de trabajo

5.3 INICIALIZANDO OPENGL

OpenGL es una biblioteca que trabaja como una máquina de estados. Cuando la utilizamos, lo primero que hay que realizar es activar y desactivar opciones, realizar ciertas acciones que tendrán como fruto una representación en pantalla de una serie de datos, dependiendo del estado en que nos encontremos.

En este proyecto primeramente es llamada una función de inicialización que se encarga de definir los atributos de color, proyección y Z-Buffer (array que guarda información de la profundidad que permite sobreponer objetos en el orden correcto).

```

void init() {
    glDepthFunc(GL_LEQUAL);
    glEnable(GL_DEPTH_TEST);
    glClearColor(0.0, 0.0, 0.0, 1.0);
    glClearDepth(1.0);

    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluPerspective(60.0, 1.0, 1.0, 100.0);
}

```

Figura 5.10 – Función de inicialización OpenGL

Posteriormente, cada vez que se entra en la función que grafica la nueva pantalla, justo después de cargar los datos de una habitación, se ejecutan las siguientes funciones de OpenGL.

```

glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

glMatrixMode(GL_MODELVIEW);

glLoadIdentity();

gluLookAt(valorLookAtX, valorLookAtY, valorLookAtZ, 4.0, 0.0, -4.0, 0.0, 1.0, 0.0);

```

Figura 5.11 – Funciones que definen, posicionan y orientan la cámara

Su utilidad básica es indicar a OpenGL donde “mira” la cámara. Está incluida en la función de dibujo de la pantalla de Knight Lore 2006, porque en cualquier momento el usuario puede cambiar la perspectiva del juego.

5.4 DIBUJANDO LOS FONDOS

Las paredes de cada habitación se dibujaron utilizando las texturas del juego original. El proceso comprendió la captura de cada sprite (ver figura 5.12), que conformaba cada pared para su posterior almacenamiento en un fichero gráfico. En la sección 4.9 se explica detalladamente como están almacenadas las imágenes del juego.



Figura 5.12 – Diferentes sprites que representan paredes

La función de dibujado de fondos examina los datos dinámicos de la habitación, para obtener los identificadores de los sprites que formarán parte de la pared. Después, carga los ficheros que corresponden a los sprites de la pared y los coloca en función de su posición en el nuevo entorno.

Hay que tener en cuenta que las texturas capturadas del juego original presentan una distorsión en perspectiva ortográfica, pero que se solucionó gracias a que, mediante OpenGL, a la hora de aplicar una textura a un objeto podemos corregir su posicionamiento mediante la técnica Dewarping.

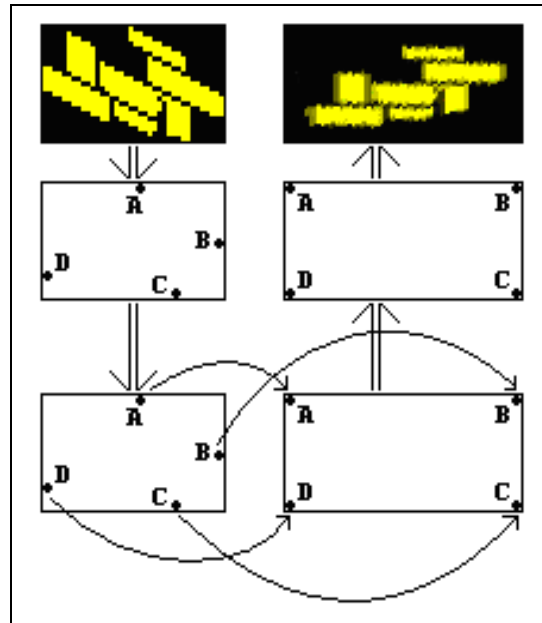


Figura 5.13 – Corrección perspectiva de textura mediante la técnica Dewarping

La técnica Dewarping, elige una serie de puntos, que se pasan como coordenadas de textura a OpenGL. Entonces éste último, realiza las correcciones necesarias para aplicar la textura para su correcta visualización, conceptualmente se muestra en la figura 5.13.

El resto de los fondos (mayoritariamente puertas) se dibujaran como cualquier otro objeto, utilizando primitivas OpenGL.

5.5 DIBUJANDO LOS OBJETOS

Todos los objetos son dibujados mediante primitivas OpenGL, un ejemplo podemos observarlo en la figura 5.14. La función que se encarga de dibujarlos es llamada por la función de principal tantas veces como objetos dispone la localización. Cada llamada incluye los argumentos necesarios que describen el objeto. En función del identificador de objeto que recibe, aplica unas primitivas u otras. Si hay algún objeto no definido entonces se dibuja el objeto por defecto, que en nuestro caso es una pequeña esfera.

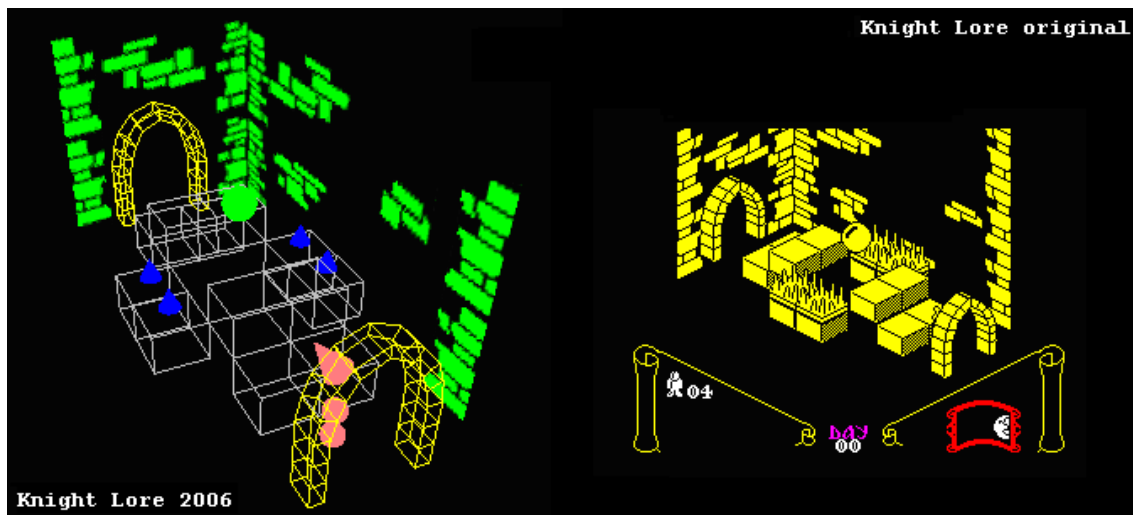


Figura 5.14 – Habitación con objetos

Hay que destacar que aunque hay varios objetos que, aunque comparten un aspecto visual, de cara a la interacción en el juego se comportan de diferente manera e incluso no disponen del mismo identificador de sprite. Es el caso de un objeto muy común, el bloque (ver figura 5.15), el cual disponen de diferentes entidades (bloque fijo, bloque que desaparece al ser pisado, bloque con movimiento), que aunque internamente el juego los trata de manera diferente, su graficación no requiere de esa diferenciación y los considera y dibuja como si fueran un mismo objeto.



Figura 5.15 – Diferentes sprites con la misma imagen

Los objetos que disponen de animación (pelota, fuego, soldado, etc) requieren de diferentes sprites para representarla. También en este caso es algo transparente para el nuevo entorno del juego y es el juego original que va modificando la memoria de trabajo en función del sprite que requiere cada objeto animado en función de su estado.



Figura 5.16 – Sprites que representan varias animaciones

5.6 DIBUJANDO EL PERSONAJE PRINCIPAL

La función que dibuja el personaje principal, primeramente obtiene las coordenadas (x,y,z) situadas en las posiciones de memoria 5C09h, 5C0Ah y 5C0Bh (ver sección 4.8). Seguidamente convierte estos valores de coordenadas 3D High al entorno OpenGL (ver apartado 4.7). Una vez conocida la posición el siguiente paso es deducir el estado del personaje (hombre o lobo) y su orientación. Consultando la posición de memoria 5C41h obtendremos el identificador del sprite del cuerpo del personaje, ello nos permite saber su estado y parte de su orientación. Para terminar de saberla, deberemos consultar el valor de la posición de memoria 5C0Fh, en función de su resultado podremos deducir definitivamente la orientación.

```
coordPerX = readmem(0x5c09);
coordPerY = readmem(0x5c0a);
coordPerZ = readmem(0x5c0b);

cordX = convertirCoordenada(coordPerX);
cordY = convertirCoordenada(coordPerY);
cordZ = convertirCoordenada(coordPerZ);

spriteCuerpo = readmem(0x5c41);

if ((spriteCuerpo == 0x18) || (spriteCuerpo == 0x19))
    cuerpo = 1; // Hombre
else
    cuerpo = 2; // Lobo

if (readmem(0x5c0f) >= 0x4c) {
    direccion = 2; // Sur
    if ((spriteCuerpo == 0x18) || (spriteCuerpo == 0x1d)) direccion = 1; // Norte
} else {
    direccion = 3; // Este
    if ((spriteCuerpo == 0x18) || (spriteCuerpo == 0x1d)) direccion = 4; // Oeste
}
```

Figura 5.17 – Código que obtiene la posición y orientación del personaje principal

En la figura 5.17 podemos observar como primeramente la función obtiene las coordenadas de la memoria de trabajo y las convierte al entorno OpenGL. Seguidamente obtiene el byte situado en la dirección de memoria 5C41h para discernir si el personaje es hombre o lobo (ver figura 5.18). Por último en función del valor de la posición de memoria 5C0Fh y el sprite determinará la orientación (ver sección 4.8).









Id	Sprite	Flip horizontal	Id	Sprite	Flip horizontal
18		<input type="checkbox"/>	1D		<input type="checkbox"/>
18		<input checked="" type="checkbox"/>	1D		<input checked="" type="checkbox"/>
19		<input type="checkbox"/>	1E		<input type="checkbox"/>
19		<input checked="" type="checkbox"/>	1E		<input checked="" type="checkbox"/>

Figura 5.18 – Sprites personaje principal

Una vez adquiridos los datos que requerimos del personaje para dibujarlo, aplicamos primeramente una rotación, posteriormente una traslación y luego utilizamos varias primitivas de OpenGL para finalmente dibujar el personaje. En estas operaciones se ha tenido en cuenta la orientación (para eso se aplica la rotación) y las coordenadas de su posición, para su correcto posicionamiento.



Figura 5.19 – Personaje principal en el viejo y nuevo contexto

6. RESULTADOS, CONCLUSIONES Y TRABAJOS FUTUROS

6.1 RESULTADOS

El resultado final consta de un nuevo entorno, como podemos observar en la figura 6.1. La figura 6.2 muestra una secuencia del movimiento de un balón botando. También dispone de las opción de visualizar el juego en 3D desde diferentes perspectivas y un mejorada aspecto visual, como comprobamos en las figuras 6.3 y 6.4.

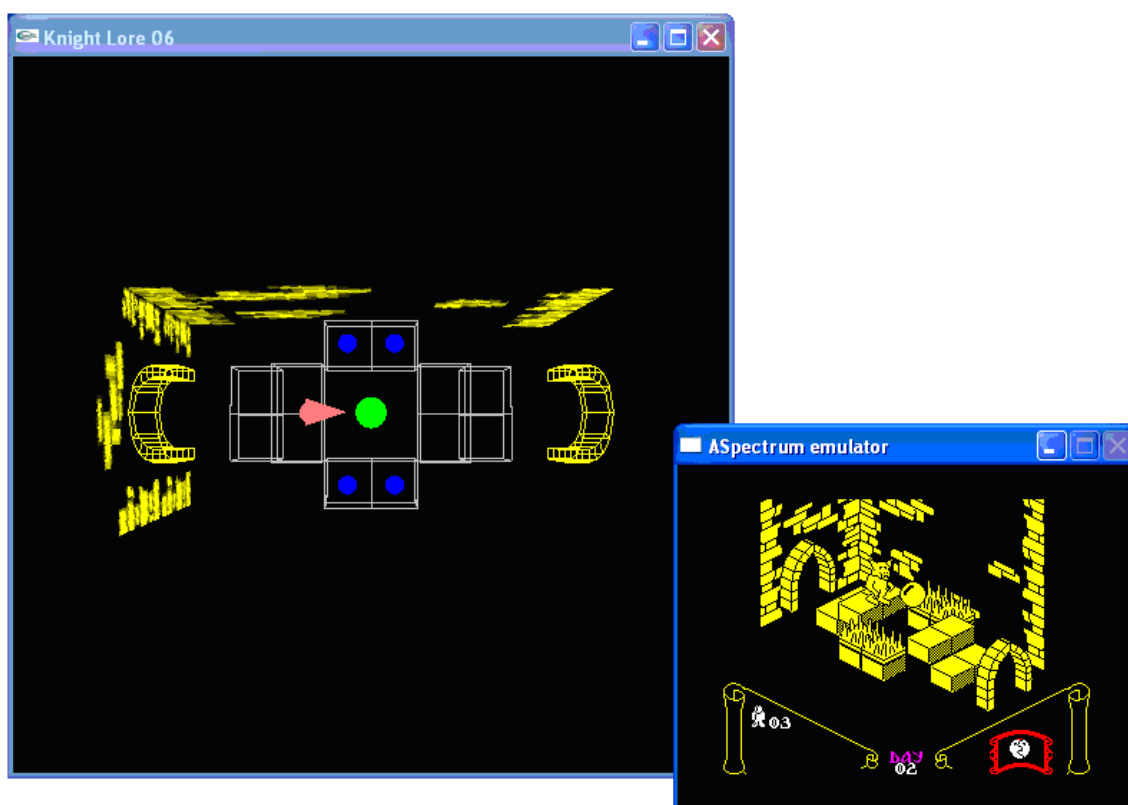


Figura 6.1 – Aspecto final de la aplicación híbrida

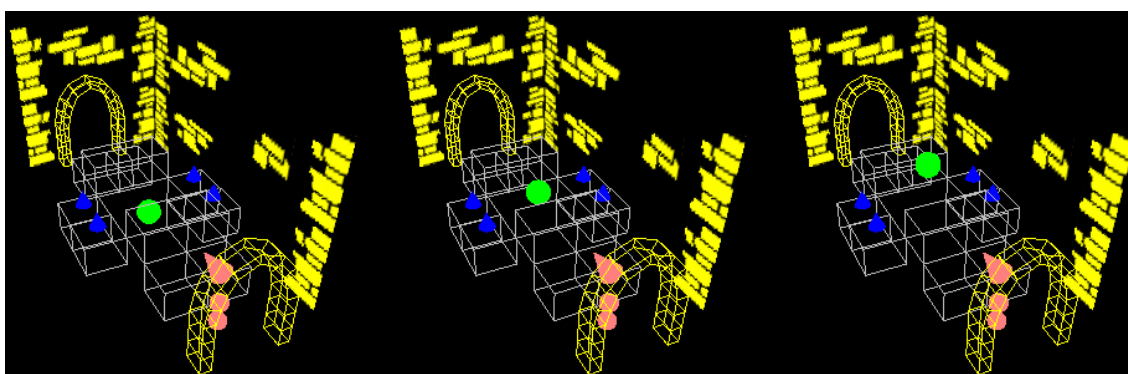


Figura 6.2 – Secuencia de un balón en movimiento en una habitación

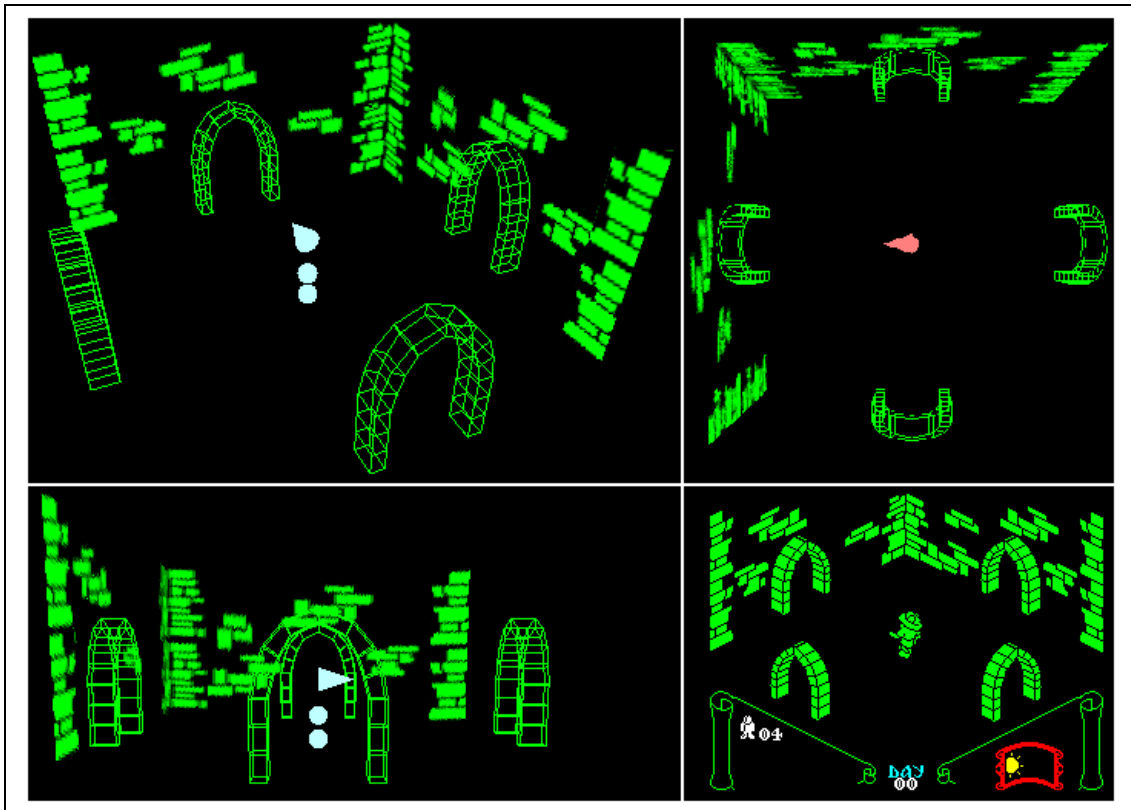


Figura 6.3 – Diferentes vistas de una habitación de Knight Lore 2006 y el original

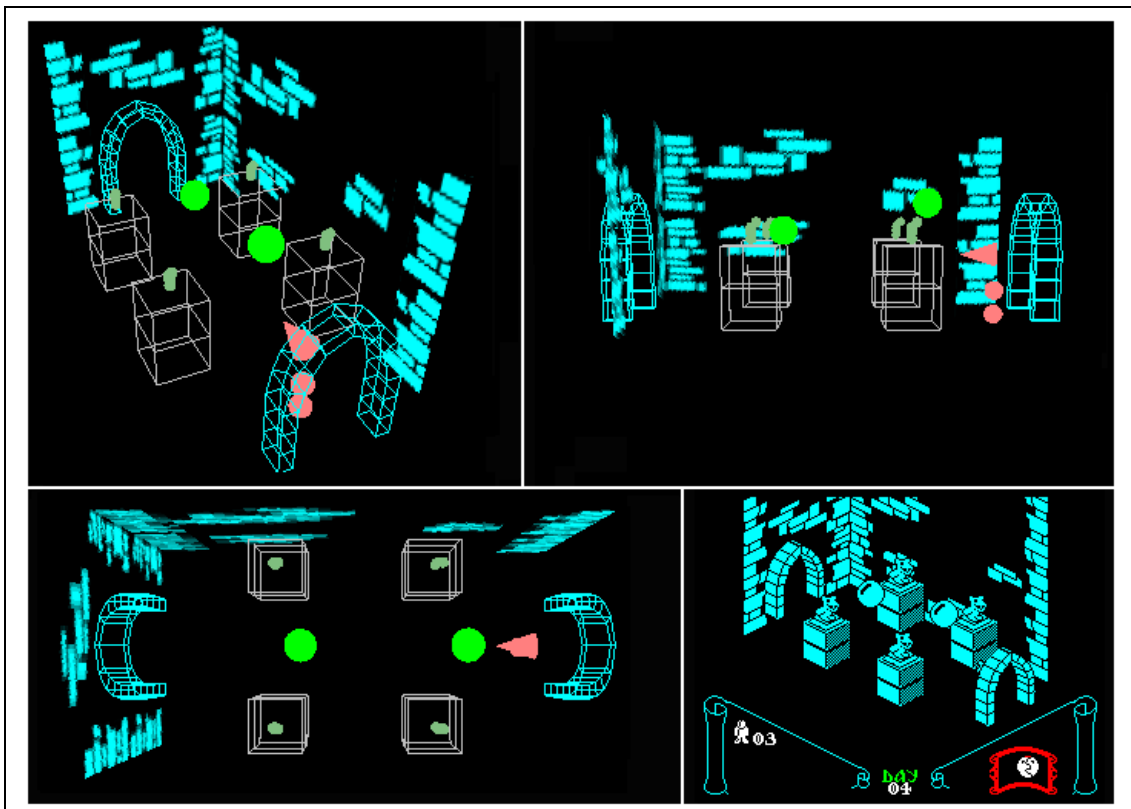


Figura 6.4 – Diferentes vistas de una habitación de Knight Lore 2006 y el original

6.2 CONCLUSIONES

Hemos conseguido desarrollar un remake de un juego clásico del ZX Spectrum, pero sin reconstruir el juego completamente sino añadiendo funcionalidades para generar un nuevo contexto gráfico, siendo desviado el flujo gráfico e implementándolo nuevamente en otro contexto 3D. Para ello hemos estudiado la arquitectura del ZX Spectrum, hecho uso de la ingeniería inversa para descubrir como trabaja el juego a nivel gráfico y creado un nuevo motor de renderización basado en elementos geométricos y texturas.

A nivel personal, concluimos el trabajo satisfechos por los resultados obtenidos, que se ciñen a los objetivos especificados al principio, que permiten disponer de una plataforma funcional para añadir nuevas funcionalidades y mejoras.

6.3 TRABAJOS FUTUROS

Respecto a los trabajos futuros a desarrollar podrían ser:

- Dibujar los objetos mediante diferentes técnicas apropiadas para representarlos y conseguir un mejor acabado final de los mismos.
- Aplicar fuentes de luz para mejorar el aspecto en consonancia a la temática del juego.
- Mejorar la técnica de captura de sprites para que se obtengan directamente del código de juego.
- Generalización del código para que se adapte a otros juegos que trabajen mediante las mismas técnicas, aunque se reduce a varios juegos sacados por la misma compañía y otras afines.

7 BIBLIOGRAFÍA

El Microprocesador Z-80

Primitivo de Francisco

<http://www.microhobby.org/varios/MICROHOBBY-ElmicroprocesadorZ80.pdf>

Curso de Código Máquina del ZX-Spectrum

Jesús Alonso Rodríguez

<http://www.microhobby.org/varios/cursocodigomaquina.zip>

Iniciación al sistema Filamation

Revista MicroHobby 96, 97, 98 y 99

<http://www.microhoby.org>

On filmation

Neil Walker

<http://retrospec.sgn.net/users/nwalker/filmation/>

Knight Lore data format

Chris Wild

<http://www.icemark.com/dataformats/knightlore/index.html>

Emuladores del Spectrum

Santiago Romero

<http://www.speccy.org/sromero/spectrum/emuls/>

The OpenGL Programming Guide

Dave Shreiner, Mason Woo, Jackie Neider and Tom Davis

http://opengl.org/documentation/red_book/

The OpenGL Utility Toolkit (GLUT) Programming Interface

Mark J. Kilgard

<http://www.opengl.org/documentation/specs/glut/>

Curso de introducción a OpenGL

Jorge García Ochoa de Aspuru

<http://www.bardok.net/content.php?lang=1&article=2>

Apuntes de OpenGL y GLUT

Cristina Cañero Morales

<http://www.cvc.uab.es/shared/teach/a21306/doc/Apuntes%20de%20OpenGL.pdf>

Aula Macedonia – Curso de programación gráfica en OpenGL

Oscar García

<http://usuarios.lycos.es/macedoniamagazine/opengl.htm>

Prácticas de informática gráfica

Arno Formilla y M^a Victoria Luzón García

<http://trevinca.ei.uvigo.es/~formella/doc/ig02/>

Informática Gráfica

Inma Remolar, Óscar Belmonte y J. Francisco Ramos

<http://graficos.uji.es/grafica/>

Prácticas de Infografía I

Alejandro Ramírez Montero














<http://www.ucbcba.edu.bo/maestrias/MATERIAS/grafismo/infografia/index.html>


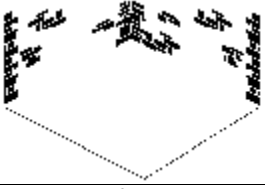









NeHe Productions – OpenGL Lessons

<http://nehe.gamedev.net/>





























APÉNDICES

A. LOS FONDOS

ID	NOMBRE	GRÁFICO
00h	Arco norte	
01h	Arco este	
02h	Arco sur	
03h	Arco oeste	
04h	Arco árbol norte	
05h	Arco árbol este	
06h	Arco árbol sur	
07h	Arco árbol oeste	
08h	Reja norte	
09h	Reja este	
0Ah	Reja sur	
0Bh	Reja oeste	
0Ch	Habitación muros tamaño 1	

0Dh	Habitación muros tamaño 2	
0Eh	Habitación muros tamaño 3	
0Fh	Habitación árbol 1	
10h	Arboleda de relleno oeste	
11h	Arboleda de relleno norte	
12h	Hechicero	
13h	Caldero	
14h	Arco este alto	
15h	Arco sur alto	
16h	Arco este alto base	
17h	Arco sur alto base	

B. LOS OBJETOS

ID	NOMBRE	GRÁFICO
00h	Bloque	
01h	Fuego	
02h	Bola con movimiento vertical	
03h	Roca	
04h	Gárgola	
05h	Bloque de pinchos	
06h	Baúl empujable	
07h	Mesa empujable	
08h	Guardia con movimiento este - oeste	
09h	Fantasma	
0Ah	Fuego con movimiento norte - sur	
0Bh	Bloque alto	
0Ch	Bola con movimiento vertical	
0Dh	Guardia patrullando perímetro	
0Eh	Bloque movable este – oeste	
0Fh	Bloque movable norte – sur	
10h	Bloque movable	
11h	Bloque pinchos alto	
12h	Bola con pinchos cayendo	
13h	Bola con pinchos cayendo del techo	
14h	Fuego con movimiento este - oeste	
15h	Bloque arrastrable	
16h	Bloque inamovible	
17h	Bola con movimiento aleatorio	
18h	Bola cayendo desde arriba	
19h	Hechizo asesino	
1Ah	Reja con movimiento vertical	
1Bh	Reja con movimiento vertical	
1Ch	Bola con movimiento vertical	