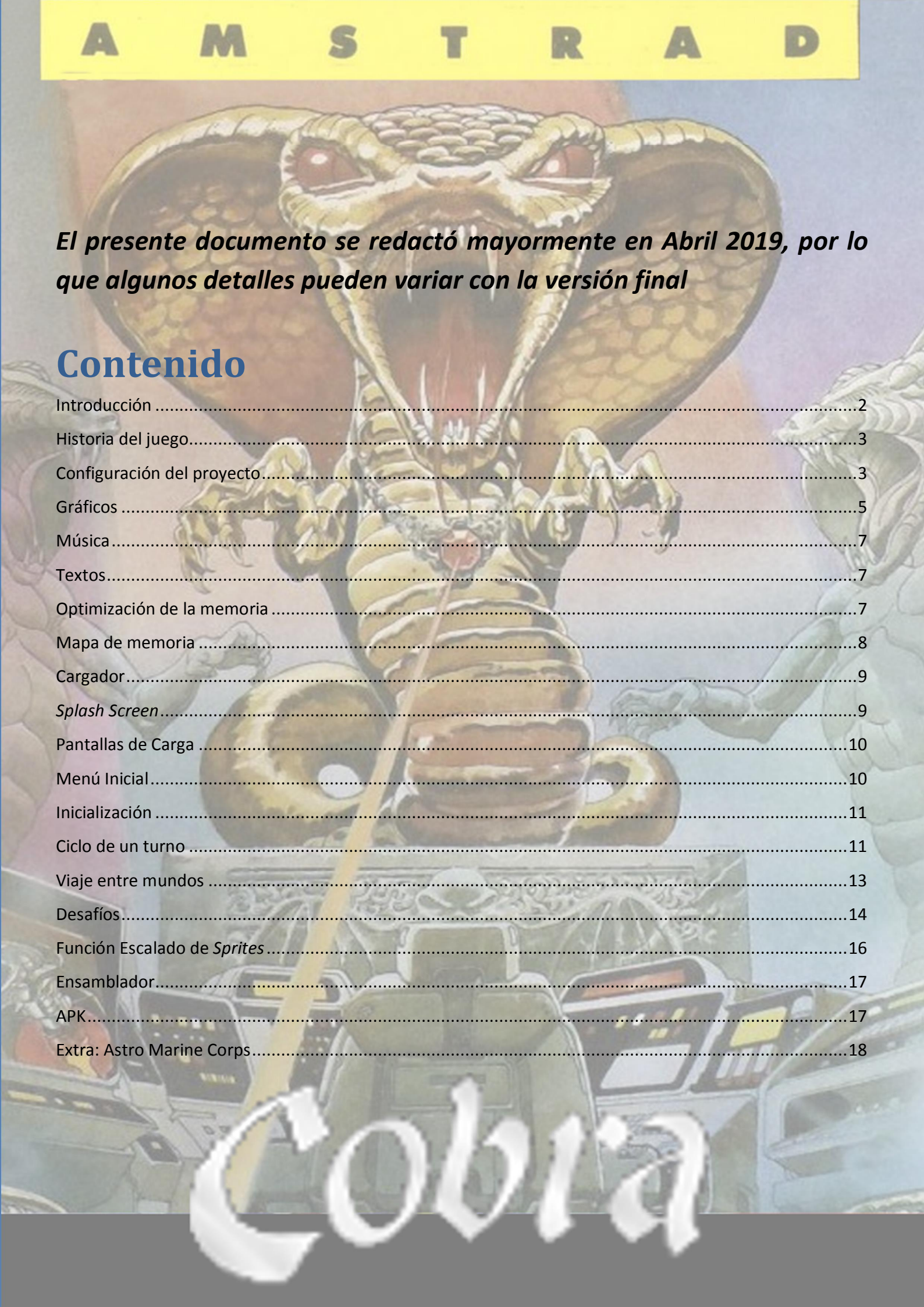


El presente documento se redactó mayormente en Abril 2019, por lo que algunos detalles pueden variar con la versión final

Contenido

Introducción

Este documento recoge los puntos más importantes de la creación del juego **COBRA** (basado en el conocido juego de mesa "En busca del Imperio Cobra"). A lo largo de las distintas secciones se irán comentando los distintos aspectos de la creación. Para más información, el código fuente original muestra abundantes comentarios sobre las funciones que realiza cada parte del mismo.

Para este desarrollo quería pintar varias pantallas completas (los tableros del juego de mesa). Una pantalla completa ocupa 16KB en condiciones normales. Queriendo hacer uso hasta de 3 pantallas completas (uno por cada mundo), se veía imposible conseguirlo en los **64KB** (48KB reales) del CPC 464. No quedaría ya espacio para el resto, ocuparíamos toda la memoria disponible sin haber puesto siquiera una línea de código. [*Recordemos que las reglas de cpcretrodev impiden hacer uso de carga de datos una vez empezada la partida, por lo que ir cargando cada tablero de disco a memoria según fuera necesario no era posible*]

Por eso decidí pasar a la **versión 1.5** de cpctelera (aunque esté en desarrollo), tiene una opción muy cómoda de comprimir archivos binarios, que es la solución perfecta para mi problema. También decir que en todo el desarrollo no me he encontrado con ningún problema con esta versión de cpctelera, salvo en la instalación. Tuve que pasar todos los scripts por un conversor Linux to Windows por problemas del formato de salto de línea entre una plataforma y otra, pues yo uso cygwin bajo Windows y la versión v1.5 todavía no se ha preparado para este sistema.

La facilidad de cpctelera me ha hecho desarrollar el 99% del juego en C, dejando sólo unos pocos procedimientos en ensamblador, que comentaré más adelante.

Desarrollar en una máquina tan limitada como el Amstrad CPC 464 en comparación con las actuales es todo un desafío; hace que no siempre valga la primera solución que se te ocurra y te hace buscar continuamente optimizaciones para dotar al juego de mayor calidad y contenido. Incluso a veces hay que decidir si conviene gastar más ciclos de reloj y desarrollar un código más corto para ahorrar memoria; o al revés, programar un código más largo que ocupe más memoria, pero que tarde menos ciclos en ejecutarse.

La mayoría del proyecto se desarrolló entre diciembre 2018 y mayo 2019 quedando el juego terminado con escasos 10 Bytes libres. A mediados de septiembre se ofreció **Ricardo Oyón Rodriguez** a mejorar los tableros (que en aquel momento estaban simplemente importados de los escaneados originales, sin tratamiento posterior) y a hacer un pequeño testeo dado que tuvo bastante experiencia en este campo al haber trabajado para Sony Entertainment (Play Station).

Con su incorporación al desarrollo, el juego cambio bastante en concepto, mejorando notablemente tanto la jugabilidad como la gestión de la memoria. Con los nuevos tableros del juego se liberó casi 4 KBytes de memoria, dando lugar a la incorporación de mayor contenido así como afinar algunos detalles a favor de una mejor experiencia de juego. Además de que el aspecto gráfico ha mejorado muchísimo teniendo ahora los tableros la personalidad que se merecen.

Historia del juego

Como se puede apreciar tras un primer vistazo rápido, el juego está basado en el famoso juego de mesa "**En Busca del Imperio Cobra**" de CEFA (creado por Pepe Pineda e ilustrado por Isidre Monés). No quería hacer una trasposición 100% fiel al original, sino una adaptación libre.

En el original, se te asignaban una serie de misiones, que consistían en ir a un lugar en concreto del mapa con un arma específica. Una vez completadas las misiones había que viajar a la isla cobra para el enfrentamiento final.

En esta **adaptación** cambian bastantes cosas para hacer un juego más acorde a los videojuegos, siendo las principales características:

- De 1 a 3 Jugadores, pudiendo ser humanos o controlados por el ordenador, con todas las combinaciones que se puedan producir. Es decir, desde jugar un jugador (sólo contra el tiempo), a que el ordenador juegue con los 3 personajes.
- Sólo hay que completar dos misiones para pasar a la fase final (en vez de tres)
- No existen cartas como el original, para obtener armas hay que recogerlas en posiciones aleatorias de las casillas de los tableros.
- Para tirar el dado, primero hay que superar un desafío. En el caso de haber fallado dos desafíos consecutivos, al tercer turno se tiran directamente los dados, no hay desafío. Si el jugador es el ordenador, el que tenga derecho a tirar dados o no dependerá de una probabilidad (75%). Se mostrará el borde verde si se ha superado el desafío; o el borde en rojo oscuro en caso contrario.
- Hay 9 desafíos, de aparición aleatoria. Se ha intentado que los desafíos sean muy diferentes entre sí para que la cualidad necesaria para superarlos sea diferente: memoria, habilidad, inteligencia,… Normalmente contaremos con tres intentos para superar cada desafío. A medida que avanza la partida los desafíos se harán un poco más difíciles.
- Tras completar las dos misiones, en el siguiente turno habrá un desafío final para arrebatar el ojo mágico al rey Cobra. Si se completa con éxito, ganaremos la partida.
- Hay un desafío especial para viajar de un tablero (mundo) a otro, un viaje en balsa por el mar.
- Las partidas están limitadas por tiempo, es decir, por el número de turnos (32 en total). Se muestra mediante un reloj de arena en la parte inferior derecha. Cuando llevemos la mitad del máximo de turnos, se activará el modo difícil, que hará que algunos desafíos sean un poco más complicados.

Configuración del proyecto

Para el proyecto cambiaré la configuración de los siguientes ficheros:

- music_conversion.mk: Aquí añadiré las músicas que he compuesto con Arkos tracker v1 para que directamente se integren en el proyecto y pueda referenciarlas en otros archivos. *(Hay una música que no irá directamente especificada aquí que comentaré más adelante).* A pesar de haber cambiado el formato de configuración con cpctelera v1.5, no se hace complicado su uso pues es muy similar al anterior.

- compression.mk: En el directorio gfx dejo los 3 tableros en binario (16KB cada uno). En este fichero se indica que se genere la estructura de datos comprimida para que se pueda usar en el proyecto.

- build_config.mk: se cambian los parámetros habituales: el nombre del proyecto, la dirección desde la que se tiene que ir generando el código y los Z80CCFLAGS, en este caso para optimizar el espacio del código resultante pues es la mayor limitación del proyecto (*--max-allocs-per-node 2000000 --opt-code-size*).

Gráficos

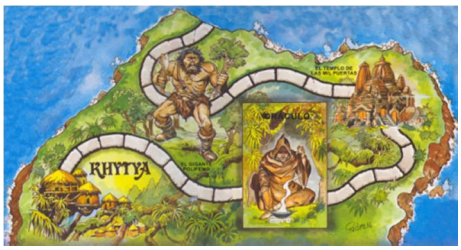
Para la edición de gráficos se ha usado la herramienta RGAS (Retro Game Asset Studio) incluida con cpctelera. Se trata de un cómodo editor de cuadrillas de pixels, que está preparado para Amstrad CPC y cpctelera. Permite ver directamente la paleta de colores del CPC, usar varios modos gráficos y finalmente exportar a código directamente usable en cpctelera, además de otras funcionalidades muy cómodas.

Para los **tableros**, he cogido el tablero original del juego de mesa (escaneado por www.imprimiryjugar.com) y lo he dividido en tres partes, uno por cada mundo. Posteriormente he editado con GIMP las imágenes para eliminar partes que no me serán de utilidad, como por ejemplo lo que había dibujado en la zona del mar.

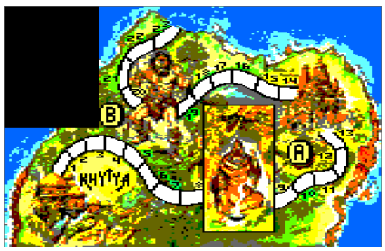
Una vez que tenía la imagen definitiva en formato png, es la hora de pasarlo a RGAS. Primero debo seleccionar qué 16 colores CPC (de los 27 existente) se adaptan mejor a la imagen a importar, pues cada tablero tendrá su propia paleta (aunque habrá unos colores comunes). A la hora de importar, RGAS permite configurar unos parámetros en la importación y ver directamente cómo será el resultado final. Por eso he preferido utilizar esta herramienta a la opción de image_conversion.mk de cpctelera.

A la hora de crear el nuevo gráfico, hay que indicar que se trata de un *loading screen* para que me meta los *spare video bytes*[1] en el archivo resultante, pues a la hora de descomprimir estos tableros, lo haré directamente sobre la memoria de vídeo.

Como la importación añade mucho ruido, se hizo un trabajo manual de limpieza del "ruido" (zonas con pixeles de muchos colores diferentes) que además no terminaban de definir la imagen. Así se conseguía mayor claridad del tablero y que luego la compresión fuese más efectiva.



Tablero Original



Importado por RGAS

Rediseño manual

Una vez tengo el array de datos que me genera RGAS, tengo que pasarlo a binario, es decir dejar un fichero donde sólo estén los bytes, para ello cojo la declaración en C del tablero y le quito todos los caracteres que no sean Bytes: comas, llaves, saltos de línea, tabulación, el prefijo 0x, … quedando sólo los bytes. El editor Sublime me permite guardar ese texto a un fichero hexadecimal (previamente hay que pasar las letras a minúsculas; sublime lleva una macro para hacerlo automáticamente)

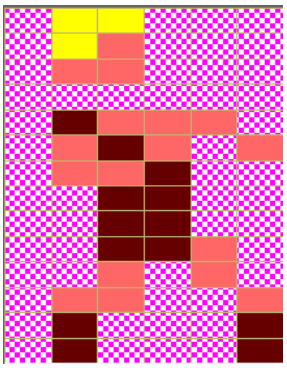
[1] La memoria de vídeo reserva 16KBytes (de C000 a FFFF), pero realmente la resolución es 80 Bytes * 200 filas = 16000, por lo que hay 384 Bytes que no se usan y no tienen reflejo en pantalla.

```
const unsigned char G_Vendha[16000] = {
        0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
        0x00, 0x00, 0x00, 0x00, 0x03, 0x03, 0x03, 0x03, 0x03, 0x03, 0x03, 0x03, 0x03, 0x03, 0x03, 0x03,
        0x03, 0x03, 0x03, 0x03, 0x03, 0x03, 0x03, 0x03, 0x03, 0x03, 0x03, 0x03, 0x03, 0x03, 0x03, 0x03,
```

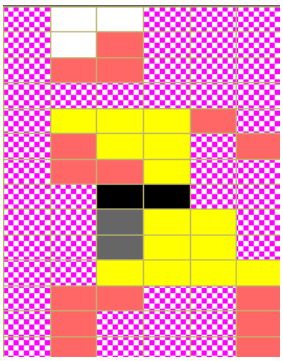
```
0000 0000 0000 0000 0000 0000 0000 0000
0000 0000 0303 0303 0303 0303 0303 0303
0303 0303 0303 0303 0303 0303 0303 0303
```

Para el **menú inicial**, se han usado gráficos en los tres modos (0, 1 y 2). Quería experimentar con el *split screen* en una parte del juego que sea casi estática. Para los gráficos del menú he usado algunas importaciones de material original del juego, y un mosaico propio para albergar las opciones de inicio.

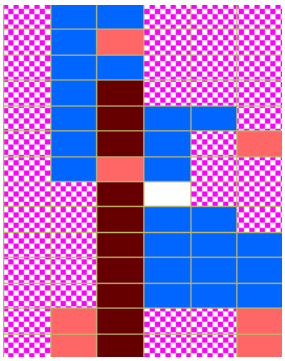
Para el **juego**, son gráficos propios diseñados en modo 0. Uso el propio editor RGAS para su composición. Algunos sprites usarán máscara, que el propio RGAS permite definir. Estos sprites ocuparán el doble de tamaño para guardar la información de la máscara, pero luego se podrán pintar encima de un fondo. Sólo unos pocos sprites necesitarán este comportamiento, quedando el resto como sprites normales sin máscara. Como había espacio suficiente, he diseñado un sprite personalizado para cada uno de los 3 jugadores.



Jugador de Khytya



Jugador de Vendha



Jugador de Hyrca

Música

Las músicas y efectos que acompañan al juegos están compuestas con la herramienta Arkos Tracker v1, incluido en cpctelera. Los instrumentos usados parten mayoritariamente de la propia carpeta de datos que incluye la herramienta, algunos con ligeras modificaciones.

Hay una composición simple por cada tablero, inspirada según el mundo: percusión (**Khytya**), escala pentatónica egipcia (**Vendha**) y escala pentatónica oriental (**Hyrca**). La tonadilla sonará según sea el tablero que esté en pantalla, creándose así una composición orgánica que va fluyendo de una melodía a otra según estemos jugando, resultando siempre diferente en cada partida. El tema principal se usa en los desafíos, e intenta ser una composición donde se funden los 3 mundos, reservando uno de los tres canales para cada uno de estos tres mundos, y donde cada uno de ellos tiene un momento destacado durante la melodía.

Para hacer sonar las melodías uso el habitual 'truco' de parchear las interrupciones del Amstrad CPC para reproducir la siguiente sección de melodía a 50Hz. Centralizo los cambios de melodías en un procedimiento genérico para que sea fácil el cambio, bastará con cambiar el valor a variable global para que la reproducción pase de una melodía a otra.

Para los **FXs** he vuelto a optar por cargar los instrumentos directamente en memoria de manera manual como en el proyecto anterior. Desconozco si en cpctelera 1.5 ya se puede hacer de otra forma más sencilla usando los procedimientos disponibles. Para cargarlo de manera manual, hay que crear un fichero Arkos Tracker (.aks) con sólo los instrumentos, y exportarlo como binario indicando la dirección de memoria donde luego lo situaremos. El binario lo pasamos a Array para lenguaje C y lo incluimos en nuestro código fuente.

Textos

cpctelera v1.5 tiene bastante optimizado con respecto a su versión anterior el uso de mostrar texto en pantalla, sin embargo he decido usar procedimientos propios, sobre todo para poder definir una fuente de caracteres propia que tenga un ancho de sólo 2 Bytes (4 pixels) a diferencia de los 4 bytes (8 pixels) por carácter de los caracteres propios de AMSTRAD cuando se usa el modo de vídeo 0. Así disponía de un ancho de 40 caracteres por línea en vez de 20.

Defino sólo los caracteres que usaré porque ocuparán zona de memoria, no todo el alfabeto extendido. Y tengo un procedimiento que pintará lo solicitado en pantalla en la posición que indica un puntero global. También tengo la opción de ir mostrando el texto poco a poco (efecto teletipo), o de golpe.

Optimización de la memoria

De los 64KB inicialmente disponibles en un AMSTRAD CPC 464, 16KB de ellos están reservados para la memoria de vídeo, por lo que sólo disponemos de 48 KB reales para nuestro proyecto. Sin embargo, este número también se ve reducido por el sistema de carga del juego. En la parte alta de la memoria (a partir de #A6FC) reside el **firmware** ("sistema operativo"). El cuál es necesario para cargar el programa a la memoria tras encender el ordenador, por lo que nuestro proyecto no puede contener datos o código más allá de esa dirección; si no, nuestro programa no se cargará al sobrescribirse código del firmware. Lo que nos limita a sólo 42KB aproximadamente.

Sin embargo, hay maneras de solventar esta limitación y usar el máximo de la memoria. La más rápida, es usar un cargador propio que no tenga que usar el firmware y por tanto liberar esa zona de memoria. La versión v1.5 ya incluye algo similar, pero sólo para el formato cinta (cassette). Dado que quería desarrollar el juego tanto en cinta como en disco, opté por otra solución más enrevesada, pero que implica menos conocimientos técnicos por mi parte.

Mi solución consiste en hacer **dos cargas** antes de empezar el juego. El código generado por cpctelera lo tendré limitado a #A6FC (realmente #A6FA porque usaré dos Bytes que se explicarán más adelante). Por otro lado, tendré un fichero binario extra de 6KBytes donde guardo datos estáticos como gráficos, texto y la melodía principal. El ciclo de ejecución sería el siguiente:

1. Cargar el programa principal de manera normal (como es menor que #A6FC, no hay colisión con el firmware)
2. Cargar el fichero extra de datos en la memoria de vídeo (saldrán caracteres raros en pantalla pero así puedo volcar toda la información en un lugar temporal que no se modificará)
3. Desactivar el firmware para liberar todo el contenido a partir de #A6FC, ya no lo necesitaré más.
4. Copiar lo que he dejado temporalmente en memoria de vídeo (gráficos, texto y melodía principal) a la parte de memoria liberada (direcciones referenciadas previamente en main.h)

De esta manera sencilla, aunque no elegante (pues se ven momentáneamente pixeles aleatorios en pantalla), puedo usar toda la memoria disponible, muy necesaria dada la limitación. Para mayor optimización de la memoria, realmente existen 2 archivos de datos extra, uno por cada idioma (inglés y español), aunque realmente sólo difieren en los primeros 1024 bytes que es donde está almacenada la información de texto.

Mapa de memoria

Aquí pongo un resumen del mapa de memoria que he usado para tener una instantánea rápida de cómo está organizado el juego.

#0000-#003F: No usada (Reservada para el sistema)

#0040-#010C: Instrumentos para FXs

#010D-#0478: Melodías de cada mundo

#0479-#0481: Nombre del fichero de datos extra (data1.bin o data2.bin)

#0482-#A6DA: Código y constantes generado por cpctelera

#A6FA-#A88F: Variables globales (que no se necesitan antes de la carga de datos extra)

#A890-#AC8F: Textos que se mostrarán en el juego

#AC90-#BD6F: Gráficos

#BD70-#BF10: Melodía principal

…-#BFFF: Pila (va en sentido decreciente)

#C000-#FFFF: Memoria de vídeo

La pila del sistema empieza en #BFFF y va hacia atrás, por lo que puede llegar a escribir en una dirección que tenga yo usada, es por eso que he decidido poner al final la melodía principal. En caso de colisión de la pila, se corromperían datos de la melodía principal y sonaría raro pero no afectaría al juego ni lo dejaría parado. En las pruebas realizadas nunca ha llegado a producirse esta colisión.

Cargador

El juego se cargará con el comando habitual RUN desde el BASIC que aparece tras encender el ordenador. Para ello hay que escribir un pequeño **programa BASIC** que haga las cargas necesarias de los binarios. Esta es la manera más sencilla para poner una pantalla de carga que amenice la carga posterior del código.

Para este proyecto he creado un cargador distinto según versión disco o cinta. La versión **disco** ejecutará una asignación aleatoria para que la pantalla de carga a mostrar sea una de las 3 incluidas. Anteriormente a la carga de pantalla, hay que cambiar el modo gráfico y establecer la paleta de color correspondiente. Posteriormente se cargará el programa principal. La versión **cinta** siempre carga la misma pantalla de carga. Al ser un dispositivo secuencial, penalizaría mucho tener que cargar una pantalla aleatoria, pues tendría que pasar por el proceso de lectura de las otras dos aunque no las pinte en pantalla.

Este cargador también marca en un byte de memoria el dispositivo con el que hemos cargado el juego (cinta o disco), para tenerlo en cuenta posteriormente en la carga de datos extra. Después, inicializará otro byte para que el jugador seleccione el idioma. Cuyo uso será necesario posteriormente.

Splash Screen

En la versión disco también incluí una pequeña "splash screen" con el logo de cpctelera. Esto me vino motivado para estudiar cómo poder concatenar varios programas binarios en la carga del juego y así poder hacer introducciones más trabajadas en proyectos futuros (como en Dragon Attack [Bitplane Technomantes, 2016] donde se va contando la historia en varias pantallas durante la carga del juego). En este caso se podría hacer uso de todos los recursos de la máquina, porque una vez cargadas estas introducciones se libera toda la memoria usada y se puede empezar a cargar el código propio del juego.

Para ello hay que crear un nuevo proyecto cpctelera por cada programa binario que queramos cargar, en esa parte no hay misterio alguno. En mi caso es reproducir una pequeña melodía y pintar el logo (importado a partir de un fichero png usando la herramienta image_conversion.mk). Donde he tenido que investigar es a la hora de cargar el fichero resultante en mitad de mi proyecto. Esto es porque si pongo directamente en mi cargador BASIC *RUN"splash.bin"*, cargará mi pantalla con el logo de cpctelera pero luego se reseteará la máquina cortándose la ejecución en ese punto. Para evitar esto hay que cargar el programa intermedio en memoria con **LOAD** y luego lanzar la ejecución manualmente (**CALL**); así al terminar seguirá nuestro programa BASIC cargador en vez del reset:

MEMORY &4000 (Liberar zona de memoria para cargar el código del programa intermedio)
LOAD"!SPLASH.BIN",&4000 (Cargar el programa a partir de 0x4000, que es la dirección de inicio para el proyecto splash)
CALL &44F3 (Ejecutar el programa splash llamando a la dirección de memoria donde está el procedimiento main();se puede averiguar estudiando los ficheros resultantes del ensamblado, como por ejemplo el .map)

Esta pantalla "splash" sólo está incluida en la versión disco del juego. Esto es porque el fichero binario ocupa hasta 5KBytes. Para una carga disco no supone más de un par de segundos, pero para la versión cinta supondría bastante más tiempo y no tendría sentido alargar innecesariamente la ya de por si extensa carga del juego.

Pantallas de Carga

Para la ocasión estuve experimentando con 3 pantallas de carga, cada una de ellas en un modo gráfico. Como no conseguía decidirme por ninguna, busqué la manera de incluir las 3 y que se cargase una de ellas de manera aleatoria cada vez.

Las pantallas de carga están mayormente compuestas usando la utilidad de importar archivo de imagen de RGAS sobre material original escaneado del juego de mesa, con una pequeña edición posterior para limpiar zonas o incluir logo (usando también RGAS).

Incluir las pantallas de carga en el dsk o cdt resultante es sencillo. De manera similar a la anteriormente descrita para los tableros, obtendremos un fichero binario usando RGAS y el editor de textos, sólo queda incluirlos en la carpeta del proyecto dsk. El propio *make* de cpctelera incluirá esos ficheros al dsk final. Para el cdt es necesario realizar unas instrucciones de manera manual, pero resulta muy sencillo usando otra de las herramientas también incluidas en cpctelera: cpc2cdt

Modo 0 (160x200, 16 colores)

Modo 1 (320x200, 4 colores)

Modo 2 (640x200, 2 colores)

Menú Inicial

Tras la carga del juego principal se muestra una pantalla minimalista para seleccionar idioma. 1 para Inglés; 2 para Español; Cualquier otra tecla pulsada (o joystick) tomará como selección por defecto: el inglés. Una vez seleccionado se procede a la carga de datos extra y aparece el menú inicial.

Lo más destacable del menú es el uso de "***split screen***" donde conviven a la vez los 3 modos gráficos del AMSTRAD CPC mientras suena la melodía inicial. En la parte superior está el logo en modo2; en el medio, una figura de la cobra en modo 1 (4 colores); en la parte inferior aparece un marco en modo 0 junto con las opciones de inicio:

Jugador 1 (Rojo oscuro): Humano u ordenador
Jugador 2 (Amarillo): Humano, ordenador o deshabilitado
Jugador 3 (Azul): Humano, ordenador o deshabilitado
Empezar

Inicialización

Cada vez que comencemos una nueva partida es necesario inicializar una serie de variables para que el juego siempre empiece en una posición fija inicial. Aprovecho para ir comentando las variables globales más relevantes

- Valores de cada jugador, como el mundo y casilla donde se encuentran (hay que situar cada jugador en su mundo en la primera casilla), el sprite que llevan asociado y si ya han visitado el oráculo (es donde se asignan las misiones). Una tabla de 3 elementos (tipo TJugador) contendrá los valores de cada jugador
- Nº de turnos hasta que el juego termine por timeout (valor = 32)
- Inicializar modo_dificil a 0. Es un 'flag' para hacer los mini juegos un poco más difíciles cuando llevemos la mitad de la partida.
- Resetear el display donde aparecen las misiones que tiene que hacer cada jugador (inicialmente mostrará que se vaya al oráculo)
- Situar las armas de manera aleatoria por los tableros (2 en cada mundo para que sea lo más equitativo posible dentro de la aleatoriedad). Hay algunas casillas que se marcan como no disponibles para contener armas por temas de jugabilidad.

Ciclo de un turno

1. Cada 4 turnos actualizar el dibujo del **reloj de arena** (hay 8 posiciones diferentes)
2. Pintar el **tablero** donde está el jugador que le toca turno. Esto se hace descomprimiendo el tablero directamente a la memoria de vídeo (#C000). Posteriormente se pintarán las armas (si ya se desbloqueó su misión correspondiente) y las ficha de los jugadores en ese tablero (El jugador actual se pinta el último para que luego se restaure el fondo correctamente). Antes siempre hay que cambiar a la paleta de colores específica de cada tablero.
3. Si el jugador está en la última casilla del tablero, podrá viajar a otro tablero. Si falla en el **viaje**, habrá terminado su turno. Si viaja con éxito o decide quedarse en ese mundo, seguirá su turno.
4. Aparecerá uno de los **desafíos** de manera aleatoria (Para controlar que no se repitan los desafíos de manera descontrolada, se inicializa una tabla con los ID de los desafíos y se "barajan". Cada vez que se hayan jugado todos ellos, se vuelve a barajar. Así no dejamos la elección del desafío a una función puramente aleatoria que podría mostrar el mismo desafío varias veces o incluso dejar sin mostrar alguno durante una partida).

   Si el jugador es el ordenador, se le asignará desafío completado o no según una probabilidad del 75%. Si ocurre ese 25% donde NO va a tirar los dados, sonará el efecto de sonido de cuando perdemos una vida, para resaltar que no se moverá. También se pinta el borde de pantalla en verde o rojo, según se haya superado el desafío o no.

   Si un jugador falla dos veces consecutivas un desafío, la tercera vez tirará directamente los dados. Es una medida para no frustrar al jugador y pueda avanzar a pesar de fallar los desafíos.

   Si hemos superado con éxito el desafío propuesto, tiraremos el dado y procederemos a elegir a qué casilla queremos ir. Si el jugador es el ordenador tomará la decisión de avanzar o retroceder en base a una función de IA:

   a. Si no ha visitado aún el oráculo para que le asigne las misiones => avanzar
   b. Averiguar el objetivo más cercano. Esto puede ser una de las armas que necesite o uno de las localizaciones donde completar la misión si ya tiene esa arma.
   c. Se moverá al objetivo más cercano

5. Si superamos la casilla 8 y no habíamos visitado el **oráculo**, se muestran las misiones asignadas al jugador (realmente se asignan al principio de la partida por simplicidad, pero permanecen ocultas hasta este momento)
6. Se va pintando la ficha del **jugador**, casilla a casilla. Para esto, previamente guardo el fondo donde se pinta la ficha para restaurarlo según se va moviendo.
7. Si el jugador ya ha visitado el oráculo y pasa por encima de una casilla donde hay un arma que necesita, la recoge. Si tiene el arma y pasa por una localización final, se marca la misión como completada. También salta un efecto de **borde multicolor** para resaltar este hecho, así como un efecto sonoro.
8. Se actualiza el **display** del jugador para mostrar las armas recogidas o misiones completadas

Al acabar y pasar el turno al siguiente jugador, se ejecuta un efecto básico de **CRTC**, que consiste en reducir el número de filas y columnas que se muestran, para simular un efecto de plegado y desplegado con el fin de resaltar el cambio de turno.

Fin del juego

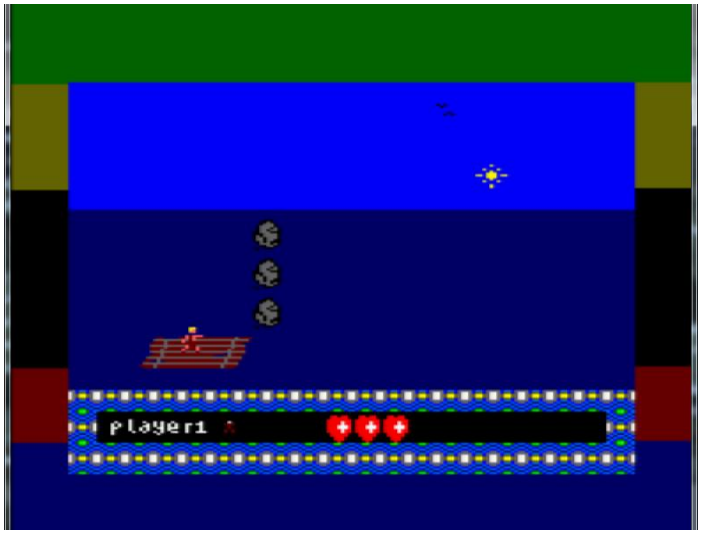
El juego acabará después de que el jugador haya completado las dos misiones y haya superado el **desafío final**. Si el jugador es el ordenador, bastará con que complete las dos misiones, la misión final se le dará como completada a la primera. Si se completan 32 turnos sin que ningún jugador haya completado el juego, la partida terminará por tiempo.

Viaje entre mundos

Para viajar entre un tablero y otro hay que completar un mini juego en el que el jugador está sobre una balsa y durante un tiempo limitado debe evitar las piedras que hay en el mar. Aunque es un planteamiento simple fue la parte donde más complicaciones surgieron en el proceso de **renderizado** (pintar en pantalla).

El principal problema es que hay varios sprites que dibujar y borrar en pantalla, y además de distinto tipo. En el peor momento del rederizado (cuando hay que pintar 3 piedras), el personaje del jugador sobre la balsa aparecía a menudo cortado, no terminaba de pintarse al completo, sobre todo si estaba en la posición más alta. Esto es porque cuando se ejecutaba las instrucciones de pintado del jugador, el CRTC ya había pasado por esa parte de la pantalla y no lo muestra. Para solucionarlo primero estudié el tiempo de procesamiento de cada sección con el conocido procedimiento de cambiar el borde de color para ver visualmente cuánto tarda cada parte del código.

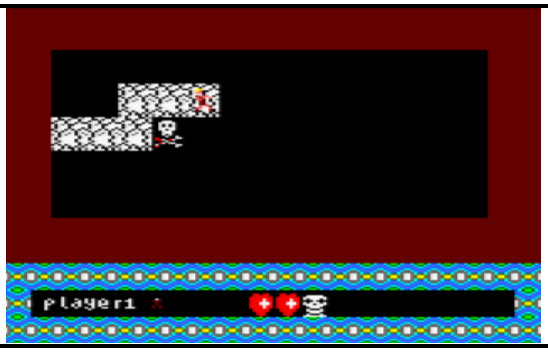
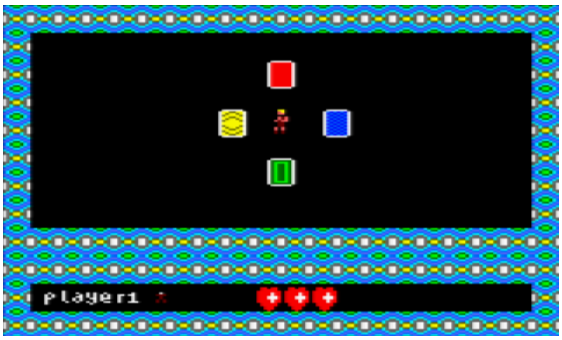
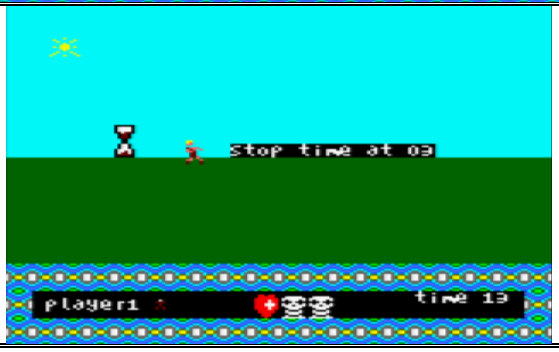
Primero descubrí que no era posible hacer todo el renderizado desde que sincronizo el VSync hasta que el CRTC llega al primer pixel de pantalla. Por lo que la solución pasaba por reordenar el código para renderizar los elementos teniendo en cuenta dónde está el CRTC. También noté que el proceso de dibujado de las piedras era variable según el nº de piedras a pintar (algo bastante lógico), por lo que si quería jugar con ejecutar código con respecto a dónde estaba el CRTC tenía que volver todo el proceso más o menos **constante** en su dibujado. Para ello modifico la parte del dibujado de piedras; si son menos de 3, dibujo un cuadrado desechable en otro lugar de la pantalla y así el tiempo de pintado se vuelve constante independientemente del nº de piedras que luego se ven en pantalla. En la ilustración adjunta se puede ver cómo logre sincronizar el proceso de renderizado para evitar *flickering* o *tearing*:



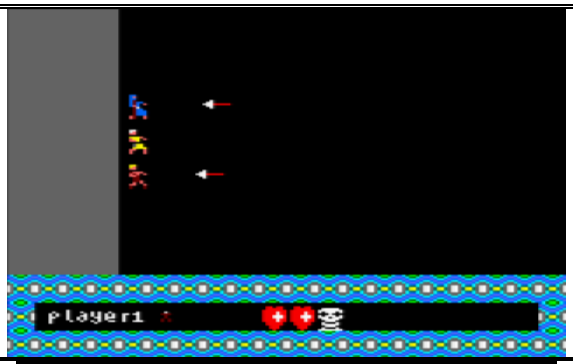
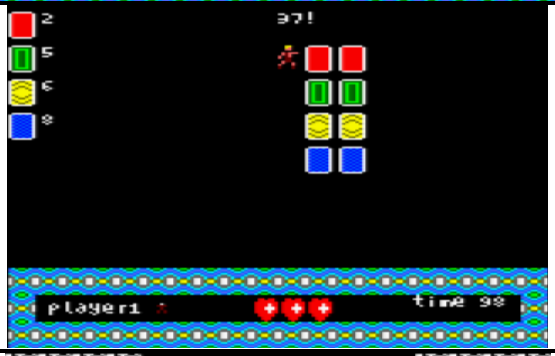
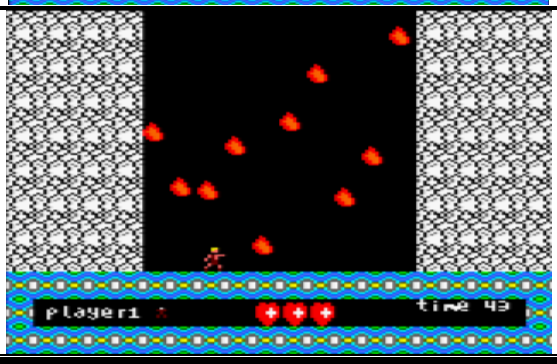
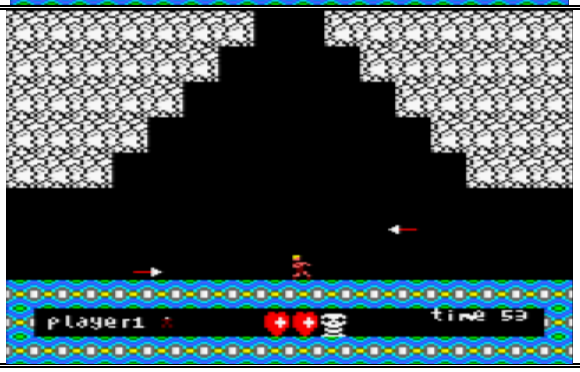
- Mientras el CRTC está en la parte **verde**, escribo en la memoria de vídeo las piedras. Como se pintan en la parte media de la pantalla no importará que esté el CRTC arriba.
- Mientras el CRTC está en la parte **amarilla**, escribo en la memoria de vídeo la balsa y el jugador. Coincide con el cielo y termina justo donde empieza el mar que es su zona de visualización. Es decir cuando el CRTC llegue a la zona del mar la información de la balsa y del jugador acaba de ser actualizada.
- La parte **negra** es la parte que nos sobra de toda la pasada del CRTC o mientras se hacen las actualizaciones de posición o control de colisiones. Ahí no se pinta nada.

- La parte **roja** es la encargada de actualizar la información de borrado y pintado del sol y los pájaros. Y también borrar la balsa. Como se ve en la ilustración, el CRTC ya está pintando el marcador por lo que no le afectan estos cambios.
- Finalmente la parte **azul** es para actualizar la información del borrado de las piedras (su posición anterior). Ya se produce cuando el CRTC está prácticamente fuera de la pantalla de juego.

Desafíos

DESAFIO	PANTALLA	DESCRIPCION	MODO DIFICIL[2]
1. Camino oculto		Se muestra un camino lleno de trampas y luego desaparece, el jugador debe encontrar la salida de memoria evitando las trampas.	El camino se muestra sólo la mitad de tiempo
2. Repetir patrón		Como el juego del Simon, hay que repetir la melodía. Tiene la ayuda tanto de distintos colores como de distintos sonidos, como el original. Los botones se han rellenado con un entramado diferente por si se juega en monocromo.	Las notas se reproducen más rápidas
3. Supervivencia		Hay que evitar las calaveras hasta que se agote el tiempo.	Las dos calaveras se moverán al doble de velocidad en vez de sólo una.
4. Parar tiempo		Hay que parar el temporizador en el número que nos indica en el mensaje	Empieza el contador desde 30 en vez de 40, dando menos tiempo de reacción al jugador

[2] El modo difícil se activa a mitad de la partida (cuando llevemos 16 turnos)

5. Suerte		Una prueba de suerte, hay que pulsar una tecla y el personaje que queda en el centro debe ser el del jugador actual	Este desafío no tiene diferencia en modo difícil
6. Suma		Hay que sumar la cifra indicada usando los botones de colores, cada botón de color tiene asignado un valor.	El temporizador empieza desde 70 en vez de 99
7. Lluvia Ácida		Esquivar las gotas que van cayendo hasta que se acabe el temporizador.	El temporizador empieza desde 99 en vez de 60
8. Tiro al Arco		Disparar a la calavera que aparecerá al fondo	Las calaveras no aparecen en los bordes, sino un poco después. Dando menos tiempo de reacción al jugador
9. Saltar Lanzas		Hay que esquivar las lanzas que aparecen a cada lado (es posible saltar)	El temporizador empieza desde 99 en vez de 60

Función Escalado de *Sprites*

Después de los nuevos tableros retocados manualmente, la compresión de los mismo mejoró muchísimo dejando bastante espacio de memoria para añadir nuevos desafíos y la posibilidad de probar una función que fuese capaz de aumentar los sprites como si una lupa se aplicase sobre ellos, es decir, similar a la técnica de **sprite-scaling** que se usó en juegos como Out Run, Space Harrier, …

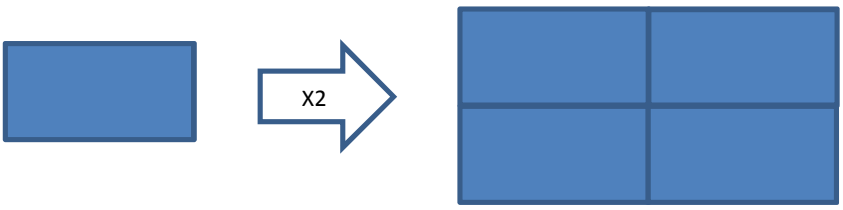
Aquí lo he aprovechado para el dado, es un sprite de 8x14 pixels, pero cuando termina de girar se muestra con un aumento de x2 y x3 resultando en un sprite de hasta 24x42 pixels sin cambiar la definición del gráfico.

Para ello me construí una función que se encarga primero de calcular el centro de la imagen original (pues quería que el escalado se produjese sin descentrar el sprite). Una vez tengo el centro y el factor de escalado, puedo obtener la coordenada **x** e **y** desde donde tendré que empezar a dibujar el sprite escalado, así como su ancho y alto.

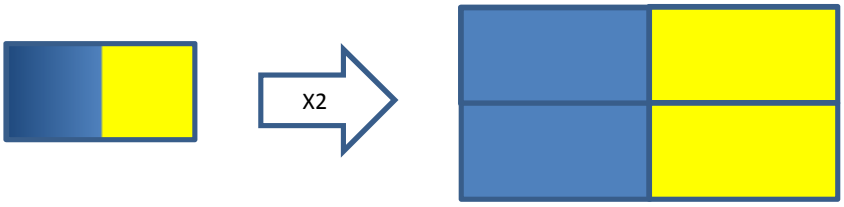
Recorreré byte a byte del sprite original (por líneas) aumentando su pixel en función del factor de escalado. Por ejemplo si es x2, cada byte gráfico del sprite original se converitrá en un cuadrado 2x2. Si es x3, un cuadrado 3x3, …

La complicación viene porque esta transformación no es directa. En primer lugar porque el sprite está **codificado** por bytes. Al estar en el modo gráfico 0, esto quiere decir que cada byte contiene la información del color de 2 pixels. Además esta codificación del color no es trivial, un color está codificado usando los bits pares y el otro los impares, y siguiendo además un orden no lineal. Luego tenemos el problema que en un factor de escalado impar hay que tener en cuenta qué color mostrar en el byte del medio. Ejemplos:

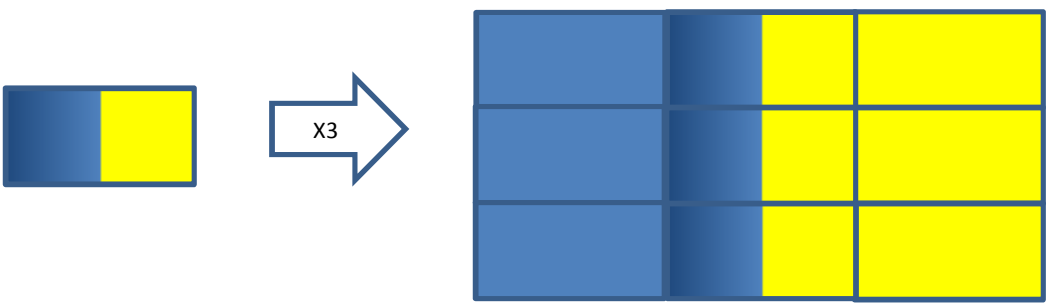
Si tenemos un byte con los colores iguales es sencillo, se transformará en un cuadrado factor x factor de ese color



Si tenemos un byte con 2 colores y un factor Par tampoco supone gran problemática, la mitad irá de un color y la otra del otro



Para un factor impar tendremos que dejar la columna central para un byte que mezcle ambos colores como en el original



Ensamblador

La mayoría del desarrollo está hecho en lenguaje C, pero hay unas pocas funciones que están escritas en ensamblador por los motivos que comento a continuación.

borrarPantalla: Sirve para poner la pantalla y borde en negro (color 0 de la paleta), es más corta que la que incluye cpctelera porque aquí no hay que indicar parámetros del color, siempre uso color negro.

cpct_mygetScreenPtr: en vez de usar el que viene por cpctelera, creo éste que no tiene el parámetro de indicar el comienzo de la memoria de vídeo. Dado que es fijo en todo el proyecto, me ahorro de enviar siempre el mismo parámetro.

initDrawChar: Para usar los drawString de cpctelera, primero hay que usar una función para inicializar los valores de la tinta y del fondo que usarán los caracteres. Como yo sólo lo necesito para indicar los colores una vez, puedo optimizarla para que sea más pequeña, poniendo unos valores fijos. (13 Bytes en vez de 84 Bytes del original)

APK

En el momento de terminar el juego aún no existía una versión definitiva para generar el apk directamente del proyecto cpctelera usando cygwin, pero sí se podía usar la rama de desarrollo del github y con unas pequeñas modificaciones conseguí generarlo.

- Modificar la variable ANDROID_PATH := ../cpctelera/tools/android/
- Usar versión java a 8 u101 o superior
- Modificar la variable ZIPALIGN := $(ANDROID_PATH)bin/zipalign/win/32/zipalign.exe

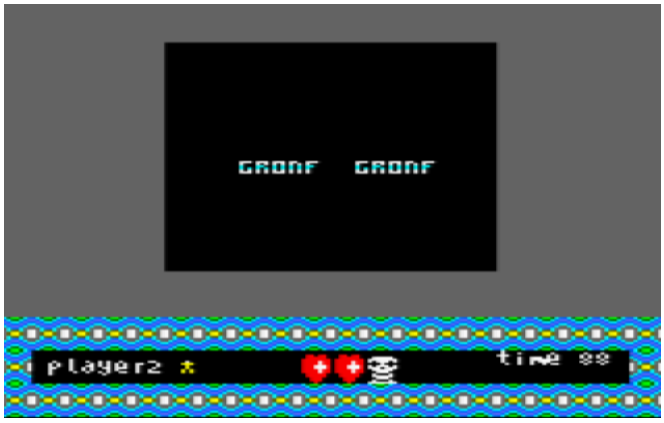
Para crear el apk es necesario un snapshot (.sna), es decir un volcado de memoria del juego. Dadas las características de mi proyecto no me vale el snapshot que crea cpctelera, pues es necesario esa carga extra de datos con los gráficos, textos y música principal. Por tanto, necesito crear un .sna de manera manual; para ello cargo el juego en el emulador RetroVirtualMachine y uso la función de Save State, sobreescribiendo el fichero de cpctelera. Ahora ya está listo para ejecutar el script de creación del apk.

Adicionalmente modifico el fichero de configuración del layout del gamepad virtual (.\exp\android\assets\conf.json) para que sólo muestre un botón de disparo en vez de los cuatro por defecto, pues sólo es necesario uno. También cambio los iconos asociados a la aplicación (.\exp\android\res\mipmap-*)

Extra: Astro Marine Corps

Para la presente edición de la cpcretrodev 2019 se invita a hacer un guiño al momento en que la planta carnívora escondida te mataba en el A.M.C. de CreepSoft por su 30º aniversario.

En el desafío de Supervivencia, cuando perdemos una vida al colisionar con una calavera aparecerá el mensaje “Gronf Gronf” con la misma fuente de texto al igual que sucedía cuando te comía dicha planta en el Astro Marine Corps.



Gracias por su interés en leer este documento