

Documentación de MTE MK1

En este directorio está disponible la documentación existente sobre **MTE MK1**. Al contenido de este directorio podríamos añadir los *post mortem* de los ejemplos y contribuciones, que enlazamos al final.

El tutorial

- [Capítulo 1: Introducción](#)
- [Capítulo 2: Tileset](#)
- [Capítulo 3: Mapas](#)
- [Capítulo 4: Sprites](#)
- [Capítulo 5: Terminando los gráficos](#)
- [Capítulo 6: Colocando cosas](#)
- [Capítulo 7: Primer montaje](#)
- [Capítulo 8: Empezando con el scripting](#)
- [Capítulo 9: Scripting básico](#)
- [Capítulo 10: Música y FX \(48K\)](#)
- [Capítulo 11: Code Injection Points](#)
- [Capítulo 12: Juegos multinivel \(48K\)](#)
- [Capítulo 13: Un juego de 128K](#)
- [Capítulo 14: Sonido 128K](#)
- [Capítulo 15: Custom Vertical Engine](#)

Referencias

- [API de MTE MK1](#)
- [Sistema de scripting MSC3](#)
- [Code Injection Points](#)
- [Herramientas de MTE MK1](#)

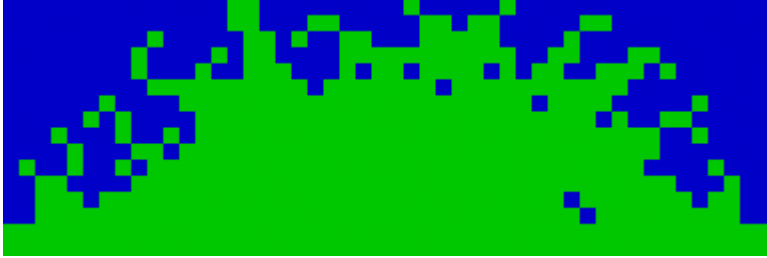
Post Mortems

- [Godkiller 2](#)
- [Sami Troid](#)
- [Cheril the Goddess](#)
- [Sgt. Helmet's Training Day](#)

Pronto más.

Capítulo 1: Introducción

¿Pero qué seto?



Eso digo yo ¿Pero qué seto? Ufff. Hay tanto que decir, y tan poco espacio. Podría tirarme horas charlando y diciendo chorradas, pero intentaré no hacerlo. Me han dicho que tengo que ser claro y conciso y, aunque me cueste, trataré de serlo.

Empecemos por el principio. En realidad un poquito más adelante, que hay mucha gresca entre creacionistas y evolucionistas. Vayámonos a 2010. A principios de ese año tuvimos una idea en Mojonía. Básicamente estábamos hartos de copiar y pegar a la hora de hacer güegos. Porque, a ver, no todo es cuestión de copiar y pegar, pero bien es cierto que cantidad de cosas siempre se hacen igual, cambiando varios parámetros. A ver, si te piensas que los que hacemos güegos escribimos la misma rutina de pintar la pantalla con tiles cada vez que hacemos un güego... po no. También estábamos hartos del arduo trabajo manual. Que si pasar a mano los sprites al formato de la splib2, que si ordenar a mano los tiles para que SevenUP los cogiese en el orden correcto, que si pasar el mapa, que si colocar enemigos con una hoja de cuadritos... Había mil quehaceres a la hora de hacer güegos que resultaban tediosos y aburridos. ¿Y quién quiere aburrirse mientras hace algo que supuestamente le gusta? Nosotros no. Y tú tampoco, supongo.

Ya lo sé. Que eso de diseñar el güego y hacer las cosas en papel milimetrado es muy de los 80 y tal pero, a ver, en serio, es un coñazo. Uno puede ser friki, pero masoquista no.

Se nos ocurrió que lo que necesitábamos era un framework, que le llaman ahora, que nos permitiese disponer de los módulos de código que queríamos utilizar de forma sencilla, y que nos hiciese llevadero todo el tema de la conversión e integración de datos (gráficos, mapas, posicionamiento de enemigos y objetos...). Empezamos tímidamente escribiendo las utilidades de conversión, para luego ir levantando, usando trocitos de aquí y de allá, un engine que sirviese como base.

Teníamos un montón de ideas para el engine. Podríamos habernos puesto a desarrollarlo poco a poco y luego sacar el güego definitivo, pero nosotros no funcionamos así. Como se nos iban ocurriendo paranoias, íbamos sacando güegos cada vez que le metíamos una cosa nueva al engine. Así, en cuanto estuvo listo el "mínimo operativo", lo estrenamos con **Lala the Magical**, **Cheril of the Bosque**, **Sir Ababol** y **Viaje al Centro de la Napia**.

Como la gracieta, dentro de la retroescena, era que nosotros hacíamos juegos “como churros” (¡pero qué churros, señora!), decidimos llamarle al sistema *MTE MK1 (Mojon Twins Engine MK1)***, o la **Churrera**. Y así empezó todo... Mala idea, por cierto, porque hay gente a la que por lo visto le ha sentado muy mal que hagamos un motor reutilizable y usen el nombre de forma muy despectiva.

Pero entonces ¿qué es exactamente MTE MK1?

Pues eso mismo: **MTE MK1** es un framework que se compone de varias cosas muy chulas:

1. El *engine*, o *motor*, el corazón de **MTE MK1**. Se trata de un pifostio de código bestial que se “gobierna” mediante un archivo principal llamado “config.h” en el que decimos qué partes del motor usaremos en nuestro güego y cómo se comportarán.
2. Las utilidades de conversión, que nos permiten diseñar nuestro güego en nuestros editores preferidos y, de forma incolora, inodora e insípida, meter todos esos datos en nuestro güego.
3. Un montón de monos, imprescindibles para cualquier cosa que se quiera hacer en condiciones.
4. Tortilla.

MTE MK1 ha tenido muchas versiones a lo largo de los últimos tres años. Desde 2010 hasta principios de 2013 fuimos evolucionando hasta llegar a la versión 4.7, pero el pifostio se nos lió tanto que no era, para nada, presentable. Cuando por aquella época se nos ocurrió hacer un tutorial decidimos irnos para atrás un poco, a un punto del pasado en el que el tema era aún manejable: la versión 3.1 (**Trabajo Basura, Zombie Calavera Prologue**).

Durante un par de meses nos dedicamos exclusivamente a coger la versión 3.1, corregirle todas las cosas que estaban chungas, cambiar la mitad de los componentes para hacerlos más rápidos y más compactos, y añadir un montón de características. Así construimos las versiones **3.99**, que fueron las que pusimos a vuestra disposición y fuimos evolucionando poco a poco hasta que, de forma poco natural, mutó en **MK2** a mediados de 2014.

Años después, aprovechando el décimo aniversario de **MTE MK1**, hemos revisado la última versión, actualizado el *toolchain* para automatizar al máximo todo el proceso, e incorporado en el *engine* muchísimas mejoras propias de motores más modernos, para hacer un híbrido tan sencillo como **MTE MK1** original pero que rindiese como nuestros últimos productos.

Para ponerlos un ejemplo, el *proyecto por defecto* (que no es más que la versión beta de **Lalal Prologue** portada a **MTE MK1 v5.0**) ocupa 2.5Kb menos y corre un 20% más rápido (con cambio de pantallas instantáneo) que utilizando la versión 3.99 del motor. Además, grabada en un diskette da la consistencia necesaria al material magnético para que, lanzado en plan ninja, cercene limpiamente las cabezas de los enemigos.

Pero ¿qué se puede hacer con esto?

Pues un montón de cosas. A nosotros se nos han ocurrido ya un montón. Si bien es cierto que hay elementos comunes y ciertas limitaciones, muchas veces te puedes sacar de la manga una paranoia nueva solamente combinando de forma diferente los elementos que tienes a tu disposición. ¿Quieres ejemplos? Pues para eso mismo hemos acompañado el lanzamiento de la rama 3.99 con la **Mojon Twins Covertape #2**. Si todavía no la tienes, bájtela. AHORA.



Para la **Mojon Twins Covertape #2** lo que hicimos fue contratar a una tribu de indios pies-sucios (oriundos de la Jungla de Badajoz). A cada uno le escribimos una característica de **MTE MK1** en la espalda y otra en el pecho, y les animamos a descender por las colinas haciendo la croqueta. Cuando llegaban abajo hacíamos una foto y anotábamos las combinaciones. Con esas combinaciones hacíamos un güego. Luego llamábamos a Alberto, el Mono Tuerto, que el tío se inventa una historias de la leche, para que las justificase con un argumento convincente. Y funciona, en serio.

Vamos a echarle un vistazo a las cosas que tenemos.

1. **Valores:** Todos los valores relacionados con el movimiento del protagonista son modificables. Podemos hacer que salte más o menos, que caiga más despacio, que resbale más, que corra poco o mucho, y más cosas.
2. **Orientación:** Podemos hacer que nuestro güego se vea de lado o desde arriba (lo que, en mojonía, conocemos como "perspectiva genital"). Es lo primero que tendremos que decidir, porque esto condicionará todo el diseño del güego.
3. **¿Sartar? ¿Volar? ¿cómo?:** Si elegimos una perspectiva lateral (que el güego se vea de lado) tendremos que decidir cómo se moverá el muñeco. Podemos hacer que sarte cuando se pulse salto (**Lala Lah, Julifrustris, Journey to the Centre of the Nose, Dogmole Tuppowski...**), que sarte siempre (**Bootee**)... También podemos hacer que vuele (**Jetpaco**).
4. **Rebotar contra las paredes.** Si elegimos que nuestro güego tenga perspectiva genital, podemos hacer que el prota rebote un poco cuando se choque con una pared.
5. **Bloques especiales.** Podemos activar o desactivar llaves y cerrojos, o bloques que se pueden empujar. Esto funciona para ambas orientaciones, si bien en la vista lateral los bloques solo pueden empujarse lateralmente.

6. **Disparar.** También podemos hacer que el prota dispare. En los güegos de vista genital disparará en cualquiera de las cuatro direcciones principales. También podemos indicarle al motor qué dirección (vertical u horizontal) tiene preferencia en las diagonales. En los de vista lateral, disparará en un sentido u otro según mire a izquierda o derecha.
7. **Enemigos voladores:** que te persiguen sin tregua.
8. **Pinchos y cosas del escenario que te matan,** no necesariamente pinchos.
9. **Matar:** en los güegos de perspectiva lateral podemos hacer que los enemigos, un cierto tipo de enemigos, se mueran si les pisas la cabeza.
10. **Objetos coleccionables:** para que haya cosas que ir recopilando y guardándose en la buchaca.
11. **Scripting** Si lo de arriba no es suficiente, podemos inventarnos muchas más cosas usando el sencillo lenguaje de scripting incorporado.

Y más cosas que ahora mismo no recuerdo pero que irán surgiendo a medida que vayamos haciendo cosas.

El truco está en combinar estas cosas, echarle imaginación, timar un poco con los gráficos, y, en definitiva, ser un poco creativo. Por ejemplo, si ponemos una gravedad débil (que hará que el prota caiga muy lentamente) y habilitamos la capacidad de volar con muy poca aceleración, ponemos un fondo azul y el personaje tiene forma de buzo, podemos hacer como si estuviésemos debajo del agua. También se pueden probar cosas extremas, como por ejemplo poner los valores de aceleración vertical y de gravedad en negativo, con lo que el muñeco se vería empujado hacia arriba y haría fuerza para hundirse... cosa que, por cierto, no hemos probado nunca y que me acaba de dar una idea... En cuanto hable con Alberto y se invente un argumento tenemos güego nuevo.

¿Véis? ¡Así funciona! Mandadme un email y os doy el teléfono de Alberto.

Entonces ¿Cómo empezamos?

Con imaginación. No me vale que cojas un juego nuestro que ya esté hecho y le cambies cosas. No. Así no vas a llegar a ningún sitio. La gente se cree que sí, pero NO. Invéntate algo, empieza de cero, y lo vamos construyendo poco a poco.

Si no se te da bien dibujar, búscate a un amigo que sepa, que siempre hay. En serio, siempre hay. Si no encuentras a nadie, no pasa nada: puedes usar cualquier gráfico que hayamos hecho nosotros. En los paquetes de código fuente de todos los güegos están todos los gráficos en png y tal. Aprender a recortar y pegar con un editor de gráficos es un mínimo que deberás aprender.

De todos modos, para empezar, y siendo conscientes de que realmente no sabéis qué se puede y qué no se puede hacer, os invito a que vayamos construyendo, poco a poco, el juego **Dogmole Tuppowski**. ¿Por qué este? Pues porque usa un montón de cosas, incluyendo el scripting. Pero no quiero que vayas, te pilles el paquete de fuentes de la **Covertape #2**, y te limites a seguir el tutorial

mirando los archivos y tal. No. Lo suyo es que empieces con el paquete del engine vacío que os vamos a ofrecer más abajo, y que, para cada capítulo, vayas obteniendo los diferentes recursos y realizando las acciones necesarias, como si realmente estuvieses creando el juego desde cero.

¿Para qué tanto teatro? Pues porque cuando te quieras poner a hacer el tuyo ya no será la primera vez y, créeme, es toda una ventaja. Y porque el teatro mola. ¡Salen tetas!

Venga, va. Vamos a ello

Lo primero es inventarse una historia que apunte al gameplay. No vamos a escribir todavía una historia para el güego (porque ahora no la necesitamos) – eso vendrá dentro de poco. Primero vamos a decidir qué hay que hacer. Vamos a hacer un güego con **Dogmole Tuppowski**, un personaje que nos inventamos hace tiempo y que tiene esta pinta:



En primer lugar vamos a hacer un juego de perspectiva lateral, de plataformas, en el que el personaje salte. No saltará demasiado, pongamos que podrá cubrir una distancia de unos cuatro o cinco tiles horizontalmente y dos tiles verticalmente. Esto lo tenemos que decidir en este punto porque tendremos que diseñar el mapa y habrá que asegurarse de que el jugador podrá llegar a los sitios a los que decidamos que se puede llegar.

Vamos a hacer que en el juego haya que llevar a cabo dos misiones para poder terminarlo. Esto lo conseguiremos mediante scripting, que será algo que dejaremos para el final del desarrollo. Las dos

misiones serán sencillas y emplearán características automáticas del motor para que no haya que hacer un script demasiado complicado:

1. Habrá cierto tipo de enemigos a los que tendremos que eliminar. Una vez eliminados, tendremos acceso a la segunda misión, porque se eliminará un bloque de piedra en la pantalla que da acceso a una parte del mapa.
2. Habrá que llevar, uno a uno, objetos a la parte del mapa que se desbloquea con la primera misión.

Para justificar esto, explicaremos que los enemigos que hay que eliminar son unos brujos o monjes o algo así mágico que hacen un podewwwr que mantiene cerrada la parte del escenario donde hay que llevar los objetos. ¡La historia se nos escribe sola!

Por tanto, ya sabemos que tendremos que construir un güego de vista lateral, con saltos, que se pueda pisar a cierto tipo de enemigos y matarlos, que sólo se pueda llevar un objeto encima, y que necesitaremos scripting para que se pinte la piedra en la entrada del sitio donde hay que llevar los objetos si no hemos matado a los enemigos. Además, el hecho de llevar las cosas una a una para dejarlas en un sitio necesitará otro poquito de scripting también.

Como lo tenemos a huevo, nos inventamos la historia, que, si te leíste en su día la ficha del **Covertape #2**, ya conocerás:

Dogmole Tuppowski es el patrón de un esquife de lata que hace transportes de extraperlo de objetos raros y demás artefactos de dudosa procedencia (algunos dicen que con propiedades mágicas) para cierto departamento de la Universidad de Miskatonic (provincia de Badajoz). Sin embargo, una noche, justo cuando iba a hacer su entrega con cajas repletas de misteriosos y mágicos objetos, se levantó una tempestad del copetín que estrelló su barco contra un montón de pieras hostioneras con lo que el reventón fue de aúpa, y todas las cajas quedaron desperdigadas por la playa y alrededores.

La señorita Meemaide de Miskatonic, que se encontraba peinando sus Nancys a la luz de la luna llena en su torreón del acantilado, lo vio todo y, consciente de que el contenido de las cajas podría ser muypreciado por ocultistas y demás gente raruna, decidió llevárselas para ella. Como su camión estaba roto y reparándose y no lo tendría hasta el día siguiente, puso a sus esbirros de dudosa procedencia a guardar las cajas, y, por si acaso, también mandó a los veinte Brujetes de la Religión de Petete que hicieran un hechizo mental para cerrar las puertas de la Universidad.

La misión de Dogmole, por tanto, es doble: primero tiene que eliminar a los veinte Brujetes de la Religión de Petete y, una vez abierta la puerta de la Universidad, tendrá que buscar y llevar, una por una, cada una de las diez cajas al vestíbulo del edificio, donde deberá dejarlas en donde pone "BOXES" pulsando "A".

Y ya, con esto, podemos empezar a diseñar nuestro juego. En realidad el tema suele salir así. En nuestro a veces hacemos trampas y jugamos con ventaja: muchos de nuestros juegos han surgido porque hemos añadido una nueva capacidad a **MTE MK1** y había que probarla, como ocurrió con

Bootee, Balowwn, Zombie Calavera o Cheril the Goddess. El proceso creativo es insondable y requiere que tengas algo de inventiva e imaginación y eso, desgraciadamente, no es algo que se pueda enseñar.

Ah, que se me olvidaba. También hicimos un dibujo de la Meemaid. Es esta:



Metiendo la cara en el barro

Vamos a empezar a montar cosas. En primer lugar, necesitarás **z88dk**, que es un compilador de C, y **splib2**, que es la biblioteca de gráficos y demás E/S que usamos. Como no tenemos ganas de que te compliques instalando cosas (sobre todo porque la splib2 es muy vieja y es complicado compilarla usando un z88dk moderno, y porque la splib2 que usamos en el motor está modificada por Elías, el mono modificalibrerías), hemos preparado, para los usuarios de Windows, un paquete **z88dk10-stripped.zip** que encontrarás en el directorio `env` de este repositorio y que deberás descomprimir en C:.

También vamos a necesitar un editor de textos. Si eres programador ya tendrás uno que te guste de la hostia. Si no lo eres, por favor, no utilices el bloc de notas de Windows.

Nos hará falta el editor Mappy para hacer los mapas del juego. La versión mojona **mappy-mojon.zip**, que está modificada con las cosas que necesitamos y un par de características custom que vienen muy bien, también la hemos incluido en el directorio `env` de este repositorio.

Cuando lleguemos a la parte del sonido hablaremos de las utilidades Beepola y BeepFX. Puedes buscarlas y descargarlas ahora si quieres, pero no las utilizaremos hasta el final.

Otra cosa que necesitarás será un buen editor de gráficos para pintar los monitos y los cachitos de escenario. Te vale cualquiera que grabe en `.png`. Te reitero que si no sabes dibujar y no tienes ningún amigo que sepa puedes pillar los gráficos de Mojon Twins. Igualmente vas a necesitar un editor gráfico para recortar y pegar nuestros gráficos en los tuyos. Te vale cualquier cosa. Yo uso una versión super vieja de Photoshop o el genial Aseprite porque es a lo que estoy acostumbrado. Mucha gente usa Gimp. Hay un montón, elije el que más te guste. Pero que grabe `.png`. Remember. Se bebes no kandusikas.

Una vez que eso esté instalado y tal, necesitaremos **MTE MK1**. Descarga este repositorio a tu disco duro.

Probando que todo está bien instalado

Para probar que todo está correctamente instalado, abre una ventana de línea de comandos, muévete al directorio `src/dev`, y:

1. Ejecuta `setenv.bat` la primera vez, para establecer las variables de entorno que harán que **z88dk** esté encontrable. Si todo está donde debe estar, deberás obtener este mensaje en tu consola:

```
zcc - Frontend for the z88dk Cross-C Compiler
v2.59 (C) 16.4.2010 D.J.Morris
```

2. Ejecuta `compile.bat` para compilar el juego de ejemplo. Deberás ver el proceso en tu consola:

```
Compilando script
Convirtiendo mapa
Convirtiendo enemigos/hotspots
Importando GFX
Compilando juego
lala_beta.bin: 28566 bytes
Construyendo cinta
Limpiando
Hecho!
```

Esto generará el archivo `lala_beta.tap` que podrás ejecutar en tu emulador de Spectrum favorito.



Empezando un nuevo proyecto

1. Copia el contenido de `src` en un nuevo directorio con el nombre de tu juego.
2. Luego entra en `dev` y edita el archivo `compile.bat`. Al principio encontrarás una línea parecida a esta:

```
set game=lala_beta
```

Simplemente cambia `lala_beta` por el nombre de tu juego.

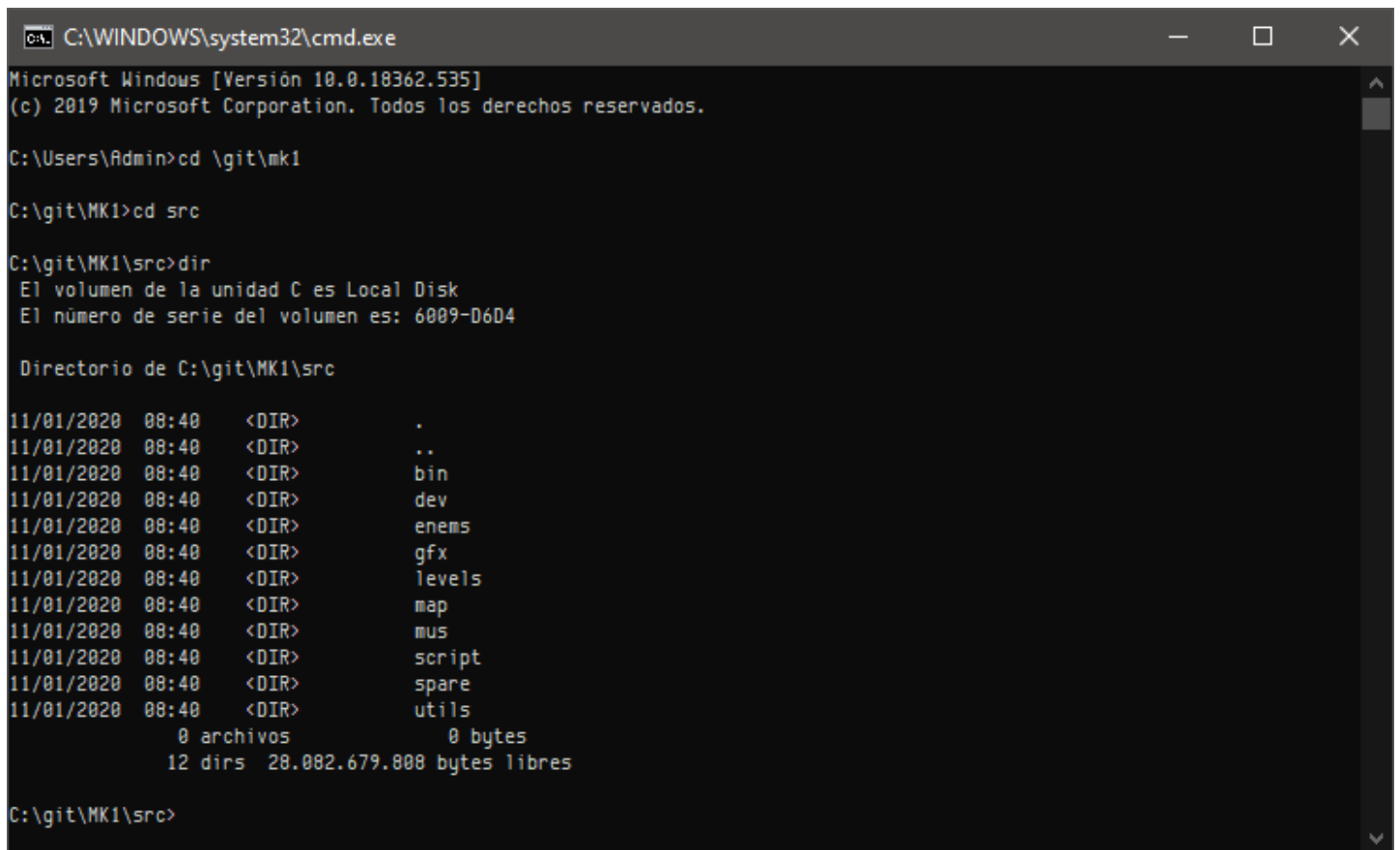
Este archivo `compile.bat` contiene todos los pasos que hay que ejecutar para convertir cada archivo de datos (imagenes, mapa, ...) de tu juego y luego compilar el motor para obtener un archivo `.tap`. En los siguientes capítulos veremos cada conversión e incluso la ejecutaremos de forma manual para entender qué está pasando tras las entretelas de los visillos, pero a la hora de hacer un juego, a menos que estés haciendo algo muy avanzado, te bastará con sustituir los archivos que vienen en el proyecto de ejemplo y ejecutar `compile.bat` cada vez.

Ahora ya estamos listos para que empiece lo bueno.

Capítulo 2: Tileset

Antes de empezar

En este capítulo y en prácticamente todos los demás tendremos que abrir una ventana de línea de comandos para ejecutar scripts y programillas, además de para lanzar la compilación del juego y cosas por el estilo. Lo que quiero decir es que deberías tener alguna noción básica de estos manejos. Si no sabes lo que es esto que te pongo aquí abajo, es mejor que consultes algún tutorial básico sobre el manejo de la ventana de línea de comandos (o consola) del sistema operativo que uses. O eso, o que llames a tu amigo el de las gafas y la camiseta de Piedra-Papel-Tijeras-Lagarto-Spock.



```
C:\WINDOWS\system32\cmd.exe
Microsoft Windows [Versión 10.0.18362.535]
(c) 2019 Microsoft Corporation. Todos los derechos reservados.

C:\Users\Admin>cd \git\mk1

C:\git\MK1>cd src

C:\git\MK1\src>dir
El volumen de la unidad C es Local Disk
El número de serie del volumen es: 6009-D6D4

Directorio de C:\git\MK1\src

11/01/2020  08:40    <DIR>        .
11/01/2020  08:40    <DIR>        ..
11/01/2020  08:40    <DIR>        bin
11/01/2020  08:40    <DIR>        dev
11/01/2020  08:40    <DIR>        enems
11/01/2020  08:40    <DIR>        gfx
11/01/2020  08:40    <DIR>        levels
11/01/2020  08:40    <DIR>        map
11/01/2020  08:40    <DIR>        mus
11/01/2020  08:40    <DIR>        script
11/01/2020  08:40    <DIR>        spare
11/01/2020  08:40    <DIR>        utils
               0 archivos             0 bytes
              12 dirs 28.002.679.808 bytes libres

C:\git\MK1\src>
```

Podéis echar un vistazo por ejemplo a [este tutorial](#). Con los comandos listados en la sección *Lista de comandos básicos* tendréis más que suficiente.

Material

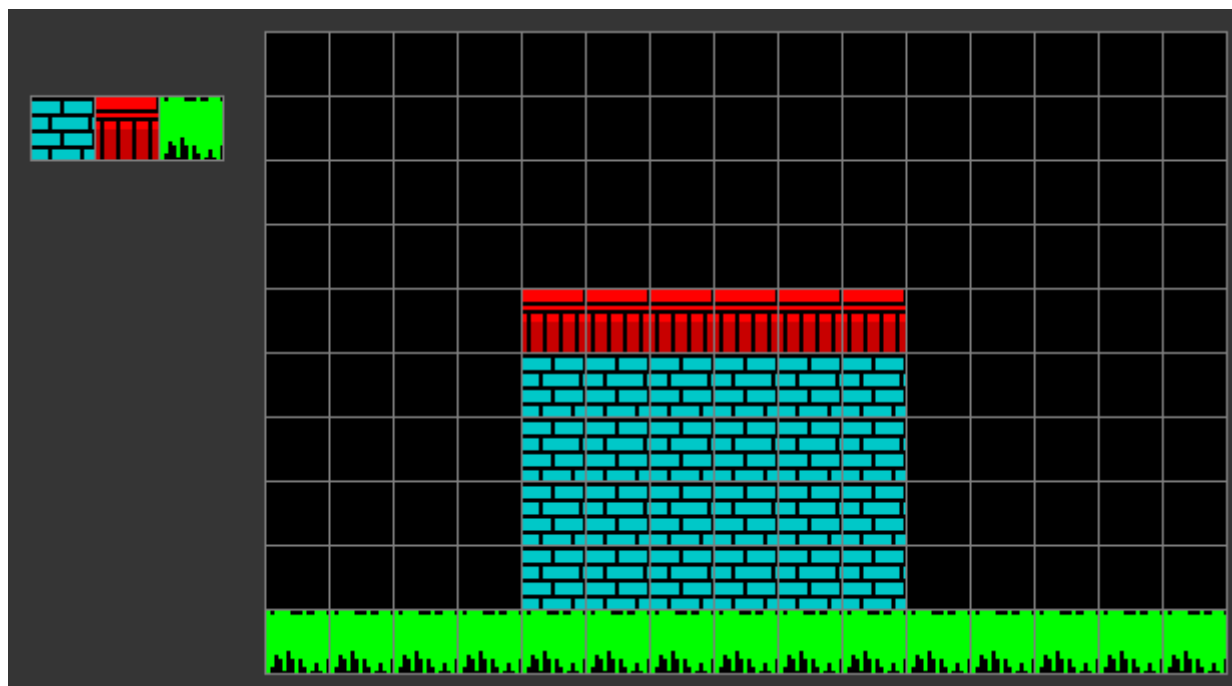
El material necesario para seguir este capítulo os lo he dejado aquí:

[Material del capítulo 2](#)

Descárgalo y ponlo en una carpeta temporal, que ya iremos poniendo cosas en nuestro proyecto según las vayamos necesitando. Dentro hay cosas bonitas.

Tileset... ¿De qué leches estamos hablando?

Pues de tiles. ¿Que qué es un tile? Pues para ponerlo sencillo, no es más que un cachito de gráfico que es del mismo tamaño y forma que otros cachitos de gráficos. Para que lo veas, busca la traducción: tile significa "azulejo", (aunque nosotros preferimos pensar que en realidad se trata de las siglas "Tengo Ideas Locas y Estrafalarias"). Ahora piensa en la pared de tu cuarto de baño, e imagina que en cada azulejo hay un cachito de gráfico. Tenemos el azulejo con un cachito de ladrillo, el azulejo con un cachito de hierba, y el azulejo negro y el azulejo con un cachito de suelo. Con varios de cada uno podemos ordenarlos de forma que hagamos un dibujo que se parezca a una casa de campo. Un cuarto de baño así molaría de la hostia, por cierto. Siempre hay que hacer pipí. Y caca.



Esto es lo que usa **MTE MK1** para pintar los gráficos de fondo. Como guardar una pantalla gráfica completa ocupa un huevo, lo que se hace es guardar un cierto número de cachitos y luego una lista de qué cachitos ocupa cada pantalla. La colección de cachitos de un güego es lo que se conoce como "tileset". En este capítulo vamos a explicar cómo son los tilesets de **MTE MK1**, cómo se crean, cómo se convierten, cómo se importan y cómo se usan. Pero antes necesitamos entender varios conceptos. Vete preparando un zumito.

Colisión

MTE MK1, además, usa los tiles para otra cosa: para la colisión. Colisión es un nombre muy chulo para referirse a algo muy tonto: el protagonista del güego podrá andar por la pantalla o no dependiendo del tipo del tile que vaya a pisar. O sea, que cada tile tiene asociado un comportamiento. Por ejemplo, al tile negro del ejemplo de arriba podríamos ponerle un comportamiento "traspasable" para que el jugador pudiera moverse libremente por el espacio ocupado por estos tiles. En cambio, el tile de la hierba debería ser "obstáculo", entendiendo que debe impedir que el protagonista se mueva por el espacio que ocupan. Un güegos de plataformas, por ejemplo, el motor hará caer al protagonista siempre que no haya un tile "obstáculo" bajo sus pies.

En los güegos de **MTE MK1** tenemos los siguientes tipos de tiles, o, mejor dicho, los siguientes comportamientos para los tiles. Cada uno, además, tiene un código que necesitaremos saber. Ahora no, sino más adelante, cuando ya tengamos todo el material y estemos montando el güego. Por ahora nos basta con la lista:

1. *Tipo "0", traspasable.* En los güegos de plataformas puede ser el cielo, unos ladrillos de fondo, el cuadro del tío Narciso, un florero feo o unas montañas a tomar por culo. En los güegos de vista genital los usaremos para el suelo por el que podemos andar. O sea, cosas que no detengan la marcha del muñeco.
2. *Tipo "1", traspasable y matante:* se puede traspasar, pero si se toca se resta vida al protagonista. Por ejemplo, unos pinchos, un pozo de lava, pota radioactiva, cristales rotos, o las setas en el **Cheril of the Bosque** (¿recuerdas? ¡no se podían de tocá!).
3. *Tipo "2", traspasable pero que oculta.* Son los matojos de **Zombie Calavera**. Si el personaje está quieto detrás de estos tiles, se supone que está "escondido". Los efectos de estar escondido son muy poco interesantes ya que solo afectan a los murciélagos del **Zombie Calavera**, pero bueno, ahí está, y nosotros lo mencionamos.
4. *Tipo "4", plataforma.* Sólo tienen sentido en los güegos de plataformas, obviamente. Estos tiles sólo detienen al protagonista desde arriba, o sea, que si estás abajo puedes saltar a través de ellos, pero si caes desde arriba te posarás encima. No sé cómo explicártelo... como en el **Sonic** y eso. Por ejemplo, si pintas una columna que ocupe tres tiles (cabeza, cuerpo, y pie), puedes poner el cuerpo de tipo "0" y la cabeza de tipo "4", y así se puede subir uno a la columna. También queda bien usar este tipo para plataformas delgadillas que no pegue que sean obstáculos del todo, como las típicas plataformas metálicas que salen en muchos de nuestros güegos.
5. *Tipo "8", obstáculo.* Este para al personaje por todos los lados. Las paredes. Las rocas. El suelo. Todo eso es un tipo 8. No deja pasar. Piedro. Duro. Croc.
6. *Tipo "10", interactuable.* Es un obstáculo pero que sea de tipo "10" hace que el motor esté coscao y sepa que es especial. De este tipo son, por ahora, los cerrojos y los bloques que se pueden empujar. Hablaremos de ellos dentro de poco.
7. *Tipo "10", destructible.* Son tiles que se pueden romper disparándoles.

Vaya mierda, pensarás, ¡si faltan números! Y más que faltaban antes. Esto está hecho queriendo, amigos, porque simplifica mucho los cálculos y permite **combinar comportamientos** sumando los números, hasta donde tenga sentido. Por ejemplo, un tile *obstáculo que mata* ($8+1 = 9$) no tiene sentido porque no lo vamos a poder tocar nunca, pero un *obstáculo destructible* ($8+16 = 24$) sí que lo tiene. De hecho, si no ponemos los destructibles como obstáculos *se podrán traspasar*.

En un futuro, además, se puede ampliar fácilmente, como hemos dicho. Por ejemplo, se podría añadir código a **MTE MK1** para que los tiles de tipo "5" y "6" fueran como cintas transportadoras a la izquierda y a la derecha, respectivamente. Se podría añadir. Podría. A lo mejor, al final, hacemos un

capítulo de arremangarse y meter código nuevo en **MTE MK1**... ¿por qué no? Creo que además ya sabes de sobra que [me encanta el café](#)...

Interactuables

Hemos mencionado los tiles interactivables. En la versión actual de **MTE MK1** son dos: cerrojos y empujables. Tienes que decidir si tu güego necesita estas características.

Los **cerrojos** los necesitarás si pones llaves en el güego. Si el personaje choca con un cerrojo y lleva una llave, pues lo abre. El cerrojo desaparecerá y se podrá pasar.

Los **tiles empujables** son unos tiles que, al empujarlos con el protagonista, cambiarán su posición si hay sitio. En los güegos de plataformas sólo se pueden empujar lateralmente; en los de vista genital se pueden empujar en cualquier dirección. Como ya veremos en su momento, podemos definir un par de cosas relacionadas con estos tiles empujables, como por ejemplo si queremos que los enemigos no puedan traspasarlos (y así poder usarlos para confinarlos y “quitarlos de enmedio” como ocurre en **Cheril of the Bosque** o **Monono**, por ejemplo).

Vamos al lío ya ¿no?

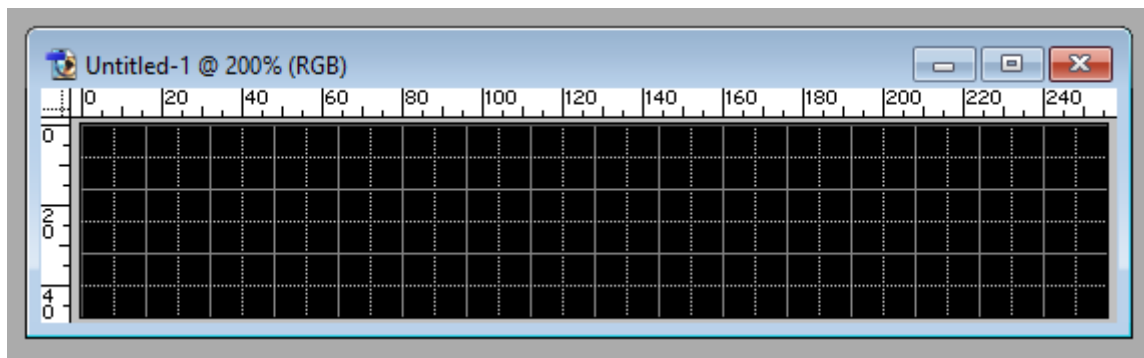
Eso, vamos al lío ya. Vamos a dibujar nuestro tileset, o a rapiñarlo de por ahí, o a pedir a nuestro amigo que sabe dibujar que nos lo haga. Que sí, hombre, que te busques uno, que hay muchos grafistas faltos de amor. Lo primero que tenemos que decidir es si vamos a usar un tileset de 16 tiles diferentes o de 48, que son los dos tamaños de tilesets que soporta **MTE MK1**. Qué tontería, estarás pensando, ¡de 48! ¡son más! Por supuesto que son más, mi querido Einstein, pero ocurre una cosa: 16 tiles diferentes se pueden representar con un número de 4 bits. Eso significa que en un byte, que tiene 8 bits, podemos almacenar dos tiles. ¿Adónde quiero llegar? ¡Bien, lo habéis adivinado! Los mapas ocupan exactamente la mitad de memoria si usamos tilesets de 16 tiles en lugar de tilesets de 48.

Ya sé que 16 pueden parecer pocos tiles, pero pensad que la mayoría de nuestros güegos están hechos así, y muy feos no quedan. Con un poco de inventiva podemos hacer pantallas muy chulas con pocos tiles. Además, como veremos más adelante en este mismo capítulo, usar tilesets de 16 tiles nos permitirá activar el efecto de sombras automáticas, que hará que parezca que tenemos bastantes más de 16 tiles.

Otra cosa que se puede hacer es hacer juegos de varios niveles cortos en el que cambiemos de tileset para cada nivel. Esto dará suficiente variedad y permitirá usar mapas que ocupen la mitad.

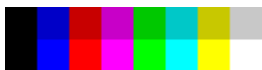
Por lo pronto id abriendo vuestro programa de edición gráfica preferido y creando un nuevo archivo de 256×48 píxels. Seguro que vuestro programa de edición gráfica tiene una opción para activar una rejilla (o grid). Colocadla para que haga recuadros de 16×16 píxels y, a ser posible, que tenga 2 subdivisiones, para que podamos ver donde empieza cada carácter. Esto nos ayudará a hacer los gráficos siguiendo las restricciones del Spectrum, o a poder saber donde empieza y termina cada tile

a la hora de recortarlos y/o dibujarlos. Yo uso una versión super vieja de Photoshop y cuando creo un nuevo tileset me veo delante de algo así:



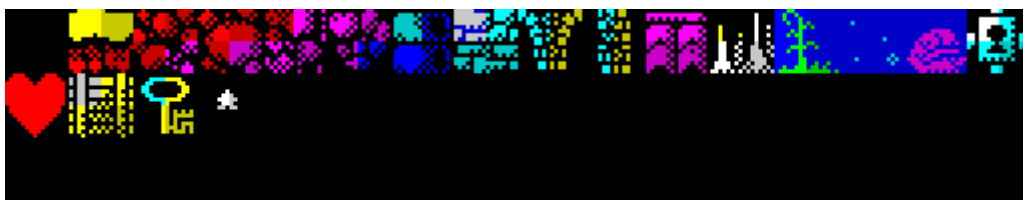
Con respecto a la paleta del Spectrum, como todas estas cosas, los valores que soportan por defecto los conversores incluidos en el toolchain son bastante arbitrarios. Para que todo vaya bien, usa unos valores de R, G, B de 200 si quieres representar los colores sin BRIGHT y de 255 si quieres representar los colores con BRIGHT. A Mappy no le gusta el magenta intenso (255, 0, 255), así que para este color puedes usar por ejemplo (254, 0, 255).

Si no te quieres rayar, usa los colores de esta paleta:



Haciendo un tileset de 16 tiles

Si has decidido ahorrar memoria (por ejemplo, si planeas que el motor del güego termine siendo medianamente complejo, con scripting y muchas cosas molonas, o si prefieres que tu mapa sea bien grande) y usar tilesets de 16 tiles, tienes que crear algo así:



El tileset se divide en dos secciones: **la primera, formada por los primeros dieciséis tiles, es la sección de mapa**. Es la que usaremos para hacer nuestro mapa. La segunda, formada por los cuatro siguientes, es la **sección especial** que se usará para pintar cosas especiales.

La sección de mapa

Por convención, el tile 0, o sea, el primero del tileset (los verdaderos desarrolladores empiezan a contar en el 0), será el tile de fondo principal, el que ocupe la mayoría del fondo en la mayoría de las pantallas. Esto no es un requerimiento, pero nos facilitará la vida cuando hagamos el mapa por motivos obvios: al crear un mapa vacío ya estarán todos los tiles puestos a 0.

Los **tiles del 1 al 13** pueden ser lo que quieras: tiles de fondo, obstáculos, plataformas, matantes...

El **tile 14** (el penúltimo), si has decidido que vas a activar los tiles empujables, será el **tile empujable**. Tiene que ser el 14, y ningún otro.

El **tile 15** (el último), si has decidido que vas a activar las llaves y los cerrojos, será el **tile de cerrojo**. Tiene que ser el 15, y ningún otro.

Si no vas a usar empujables o llaves/cerrojos puedes usar los tiles 14 y 15 libremente, por supuesto.

La sección especial

Está formada por cuatro tiles que son, de izquierda a derecha:

1. El **tile 16** es la **recarga de vida**. Aparecerá en el mapa y, si el usuario la pilla, recargará un poco de vida.
2. El **tile 17** representa a los **objetos coleccionables**. Son los que el jugador tendrá que recoger durante el güego, si decidimos activarlos.
3. El **tile 18** representa las **llaves**. Si hemos decidido incluir llaves y cerrojos, pintaremos la llave en este tile.
4. El **tile 19** es el **fondo alternativo**. Para dar un poco de *age* a las pantallas, de forma aleatoria, se pintará este tile de vez en cuando en vez del tile 0. Por ejemplo, si tu tile 0 es el cielo, puedes poner una estrelica en este tile. O, si estás haciendo un güego de perspectiva genital, puedes poner una variación del suelo. **Este fondo alternativo sólo funciona en tilesets de 16 tiles.**

¿Se pilla bien? Básicamente hay que dibujar 20 tiles: 16 para hacer el mapa, y 4 para representar objetos y restar monotonía de los fondos. Huelga decir que si, por ejemplo, no vas a usar llaves y cerrojos en tu güego, te puedes ahorrar pintar la llave en el tile 18.

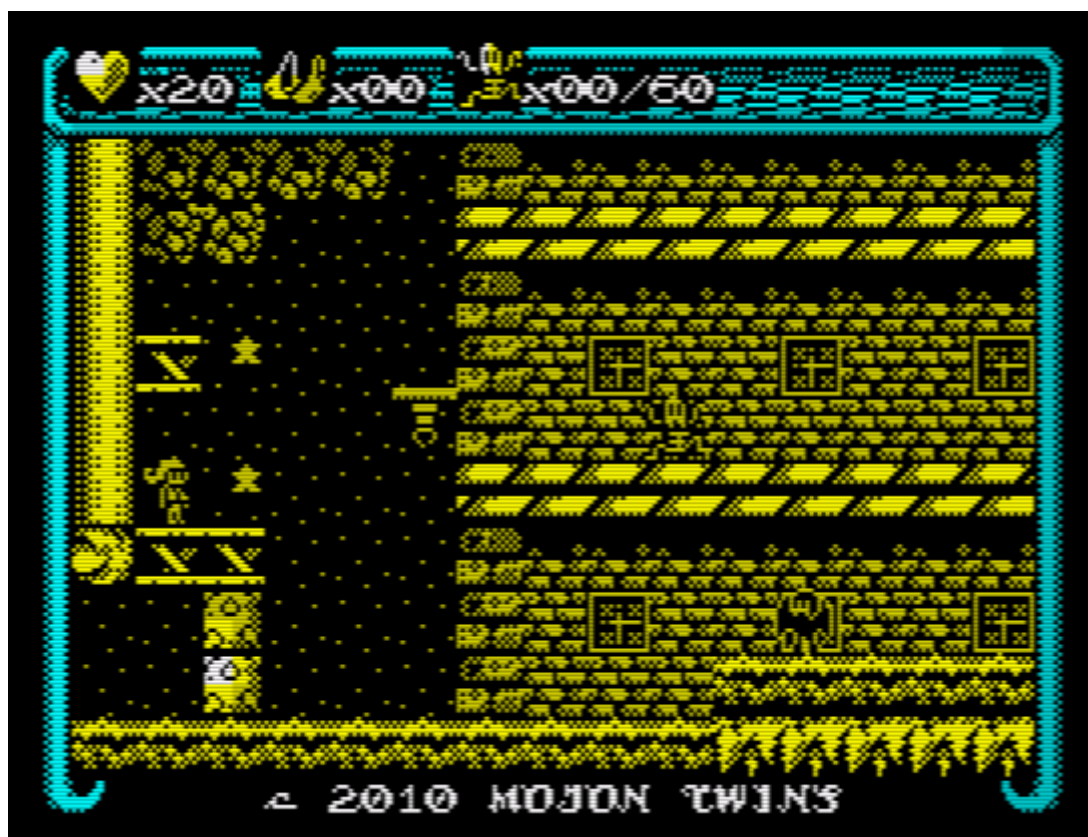
En el dogmole, por ejemplo, no hay tiles empujables. Por eso nuestro tile 14 es un mejillón del cantábrico que, como todos sabemos, no se puede empujar.

Sombreado automático

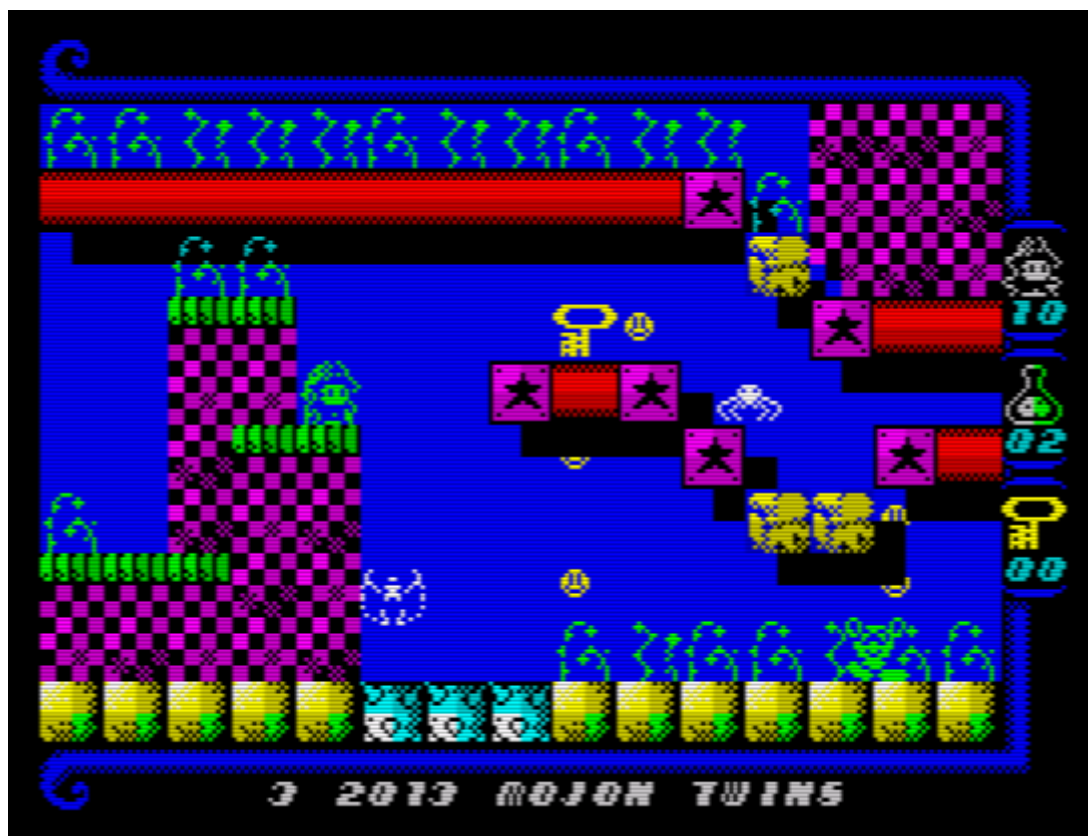
El sombreado automático puede hacer que nuestras pantallas se vean mucho más chulas. Si lo activamos, **MTE MK1** hará que los tiles obstáculo proyecten sombras sobre los demás. Para conseguirlo, necesitamos definir una versión alternativa de la sección de mapa con los tiles que no sean obstáculo sombreados, que pegaremos en la fila inferior de nuestro archivo de tiles. Por ejemplo, este es el tileset de **Cheril Perils**:



Tendremos total control, por tanto, de cómo se proyectan las sombras. El resultado obtenido lo podéis ver aquí abajo:



Otro ejemplo es Lala Lah, dentro pantallazo:



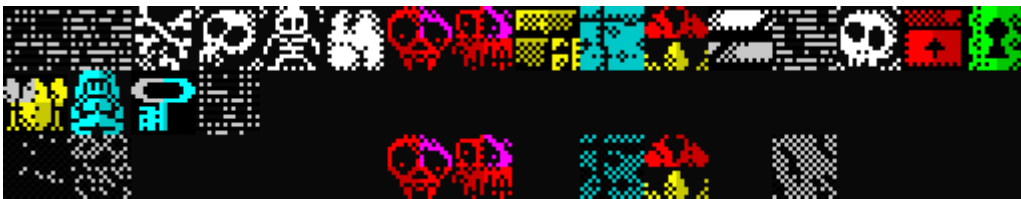
En **Dogmole** no vamos a usar esto porque necesitamos el espacio que ocuparían las sombras automáticas para otra cosa que ya veremos en su momento.

Ejemplos

Para verlo, vamos a echarle el ojetito a algunos tilesets de nuestros güegos, a modo de ejemplo, para que veáis cómo están diseñados.



Aquí tenemos el tileset de **Lala lah**. Como vemos, el primer tile es el fondo azul que se ve en la mayoría de las pantallas. Le sigue un trozo de plataforma que también es un tile de fondo, y después el rebordecico que es un tile tipo “plataforma” (tipo 4). Si juegas al güego verás cómo se comporta este tile, para terminar de entenderlo. El piedro amarillo que le sigue es un obstáculo (tipo 8). Luego hay dos matojitos psicodélicos para adornar de fondo (tipo 0). Luego otro piedro (8), un fondo enladrillado (0), una variación de los cuadros (0), una bola de pinchos que mata (tipo 1), una caja con estrella (8), dos tiles para hacer tiras no traspasables (y, por tanto, de tipo 8), una plataforma to roneona (tipo 4), y para terminar el tile nº 15 será de tipo 10, porque usamos cerrojos y los cerrojos tienen que ser obstáculos interactivables. Luego tenemos la recarga, el objeto y la llave, el tile alternativo para el fondo, y en la tira de abajo los que se usan en el sombreado automático. Vamos a ver otro:



Este es el tileset de **D'Veel'Ng**, un güego de perspectiva genitral. Este empieza con dos tiles para suelo (tipo 0), seguidos por cuatro obstáculos (tipo 8) – los buesos, la calavera, la canina y el piedro, dos tiles matadores y malignos (tipo 1), que son esas calaveras rojas tan feas, otro obstáculo en forma de ladrillos amarillos (tipo 8), otro suelo embaldosado (tipo 0), otro tile que te mata en forma de seta maligna (tipo 1), ladrillos blancos (tipo 8), más suelo (tipo 0), y otra calavera obstaculizante (tipo 8). Este güego tiene tiles que se empujan, por lo que el tile 14 es una caja roja de tipo 10. También tenemos llaves, por lo que el tile 15 es un cerrojo, también de tipo 10. Luego tenemos la típica recarga de vida, el objeto y la llave, y el tile alternativo para el fondo que se pinta aleatoriamente. En la fila de abajo, volvemos a tener una versión sombreada de los tiles de fondo. Fíjate como los tiles que matan los he dejado igual en la tira “sombreada”: esto es para que siempre se vean bien destacados. Venga, más:



Ahora toca el del **Monono**. Este es muy sencillo de ver: empezamos con el tile de fondo principal, vacío del todo (tipo 0). Seguimos con seis obstáculos (tipo 8). Luego tenemos dos tiles de fondo más, para adornar los fondos: la ventanica para asomarse y desi ola k ase y el escudo. Luego hay tres tiles-obstáculo más (tipo 8), unos pinches venenosos (tipo 1), nuestra típica plataforma metálica copyright Mojon Twins signature special (tipo 4), una caja que se puede empujar (tile 14, tipo 10) y un cerrojo (tile 15, tipo 10). Luego lo de siempre: recarga, objeto, llave, alternativo. Este no tiene sombreado automático.

Podría seguir, pero es que no tengo más ganas.

Haciendo un tileset de 48 tiles

Como hemos advertido, los mapas hechos con tilesets de 48 tiles ocupan el doble que los que están hechos con tilesets de 16. Si aún así decides seguir esta ruta, aquí tienes una explicación de cómo hacerlo.

En primer lugar, aquí no hay una diferenciación explícita entre la sección de mapa y la sección especial: está todo junto. Además, no es posible añadir sombreado automático (no tenemos sitio para las versiones alternativas de los tiles). Sobre todo, se nota que esto de los tilesets de 48 tiles fue un añadido metido con calzador huntao en manteca para Zombie Calavera que no hemos vuelto a utilizar y que no está nada refinado. Pero oye, se puede usar.

En los tilesets de 48 tiles, podemos usar los tiles de 0 a 47 para hacer las pantallas salvo los correspondientes a las características especiales: el tile 14 para el empujable, el 15 para cerrojos, el 16 para recargas, el 17 para objetos y el 18 para llaves, si es que los vas a usar. Si no los vas a usar, puedes poner tus propios tiles. En los tilesets de 48 tiles tampoco se usa el 19 como tile alternativo de fondo, así que puedes poner lo que quieras en ese espacio.

Como único ejemplo tenemos el tileset de Zombie Calavera. Si te fijas, en Zombie Calavera no hay llaves (ni cerrojos), por lo que sólo están ocupados como “especiales” el 16 para las recargas y el 17 para los objetos. Tampoco hay tiles empujables. Todos los demás se usan para pintar el escenario:



Ya tenemos el tileset pintado. Ahora ¿qué?

MTE MK2 utiliza un set de 256 "patrones" para las partes "fijas" (fondo, textos, marcadores). Un "patrón", o carácter, no es más que una imagen de 8x8 píxeles. De los 256 "patrones", los 64 primeros contendrán letras, números y símbolos, y los 192 restantes los diferentes trozos empleados para construir el tileset que has pintado.

Para conseguir tener todos sus trocitos tal y como los necesita, **MTE_MK2** dispone de un convertidor llamado `ts2bin.exe`, ubicado, como todas las utilidades, en el directorio `src/utls`. `ts2bin.exe` necesita DOS archivos para funcionar: uno con los 64 "patrones" de la fuente, y otro con tu "tileset".

Si no quieres tocar nada, lo primero que tienes que hacer es coger tu tileset, llamarlo `work.png`, y grabarlo en `/gfx` (sustituyendo el que viene por defecto). Luego necesitas hacer el archivo con la fuente.

Si estás siguiendo el tutorial con **Dogmole** sin hacer experimentos propios, puedes encontrar el `work.png` de este güego en el paquetito de archivos de este capítulo.

La fuente

El archivo con la fuente, que debe llamarse `font.png` y estar en el directorio `gfx` si no quieres editar `compile.bat`, debe contener, como hemos dicho, 64 "patrones". Estos "patrones" deben representar los caracteres ASCII desde el 32 al 95 y estar organizados en una imagen de 256x16 parecida a esta:



Lo más fácil es utilizar como plantilla el archivo `fuentes_base.png` incluido en el paquetito de archivos de este capítulo, y dibujar sobre él o adaptar alguna fuente ya existente. La fuente para **Dogmole** también está incluida en el paquetito: `font.png` que deberás copiar en `/gfx` sustituyendo la que viene.

La conversión e importación

Como hemos dicho, la utilidad `ts2bin` incluida en `/utls` se encarga de tomar una fuente y un tileset y generar un binario con todos los patrones en el formato que espera **MTE MK2**. La invocación al conversor está incluida en `compile.bat` para que tú no tengas nada más que hacer que editar los gráficos y ponerlos en su sitio (y hacer todas las modificaciones que necesites de forma indolora) (los que hayan catado la Churrera en sus versiones anteriores estará ahora dando palmas con las orejas porque antes había que hacer bastante trabajo manual).

Sin embargo, por tema del saber, que no ocupa lugar, vamos a explicar como funciona.

`ts2bin` está pensado para ser llamado desde línea de comandos (o desde un archivo de script como `compile.bat`). Si lo ejecutas a pelo desde la ventana de línea de comandos él mismo te chiva qué parámetros espera:

```
$ src\utls\ts2bin.exe
ts2bin v0.4 20200119 ~ Usage:
```

```
$ ts2bin font.png/nofont work.png|notiles|blank ts.bin defaultink
```

where:

- * font.png is a 256x16 file with 64 chars ascii 32-95
(use 'nofont' if you don't want to include a font & gen. 192 tiles)
- * work.png is a 256x48 file with your 16x16 tiles
(use 'notiles' if you don't want to include a tileset & gen. 64 tiles)
(use 'blank' if you want to generate a 100% blank placeholder tileset)
- * ts.bin is the output, 2304 bytes bin file.
- * defaultink: a number 0-7. Use this colour as 2nd colour if there's only one colour in a 8x8 cell

El primer parámetro es el nombre de archivo de la fuente (incluyendo su ubicación si es necesaria), o la palabra `nofont` si sólo quieres convertir el tileset (cosa que viene bien para hacer otras cosas que no son un juego de la churrera normal).

El segundo parámetro es el nombre del archivo con el tileset (incluyendo su ubicación si es necesaria), o la palabra `notiles` si sólo quieres convertir la fuente (bla bla bla, no con la churrera en situaciones normales), o `blank` si quieres generar el binario completo sólo con la fuente y con los tiles en negro, que es lo que llamamos un "placeholder para multi nivel" y que ya entenderás cuando veamos los multinivel.

El tercer parámetro es el nombre del archivo que quieres generar (incluyendo su ubicación si es necesaria). En el caso de la churrera, el archivo resultante ocupará 2304 y contendrá todos los patrones (fuente y tileset) y los colores que se usan en el tileset.

El cuarto parámetro es opcional, y sirve para especificar un color de tinta 0-7 que quieres que se utilice si se encuentra algún "patrón" que sea un cuadrado de color sólido como segundo color. Si sabes de Spectrum le encontrarás sentido a esto.

Si abres `compile.bat` verás que los parámetros que se emplean en la llamada para obtener los patrones se corresponden con los que hemos mencionado más arriba (la fuente se llama `font.png` y el tileset `work.png`, y se ubican en `gfx`). El archivo de salida es `tileset.bin` y se emplea `7` como `defaultink` .:

```
..\utils\ts2bin.exe ..\gfx\font.png ..\gfx\work.png tileset.bin 7
```

Si tu juego necesita sprites de otro color sobre cuadros totalmente negros tendrás que modificar esta línea con el color que necesites.

Los `..\utils\` y `..\gfx` hacen referencia a que los archivos se ubican en esas carpetas que están un nivel más *abajo* de donde está `compile.bat`. El `> nul` del final hace que las mierdas que dice `ts2bin` no se muestren.

Puedes probar a ejecutar el comando desde `dev` tal y como aparece para ver como se genera `tileset.bin` :

```
$ ..\utils\ts2bin.exe ..\gfx\font.png ..\gfx\work.png tileset.bin forcezero
ts2bin v0.3 20191202 ~ Reading font ~ reading metatiles ~ 2304 bytes written
```

```
$ dir tileset.bin
El volumen de la unidad C es Local Disk
El número de serie del volumen es: 6009-D6D4

Directorio de C:\git\MK1\src\dev

11/01/2020  16:06                2.304 tileset.bin
             1 archivos                2.304 bytes
             0 dirs  27.100.073.984 bytes libres
```

Repetimos, esta tarea, si no tienes necesidades especiales, está ya cubierta en `compile.bat` .

Vale, ya está todo.

Y con el ladrillo escamondao, os ubicamos en este mismo canal para el próximo capítulo, donde montaremos el mapa.

Capítulo 3: Mapas

Antes que nada, bájate el paquete de materiales correspondiente a este capítulo pulsando en este enlace:

[Material del capítulo 3](#)

¡Podewwwr mapa!

En este capítulo vamos a hacer el mapa. El tema se trata de construir cada pantalla del juego usando los tiles que dibujamos en el capítulo anterior. Cada pantalla es una especie de rejilla de casillas, donde cada casilla tiene un tile. En concreto, las pantallas se forman con 15×10 tiles.

Para construir el mapa usaremos **Mappy** (cuyos programadores realmente se quebraron la cabeza para ponerle el nombre). **Mappy** es bastante sencillote pero lo cierto es que funciona muy bien y, lo que es mejor, se deja meter mano. La versión de **Mappy** incluida en este repositorio (disponible en `/env/mappy-mojono.zip`) tiene un par de añadidos y modificaciones para que funcione como nosotros queremos.

Antes de empezar a ensuciarnos las manos y tal vamos a explicar un poco cómo funciona **Mappy**. El programa maneja un formato complejo para describir los mapas que permite varias capas, tiles animados y un montón de paranoias. Este formato, el nativo de **Mappy**, se llama `FMP` (que significa *Fernando Masticando Pepinos*) y es el que usaremos para almacenar nuestra copia de trabajo del mapa. Es la copia que cargaremos en **Mappy** cada vez que queramos modificar algo. Sin embargo, este formato es demasiado complejo para el importador de **MTE MK1**, que es mu tonto.

Nosotros sólo queremos una ristra de bytes que nos digan qué tile hay en cada casilla del mapa. Para eso **Mappy** tiene otro formato, el formato `MAP` (*Manolo Asando Pepinos*), que es el formato simple y personalizable. Nosotros lo hemos dejado en la mínima expresión: una ristra de bytes que nos digan qué tile hay en cada casilla del mapa, precisamente. Es el formato en el que habrá que guardar el mapa cuando queramos generar la copia que luego procesaremos e incorporaremos al güego.

Hay que tener mucho cuidado con esto, ya que es posible que actualicemos el mapa, grabemos el `MAP`, y se nos olvide guardar el `FMP`, con lo que la próxima vez que queramos ir al **Mappy** a hacer algún cambio habremos perdido los últimos. Nosotros tenemos a Colacao, el Mono Coscao, que nos avisa cuando nos olvidamos de grabar el `FMP`, pero entendemos que tú no tengas a tu lado a ningún mono coscao que cuide de tí, así que asegúrate de grabar en `MAP` y en `FMP` cada vez que hagas un cambio. Que luego jode perder cosas. No sé, ponte un post-it.

¿Que cómo es esto? Ya lo verás, pero el tema es tan sencillo como especificar `mapa.map` o `mapa.fmp` como nombre de archivo a la hora de grabar el mapa con utilizando `File → Save`. Cutre, pero efectivo.

Definiendo nuestro mapa

Lo primero que tenemos que hacer es decidir cuál será el tamaño de nuestro mapa. El mapa no es más que un rectángulo de N por M pantallas. Como es lógico, cuantas más pantallas tenga nuestro mapa en total, más RAM ocupará. Por tanto, el tamaño máximo que puede tener el mapa dependerá de qué características tengamos activadas en el motor para nuestro güego. Si se trata de un güego con motor simple, nos cabrán muchas pantallas. Si tenemos un güego más complejo, con muchas características, scripting y tal, nos cabrán menos.

Es muy complicado dar una estimación de cuántas pantallas nos cabrán, como máximo. En juegos de motor sencillo, con pocas características, como **Sir Ababol** (que, al estar construido usando la versión 1.0 de **MTE MK1**, sólo incorpora las características básicas de un juego de plataformas), nos cupieron 45 pantallas y aún sobró un porrón de RAM, con lo que podría haber tenido muchas más pantallas. Sin embargo, en otros juegos más complejos, como **Cheril the Goddess** o **Zombie Calavera**, ocupamos casi toda la RAM con mapas mucho más pequeños.

Dependiendo de cómo vaya a funcionar tu güego, a lo mejor un mapa pequeño es suficiente. Por ejemplo, en el citado **Cheril the Goddess** hay que andar bastante de un lado para otro y recorrerlo varias veces para ir llevando los objetos a los altares, por lo que el juego no se hace nada corto.

Como en **Mappy** es posible redimensionar un mapa una vez hecho, quizá sea buena idea empezar con un mapa de tamaño moderado y, al final, si vemos que nos sobra RAM y queremos más pantallas, ampliarlo.

En el capítulo anterior hablamos de tilesets de 16 y de 48 tiles, y dijimos que usando uno de los primeros las pantallas ocupaban la mitad. Esto es así porque el formato de mapa que se usa es el que conocemos como `packed`, que almacena dos tiles en cada byte. Por tanto, un mapa de estas características ocupa $15 \times 10 / 2 = 75$ bytes por pantalla. Las 30 pantallas de **Lala the Magical**, por ejemplo, ocupan 2250 bytes (30×75). Los mapas de 48 tiles no se pueden empaquetar de la misma manera, por lo que cada tile ocupa un byte. Por tanto, cada pantalla ocupa 150 bytes. Las 25 pantallas de **Zombie Calavera** ocupan, por tanto, 3750 bytes. ¿Ves como conviene usar menos tiles?

Soy consciente de que con toda esta plasta que te he soltado te he aclarado muy poco, pero es que poco más te puedo aclarar. Esto es una cosa bastante empírica. Podría haberme puesto a estudiar cuanto ocupa cada característica que activemos de **MTE MK1**, pero la verdad es que nunca me ha apetecido hacerlo. Sí te puedo decir que el motor de disparos, por ejemplo, ocupa bastante, sobre todo en vista genital (como en **Mega Meghan**), ya que necesita muchas rutinas que añadir al binario. Los enemigos voladores que te persiguen (**Zombie Calavera**) también ocupan bastante. El tema de empujar bloques, las llaves, objetos, tiles que te matan, rebotar contra las paredes, o los diferentes tipos de movimiento (poder volar (**Jet Paco**, **Cheril the Goddess**), salto automático (**Bootee**), ocupan menos. El scripting también puede llegar a ocupar mucha memoria si usamos muchas órdenes y comprobaciones diferentes.

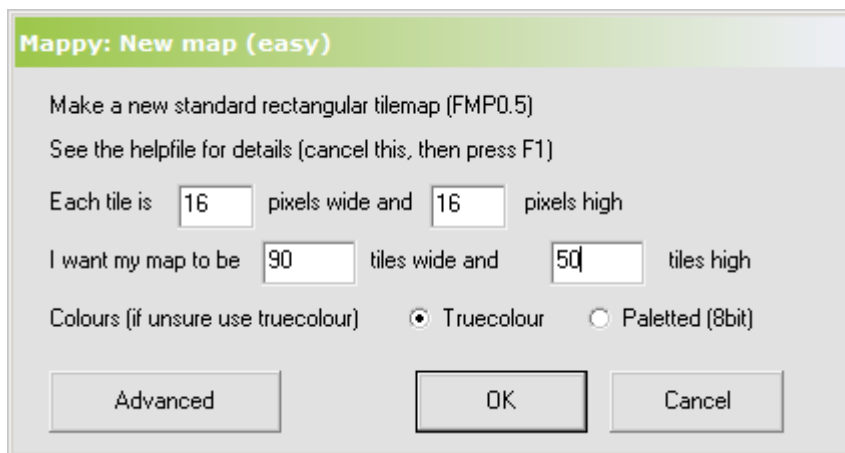
Tú fíjate un número que ronde entre las 25 y 40 pantallas y cabrá. Y si no, se recorta y listo. También me puedes [hartar a cafés](#) para que incorpore los mapas comprimidos en RLE (que significa *Ramiro Lo*

Entiende) de MK2 en MK1.

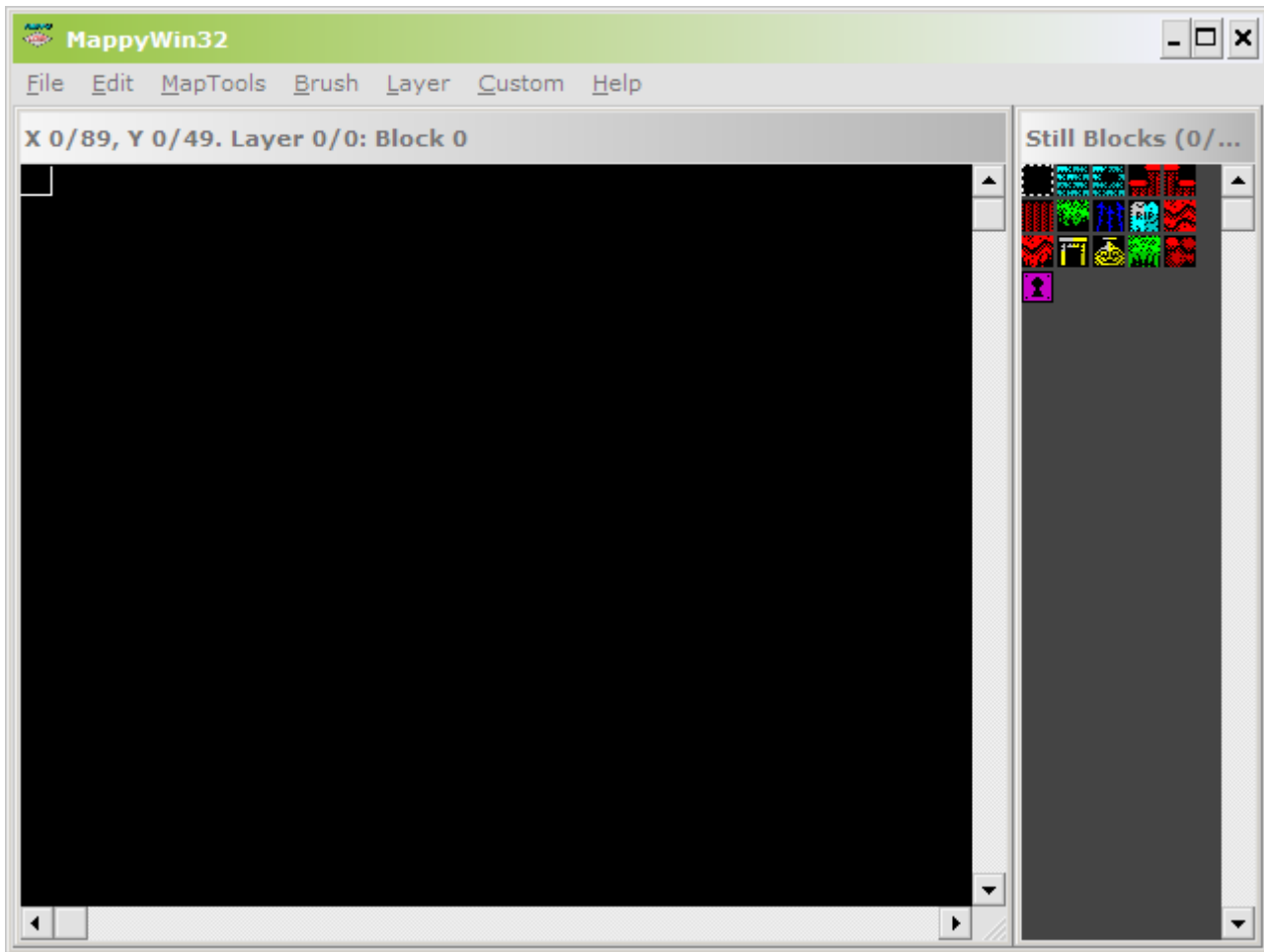
Creando un proyecto en Mappy

Lo primero que tenemos que hacer es abrir **Mappy**, ir a `File → New map`, y rellenar el recuadro donde definiremos los valores importantes de nuestro mapa: el tamaño de los tiles (16×16), y el tamaño del mapa en tiles.

En nuestro Dogmole Tuppowski, el mapa es de 8 por 3 pantallas, que María del Mar, la Mona que sabe Multiplicar, nos apunta que son un total de 24. Como cada pantalla, hemos dicho, mide 15×10 tiles, esto significa que nuestro mapa medirá $8 \times 15 = 120$ tiles de ancho y $3 \times 10 = 30$ tiles de alto. O sea, 120×30 tiles de 16×16 . Y eso es lo que tendremos que rellenar en el recuadro de valores importantes (el llamado RDVI):



Cuando le demos a OK, **Mappy**, que es muy cumplido y muy apañao, nos sacará un mensaje recordándonos que lo siguiente que tenemos que hacer es cargar un tileset. Y eso es, precisamente, lo que vamos a hacer. Nos vamos a `File → Import`, lo que nos abrirá un cuadro de diálogo de selección de archivo. Navegamos hasta la carpeta `gfx` de nuestro proyecto, y seleccionamos nuestro `work.png`. Si todo va bien, veremos como **Mappy** es obediente y carga nuestro tileset correctamente: lo veremos en la paleta de la derecha, que es la paleta de tiles:

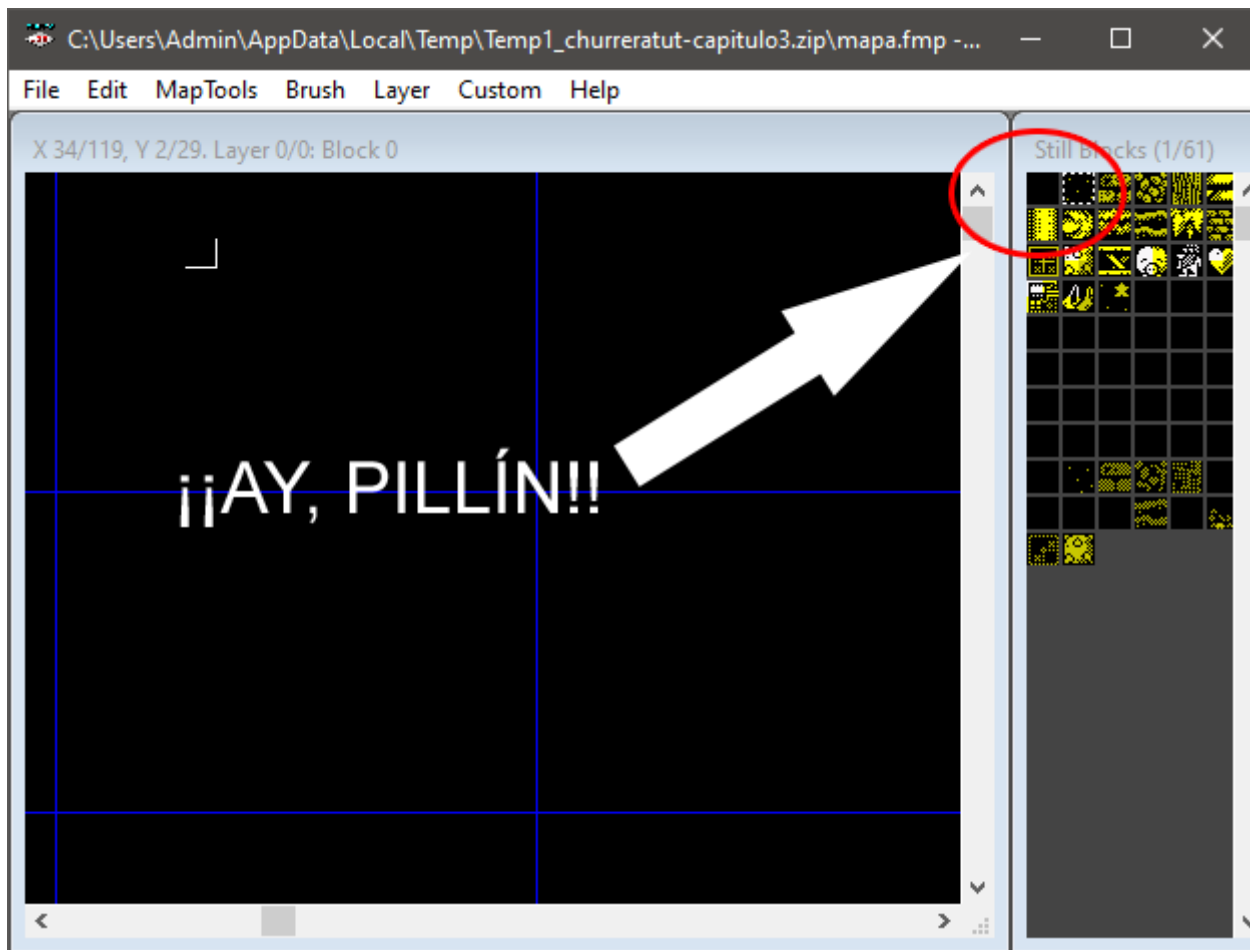


Inserto sobre PQSPDLs

Los PQSPDLs son los Programas Que Se Pasan de Listos, otro nombre para los Programas un poco tontetes. Es el caso de Mappy. Por favor, detenéos aquí que esto es **muy importante**.

Mappy necesita que el tile 0 sea un tile completamente negro. Como habrás visto, en **Dogmole** esto es así, con lo que no hay problema. El problema viene cuando el primer tile de nuestro tileset **no es completamente negro**.

Cuando intentas cargar un tileset así en **Mappy** lo que ocurrirá es que **Mappy** meterá mamporreramente su querido tile 0 negro, **desplazando todo el tileset a la derecha**:



Esto tiene **dos posibles soluciones**. La primera, y más tonta, y la que se ha venido usando de toda la vida con **MTE MK1**, consiste en **crear un tileset alternativo en el que sustituyamos el tile 0 por un tile completamente negro**. Usaremos ese tileset en **Mappy** y todo solucionado.

La segunda solución se basa en **permitir que Mappy haga sus mierdas** y meta el tile negro que tanto le gusta, pero luego tendremos que acordarnos de este detalle porque habrá que modificar `compile.bat` para añadir un parámetro extra al conversor de mapas, y a la hora de crear nuestro archivo de enemigos (en próximos capítulos). Lo mencionaremos cuando sea el momento.

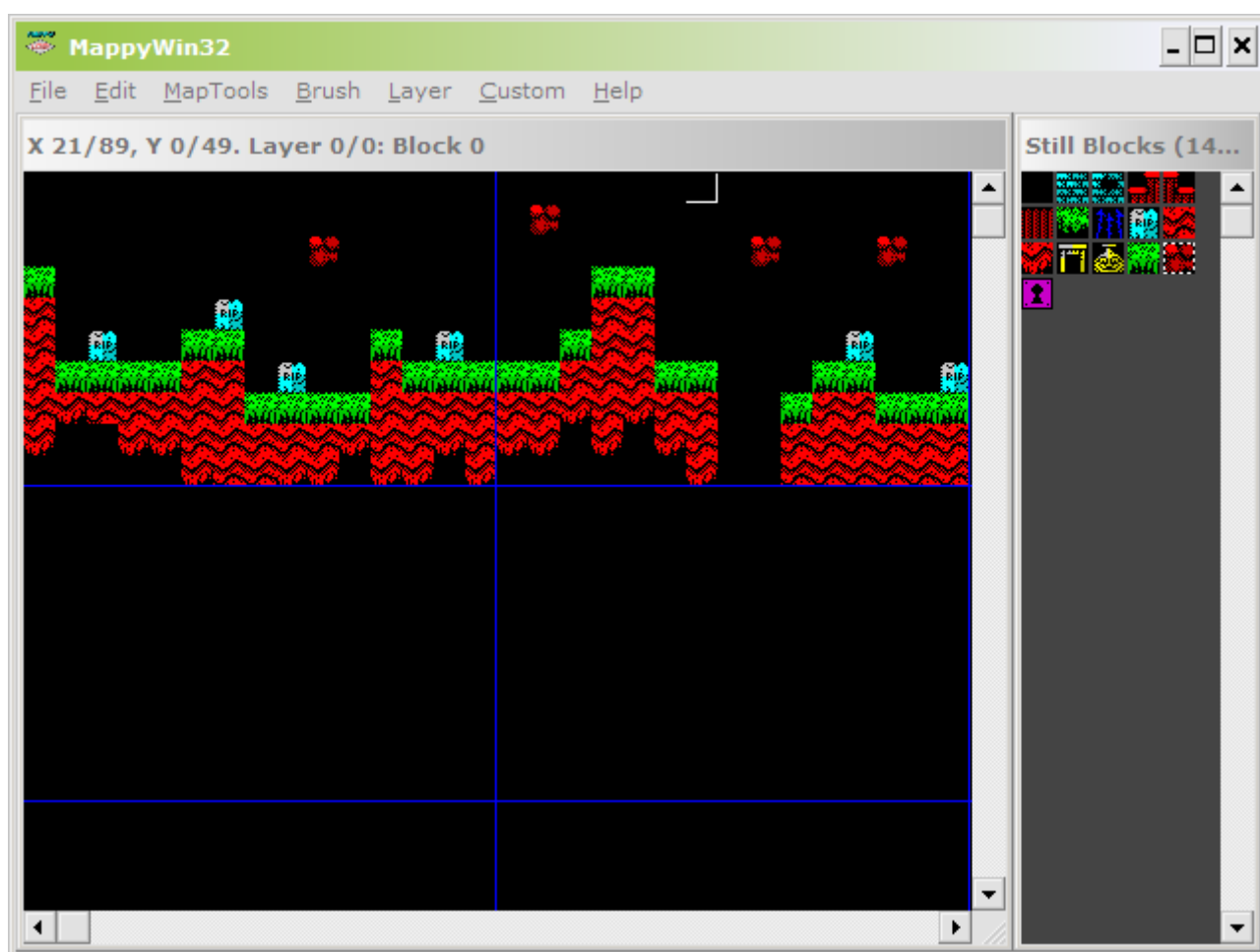
En el caso de que optemos por la segunda solución, podemos llenar todo el mapa de nuestro tile 0 (que para mappy será el 1) usando `Custom → Tile Replace`, introduciendo `0, 1` y pulsando `OK`.

Guías

Ahora sólo queda una cosa más por hacer antes de empezar: necesitamos una ayuda para saber dónde empieza y termina cada pantalla, ya que los bordes de cada pantalla tienen que ser consistentes con los de las pantallas colindantes: si hay un obstáculo al borde derecho de una pantalla, deberá haber otro obstáculo al borde izquierdo de la pantalla que esté a su derecha. A veces se nos olvida, como estaréis pensando los que nos sigáis normalmente: la mayoría de los bugs tradicionales de los Mojon Twins tiene que ver con que no hemos hecho bien el mapa. No nos tomes como ejemplo y fíjate bien en los bordes 😊

Coloquemos esas guías. Selecciona `MapTools` → `Dividers` para abrir el cuadro de diálogo de las guías. Marca la casilla `Enable Dividers` y rellena `Pixel gap x` y `Pixel gap y` con los valores 240 y 160, respectivamente, que son las dimensiones en píxeles de nuestras pantallas (esto lo sabemos de nuevo gracias a María del Mar, que ha calculado que $15 \times 16 = 240$ y $10 \times 16 = 160$). Pulsa OK y verás como el área de trabajo se divide en rectángulos con unas guías azules. Sí, lo has adivinado: cada rectángulo es una pantalla de juego. Eres un puto crack. Ídolo. Mastodonte. ¡Ya estamos preparados para empezar a trabajar!

Aquí es donde hacemos el mapa. Vamos pinchando en la paleta de tiles de la derecha para seleccionar con qué tile dibujar, y luego ponemos el tile en el área de la izquierda haciendo las pantallas. Es un trabajo laborioso, lento, y, a veces, un poco coñazo. Te recomendamos que no caigas en la vagancia y hagas pantallas aburridas con amplias zonas de tiles repetidos. Intenta que tu mapa sea orgánico, irregular y variado. Las pantallas quedan mejor. También tienes que tener en cuenta que nuestro personaje tiene que poder llegar a todos los sitios. ¿Te acuerdas que al principio decidimos que haríamos que el personaje saltase alrededor de dos tiles de alto y cuatro o cinco de ancho? Hay que diseñar el mapa con eso en cuenta. Otra cosa que hay que respetar es que no debe haber sitios de “no vuelta atrás”. Eso es muy barato y muy de juego chungo de los 80. Es una caca gorda. No caigas en eso. No confundas dificultad con mal diseño. No repitas las mierdas del pasado sólo por nostalgia.



Así que, poquito a poco, vamos construyendo nuestro mapa, que nos va quedando así (puedes cargar el archivo `mapa.fmp` del paquete de materiales de este capítulo para ver el de **Dogmole**).

Recuerda ir grabando de vez en cuando en mapa en nuestro directorio MAP como `mapa.fmp` (File → Save As). Es importante que pongas siempre `mapa.fmp` a mano o hagas click en `mapa.fmp` en la lista de archivos para que se grabe en el formato FMP y no perdamos nada.

Si tu juego tiene cerrojos (tile 15) o tiles empujables (tile 14) colócalos donde quieras que salgan al empezar el güego. Por ejemplo, en esta pantalla hemos colocado un cerrojo. No te preocupes por las llaves: las colocaremos luego, cuando pongamos los enemigos y los objetos en el mapa.



Si eres, como nuestro mono Colacao, todo un coscao, habrás visto que en el mapa de **Dogmole** hay una zona del mapa (en concreto, la zona superior izquierda) que no se puede acceder. Esta zona es la que corresponde a la Universidad de Miskatonic. Si recuerdas, cuando nos inventamos la paranoia de argumento que tiene el juego dijimos que la Universidad estaría bloqueada hasta que eliminásemos a todos los Brujetes de la Religión de Petete, que Meemaid había puesto por ahí para echar un hechizo mental que la cerrase para que no pudiésemos llevar las cajas. Cuando lleguemos al scripting veremos cómo añadir código para eliminar ese obstáculo durante el juego, cuando se detecte que los Brujetes de la Religión de Petete están todos muertos (es una cosa que podemos hacer con el scripting: modificar el mapa al vuelo). Por ahora, simplemente colocamos un piedra ahí y nos olvidamos:



Otra cosa con la que tienes que tener cuidado es con los pinchos: debe haber una forma de salir de cualquier foso con pinchos. Cuando el muñeco se rebota con los pinchos salta un poquito, pero no mucho, así que no los coloques en fosos profundos. Fíjate como están puestos en nuestros güegos.

Exportando nuestro mapa

Cuando hayamos terminado con nuestro mapa (o no, no pasa nada si montamos el juego con un cachito del mapa nada más, para ir probando), llegará el momento de exportarlo como tipo `MAP` y convertirlo para incluirlo en nuestro güego.

Nos vamos a `File → Save As` y lo grabamos como `mapa.map` en nuestro directorio `/map`. Sí, tienes que escribir `mapa.map`. Cuando lo hayas hecho, te vas de nuevo a `File → Save As` y lo grabas como `mapa.fmp` de nuevo. Escribiendo letra a letra `mapa.fmp`. Hazme caso. Hazlo, en serio. Que luego perder cosas porque se te olvidó grabar el `FMP` después de un cambio rápido es un coñazo. Que lo tenemos más que estudiao esto. Que nos ha pasado mil veces. En serio. (en rigor, se puede recuperar el `FPM` a partir del `MAP`, pero tú ten cuidado).

Convirtiendo nuestro mapa a código

Para esto se usa otra de las utilidades de **MTE MK1**: `mapcnv`. Esta utilidad pilla archivos `MAP` de **Mappy** y los divide en pantallas (para que el motor pueda construirlas de forma más sencilla, ahorrando así tiempo y espacio). Además, si estamos usando tilesets de 16 tiles, empaqueta los tiles como hemos explicado (2 por cada byte).

La llamada a `mapcnv` está también incluida en `dev/compile.bat` pero tendremos que tunearla para adecuarla a nuestro juego, por lo que conviene saberse bien los parámetros que espera. Al igual que `ts2bin`, `mapcnv` te los chiva si lo ejecutas desde la ventana de línea de comandos sin especificar parámetros:

```
$ ..\utils\mapcnv.exe
** USO **
MapCnv archivo.map archivo.h ancho_mapa alto_mapa ancho_pantalla alto_pantalla tile_cerrojo

- archivo.map : Archivo de entrada exportado con mappy en formato raw.
- archivo.h : Archivo de salida
- ancho_mapa : Ancho del mapa en pantallas.
- alto_mapa : Alto del mapa en pantallas.
- ancho_pantalla : Ancho de la pantalla en tiles.
- alto_pantalla : Alto de la pantalla en tiles.
- tile_cerrojo : Nº del tile que representa el cerrojo.
- packed : Escribe esta opción para mapas de la churrera de 16 tiles.
- fixmappy : Escribe esta opción para arreglar lo del tile 0 no negro
```

Por ejemplo, para un mapa de 6x5 pantallas para MTE MK1:

```
MapCnv mapa.map mapa.h 6 5 15 10 15 packed
```

Expliquémoslos uno a uno:

1. `archivo.map` es el archivo de entrada con nuestro mapa recién exportado de **Mappy**.
2. `archivo.h` es el nombre de archivo para la salida.
3. `ancho_mapa` es el ancho del mapa en pantallas. En el caso de **Dogmole**, 8.

4. `alto_mapa` es el alto del mapa en pantallas. En el caso de **Dogmole**, 3.
5. `ancho_pant` es el ancho de cada pantalla en tiles. Para **MTE MK1**, siempre es 15.
6. `alto_pant` es el alto de cada pantalla en tiles. Para **MTE MK1**, siempre es 10.
7. `tile_cerrojo` es el número de tile que hace de cerrojo. Para **MTE MK1** siempre ha de ser el tile número 15. Si tu juego no usa cerrojos, pon aquí un valor fuera de rango como 99. Por ejemplo, **Zombie Calavera** no usa cerrojos, así que pusimos aquí un 99 al convertir su mapa. Nosotros sí tenemos cerrojos en Dogmole, así que pondremos un 15.
8. `packed` se pone, tal cual, si nuestro tileset es de 16 tiles. Si usamos un tileset de 48 tiles, simplemente no ponemos nada. Si hacemos esto, habrá que configurar igualmente el motor activando `UNPACKED_MAP`, aunque eso ya lo veremos en el capítulo 7.
9. `fixmappy` lo pondremos si nuestro tileset no tiene un primer tile todo a negro y pasamos de hacer un tileset especial, dejando que mappy lo insertase. Así `mapcnv` lo tiene en cuenta y hace sus fullerias.

Por tanto, para convertir el mapa de **Dogmole**, tendremos que ejecutar `mapcnv` así:

```
..\utils\mapcnv.exe ..\map\mapa.map assets\mapa.h 8 3 15 10 15 packed > nul
```

Es el momento de editar `dev/compile.bat`, ubicar la llamada a `mapcnv`, y modificar los valores del proyecto por defecto por los nuestros.

Aunque no es necesario, podemos verlo en acción ejecutando esa línea desde dentro del directorio `dev/` de nuestro proyecto. Con esto, tras un misterioso y mágico proceso, obtendremos un archivo `mapa.h` listo para que el motor lo use.

```
$ ..\utils\mapcnv.exe ..\map\mapa.map assets\mapa.h 8 3 15 10 15 packed
Se escribió mapa.h con 24 pantallas empaquetadas (1800 bytes).
Se encontraron 0 cerrojos.
```

Si abris este `mapa.h` con el editor de textos, veréis un montón de números en un array de C: Ese es nuestro mapa. Justo abajo, están definidos los cerrojos en otra estructura. Como verás, habrá tantos cerrojos definidos como hayamos puesto en el mapa. Si esto no es así, es que has hecho algo mal. ¡Repasa todos los pasos!

Perfecto, todo guay, todo correcto.

Muy bien. Hemos terminado por hoy. En el próximo capítulo pintaremos los sprites del güego: el prota, los malos, las plataformas... Veremos cómo hacer un spriteset y como convertirlo para usarlo en nuestro güego.

Mientras tanto, deberías practicar. Algo que te recomendamos hacer, si es que aún no se te ha ocurrido, es descargar los paquetes de código fuente de nuestros güegos y echarle un vistazo a los

mapas. Abre los archivos FMP con **Mappy** y fíjate cómo están hechas las cosas.

Capítulo 4: Sprites

Antes que nada, bájate el paquete de materiales correspondiente a este capítulo pulsando en este enlace:

[Material del capítulo 4](#)

¿Qué son los sprites?

A estas alturas del cuento, si no sabes lo que es un sprite, mal vamos... Y no, no voy a hacer la típica gracia de la bebida carbonatada, (jajá, jijí, qué graciosos somos, los reyes de la fiesta, tralarí, alocados, divertidos, amigos de nuestros amigos). Sin embargo, como este capítulo va sobre sprites, habrá que empezar explicando que son. Por si no lo sabes, se trata de una de las leyes de los tutoriales de videogüegos. No importa de qué nivel sean o sobre qué sistema traten: es obligatorio explicar qué es un sprite. Lo manda su eminencia ilustrísima Paco Perales, el papa de los tutoriales.

Veamos: el concepto de sprite, realmente, no tiene absolutamente ningún sentido en sistemas como el Spectrum, en el que la salida gráfica se limita a una matriz de píxels. Absolutamente ningún sentido. Sin embargo, se emplea por analogía. Elaboremos: tradicionalmente, un chip de gráficos diseñado para funcionar en una máquina de güegos manejaba dos entes, principalmente: el fondo y un cierto número (limitado) de sprites. Los sprites no eran más que puñados de píxels, normalmente cuadrados o rectangulares, que el procesador gráfico se encargaba de componer sobre el fondo a la hora de enviar la imagen al monitor o la TV. Esto es: se trataba de objetos completamente ajenos al fondo y que, por tanto, podías mover sin afectar a éste. La CPU del sistema no tenía más que decirle al procesador gráfico dónde estaba cada sprite y olvidarse, ya que dicho procesador gráfico era el que se encargaba de construir la imagen enviando al monitor píxels de sprite en vez de píxels de fondo cuando era necesario. Es como si los muñequitos no estuviesen realmente ahí, de ahí su nombre: la palabra "sprite" significa "hadas" o "duendecillos".



Casi todas las videoconsolas de 8 y 16 bits (las de Nintendo y SEGA por lo menos, que los sistemas de Atari eran raros de narices y por eso se metían contigo en el colegio, por tener un Atari 5200), el

MSX y el Commodore 64, entre otros sistemas, tienen un procesador de gráficos de verdad que hace fondos y sprites.

En ordenadores como el Spectrum o el Amstrad CPC, o cualquier PC, el concepto de sprite no tiene sentido. ¿Por qué? Pues porque el hardware sólo maneja una matriz de píxels, lo que sería equivalente a sólo manejar el fondo. En estos casos, los muñequitos tiene que pintarlos la CPU: tiene que encargarse de sustituir ciertos píxels del fondo por píxels de los muñequitos. Además, el programador debe currarse algún método para mover los muñequitos: debe poder restaurar la pantalla a como estaba antes de pintar el muñequito y redibujarlo en la nueva posición. Sin embargo, por analogía, a estos muñequitos se les llama sprites.

Nosotros les llamamos sprites, y eso que a veces somos unos puristas de la hostia (purista significa frikazo).

Sprites en MTE MK1

MTE MK1 maneja cuatro sprites de 16×16 píxels para bicharracos (incluido el personaje que maneja el jugador) y generalmente hasta tres sprites para proyectiles (en los güegos de matar). Los sprites de los bicharracos se dibujan usando los gráficos que podemos definir en lo que se conoce como el spriteset. Dicho spriteset contiene 16 gráficos, de los cuales 8 se usan para animar al personaje principal y 8 para animar a los 4 tipos de enemigos (2 para cada uno, si se te dan bien las matemáticas). Un spriteset tiene esta pinta (este es el spriteset del Dogmole):



Como verás, siempre hay un gráfico de un muñeco y al lado hay una cosa rara justo a su derecha. Vamos a explicar qué es esa cosa rara antes de seguir. En primer lugar, no se llama cosa rara. Se llama máscara. Sí, ya sé que no se parece a una máscara ni de coña, pero se llama así. Tampoco los ratones de ordenador parecen ratones ni el gráfico de Horace parece un niño y no he visto a nadie quejarse todavía. La máscara se usa para decirle a la biblioteca gráfica (en este caso, nuestra **splib2** modificada), que es la que se encarga de mover los sprites (recordad que “esto es Spectrum y aquí hay que mamar”: no hay chip gráfico que haga estas cosas), qué píxels del gráfico son *transparentes*, esto es, qué píxels del gráfico NO deben sustituir los píxels del fondo.

Si no hubiera dicha máscara, todos los sprites serían cuadrados o rectangulares, y eso queda bastante feo.

Nosotros hemos ordenado nuestro spriteset de forma que cada gráfico tiene su máscara correspondiente justo a la derecha. Si te fijas, las máscaras tienen píxels negros en las zonas donde debe verse el gráfico correspondiente, y píxels de color en las zonas donde debe verse el fondo. Es como si definiésemos la silueta.

¿Por qué son necesarias las máscaras? Si eres un poco perspicaz lo habrás adivinado: en Spectrum los gráficos son de 1 bit de profundidad, lo que significa que cada píxel se representa usando un solo bit. Eso se traduce en que sólo tenemos dos posibles valores para los píxels: encendido o apagado (1 o 0). Para poder especificar qué píxels son transparentes necesitaríamos un tercer valor, y eso no es posible (¡es lo que tiene el binario!). Por eso necesitamos una estructura separada para almacenar esta información ¡la máscara! ¡las máscaras molan!.

Construyendo nuestro spriteset

Antes de construir el spriteset es muy importante saber qué tipo de vista tendrá nuestro güego: lateral o genital. Supongo que, llegados a este punto, es algo que ya tenemos decidido (vaya, si ya hemos hecho el mapa). El orden de los gráficos en el spriteset depende del tipo de vista de nuestro güego.

Spritesets de vista lateral

Para los güegos de vista lateral, los 16 gráficos de 16×16 (acompañados por sus máscaras) que componen el spriteset tienen que tener este orden:

#	Qué tiene
0	Personaje principal, derecha, andando, frame 1
1	Personaje principal, derecha, andando, frame 2, o parado
2	Personaje principal, derecha, andando, frame 3
3	Personaje principal, derecha, en el aire
4	Personaje principal, izquierda, andando, frame 1
5	Personaje principal, izquierda, andando, frame 2, o parado
6	Personaje principal, izquierda, andando, frame 3
7	Personaje principal, izquierda, en el aire
8	Enemigo tipo 1, frame 1
9	Enemigo tipo 1, frame 2
10	Enemigo tipo 2, frame 1
11	Enemigo tipo 2, frame 2
12	Enemigo tipo 3, frame 1
13	Enemigo tipo 3, frame 2

#	Qué tiene
14	Plataforma móvil, frame 1
15	Plataforma móvil, frame 2

Como vemos, los ocho primeros gráficos sirven para animar al personaje principal: cuatro para cuando mira a la derecha, y otros cuatro para cuando mira a la izquierda.

Tenemos tres animaciones básicas para el personaje: parado, andando, y saltando/cayendo:

1. **Parado:** La primera es cuando el personaje está parado (como su propio nombre indica). Parado significa que no se está moviendo por sí mismo (si lo mueve un ente externo sigue estando "parado"). Cuando el personaje está parado, el motor lo dibuja usando el frame 2 (gráfico número 1 si mira a la derecha o 5 si mira a la izquierda).
2. **Andando:** Es cuando el personaje se desplaza lateralmente por encima de una plataforma. En ese caso se realiza una animación de cuatro pasos usando los frames 1, 2, 3, 2, ... en ese orden (gráficos 0, 1, 2, 1... si miramos a la derecha o 4, 5, 6, 5... si miramos a la izquierda). A la hora de dibujar, el personaje deberá tener ambos pies en el suelo para el frame 2, y las piernas extendidas (con la izquierda o la derecha delante) en los frames 1 y 3. Por eso usamos el frame 2 en la animación "parado".
3. **Saltando/Cayendo:** Es cuando el personaje salta o cae (joder, ¡menos mal que lo he aclarado!). Entonces el motor dibuja el frame "saltando" (gráfico número 3 si mira a la derecha o número 7 si mira a la izquierda).

Los siguientes seis gráficos se usan para representar a los enemigos. Los enemigos pueden ser de tres tipos, y cada uno tiene dos frames de animación.

Para acabar, los dos últimos gráficos se usan para las plataformas móviles, que también tienen dos frames de animación. Las plataformas móviles son, precisamente, y como su nombre indica, plataformas que se mueven. El personaje principal podrá subirse en ellas para desplazarse. Para dibujar los gráficos tenemos que cuidar que la superficie sobre la que se debe posar el personaje principal debe tocar el borde superior del gráfico.

Para que quede claro, veamos otro ejemplo



El spriteset de arriba corresponde a **Cheril Perils**. Como vemos, los ocho primeros gráficos son los correspondientes a Cheril, primero mirando a la derecha y luego mirando a la izquierda. Luego tenemos los tres enemigos que vemos en el juego, y al final la plataforma móvil. Fíjate bien en el tema de la animación de andar, imagínate pasar del frame 1 al 2, del 2 al 3, del 3 al 2, y del 2 al 1.

Mira el gráfico e imagínatelo en tu cabeza. ¿lo ves? ¿ves como mueve las patitas? Ping, pong, ping, pong... Fíjate también como el frame 2 es el que mejor queda para cuando el muñeco está parado.

Spritesets de vista genital

Para los güegos de vista genital, los 16 gráficos del spriteset tienen que tener este orden:

#	Qué tiene
0	Personaje principal, derecha, andando, frame 1
1	Personaje principal, derecha, andando, frame 2
2	Personaje principal, izquierda, andando, frame 1
3	Personaje principal, izquierda, andando, frame 2
4	Personaje principal, arriba, andando, frame 1
5	Personaje principal, arriba, andando, frame 2
6	Personaje principal, abajo, andando, frame 1
7	Personaje principal, abajo, andando, frame 2
8	Enemigo tipo 1, frame 1
9	Enemigo tipo 1, frame 2
10	Enemigo tipo 2, frame 1
11	Enemigo tipo 2, frame 2
12	Enemigo tipo 3, frame 1
13	Enemigo tipo 3, frame 2
14	Enemigo tipo 4, frame 1
15	Enemigo tipo 4, frame 2

De nuevo, los ocho primeros gráficos sirven para animar al personaje principal, pero esta vez el tema es más sencillo: como tenemos cuatro direcciones en las que el personaje principal puede moverse, sólo disponemos de dos frames para cada dirección.

Los ocho gráficos restantes corresponden a cuatro tipo de enemigos, ya que en un güego de vista genital no hay plataforma móvil que valga.

algunos ejemplos:



Este es el spriteset de nuestra genial conversión de **el Hobbit** (uno de nuestros grandes honores: quedar los últimos en un concurso. Es casi mejor que quedar los primeros). Fíjate como el tema cambia: arriba tenemos 8 gráficos con 2 frames para cada dirección, y abajo tenemos cuatro tipos de enemigos en lugar de tres más las plataformas.



Este otro spriteset es el de **Mega Meghan**. Aquí volvemos a tener a la prota (Meghan, la asesina malvada) en las cuatro direcciones posibles con dos frames de animación para cada una en la tira superior, y a cuatro mujeres encueras (que hacen de enemigas) en la tira inferior.

Fíjate como, en este caso, nuestra vista genital la hemos diseñado con cierta perspectiva: cuando los muñecos van para arriba se dan la vuelta y cuando van para abajo miran de frente. Eso queda super chuli.

Dibujando nuestro spriteset

Nada más fácil que crear un nuevo archivo de 256×32 píxels, activar la rejilla para no salirnos, y dibujar los gráficos. Cuando acabemos, lo guardamos como `sprites.png` en `/gfx`. Aquí tendremos que respetar una regla muy sencilla (o dos, según se mire). Para pintar gráficos y máscaras usaremos dos colores:

En los gráficos, el **negro PURO** es el color del PAPER, y el otro color (el que queramos, es buena idea usar el blanco) es el color del INK. Recuerda que los sprites tomarán el color de los tiles que tienen de fondo al moverse por la pantalla.

En las máscaras, el **negro PURO** es la parte del gráfico que debe permanecer sólida, y cualquier otro color sirve para definir las partes transparentes (a través de las que se ve el fondo).

En nuestros ejemplos verás que usamos un color diferente (además del negro) en gráficos y máscaras, pero es más que nada por claridad. Puedes usar los colores que quieras. Yo, personalmente, uso colores diferentes porque suelo pintar la máscara y el sprite en la misma casilla de 16×16, para luego seleccionar sólo los píxeles del “color máscara” y moverlos a la casilla siguiente. Es un truco que puedes aplicar si eres mañoso con tu editor de gráficos.

Una vez que tengamos todos nuestros gráficos y sus máscaras dibujaditos en el archivo del spriteset, lo guardamos como `sprites.png` en `gfx` y estamos listos para convertirlos en código C directamente usable por **MTE MK1**.

Por cierto, vuelve a asegurarte que el negro que has usado es **negro PURO**. Todos los píxeles negros deben ser `RGB = (0, 0, 0)`.

Convirtiendo nuestro spriteset

Al igual que con todas las conversiones, la de los sprites también está incluida en `compile.bat` y no deberás preocuparte de nada más que de poner el archivo con los sprites `sprites.png` en `/gfx`.

Sin embargo, para seguir con la tradición, veamos como funciona el conversor, que en este caso se llama `sprcnv` y está, como los demás, en `/utils`. Si lo ejecutamos sin parámetros también nos los chiva:

```
$ ../utils/sprcnv.exe
** USO **
sprcnv archivo.png archivo.h [nomask]
```

Convierte un Spriteset de 16 sprites

Este es mucho más sencillo. Toma dos parámetros obligatorios: el archivo de entrada (nuestro `sprites.png`), un archivo de salida, que para **MTE MK1** debe ser `sprites.h` en `/dev/assets`, y el parámetro `nomask`, que es optativo, y que generará los sprites sin máscara (algo que solo hemos usado en *Zombie Calavera* y que por ahora dejaremos "ahí").

Si abres `/dev/compile.bat` verás que el valor de los parámetros son, precisamente, los que hemos mencionado arriba.

Jodó, ¿ya?

Sí, tío o tía. El tema de los sprites era sencillo y tenía muy poca chicha... O al menos eso creo. Si crees que algo no queda claro ya sabes qué hacer: te aguantas. No, que es coña. Si crees que algo no queda claro simplemente preguntanos. Pero tienes que poner voz de niña repipi.

En el próximo capítulo vamos a explicar el tema de las pantallas fijas: el título, el marco del juego, y la pantalla del final. Hasta entonces, ¡a hacer sprites!

Capítulo 5: Terminando los gráficos (por ahora)

Antes que nada, bájate el paquete de materiales correspondiente a este capítulo pulsando en este enlace:

[Material del capítulo 5](#)

¿Qué nos falta?

Ya queda poco que dibujar. En este capítulo vamos a explicar dos cosas: por un lado, cómo cambiar los sprites extra (explosión y bala). Por otro lado, veremos cómo hacer, convertir, comprimir e incluir el marco del juego, la pantalla de carga, la de título, y la pantalla del final.

Sprites Extra

Además del spriteset que vimos en el anterior capítulo, **MTE MK1** utiliza algunos gráficos más si se activan algunas opciones. En concreto, si los enemigos pueden morir necesitaremos un gráfico de explosión, y si el personaje dispara necesitaremos un gráfico de bala. El paquete trae unos por defecto que se pueden cambiar muy fácilmente (de nuevo, aquellos de vosotros que hayáis catado **MTE MK1** con anterioridad agradeceréis mucho este cambio).

Huelga decir que si tu güego no tiene enemigos pisables ni balas, o si te valen las que hay, puedes pasar de esta sección como de la cacota.

Para cambiar estos gráficos únicamente tenéis que sustituir los archivos `sprites_extra.png` y `sprites_bullet.png` en `/gfx`. El script `compile.bat` se encarga automáticamente de llamar a los conversores necesarios para obtener los gráficos en el formato que necesita el motor.

Cambiando la explosión

La explosión es un sprite de 16×16. Para cambiarla tendremos que sustituir `/gfx/sprites_extra.png` por un nuevo archivo. Si recordáis como estaba construido el spriteset del juego, este archivo es parecido pero solo trae un gráfico y su máscara. Por tanto, deberá tratarse de una imagen de 32×16 que tenga este aspecto (el archivo está incluido en el paquete de archivos de este capítulo):



Una vez hecho, lo guardamos, como hemos dicho, como `\gfx\sprites_extra.png`, sustituyendo al que viene por defecto.

Cambiando el disparo

El disparo original es una bolita de centrada en un cuadro de c. Pusimos una bolita porque usamos el mismo gráfico para disparos en todas las direcciones, así que una forma “orientada” no nos vale. Por lo tanto, vuestra nueva bala debe ser igualmente un gráfico de 8×8 que valga para todas las direcciones.

De forma muy parecida al sprite de la explosión tendremos que crear una imagen para definir la bala y su máscara. Esta vez, al ser el gráfico de 8×8, la imagen deberá ser de 16×8 para incluir la máscara; algo así:



Una vez hecho, lo guardamos, como hemos dicho, como `\gfx\sprites_bullet.png`, sustituyendo al que viene por defecto.

Como **Dogmole** no utiliza balas, no hay ningún archivo nuevo de bala en el paquetito de archivos de este capítulo.

Pantallas fijas

Bueno, vamos con el tema de las pantallas fijas. Básicamente, los güegos de **MTE MK1** llevan tres pantallas fijas: la pantalla de título, que es la que muestra, además, el menú (para seleccionar el tipo de control y empezar a jugar), la pantalla del marco de juego, donde se ubicarán los marcadores de vidas, objetos, y cosas así, y la pantalla del final, que será la que se mostrará una vez que el jugador haya cumplido el objetivo encomendado. Además tenemos una pantalla de carga que se muestra mientras te tomas el bocata de nocilla.

Para ahorrar memoria, además, se ofrece la posibilidad de que la pantalla de título y la pantalla con el marco del juego sean la misma, con lo que ahorraremos bastante, y que será la opción que tomaremos para nuestro Dogmole.

Otra cosa que no quiero dejar de mencionar es que las pantallas se almacenan en el güego en formato comprimido. El tipo de compresión empleado (como casi todas las compresiones) funciona mejor cuanto más sencillas sean las imagenes. O sea: cuanta más repetición y/o menos cosas haya en las pantallas, menos ocuparán al final. Ten esto muy en cuenta. Si te ves apurado de memoria, una forma de ahorrar que funciona muy bien es hacer que tus pantallas fijas sean menos complejas.

La pantalla de título

Como ya hemos mencionado, se trata de la pantalla que se muestra con el título del güego y las opciones de control. La selección de control es fija: si el jugador pulsa 1 seleccionará el control por teclado. Si pulsa 2, elegirá Kempston, y si pulsa 3 es porque quiere jugar con un joystick de la norma Sinclair (Interface 2, puerto 1). ¿Qué quiere decir esto? Pues que tendremos que dibujar una pantalla que, además del título del güego, muestre estras tres opciones. Por ejemplo, algo así:



Cuando hagas la tuya, guárdala como `title.png` en el directorio `\gfx`.

La pantalla del marco de juego

Aquí sí que hay más chicha. La pantalla del marco del juego debe reservar varias zonas importantes y luego separarlas con un adorno e indicadores. Las zonas que tenemos que reservar son:

1. El **área de juego**, que es donde pasan las cosas. Como habrás adivinado, debe ser igual de grande que cada una de nuestras pantallas. Si recuerdas, las pantallas son de 15×10 tiles y, por tanto, ocupan 240×160 píxels. Debes reservar una zona de ese tamaño como área principal de juego.
2. El **marcador de vidas/energía/vitalidad/lo que sea**: son dos dígitos que se dibujarán con la fuente que definiste cuando hicimos el tileset. Debes ubicarlo en alguna parte del marco, añadiendo algún gráfico que le de significado. Nosotros solemos poner un dibujito del protagonista y una "x", como verás en los ejemplos.
3. El **marcador de llaves** (si usas llaves en tu güego). Tiene las mismas características que el marcador de vidas.
4. El **marcador de objetos** (si usas objetos en tu güego). Idem de idem, para los objetos.
5. El **marcador de enemigos matados** (si se pueden matar enemigos y es importante contarlos). Pues lo mismo.

Todas estas cosas las podemos colocar donde nos venga en gana, siempre que estén alineadas a carácter. Además, deberemos apuntar la posición de cada una de estas cosas (en coordenadas de

carácter, esto es, 0 a 31 para la X y 0 a 23 para la Y), porque luego habrá que indicarla en la configuración de **MTE MK1** (como veremos dentro de un par de capítulos).

Veámoslo con un ejemplo. Esta es la pantalla del marco de juego de **Lala Prologue** (no beta). Veamos los diferentes elementos. En lala tenemos llaves y además hay que recoger objetos, por lo que tendremos que poner esas dos cosas, junto con el marcador de energía, en el marco, tal y como vemos:



1. El área de juego, como vemos, empieza en las coordenadas $x = 1$, $y = 2$, o sea, en (1, 2).
2. El espacio que hemos dejado para el marcador de energía está en (4, 0).
3. El marcador de objetos aparece tal que en (11, 0).
4. Por último, el marcador de llaves está en (18, 0).

Como digo, hay que apuntar todos esos valores porque habrá que usarlos a la hora de construir la configuración de nuestro güego, dentro de un par de capítulos.

Cuando tengas la tuya, se graba como `marco.png` en el directorio `\gfx`.

Pantalla de título y marco combinados

Desde muy temprano empezamos a hacer esto porque nos ayudaba a ahorrar un montón de memoria. El tema es sencillo: a una pantalla del marco de juego le añades el título y las opciones del menú dentro de la zona reservada para el área principal. La verdad es que queda bastante bien, y, reiteramos, podrás ahorrar bastante memoria.

Esto es, de hecho, lo que vamos a usar en nuestro **Dogmole**, tal y como vemos aquí:



Como vemos, por un lado tenemos el título del güego y las opciones de control, como explicamos cuando hablamos de la pantalla de título. Por otro lado, tenemos el marcador del juego. El área de juego, como se verá, se mostrará sobre el espacio que ocupa el título y las opciones de control:

1. El área de juego estará colocada en las coordenadas (1, 0).
2. El contador de enemigos eliminados (los hechiceros) se dibujará en (12, 21).
3. El contador de cajas (o sea, los objetos) se dibujará en (17, 21).
4. El contador de vidas se dibujará en (22, 21)
5. Por último, el contador de llaves lo tendremos en (27, 21).

En este caso, la pantalla de título/marco combinados se guarda como `title.png` en `\gfx`. No habrá archivo `marco.png` para los juegos que combinen título y marco en la misma pantalla, como es nuestro caso con **Dogmole**.

La pantalla del final

Es la pantalla que se muestra cuando el jugador se acaba el güego con éxito. Aquí no hay ninguna restricción: puedes dibujar lo que quieras. Si tiene gracia queda mejor, pero si te gusta hacer güegos serios a lo mejor no pega. Esta es la pantalla del final de Dogmole (¡¡SPOILER!!):



Cuando la tengamos la grabaremos como `ending.png` en el directorio `/gfx` .

La pantalla de carga

La pantalla de carga es la que se muestra mientras... bueno, ya tu sabeh. Debe guardarse igualmente en `/gfx` y llamarse `loading.png` .

Convirtiendo las pantallas a formato Spectrum y comprimiendo

Todo el proceso está automatizado en `compile.bat` pero vamos a ver como es y de qué se trata, como viene siendo costumbre.

La conversión de imagenes PNG a formato de spectrum (que se suele llamar SCR) se realiza mediante otra utilidad del toolchain, `png2scr` . De nuevo, si la ejecutamos desde la ventana de linea de comandos sin parámetros nos los chiva:

```
$ ../utils/png2scr.exe
png2scr v0.2.20191119 ~ usage:

$ png2scr.exe img.png img.scr [thirds]
  * img.png is a 256x192 image, speccy formatted
  * img.scr is the converted, output file
  * [thirds] = 1, 2, 3; don't include for full image+attrs
```

En este caso deberemos pasar como parámetros el nombre de archivo de entrada en formato PNG (que debe ser de 256x192 y respetar las restricciones del Spectrum) seguido del nombre de archivo de salida. Ambos parámetros pueden incluir ruta si es necesario. El tercer parámetro no lo vamos a necesitar en **MTE_MK1** a menos que estemos haciendo fullerías avanzadas que no entran en el tutorial.

Si abres `compile.bat` en tu editor de textos verás toda una sección dedicada a convertir al formato Spectrum las cuatro pantallas fijas (carga, título, marco y final). Fíjate como los nombres de archivo son los que hemos mencionado y que todas deben ubicarse en `/gfx` :

```
..\utils\png2scr.exe ..\gfx\title.png ..\gfx\title.scr > nul
..\utils\png2scr.exe ..\gfx\marco.png ..\gfx\marco.scr > nul
..\utils\png2scr.exe ..\gfx\ending.png ..\gfx\ending.scr > nul
..\utils\png2scr.exe ..\gfx\loading.png loading.bin > nul
```

Comprimiendo las pantallas

Como te habrás dado cuenta, los 6912 bytes que ocupa cada pantalla por tres (o por dos) son un pasote, por lo que habrá que comprimirlas. Para eso usaremos el compresor `apultra.exe` que comprime un binario en formato aplib. Tranqui, lo hemos incluido en la carpeta `utils`.

De nuevo, `compile.bat` se encargará de comprimir las pantallas de título, marco y final (la de carga va a pelo en el `.tap` ya que usaremos un sencillito cargador BASIC de toda la vida):

```
..\utils\apultra.exe ..\gfx\title.scr title.bin > nul
..\utils\apultra.exe ..\gfx\marco.scr marco.bin > nul
..\utils\apultra.exe ..\gfx\ending.scr ending.bin > nul
```

Y ya hemos terminado

Jo ¿ya? Vaya capítulo aburrido. Lo sé. Pero bueno, era necesario. En el próximo capítulo aprenderemos a colocar los enemigos, los objetos y las llaves en el mapa del güego. Y pronto, muy pronto, podremos compilar por primera vez para empezar a verlo todo en movimiento.

Capítulo 6: Colocando cosas

Antes que nada, bájate el paquete de materiales correspondiente a este capítulo pulsando en este enlace:

[Material del capítulo 6](#)

¿Ahora qué?

Ahora es cuando colocamos las cosas. Es el acto equivalente a cuando llegas del supermercado cargado de bolsas y las tienes que colocar por toda la cocina. Es un proceso tedioso, pero necesario. Puedes optar no hacerlo, por supuesto, pero luego a ver quién encuentra las salchichas. O, yo qué sé, los filtros de la cafetera. ¿Alguien sigue usando filtros para cafetera?

Básicamente lo que vamos a hacer es poner en el mapa los enemigos, los objetos y las llaves. Y no, no nos referimos precisamente a hacer esto:



Cada ítem pertenecerá a una pantalla en concreto y tendrá una serie de valores que indiquen su posición, su tipo, su velocidad en el caso de que se muevan, y otras cosas por el estilo. En realidad estamos hablando de una tabla de números enorme que sería todo un coñazo crear a mano. Por eso hemos hecho una aplicación para poder completar este trabajo de forma visual.

Quien haya catado nuestros motores recordarán el infame **colocador**. A ellos queremos dedicar su sucesor, el **ponedor**, que, entre otras cosas es:

1. Más mejor.
2. Menos peor.

La herramienta no sólo es útil para hacer güegos con **MTE MK1**. Cualquier proyecto que tengáis con tiles de 16×16 píxels, para cualquier tamaño de pantalla, podría beneficiarse de esta utilidad. Sin ir más lejos, la usamos para colocar los bichos en güegos como **Uwol 2** de CPC, **Lala Prologue** de MSDOS, o el muy nefasto port de **Cheril Perils** para Megadrive.

Conceptos básicos: enemigos y hotspots

Vamos a empezar explicando algunos conceptos básicos antes de ponernos a chulear de aplicación ponedora, más que nada porque la nomenclatura que usamos suele ser menos intuitiva que el control de Uchi-Mata.



Enemigos

En primer lugar tenemos el concepto de enemigo. Un enemigo, para **MTE MK1**, son esas cosas que se mueven en la pantalla y que te matan y, además, las plataformas móviles. Sí, para **MTE MK1** las plataformas móviles son enemigos. Así que cuando hablamos de colocar enemigos, también hablamos de colocar plataformas móviles. Y cuando decimos que en la pantalla puede haber un máximo de 3 enemigos, hay que contar también las plataformas móviles. Sí, podríamos haberlos llamado *móviles*, por ejemplo, pero lo cierto es que las plataformas móviles se nos ocurrieron cuando ya teníamos empezado el motor y Amador, nuestro mono programador, no consintió cambiarles el nombre porque estaba pegao y ni con rasqueta.

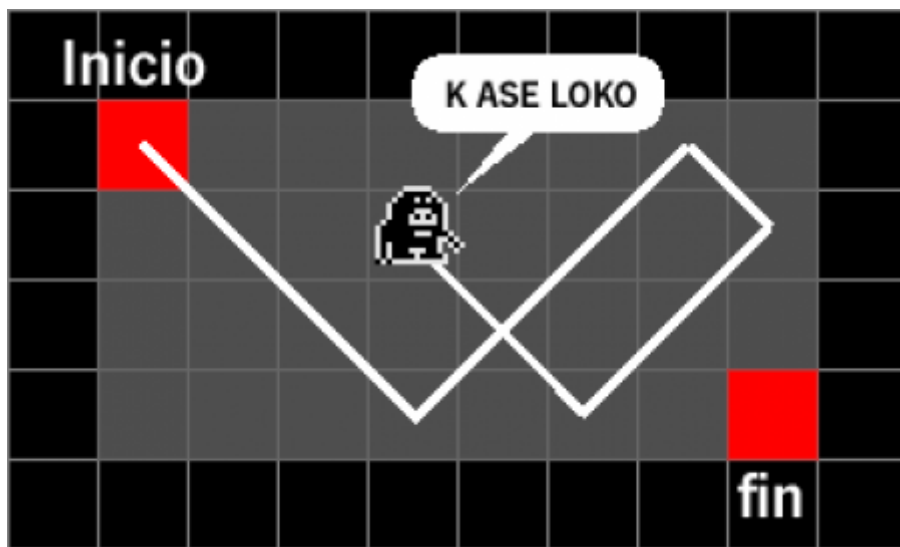
Los enemigos tienen asociados diversos valores y algo muy importante: el tipo. El tipo de enemigo define su comportamiento y, además, el gráfico con el que se pinta. Atención porque esto es un tanto confuso, sobre todo porque no se diseñó a priori y está algo parcheado (léase más arriba el asunto de las plataformas móviles):

Los enemigos de **tipos 1, 2 o 3** (y el 4 en los güegos de vista genital) describen trayectorias lineales y se dibujan con el primer, segundo o tercer (o cuarto) sprite de enemigos del tileset. Cuando hablamos de trayectorias lineales nos referimos a dos posibles casos:

1. **Trayectorias rectilíneas**, verticales u horizontales: se definen el punto de inicio y el punto de final describiendo una línea recta vertical u horizontal. El enemigo sigue esa línea imaginaria yendo y viniendo ad infinitum.



2. **Trayectorias diagonales:** estas surgieron de un *feature* del motor (efecto colateral no planado debido a un algoritmo cutre, o, lo que es lo mismo, cuando un bug te sale bien), pero las dejamos porque molan. Las hemos usado mucho. Si el punto de inicio y el del final no están en la misma fila o columna, el muñeco se mueve dentro del rectángulo que describen ambos puntos, rebotando en sus paredes y moviéndose en diagonal.



Los enemigos de **tipo 4**, cuando la vista es lateral, son **plataformas móviles**. Se comportan exactamente igual que los enemigos lineales, pero con una limitación: aunque podemos definirles una trayectoria diagonal, no garantizamos que el funcionamiento sea el correcto en todos los casos. Por tanto, sólo pueden usarse trayectorias verticales u horizontales.

Los enemigos de **tipo 6** son **voladores perseguidores**. Simplemente persiguen al personaje principal por toda la pantalla y son la peste. Todos se pintan usando el gráfico 1, 2, 3 o 4 del tileset según se configure en el motor. Son como fantasmas porque pueden traspasar el escenario. Se pueden configurar normales o de tipo "HOMING", que aparecen en el sitio donde se colocan y te persiguen si te acercas. Si te alejas vuelven a su sitio. Como los celebros de **Goku Mal**.

Los enemigos de **tipo 7** son los que denominamos **EIJ**, o sea, **Enemigos Increíblemente Jartibles**. Se trata de perseguidores lineales. Estos enemigos aparecen en el punto donde los creas y se dedican a perseguir al jugador. No pueden traspasar el escenario. Si el jugador puede disparar y los mata, volverán a aparecer al ratito en el mismo sitio de donde salieron por primera vez. El gráfico con el que se dibujan se elige al azar entre todos los disponibles a menos que configures uno fijo. Funcionan mejor en güegos de vista genital y puedes verlos en acción en **Mega Meghan**.

El código para gestionar los enemigos lineales (y plataformas) se añade por defecto al motor; los enemigos de tipo 6 o 7 hay que activarlos de forma explícita cuando configuremos el motor.

Nosotros en **Dogmole** sólo vamos a usar los enemigos de tipos 1 a 3 y las plataformas móviles de tipo 4.

Si [me invitáis a café](#), al final del tutorial podemos ver cómo modificar todo el motor de enemigos para poder usar más de 4 tipos lineales básicos de dos formas: usando 8 tipos diferentes sin animación, o añadiendo más frames para tener 8 tipos diferentes con animación (gastando un montón de memoria en el proceso). Pero para esto todavía queda un montón. No te vayas a emocionar ya, que estás hecho un bochinche.

Hotspots

Los **hotspots** son, simple y llanamente, una posición dentro de la pantalla donde puede haber un objeto, una llave, una recarga de vida, una recarga de munición, o una recarga de tiempo. En cada pantalla se define un único hotspot. Cada hotspot tiene asociado un valor:

#	Significado
1	Objeto o ítem coleccionable
2	Llave
3	Recarga de vida
4	Recarga de munición
5	Recarga de tiempo

Cuando hablamos de objetos nos referimos al ítem que se dibuja con el tile número 17 del tileset y que será contado automáticamente por el motor del juego sin necesidad de tener que hacer nosotros nada más que decirle al motor que vamos a usar objetos. Se trata de las pócimas en **Lala Prologue**, o las cruces de **Zombie Calavera**, los lápices en **Viaje al Centro de la Napia** o los diskettes en **Trabajo Basura**. Nosotros vamos a usar objetos para representar las cajas que tenemos que recoger y llevar a la universidad de Miskatonic en nuestro güego. Luego, mediante scripting y configuración, haremos que el comportamiento de los objetos sea diferente (no se contará el objeto hasta que, después de cogerlo, lo hayamos depositado en un punto determinado de la universidad, de forma muy parecida a cómo funcionan los diskettes en **Trabajo Basura**), pero básicamente se trata de objetos que habrá que colocar en el mapa usando hotspots de tipo objeto (tipo 1).

Como sé que lo vais a preguntar: no, no se puede asignar más de un hotspot por pantalla tal y como está programado el motor.

Preparando los materiales necesarios

Bien. Vamos a ponernos manos a la obra. Lo primero que tendremos que hacer es copiar los materiales necesarios al directorio `/enems` . Necesitamos dos cosas: el mapa del güego en formato MAP , y el tileset del juego `work.png` . Por tanto, copiamos `/map/mapa.map` y `/gfx/work.png` en `/enems` . ¡Ya estamos preparados para la marcha!

Configuración de nuestro proyecto

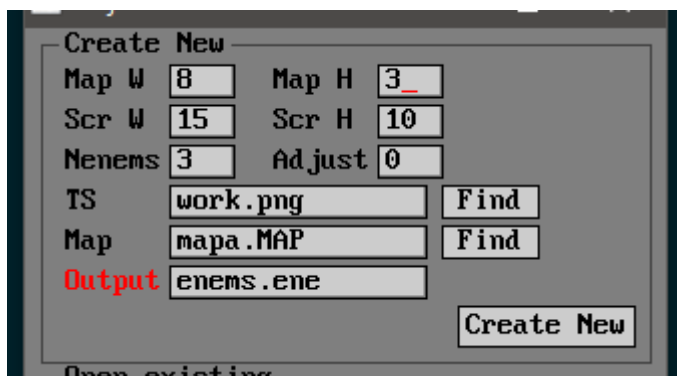
Ponedor era originalmente una aplicación que, aunque tiene su interfaz gráfica, estaba pensada para ser lanzada desde la ventana de línea de comandos. Sin embargo, por petición popular, le he añadido un diálogo de entrada para que se pueda usar tal y como se usaba el viejo colocador. Si estáis interesados en usarlo desde la ventana de línea de comandos, ejecutad el programa con e parámetro `-h`:

```
$ ../utils\ponedor.exe -h
Edit an existing set:
$ ponedor.exe file.ene
```

```
Create new set:
$ ponedor.exe new out=file.ene map=file.map tiles=file.png|bmp [adjust=n] size=w,h
[scrsz=w,h] [nenems=n] [x2]
```

out	output filename
map	map file (raw, headerless, 1 byte per tile, row order)
tiles	tileset in png or bmp format.
adjust	subtract this number from every byte read from the map file. Def=0
size	map size in screens
scrsz	screen size in tiles. Def=16,12
nenems	number of enemies per screen. Def=3
x2	zoom x2 (hacky)

Cuando ejecutas el Ponedor (por ejemplo, haciendo doble click sobre `ponedor.bat` en `/enems`) aparecerá la pantalla principal en al que podemos cargar un proyecto existente o configurar uno nuevo. Como no tenemos un proyecto existente, crearemos uno nuevo rellenando las casillas del recuadro superior:



Aquí hay un porrón de casillas que rellenar. Vamos a ir explicando qué son cada una de ellas, aunque la verdad es que a estas alturas casi todo debería resultar bastante intuitivo. Empezando desde arriba:

1. `Map W` y `Map H` son el **ancho** y el **alto** de nuestro mapa en **pantallas**, o sea, las **dimensiones del mapa**. En el caso de **dogmole** habría que rellenar 8 y 3.
2. `Scr W` y `Scr H` son las **dimensiones de cada pantalla**, en **tiles**. Para todos los güegos de **MTE MK1** estos valores son 15 y 10 (de hecho, vienen ya puestos por defecto). No toques aquí.
3. `Nenems` es el número de enemigos máximo por pantalla. En **MTE MK1** debe ser 3, ni más ni menos. También viene puesto por defecto. No toques aquí tampoco.
4. `Adjust` podrá valer 0 o 1, dependiendo se Mappy hizo fullerías. Si recordarás, a la hora de explicar cómo se montaba el mapa, mencionamos que Mappy quiere un tile negro como tile 0 y que si no se lo das, él se lo pone desplazando un espacio todos los tuyos. **Si te pasó esto, deberás cambiar el 0 que aparece en esta casilla por un 1**. De ese modo nuestro Ponedor estará coscado y pintará bien el mapa.
5. Donde pone `TS` tenemos que poner la ruta del archivo con el tileset, que debería ser `work.png` a secas si lo has copiado en `enems`, aunque puedes usar el botón `Find` para ubicarlo desde un explorador si no te apetece escribir 8 caracteres.
6. En `Map` hay que poner la ruta del archivo con el mapa, que debería ser `mapa.map` a secas si lo copiaste a `enems`. También puedes usar `Find`.
7. No te olvides de poner el **nombre de archivo de salida** en la casilla `Output`. Si no quieres tocar `compile.bat`, este nombre debe ser `enems.ene`.

En la parte inferior izquierda de la ventana, por cierto, verás un botón que pone `Size`. Puedes pulsarlo para que en vez de `Normal` salga `Double` y así el Ponedor se mostrará ampliado 2X y se verá mejor todo. Consciente que soy de que cada vez se manejan resoluciones más bestias y que cada vez estamos más cegatones, tengo en mente añadir más opciones (3x y 4x).

Cuando esté todo relleno **revísalo**, que luego vienen los llantos y el crujiir de dientes (nunca entendí esta expresión, pero la decía mucho una maestra de Badajoz que tuve), y cuando estés seguro de que todo está en su sitio, pulsa `Create New`.

Manejo básico del programa

El manejo es muy sencillo. Más que nada porque el programa es muy sencillo. Si te fijas, hay una rejilla con la pantalla actual. Si no sale la rejilla, o lo que sale en la rejilla no es la primera pantalla de tu mapa, mal vamos. Revisa los pasos anteriores, sobre todo los referidos a las dimensiones del mapa y de las pantallas.

Para navegar por el mapa (para que salga otra pantalla en la rejilla) usaremos las teclas de los cursores. Pruébalo. Deberías poder hacer un bonito recorrido turístico por tu mapa.

La pantalla se divide en tres zonas, de arriba a abajo:

Indicadores

En la parte superior de la pantalla ves un montón de indicadores que te vendrán muy bien para algunas cosas (por ejemplo para hacer el script del juego). De izquierda a derecha:

1. Coordenadas de la pantalla actual dentro del mapa (XP, YP).
2. Número actual de la pantalla (que será $YP * MAP_W + XP$).
3. `2b` o `3b` dependiendo si el archivo que estás editando es de formato antiguo o de formato nuevo, respectivamente. Con la versión 5.0 de **MTE MK1** verás siempre `3b`. Se puede convertir de formato pulsando `L`.
4. Las coordenadas de la casilla sobre la que pasa el ratón, cuando está en el área de edición.

Área de edición

Donde ponemos las cosas que hay que poner.

Botones

Abajo del todo verás un montón de botones. El primero pone `Save`, y graba una copia de lo que estás haciendo en el archivo que configuraste como archivo de salida. Púlsalo a menudo. Muy a menudo. También puedes darle a la tecla `S`.

Los que hayáis sufrido el colocador recordaréis lo divertido que era pulsar `ESC` sin querer y perder todos los cambios. Ahora ya no, te sale un diálogo de confirmación. Igualmente puedes pulsar el botón `Exit`.

El botón **Grid** activa o desactiva la rejilla. No sé para qué lo puse o para qué carajo puede servir. También vale pulsar la tecla `G`.

El siguiente botón pone **Reload** y sirve para recargar los recursos (mapa y gráficos) de nuevo. A veces puedes estar poniendo enemigos y ver una cagada en el mapa. Entonces abres mappy, apañas, copias el `mapa.map` de nuevo en `enems`, y pulsar **Reload** para que el Ponedor se cosque de los cambios.

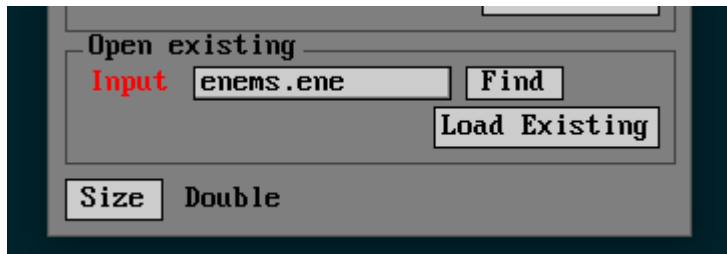
El último botón que pone **H** es por si quieres trabajar con la churrera antigua o con las primeras versiones de MK1_NES (lo que ahora es AGNES). Genera un archivo `enems.h` con los enemigos en un array de código C, **pero esto ya no se usa y no es necesario**. Ahora en `compile.bat` hay una llamada a un conversor que pilla lo que necesita directamente de tu `enems.ene`. Así que olvídate de exportar y hostias como en la Churrera vieja. No he quitado este botón porque ¿y si hay que modificar un juego viejo? Pues eso.

Grabando y cargando

Aunque no hayas puesto nada todavía, graba tu proyecto pulsando `Save` o `S`. Verás que en el directorio `/enems` aparecerá el archivo `enems.ene` que referenciaste en el diálogo inicial al crear el

proyecto. Ahora cierra el Ponedor. Ahora vuelve a ejecutarlo pulsando sobre `ponedor.bat` en `/enems`.

Esta vez, en lugar de rellenar los valores, escribe `enems.ene` en el cuadro `Input` situado en el recuadro inferior que está etiquetado `Open existing` y haz click sobre el botón que pone `Load Existing`. Si todo sale bien, te debería volver a salir la primera pantalla del mapa.



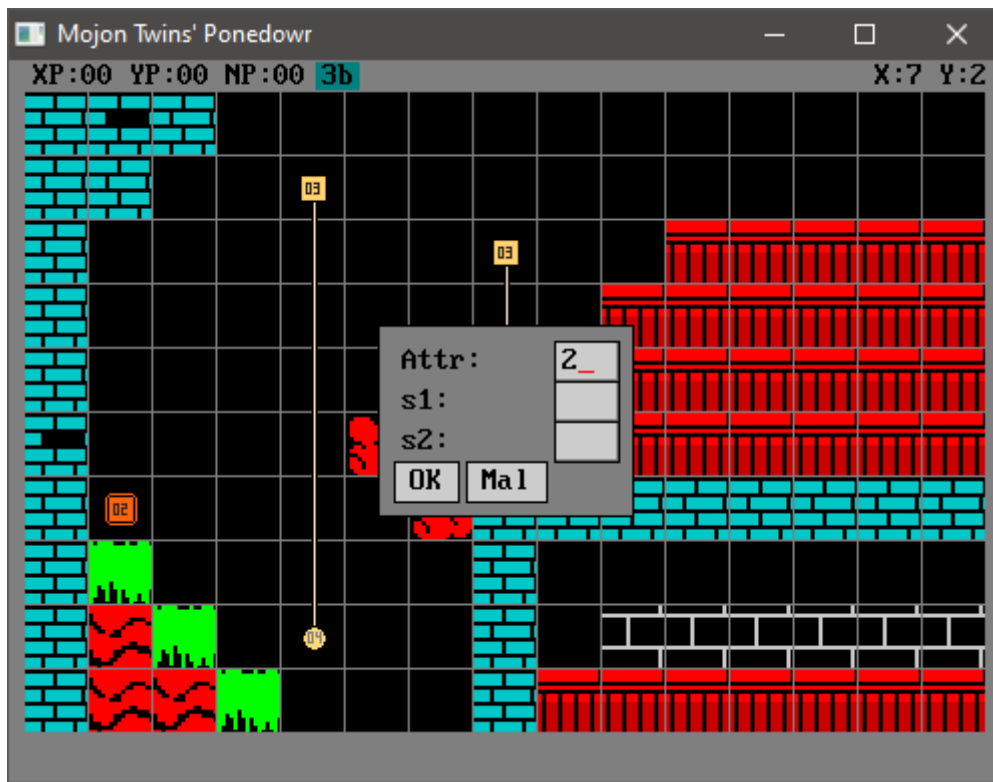
Como mencionamos antes, sólo tenemos que preocuparnos de poner las cosas en el Ponedor y grabar el archivo `enems.ene` a menudo. `compile.bat` se encargará de hacer las conversiones necesarias y de meter los numeritos en el juego.

Poniendo enemigos y plataformas

Lo primero es aprender a colocar enemigos lineales (horizontales, verticales, o los raros esos diagonales que vimos antes). Lo que se hace es definir una trayectoria, un tipo, y una velocidad. Vamos a hacerlo.

Para empezar, colocamos el ratón sobre la casilla de inicio de la trayectoria y hacer click. Esta posición será la inicial, donde aparecerá el enemigo, y además servirá como uno de los límites de su trayectoria (mirad los dibujitos de más arriba, de cuando hablábamos de los tipos de enemigos). Cuando hagamos esto, aparecerá un cuadro de diálogo donde deberemos introducir el tipo del enemigo. Recuerda que en el caso de enemigos lineales será un valor de 1 a 4, 4 para las plataformas en los güegos de vista lateral. Ponemos el numerito y pulsamos OK.

Ahora lo que el programa espera es que le digamos donde acabaría la trayectoria. Nos vamos a la casilla donde debe acabar la trayectoria y hacemos click de nuevo. Veremos como se nos muestra gráficamente la trayectoria y aparece un nuevo cuadro de diálogo algo más complejo:



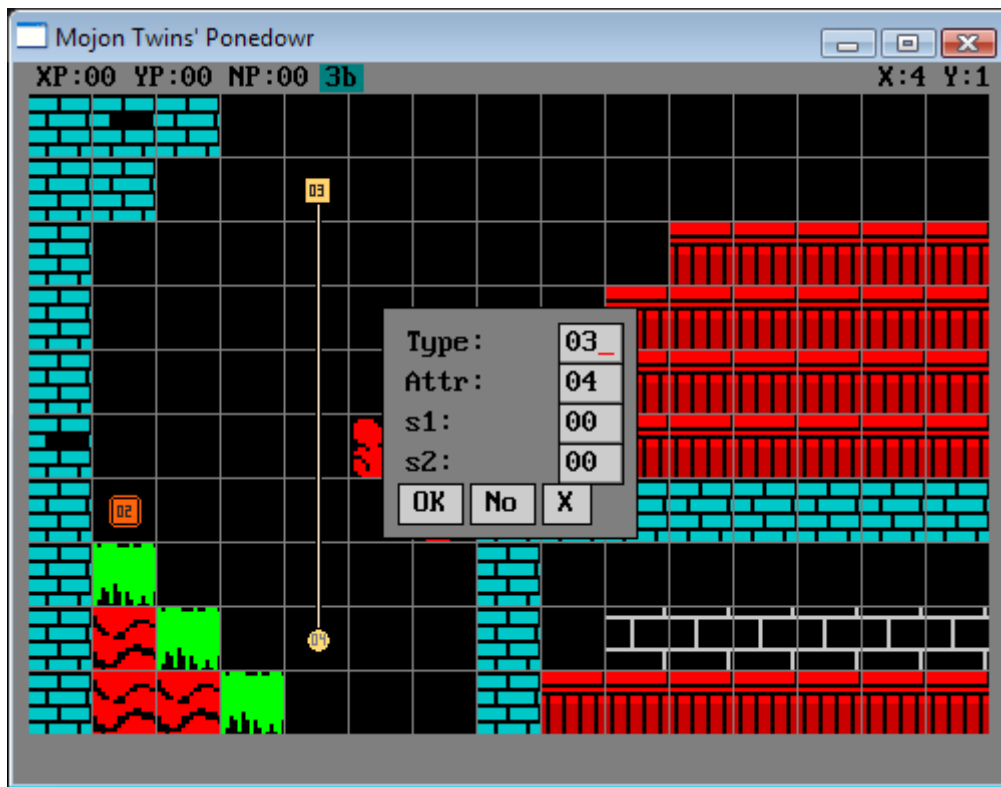
En **MTE MK1** sólo tendremos que rellenar el recuadro **Attr** . Los otros dos están ahí porque Ponedor pretende ser una herramienta más general y **la usamos con otros motores que sí que necesitan más parámetros**.

El valor introducido en **Attr** será el número de píxeles que avanzará el enemigo o plataforma por cada cuadro de juego. Estos valores, para que no haya problemas, deberían ser potencias de dos. O sea, 1, 2, 4, 8... En realidad, los valores que valen son 1, 2 o 4. Algo más allá es ya demasiado rápido y daría problemas de todas clases. Valores que no sean potencias de dos también pueden dar problemas. Si quieres puedes probar a poner un 3 o algo a ver qué pasa, pero te digo ya que es probable que termines con el enemigo yéndose a pescar fuera de la pantalla o cosas peores. Una vez que hayamos puesto el numerico, pulsamos OK y ya tenemos nuestro primer enemigo colocado.

Para los enemigos de tipo 6 (los que vuelan, que nosotros llamamos **Fantys**) y los de tipo 7 (los que aparecen continuamente en un punto fijo y se tiran para tí a toda leche, o **EIJ**) sólo importa la casilla de inicio. La otra casilla da igual donde la pongas. Para no guarrear, se suele poner en la casilla de al lado. Para ambos tipos de enemigos puedes dejar vacío el cuadro **Attr** , ya que no se usa tampoco.

Puedes poner un máximo de tres por pantalla (el programa no te dejará meter más). No tiene por que haber tres en cada pantalla, puedes tener pantallas con uno, con dos, o con ninguno.

Para eliminar o editar los valores de un enemigo que ya hayamos colocado, basta con hacer click sobre la casilla de inicio de la trayectoria (donde aparece el numerito que indica el tipo). Entonces nos aparecerá un cuadro de diálogo donde podremos cambiar su tipo o la velocidad o eliminarlo completamente.



Y así, poco a poco, iremos colocando nuestros enemigos y plataformas en el mapa, con un máximo de 3 por pantalla, cuidándonos de no meter un valor fuera de rango como tipo de enemigo, y no olvidándonos de que la velocidad debe ser 1, 2 o 4.

Colocando hotspots

Como dijimos antes, un hotspot es la casilla donde aparecerá una llave, un objeto o una recarga de vida, munición o tiempo.

Se recordáis, mencionamos que cada hotspot tiene un tipo. Para **MTE MK1**, este tipo puede ser 0 (desactivado) o uno de esta tabla que pusimos antes y que ponemos de nuevo porque github es gratis:

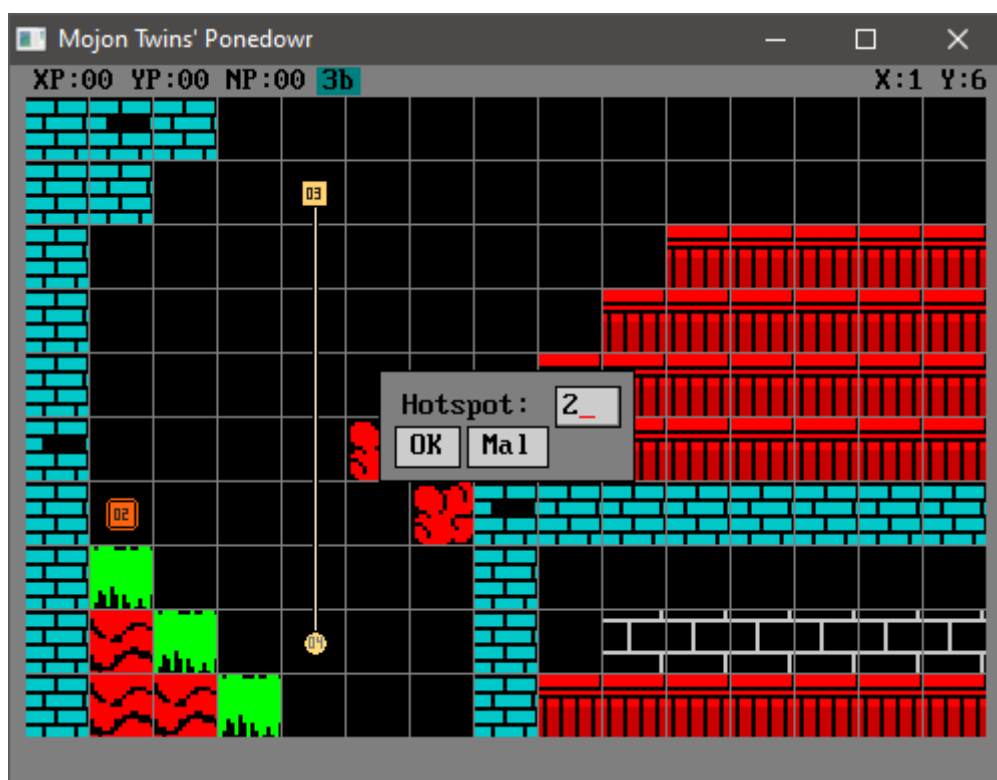
#	Significado
1	Objeto o ítem coleccionable
2	Llave
3	Recarga de vida
4	Recarga de munición
5	Recarga de tiempo

Recordemos que **cada pantalla admite un único hotspot**. Eso significa que el número total de llaves y objetos necesarios para terminar el güego no puede exceder el número de pantallas. Por ejemplo, en Lala Prologue tenemos 30 pantallas. Para terminar el güego con éxito hay que recoger 25 objetos,

y además hay cuatro cerrojos que abrir, por lo que necesitamos 4 llaves. Esto significa que en 25 de las 30 pantallas habrá un objeto, y en 4 habrá una llave. Perfecto: necesitamos 29 pantallas de las 30. Es importante planear esto de antemano. Si te emocionas colocando un montón de cerrojos puede que luego no te quepan todas las llaves y todos los objetos que hay que recoger.

También deberías asegurarte de que pones suficientes llaves para abrir todos los cerrojos y que siempre hay una forma de terminarse el juego sin quedarnos encerrados. Ten cuidado con esto, es posible construir combinaciones de llaves y cerrojos incorrectas si haces bifurcaciones y puede que el jugador gaste las llaves en una ruta que no tenías planeada y luego no pueda seguir avanzando.

Para colocar el hotspot de la pantalla actual, simplemente **hacemos click con el botón derecho en la casilla donde queremos que aparezca la llave, el objeto o la recarga**, con lo que haremos aparecer un pequeño cuadro de diálogo donde deberemos introducir el tipo:



Huelga decir que no todas las pantallas tienen por qué tener un hotspot. Para editar o borrar un hotspot, simplemente haz click *con el botón derecho* sobre él para mostrar el diálogo de edición desde el que también te lo puedes cargar.

Conversión e importación

Esto antes era una tarea manual. Ahora es algo de lo que se encarga automáticamente `compile.bat`. En concreto, lo hace gracias a la utilidad `ene2h`, que lee nuestro archivo `.ene` y, mediante un mágico y misterioso proceso, obtiene el código que necesita para enchufarle al motor. Como ya es costumbre, veamos como funciona, sólo por el placer de adquirir conocimiento. Ejecutándolo en la ventana de línea de comandos, como siempre, nos chiva los parámetros:

```
$ ..\utils\ene2h.exe  
$ ene2h.exe enems.ene enems.h [2bytes]
```

The 2bytes parameter is **for** really old .ene files which stored the hotspots 2 bytes each instead of 3 bytes.

As **a** rule of thumb:

.ene file created with ponedor.exe -> 3 bytes.

.ene file created with colocador.exe **for** MK1 -> 2 bytes.

Para **MTE MK1** sólo utilizaremos el parámetro de entrada, que deberá apuntar a nuestro archivo .ene , conteniendo la ruta si es necesario, nuestro archivo de salida, que para **MTE MK1** **debe** ser assets/enems.h , y listo. De hecho, si editas una vez más `compile.bat` encontrarás la línea de conversión tal que así:

```
..\utils\ene2h.exe ..\enems\enems.ene assets\enems.h
```

¡Y hemos terminado!

Además, con más emoción: en el próximo número vamos a configurar y compilar por primera vez nuestro güego, aunque sea sin scripting, para ver cómo va quedando.

Capítulo 7: Primer montaje

Antes que nada, bájate el paquete de materiales correspondiente a este capítulo pulsando en este enlace:

[Material del capítulo 7](#)

Venga ya. ¡Quiero ver cosas moviéndose!

En eso estamos. Este capítulo va a ser densote, pero densote. Prepárate para absorber cual porífero abisal toneladas y toneladas de información. Para seguir este capítulo recomendamos que te comas un buen tico-tico de sandía, que fortalece el cerebro y el peshito.



Para empezar de fácil, vamos primero a hacer una especie de recapitulación para ver que tenemos todo lo que necesitamos.

1. El primer paso fue hacerse con una copia del repositorio y copiar el contenido de la carpeta `/src` a una nueva con el nombre de tu proyecto.
2. El segundo paso se trató de editar `/dev/compile.bat` y cambiar el nombre del juego en la línea que empieza por `set game=` .
3. Todos los gráficos deberían estar en `/gfx` con los nombres correctos, a saber: `font.png` y `work.png` para los patrones fijos, `sprites.png` , `sprites_extra.png` y `sprites_bullet.png` para los sprites, y `loading.png` , `title.png` , `marco.png` (si usas marco separado de la pantalla de título) y `ending.png` para las pantallas fijas.

4. El cuarto paso fue poner el mapa en `/map` exportado en formato `MAP`, y luego volver a editar `/dev/compile.bat` para modificar la línea del conversor (`mapcnv`) con el tamaño de tu mapa: Para **Dogmole** sería:

```
..\utils\mapcnv.exe ..\map\mapa.map 8 3 15 10 15 packed > nul
```

5. Seguidamente pusimos los enemigos y los hotspots y los guardamos en `/enems` como `enems.ene`.

¿Lo tenemos todo? ¿Seguro? Bien. Entonces podemos empezar a configurar el motor para hacer nuestra primera compilación. Agárrense, que vienen curvas. Y mucho texto.

El archivo de configuración

Si te fijas en nuestra carpeta de desarrollo `/dev`, existe una subcarpeta `/dev/my`, y dentro de esta un archivo llamado `config.h`. Quédate con su cara: es el archivo más importante de todos, ya que es el que decide qué trozos del engine se ensamblan para formar el güego, y qué valores de parámetros se emplean en dichos trozos. Tendrías que haberlo hecho ya, pero por si acaso: abre `/dev/my/config.h` en tu editor de textos favorito (que no sea mierder). Y ahora flípalo con la cantidad de cosas que hay. Vamos a ir por partes, explicando cada sección, para qué sirven los valores, cómo interpretarlos, y rellenándolos para nuestro Dogmole de ejemplo.

En este archivo lo que hay es un porrón de directivas `#define`, que definen (!) símbolos que luego están por el código diciéndole al compilador qué partes o no incluir, o qué valor dar a algunas partes. Por lo general, nuestro cometido aquí será *comentar* las directivas que no queramos, y *descomentar* (!) las que sí queramos. La expresión *comentar*, en este contexto, es un false friend de una elipsis y más o menos significa "convertir en un comentario", o sea, en texto que el compilador ignore. **MTE MK1** está escrito en lenguaje `c`, por lo que la forma de hacer que una línea sea ignorada es ponerle dos barras delante: `//`. Cada vez que hablemos de *desactivar* o *comentar* una directiva nos estaremos refiriendo a ponerle `//` delante, y cada vez que digamos *activar* o *descomentar* nos estaremos refiriendo a quitarle el `//`.

Aquí es importante tener un buen editor de textos *no mierder* que además tenga resaltado de sintaxis, porque eso nos ayudará a ver muy fácilmente qué está activo y qué no, ya que los comentarios suelen ponerse todos de un color especial.

```

16  #define SCR_INICIO          16      // Initial screen
17  #define PLAYER_INI_X       1       //
18  #define PLAYER_INI_Y       7       // Initial tile coordinates
19  #define SCR_FIN             99      // Last screen. 99 = deactivated.
20  #define PLAYER_FIN_X       99      //
21  #define PLAYER_FIN_Y       99      // Player tile coordinates to finish game
22  #define PLAYER_NUM_OBJETOS  99      // Objects to get to finish game
23  #define PLAYER_LIFE         15      // Max and starting life gauge.
24  #define PLAYER_REFILL       1       // Life recharge
25  // #define COMPRESSED_LEVELS  // us...ls...of...h (!)
26  // #define MAX_LEVELS       // #...compressed...ls
27  // #define REFILL_ME        // If defined, refill player on each level
28
29  // =====
30  // II. Engine type
31  // =====

```

Un color especial

Otra convención que hemos tenido a bien tomar en **MTE MK1** es que un valor de 99 significa "no uses esto" o "ignora este parámetro". Esto es cierto en general, excepto para el máximo de vida del jugador (`PLAYER_LIFE`), que puede ser 99 perfectamente.

Tomemos aire y empecemos.

Configuración general

Esta sección empieza con un `#define MODE_128K` que está desactivado y que por ahora ignoraremos amablemente. Sin aspavientos.

En este apartado se configuran los valores generales de juego: el tamaño del mapa, el número de objetos (si aplica), y cosas por el estilo:

Tamaño del mapa

```

#define MAP_W      8      //
#define MAP_H      3      // Map dimensions in screens

```

Estas dos directivas definen el tamaño que tiene nuestro mapa. Si recordamos, para nuestro **Dogmole** el mapa mide 8×3 pantallas. Rellenamos, por tanto, 8 y 3 (W de Width, anchura, y H de Height, altura).

Posición de inicio

```

#define SCR_INICIO  16      // Initial screen
#define PLAYER_INI_X 1       //
#define PLAYER_INI_Y 7       // Initial tile coordinates

```

Aquí definimos la posición del personaje principal cuando se empieza una nueva partida. `SCR_INICIO` define qué pantalla, y `PLAYER_INI_X` e `Y` definen al coordenada del tile donde aparecerá el personaje. Nosotros empezamos en la primera pantalla de la tercera fila, que (si contamos, o calculamos) es la pantalla número 16. Hemos colocado a Dogmole sobre el suelo y junto a la pared

de roca, en la casilla de coordenadas (1, 7) (¡recuerda! Los verdaderos desarrolladores empiezan a contar en el 0).

Posición de fin

```
//#define SCR_FIN          99      // Last screen. 99 = deactivated.  
//#define PLAYER_FIN_X    99      //  
//#define PLAYER_FIN_Y    99      // Player tile coordinates to finish game
```

Aquí definimos la posición final a la que debemos llegar para terminar el güego. Puede interesarnos hacer un güego en el que sencillamente tengamos que llegar a un sitio en concreto para terminarlo. En ese caso rellenaríamos estos valores y **descomentaríamos las macros**. Como en el güego que nos ocupa esto no lo vamos a usar, las comentamos para que se ignore.

Número de objetos

```
//#define PLAYER_NUM_OBJETOS    99      // Objects to get to finish game
```

Este parámetro define el número de objetos que tenemos que reunir para terminar el güego. En güegos sencillos como **Lala Prologue**, el conteo de objetos y la comprobación de que lo tenemos todos es automática y emplea este valor: en cuanto el jugador tenga ese número de objetos se mostrará la pantalla del final. En nuestro caso, no: nosotros vamos a usar scripting para manejar los objetos y las comprobaciones de que hemos hecho todo lo que teníamos que hacer para ganar la partida, así que no vamos a necesitar para nada esto. Por tanto, **la dejaremos comentada** para que el motor ignore el conteo de objetos automático. Si estás haciendo un güego por tu cuenta en el que simplemente hay que recoger todos los objetos, como en tantos que hemos lanzado, coloca aquí el número máximo de objetos necesarios.

Vida inicial y valor de recarga

```
#define PLAYER_LIFE          15      // Max and starting life gauge.  
#define PLAYER_REFILL        1      // Life recharge
```

Aquí definimos cuál será el valor inicial de vidas del personaje y cuánto se incrementará cuando cojamos una vida extra. En Dogmole, empezaremos con 15 vidas y recargaremos de 1 en 1 al encontrar corazones. Como vamos a configurar el motor para que una colisión con un enemigo nos haga parpadear, las vidas se perderán poco a poco. En güegos como Lala Prologue, en los que las colisiones producen rebotes y el enemigo nos puede golpear muchas veces, las “vidas” se consideran “energía” y se usan valores más altos, como 99, y recargas más generosas, de 10 o 25.

Juegos multi nivel

Por ahora vamos a pasar de los juegos multi nivel. Pero, como ya hemos mencionado (y habrás visto), es algo totalmente posible. Internamente se basa en tener reservado espacio para mapa y enemigos, y quizá también gráficos y comportamientos, y luego tener una colección de recursos comprimidos que se descomprimen sobre este espacio para cada nivel que se carga, siguiendo diferentes configuraciones.

Tipo de motor

Ahora empezamos con lo bueno: la configuración de las partes de código que serán ensambladas cual transformer múltiple para formar nuestro engendro. Aquí es donde está toda la chicha. Es divertido porque podemos experimentar usando combinaciones extrañas (lo hemos hecho a saco en la Covertape #2) y probando los efectos. Como ya mencionamos, es imposible activarlo todo, y no sólo porque haya cosas que se anulan entre sí, sino porque, sencillamente, no cabría.

Vayamos por partes, como dijo Victor Frankenstein...

Tamaño de la caja de colisión

```
// Bounding box size
// -----

#define BOUNDING_BOX_8_BOTTOM           // Comment both for normal 16x16 bounding box
// 8x8 aligned to bottom center in 16x16
//#define BOUNDING_BOX_8_CENTERED       // 8x8 aligned to center in 16x16
//#define SMALL_COLLISION                // 8x8 centered collision instead of 12x12
```

La caja de colisión se refiere al cuadrado que ocupa “realmente” nuestro sprite. Para entendernos, nuestro sprite va a chocar contra el escenario. Dicha colisión se calcula con un cuadrado imaginario que puede tener dos tamaños: 16×16 u 8×8. El motor sencillamente comprueba que ese cuadrado no se meta en un bloque obstáculo.

Para que entiendas la diferencia, fíjate cómo interactúa Lala con el escenario en **Lala Prologue** (que tiene una caja de colisión de 16×16 que ocupa todo el sprite) y en **Lala Lah** (en la que usamos una caja de colisión de 8×8, más pequeña que el sprite).

Si elegimos una colisión de 8×8 con el escenario, tenemos dos opciones: que el recuadro esté centrado en el sprite o que esté en la parte baja:



La primera opción (recuadro centrado) está pensada para güegos con vista genital, como **Balowwwn** o **D'Veel'Ng**. La segunda funciona bien con güegos de vista lateral o güegos con vista genital “con un poco de perspectiva”, como **Mega Meghan**.

Las dos primeras directivas se refieren a colisiones **contra el escenario**. Solo una de las dos directivas puede estar activa (porque son excluyentes): si queremos colisión 8×8 centrada activamos `BOUNDING_BOX_8_CENTERED` y desactivamos la otra. Si queremos colisión 8×8 en la parte baja activamos `BOUNDING_BOX_8_BOTTOM` y desactivamos la otra. **Si queremos colisión de 16×16 desactivamos ambas.**

La tercera directiva se refiere a las colisiones **contra los enemigos**. Si activamos `SMALL_COLLISION`, los sprites tendrán que tocarnos mucho más para darnos. Con `SMALL_COLLISION` los enemigos son más fáciles de esquivar. Funciona bien en güegos con movimientos rápidos, como **Bootee**. Nosotros la vamos a dejar desactivada para **Dogmole**.

Directivas generales

```
// #define PLAYER_CHECK_MAP_BOUNDARIES // If defined, you can't exit the map.
```

Si se activa, el motor comprobará que no nos salimos del mapa. Si tu mapa está bien delimitado con tiles obstáculos por todos los lados, puedes desactivar esta directiva y ahorrar espacio. Tampoco hace falta activarla si el único sitio no delimitado es la parte superior del mapa, como es nuestro caso en **Dogmole**, por lo que la dejamos deshabilitada.

```
#define DIRECT_TO_PLAY // If defined, title screen is also the game f
```



Activamos esto para conseguir lo que dijimos cuando estábamos haciendo las pantallas fijas: que el título del juego sirva también como marco. Si tienes un `title.png` y un `marco.png` separados, deberás desactivarla. En nuestro caso, en el que sólo tenemos un `title.png` que también incluye marco, la dejamos activada.

```
//#define DEACTIVATE_KEYS           // If defined, keys are not present.
//#define DEACTIVATE_OBJECTS       // If defined, objects are not present.
//#define DEACTIVATE_REFILLS       // If defined, life refills are not present.
```

Estas tres directivas sirven para desactivar llaves, objetos o recargas. Si tu juego no usa llaves y cerrojos, deberás activar `DEACTIVATE_KEYS`. Si no vas a usar objetos, activamos `DEACTIVATE_OBJECTS`. Idem con las recargas de vida. Así ahorramos toneladas de código.

Seguramente alguno de vosotros estará pensando ¿por qué no activamos `DEACTIVATE_OBJECTS` en **Dogmole**, si hemos dicho que los objetos los vamos a controlar por scripting? ¡Buena pregunta! Es sencillo: lo que vamos a controlar por scripting es el **conteo de objetos** y la **condición final**, pero necesitamos que el motor gestione la **recogida** y **colocación** de los objetos. No nos los podemos fumar.

```
#define ONLY_ONE_OBJECT           // If defined, only one object can be carried
#define OBJECT_COUNT              1 // Defines which FLAG will be used to store th
```

Seguimos con dos directivas con una aplicación muy específica: si activamos la primera, `ONLY_ONE_OBJECT`, sólo podremos llevar un objeto *cada vez*. Una vez que cojas un objeto, la recogida de objetos se bloquea y no puedes coger más. Para volver a activar la recogida de objetos hay que *liberar el objeto*, y para eso tendremos que usar scripting o *inyección de código* (que algún día explicaremos en otro tutorial).

Con esto conseguimos en **Dogmole** el efecto que necesitamos de que haya que ir llevando las cajas una a una: configuramos el motor para que sólo permita que llevemos un objeto (una caja), y luego, cuando hagamos el script, haremos que cuando llevemos la caja al sitio donde hay que ir las depositando (un sitio concreto de la Universidad) *liberemos el objeto* para que se vuelva a activar la recogida de objetos y así podamos ir a por la siguiente caja.

La segunda directiva, `OBJECT_COUNT`, sirve para que en el **marcador de objetos**, en lugar de la cuenta interna de objetos recogidos, **se muestre el valor de uno de los *flags* del sistema de scripting**. Ya lo veremos en el futuro, cuando expliquemos el motor de scripting, pero los scripts tienen hasta 32 variables o *flags* que podemos usar para almacenar valores y realizar comprobaciones. Cada variable tiene un número. Si definimos esta directiva, el motor mostrará el valor del flag indicado en el contador de objetos del marcador, en lugar del contador interno de objetos. Desde el script iremos incrementando dicho valor cada vez que el jugador llegue a la Universidad y deposite un objeto.

En definitiva, sólo necesitaremos definir `OBJECT_COUNT` si somos nosotros los que vamos a llevar la cuenta, a mano, desde el script (o mediante inyección de código), usando uno de sus flags. Si no vamos a usar scripting, o no vamos a necesitar controlar a mano el número de objetos recogidos, tendremos que comentar esta directiva para que no sea tomada en cuenta.

```
// #define DEACTIVATE_EVIL_TILE // If defined, no killing tiles (behaviour 1)
```

Activa esta directiva si quieres desactivar los tiles que te matan (tipo 1). Si no usas tiles de tipo 1 en tu juego, descomenta esta línea y ahorrarás espacio, ya que así la detección de tiles matantes no se incluirá en el código.

```
#define PLAYER_BOUNCES // If defined, collisions make player bounce
// #define FULL_BOUNCE // If defined, evil tile bounces equal MAX_VX,
// #define SLOW_DRAIN // Works with bounces. Drain is 4 times slower
#define PLAYER_FLICKERS // If defined, collisions make player flicker
```

Estas directivas controlan los rebotes. Si activas `PLAYER_BOUNCES`, el jugador rebotará contra los enemigos al tocarlos. La fuerza de dicho rebote se controla con `FULL_BOUNCE`: si se activa, los rebotes serán mucho más bestias porque se empleará la velocidad con la que el jugador avanzaba originalmente, pero en sentido contrario. Desactivando `FULL_BOUNCE` el rebote es a la mitad de la velocidad (y por tanto menos desagradable).

Si definimos, además, `SLOW_DRAIN`, la velocidad a la que perderemos energía si nos quedamos atrapados en la trayectoria del enemigo (acuérdense de **Lala Prologue**, **Sir Ababol** o la versión original de **Viaje al Centro de la Napia**) será cuatro veces menor. Esto se usa en **Bootee**, donde es fácil quedarse atrapado en la trayectoria de un enemigo y complicado salir de la misma. Esto hace el juego más asequible.

Como podrás imaginar, `FULL_BOUNCE` y `SLOW_DRAIN` dependen de `PLAYER_BOUNCES`. Si `PLAYER_BOUNCES` está desactivada, las otras dos directivas son ignoradas.

Activando `PLAYER_FLICKERS` logramos que el personaje parpadee si colisiona con un enemigo (siendo invulnerable durante un corto período de tiempo). Normalmente elegiremos entre `PLAYER_BOUNCES` y `PLAYER_FLICKERS`, **pero pueden funcionar juntas**. Nosotros, en **Dogmole**, queremos que el protagonista únicamente parpadee al colisionar con un enemigo, por lo que desactivamos `PLAYER_BOUNCES` y activamos `PLAYER_FLICKERS`.

```
// #define MAP_BOTTOM_KILLS // If defined, exiting the map bottomwise kill
```

Desactiva esta directiva si quieres que, en el caso de que el personaje vaya a salirse del mapa por abajo, el motor le haga rebotar y le reste vida, como ocurre en **Zombie Calavera**. Si tu mapa está

cerrado por abajo, desactívala para ganar unos bytes.

```
//#define WALLS_STOP_ENEMIES           // If defined, enemies react to the scenary
//#define EVERYTHING_IS_A_WALL         // If defined, any tile <> type 0 is a wall, o
```

Si activas `WALLS_STOP_ENEMIES`, los tiles de tipo 8 pararán la trayectoria de los enemigos lineales, haciéndolos cambiar de dirección igual que si llegasen a sus límites de trayectoria. Esto es interesante por ejemplo si quieres hacer trayectorias diagonales interesantes o tienes tiles empujables o escenario destructible.

Si además activas `EVERYTHING_IS_A_WALL`, cualquier tile de comportamiento distinto de 0 (recuerda los tipos de tile que explicamos en el capítulo 2) será considerado un obstáculo.

```
//#define BODY_COUNT_ON                2           // If defined, count enemies on flag #
```

Si configuras el motor para disparar o pisar, el número de muertes se irá contando en el flag que indiques aquí si defines esta directiva.

```
//#define DISABLE_PLATFORMS           // Disables platforms in side-view
```

Si no vas a necesitar plataformas en tu juego de vista lateral puedes activar esto para no incluir el código de interacción con las plataformas y ahorrar espacio. De regalo tienes un tipo extra de enemigo lineal.

Tipos de enemigos extra

Vamos a ver ahora un conjunto de directivas que nos servirán para activar los enemigos de tipos 6 o 7. Recordad lo que mencionamos sobre este tipo de enemigos: el 6 son los Fanties y el 7 son los enemigos que te persiguen de **Mega Meghan** y **Sgt. Helmet: Training Day**.

En **Dogmole** no usamos ningún tipo de enemigo extra. Para tus juegos, puedes experimentar con estos bicharracos. Aquí explicamos cómo se configuran y para qué sirven las cosas

Los Fanties

Los Fanties van volando por la pantalla en trayectorias suaves y persiguen al jugador. Hay dos tipos de Fanties: los sencillos, que simplemente aparecen donde los has puesto en Ponedor y persiguen al jugador, y los tipo "Homing Fanties", que permanecen en su sitio a menos que te acerques a determinada distancia. Es como si fueran cegatos. Si no te ven, no te persiguen. Si te alejas, dejarán de verte y volverán a su sitio inicial (que nosotros llamamos *su nido*).

Los Fanties se activan con esta directiva (en **Dogmole** la dejamos desactivada):

```
//#define ENABLE_FANTIES
```

```
// If defined, Fanties are enabled!
```

Y se configuran con estas otras:

```
//#define FANTIES_BASE_CELL      2      // Base sprite cell (0, 1, 2 or 3)
//#define FANTIES_SIGHT_DISTANCE 104     // Used in our type 6 enemies.
//#define FANTIES_MAX_V          256     // Flying enemies max speed (also for custom t
//#define FANTIES_A               16     // Flying enemies acceleration.
//#define FANTIES_LIFE_GAUGE       10     // Amount of shots needed to kill flying enemi
//#define FANTIES_TYPE_HOMING     // Unset for simple fanties.
```

1. `FANTIES_BASE_CELL` selecciona con qué gráfico pintaremos a los fanties de entre los 4 disponibles (3 si estás en vista lateral y usas plataformas). El valor debe ser 0, 1, 2 o 3, porque recuerda, los verdaderos desarrolladores empiezan a contar desde 0.
2. `FANTIES_SIGHT_DISTANCE` sólo tiene sentido para los fanties de tipo "Homing". Es la distancia, en píxel, a la que ven. Si no te acercas a menos de esa distancia se quedarán en su *nido* (donde los pones con el Ponedor).
3. `FANTY_MAX_V` define la velocidad máxima. Para hacerte una idea, divide el valor entre 64 y el resultado es el número de píxels que avanzará por cada cuadro (frame) de juego como máximo. Si definimos 256, el enemigo volador podrá acelerar hasta los 4 píxels por frame.
4. `FANTY_A` es el valor de aceleración. Cada cuadro de juego, la velocidad se incrementará en el valor indicado en dirección hacia el jugador, si no está escondido. Cuanto menor sea este valor, más tardará el enemigo en reaccionar a un cambio de dirección del jugador.
5. `FANTIES_LIFE_GAUGE` define cuántos tiros deberá recibir el bicharraco para morirse, si es que hemos activado los tiros (ver más adelante).
6. `FANTIES_TYPE_HOMING` sirve para seleccionar el tipo de Fanties. Si la activas, serán de tipo "Homing". Si la desactivas, serán de los sencillos.

Los Enemigos Increíblemente Jartibles

La casilla donde los pones en Ponedor será el "punto de aparición", o el "spawning point" para los que les guste el Checoslovaco. En ese punto, cada cierto tiempo, aparecerá un enemigo que perseguirá al jugador. Cuando el enemigo muera (a tiros, normalmente), tras ese cierto tiempo, aparecerá un nuevo enemigo. Y así *ad nauseam*. De ahí el nombre del tipo de enemigos. Los enemigos que te persiguen, por cierto, se detienen con los obstáculos del escenario, lo cual es de agradecer.

Los EIJs se activan con esta directiva (en **Dogmole** la dejamos desactivada):

```
//#define ENABLE_PURSUERS
```

```
// If defined, type 7 enemies are active
```

Y se configuran con estas otras:

```
//#define DEATH_COUNT_EXPRESSION 20+(rand())&15)
```

```
//#define PURSUERS_BASE_CELL 3 // If defined, type 7 enemies are always #
```

1. **DEATH_COUNT_EXPRESSION** . Si tenemos activado el motor de disparos, cuando matemos a un enemigo de tipo 7 tardará un tiempo en volver a salir. Dicho tiempo, expresado en número de cuadros (frames), se calcula usando la expresión definida en esta directiva. La que se ve en el ejemplo es la que se emplea en **Sgt. Helmet: Training Day**: 20 más un número al azar entre 0 y 15 (o sea, entre 20 y 35 frames).
2. **PURSUERS_BASE_CELL** selecciona con qué gráfico pintaremos a los EIJs de entre los 4 disponibles (3 si estás en vista lateral y usas plataformas). El valor debe ser 0, 1, 2 o 3, porque recuerda, bla bli blu.

Motor de bloques empujables

Estas dos directivas activan y configuran los bloques empujables. Nosotros no vamos a usar bloques empujables en Dogmole, por lo que las desactivamos. Activar la primera (**PLAYER_PUSH_BOXES**) activa los bloques, de forma que los tiles #14 (con comportamiento tipo 10, recuerda) podrán ser empujados.

El motor de bloques empujables se activa con:

```
//#define PLAYER_PUSH_BOXES
```

```
// If defined, tile #14 is pushable. Must be t
```

Y se configura con:

```
//#define FIRE_TO_PUSH
```

```
// If defined, you have to press FIRE+directio
```

```
//#define ENABLE_PUSHED_SCRIPTING
```

```
// If defined, nice goodies (below) are activa
```

```
//#define MOVED_TILE_FLAG
```

```
1
```

```
// Current tile "overwritten" with block is st
```

```
//#define MOVED_X_FLAG
```

```
2
```

```
// X after pushing is stored here.
```

```
//#define MOVED_Y_FLAG
```

```
3
```

```
// Y after pushing is stored here.
```

```
//#define PUSHING_ACTION
```

```
// If defined, pushing a tile runs PRESS_FIRE
```

1. **FIRE_TO_PUSH** : Si está activa, el jugador debe pulsar fire además de la dirección en la que empuja o no. En **Cheril of the Bosque**, por ejemplo, no hay que pulsar fire: sólo tocando el bloque mientras avanzamos se desplazará. En **D'Veel'Ng**, sí que es necesario pulsar fire para

empujar un bloque. Si vas a usar bloques empujables, debes decidir la opción que más te gusta (y que mejor se adecue al tipo de gameplay que quieras conseguir).

2. `ENABLE_PUSHED_SCRIPTING` activa la integración del sistema de scripting con los bloques empujables. Dicha integración se configura con las siguientes directivas:
3. `MOVED_TILE_FLAG` , `MOVED_X_FLAG` y `MOVED_Y_FLAG` : Cuando se empuja un bloque empujable, el tile que "pisa" se almacena en el flag que diga `MOVED_TILE_FLAG` , y sus coordenadas en los flags que digan `MOVED_X_FLAG` y `MOVED_Y_FLAG` . Con esto sabemos, desde el scripting, un montón de cosas útiles sólo mirando el valor de esas flags.
4. `PUSHING_ACTION` : Si la activamos, las cláusulas de las secciones `PRESS_FIRE` de la pantalla actual serán ejecutadas tras copiar los valores a los flags definidos más arriba cuando movamos un bloque empujable. Cuando expliquemos el sistema de scripting esto no te sonará a chino, sólo a croata.

Motor de disparos

El motor de disparos es bastante costoso en cuanto a memoria. Activarlo incluye bastantes trozos de código, ya que hay que comprobar más colisiones y tener en cuenta muchas cosas, además de los sprites extra necesarios. Tiene mogollón de cosas configurables y se puede usar para cosas que en un principio no se parecen a un motor de disparos. Échale imaginación, que de eso se trata.

El motor de disparos se activa con (desactivada en **Dogmole**):

```
//#define PLAYER_CAN_FIRE // If defined, shooting engine is enabled.
```

Y se configura con todas estas, que partimos en bloques:

```
//#define PLAYER_CAN_FIRE_FLAG 1 // If defined, player can only fire when flag
//#define PLAYER_BULLET_SPEED 8 // Pixels/frame.
//#define MAX_BULLETS 3 // Max number of bullets on screen. Be careful
```

1. `PLAYER_CAN_FIRE_FLAG` se usa con scripting o inyección de código. Si está activa, el jugador sólo podrá disparar si el valor del flag que indica es 1. Puedes usarlo para que no se pueda matar hasta que se encuentre la pihtola.
2. `PLAYER_BULLET_SPEED` controla la velocidad de las balas. 8 píxels por cuadro es un buen valor y es el que hemos usado en todos los güegos. Un valor mayor puede hacer que se pierdan colisiones, ya que todo lo que ocurre en pantalla es discreto (no continuo) y si un enemigo se mueve rápidamente en dirección contraria a una bala que se mueve demasiado rápido, es posible que de frame a frame se crucen sin colisionar. Si piensas un poco en ello y te imaginas el juego en cámara lenta como una sucesión de frames lo entenderás.

3. El valor de `MAX_BULLETS` controla el número máximo de balas que podrá haber en pantalla. Ten cuidado con esto, porque valores muy altos podrían comer mucha memoria y además ralentizar bastante el juego.

```
//#define PLAYER_BULLET_Y_OFFSET    6        // vertical offset from the player's top.
//#define PLAYER_BULLET_X_OFFSET    0        // vertical offset from the player's left/right
```

Estas dos directivas definen dónde aparecen las balas cuando se disparan. El comportamiento de estos valores cambia según la vista:

- Si el güego es en **vista lateral**, sólo podemos disparar a la izquierda o a la derecha (y, si lo configuramos, también en diagonal, pero sólo influye si el jugador mira a izquierda o a derecha). En ese caso, `PLAYER_BULLET_Y_OFFSET` define la altura, en píxels, a la que aparecerá la bala contando desde la parte superior del sprite del personaje. Esto sirve para ajustar de forma que salgan de la pistola o de donde queramos (de la pisha por ejemplo). `PLAYER_BULLET_X_OFFSET` se ignora completamente.
- Si el güego es en **vista genital**, el comportamiento es igual que el descrito si miramos para la izquierda o para la derecha, pero si miramos para arriba o para abajo la bala aparecerá desplazada lateralmente `PLAYER_BULLET_X_OFFSET` píxels desde la izquierda si miramos hacia abajo o desde la derecha si miramos hacia arriba. **Esto significa que nuestro personaje es diestro** a menos que seamos más listos que la quina *y juguemos con valores negativos*. Fijáos en los sprites de **Mega Meghan**.

```
//#define ENEMIES_LIFE_GAUGE          4        // Amount of shots needed to kill enemies.
//#define RESPAWN_ON_ENTER              // Enemies respawn when entering screen
//#define FIRE_MIN_KILLABLE            3        // If defined, only enemies >= N can be killed
//#define CAN_FIRE_UP                  // If defined, player can fire upwards and dia
```

Estas de aquí sirven para controlar cosas misceláneas balísticas

1. `ENEMIES_LIFE_GAUGE` define el número de tiros que deberán llevarse para morirse.
2. Si activamos `RESPAWN_ON_ENTER`, los enemigos habrán resucitado si salimos de la pantalla y volvemos a entrar. Como en los güegos clásicos, illo.
3. `FIRE_MIN_KILLABLE` sirve para que algunos enemigos no se puedan matal. Sólo morirán los que tengan un tipo mayor o igual al valor de esta directiva.
4. `CAN_FIRE_UP` deja que el jugador dispare en diagonales y para arriba en juegos de vista lateral, como en **Goku Mal**.

Por cierto, los enemigos matados se van contando y dicho valor puede controlarse desde el script.

Las balas, además, pueden tener un alcance limitado, lo que da mucho juego para hacer *cosas que no son balas con balas*. Este comportamiento se puede configurar con las siguientes directivas:

```
// #define LIMITED_BULLETS           // If defined, bullets die after N frames
// #define LB_FRAMES                  4      // If defined, defines the # of frames bullets
// #define LB_FRAMES_FLAG             2      // If defined, defines which flag determines t
```

Si activamos `LIMITED_BULLETS`, las balas durarán solo cierto número de frames. Este número de frames será el valor del flag `LB_FRAMES_FLAG` si se activa esta directiva, o el valor de `LB_FRAMES` directamente si queremos que sea fijo.

Tiles destructibles

Los tiles destructibles están orientados a los juegos que lleven el motor de disparos. Recordemos que el tipo de tile destructible era el 16. Estos tiles se romperán y desaparecerán (serán sustituidos por el tile 0) tras ser alcanzados por cierto número (configurable) de disparos. Para incluirlos, además de tener tiles definidos con el tipo 16, tenemos que configurar algunas cosas:

```
// #define BREAKABLE_WALLS           // Breakable walls
// #define BREAKABLE_WALLS_LIFE      1      // Amount of hits to break wall
```

La primera directiva, `BREAKABLE_WALLS`, activa esta característica e incluye todo el código necesario para que haya tiles destructibles (si no la activas, por mucho que tengas tiles de tipo 16 no pasará nada). La segunda, `BREAKABLE_WALLS_LIFE`, define el número de disparos que deben recibir los tiles destructibles para romperse. Obviamente, tienes que poner un número mayor de 0. Si pones un 3, el tile se romperá al tercer disparo que reciba.

Recuerda, además, que el tipo 16 por sí mismo no hace nada, sino que **debe ser combinado con otro comportamiento** para que tenga sentido: con obstáculos ($8+16 = 24$) o con tiles que matan ($1+16 = 17$).

Scripting

Las siguientes directivas sirven para activar el motor de scripting y definir un par de cosas relacionadas con el mismo. Por ahora las vamos a dejar sin activar, para poder ir compilando y probando el juego ya sin tener que hacer un script. Porque tenemos ganas ya, ¿no? Tranquilos, volveremos a ellas más adelante.

Con la primera activamos el sistema de scripting:

```
// #define ACTIVATE_SCRIPTING           // Activates msc scripting and flag related stuff
```

Con las siguientes lo configuramos:

```
#define MAX_FLAGS          32
#define SCRIPTING_DOWN      // Use DOWN as the action key.
//#define SCRIPTING_KEY_M    // Use M as the action key instead.
//#define SCRIPTING_KEY_FIRE // User FIRE as the action key instead.

#define ENABLE_EXTERN_CODE   // Enables custom code to be run from the script u
//#define ENABLE_FIRE_ZONE    // Allows to define a zone which auto-triggers "FI
```

1. `MAX_FLAGS` sirve para configurar el número de *flags* que estarán disponibles para el sistema de scripting. 32 es un valor muy adecuado para la mayoría de las cosas, pero siempre puedes ir aquí y ajustar el número una vez que hayas terminado de hacer el juego para afeitar unos cuantos bytes.
2. Las tres siguientes directivas sirven para elegir qué tecla de acción activará el scripting. Habrá que activar una de ellas y desactivar las otras dos. Podemos elegir entre que la tecla de acción sea "abajo", **M** o el botón de disparo (usando la configuración de controles que sea).
3. `ENABLE_EXTERN_CODE` permite que escribamos código C para responder al comando `EXTERN` del script. `EXTERN` tomará un parámetro y el intérprete lo pasará tal cual a una función especial `do_extern_action` que está en `/dev/my` y donde podremos añadir nuestro código. Ya lo trataremos en más detalle cuando hablemos del scripting.
4. `ENABLE_FIRE_ZONE` permite utilizar el comando `SET_FIRE_ZONE` del script, que activa un rectángulo en pantalla que lanzará las cláusulas de las secciones `PRESS_FIRE` de la pantalla actual cuando el personaje la toque. De nuevo, ya lo trataremos más adelante.

Como te he dicho, por ahora dejamos `ACTIVATE_SCRIPTING` desactivada. Ya la activaremos cuando empecemos a hacer nuestro script.

Timer

Se trata de un temporizador que podemos usar de forma automática o desde el script. El temporizador toma un valor inicial, va contando hacia abajo, puede recargarse, se puede configurar cada cuántos frames se decrementa o decidir qué hacer cuando se agota.

Con esta directiva activamos el sistema de timer. En **Dogmole** no lo usaremos.

```
//#define TIMER_ENABLE
```

Con las siguientes lo configuramos. Son unas cuantas. Veámoslas:


```
#define TIMER_INITIAL      99
#define TIMER_REFILL      25
#define TIMER_LAPSE       32
#define TIMER_START
```

1. `TIMER_INITIAL` especifica el valor inicial del temporizador.
2. Las recargas de tiempo, que se ponen con el Ponedor como *hotspots* de **tipo 5**, recargarán el valor especificado en `TIMER_REFILL`. El valor máximo del timer, tanto para el inicial como al recargar, es de 99.
3. `TIMER_LAPSE` controla intervalo de tiempo que transcurre entre cada decremento del temporizador, y está medido en frames. Los juegos de **MTE MK1** suelen ejecutarse a una media de 20 fps, para que te hagas una idea.
4. Si se define `TIMER_START`, el temporizador estará activo desde el principio del juego

Tenemos, además, algunas directivas que definen qué pasará cuando el temporizador llegue a cero. Hay que activar solo una de ellas, dejando las demás comentadas.:

```
#define TIMER_SCRIPT_0
//#define TIMER_GAMEOVER_0
//#define TIMER_KILL_0
```

1. `TIMER_SCRIPT_0` : Cuando llegue a cero el temporizador se ejecutará una sección especial del script, `ON_TIMER_OFF`. Es ideal para llevar todo el control del temporizador por scripting, como ocurre en **Cadàveriön**.
2. `TIMER_GAME_OVER_0` : El juego terminará cuando el temporizador llegue a cero (Game Over).
3. `TIMER_KILL_0` : Se restará una vida cuando el temporizador llegue a cero. Si se define esta última, además, se hace caso de las tres siguientes:

```
//#define TIMER_WARP_TO 0
//#define TIMER_WARP_TO_X 1
//#define TIMER_WARP_TO_Y 1
```

1. Si se define `TIMER_WARP_TO_0` además de `TIMER_KILL_0`, al morir cambiaremos a la pantalla indicada.
2. Si se define `TIMER_WARP_TO_X` y `TIMER_WARP_TO_Y` además de `TIMER_KILL_0`, al moriri nos moveremos a esta posición.

Pueden combinarse las tres para cambiar a una pantalla y a una posición concreta dentro de la misma.

Esperad, no os vayáis, que hay más:

```
//#define TIMER_AUTO_RESET  
#define SHOW_TIMER_OVER
```

1. `TIMER_AUTO_RESET` : Si se define esta opción, el temporizador volverá al máximo tras llegar a cero de forma automática. Si vas a realizar el control por scripting, mejor deja esta comentada.
2. `SHOW_TIMER_OVER` : Si se define esta, en el caso de que hayamos definido o bien `TIMER_SCRIPT_0` o bien `TIMER_KILL_0`, se mostrará un cartel de "TIME'S UP!" cuando el temporizador llegue a cero.

Como hemos dicho, el temporizador puede administrarse desde el script. Es interesante que, si decidimos hacer esto, activemos `TIMER_SCRIPT_0` para que cuando el temporizador llegue a cero se ejecute la sección `ON_TIMER_OFF` de nuestro script y que el control sea total.

Guardar estado

Se trata de definir *check points* en el juego. Se colocan como **hotspots de tipo 6**. Cuando el jugador los toca, se almacena su estado actual:

- Valor de todos los flags. (si aplica)
- Posición. (n_pant, tile X, tile Y)
- Tiempo. (si aplica)
- Munición. (si aplica)

Cuando esto ocurre, cuando se va a iniciar una nueva partida se da la opción entre empezar de nuevo o continuar desde el último *check point*.

Nótese que no es posible almacenar llaves y objetos, ya que esto implicaría hacer copias en memoria de estructuras grandes. Esta funcionalidad está más pensada para juegos en los que la lógica venga regida por un script.

También se puede configurar el motor para que la vuelta al check point se de siempre que perdamos una vida.

```
//#define ENABLE_CHECKPOINTS           // Enable checkpoints  
//#define CP_RESET_WHEN_DYING          // Move player to checkpoint when player dies.  
//#define CP_RESET_ALSO_FLAGS          // and also restore its flags / values
```

El comportamiento por defecto es que no pase nada al morir, y cuando acabe la partida podremos continuar desde el último *check point*.

1. Si se activa `CP_RESET_WHEN_DYING` , además, cada vez que se pierda una vida se trasladará al jugador al último *check point*.
2. Si se activa `CP_RESET_ALSO_VALUES` , además de lo anterior, se restaurará el valor de los flags.

No usaremos nada de esto en **Dogmole**, por lo que todo se queda comentado.

Directivas relacionadas con la vista genital

```
//#define PLAYER_GENITAL                // Enable top view.  
//#define TOP_OVER_SIDE                 // UP/DOWN has priority over LEFT/RIGHT
```

Si activamos `PLAYER_GENITAL` , el güego será de vista genital. Si no está activada, el güego será de vista lateral. Para **Dogmole** la dejamos desactivada, por tanto.

La siguiente, `TOP_OVER_SIDE` , define el comportamiento de las diagonales. Esto tiene utilidad sobre todo si tu güego tiene, además, disparos. Si se define `TOP_OVER_SIDE` , al desplazarse en diagonal el muñeco mirará hacia arriba o hacia abajo y, por tanto, disparará en dicha dirección. Si no se define, el muñeco mirará y disparará hacia la izquierda o hacia la derecha. Depende del tipo de juego o de la configuración del mapa te interesará más una u otra opción. No, no se puede disparar en diagonal.

```
//#define PLAYER_BOUNCE_WITH_WALLS      // Bounce when hitting a wall. Only really use
```

El jugador rebota contra las paredes, como en **Balowwnn**. Si le encuentras alguna utilidad, go!

Directivas relacionadas con la vista lateral

Aquí hay un montón de cosas:

Sartar

```
#define PLAYER_HAS_JUMP                 // If defined, player is able to jump.
```

Si se define esta, el jugador puede saltar. Si no se han activado los disparos, fire hará que el jugador salte. Si están activados, lo hará la tecla "arriba". También podemos usar la configuración de dos botones (que veremos más adelante), que asignará un botón a saltar y otro a disparar.

Jetpac

```
//#define PLAYER_HAS_JETPAC             // If defined, player can thrust a vertical je
```

Si definimos `PLAYER_HAS_JETPAC`, la tecla "arriba" hará que activemos un jetpac. Es compatible con poder saltar. Sin embargo, si activas a la vez el salto y el jetpac no podrás usar los disparos... aunque no lo hemos probado, a lo mejor pasa algo raro. No lo hagas. O, no sé, hazlo. Si no sabes qué es esto, juega a **Cheril the Goddess** o **Jet Paco**.

Pisar enemigos

```
#define PLAYER_STEPS_ON_ENEMIES           // If defined, stepping on enemies kills them
//#define PLAYER_CAN_STEP_ON_FLAG    1    // If defined, player can only kill when flag
#define PLAYER_MIN_KILLABLE            3    // Only kill enemies with id >= PLAYER_MIN_KIL
```

`PLAYER_STEPS_ON_ENEMIES` activa el motor de pisoteo y las otras dos, que lo configuran, son opcionales. Con este motor activado, el jugador podrá saltar sobre los enemigos para matarlos.

1. `PLAYER_CAN_STEP_ON_FLAG` : Si se activa, el valor del flag indicado deberá valer 1 para que el pisoteo funcione. Como ganar un super poder.
2. `PLAYER_MIN_KILLABLE` nos sirve para hacer que no todos los enemigos se puedan matar. En **Dogmole**, sólo podremos matar a los hechiceros, que son de tipo 3. Ojo con esto: si ponemos un 1 podremos matar a todos, si ponemos un 2, a los enemigos tipo 2 y 3, y si ponemos un 3 sólo a los de tipo 3. O sea, se podrá matar a los enemigos cuyo tipo sea mayor o igual al valor que se configure.

Motor de Bootèe

```
//#define PLAYER_BOOTEE                     // Always jumping engine. Don't forget to disa
```

Esta directiva activa el salto continuo (el del juego **Botèe**). No es compatible con `PLAYER_HAS_JUMP` o con `PLAYER_HAS_JETPAC`. Si activas `PLAYER_BOOTEE`, tienes que desactivar los otros dos. Si no, cacota.

Definición del teclado

MTE MK1 soporta dos configuraciones de controles, una basada en un sólo botón, que además se puede emplear cómodamente con joysticks tipo Sinclair o Kempston, y otra basada en dos botones, más pensada para usar el teclado como si fuera un pad de Master System y NES: con una "cruceta" virtual y dos botones.

```
//#define USE_TWO_BUTTONS
```

Por defecto la configuración será la de un solo botón. Para ello dejamos `USE_TWO_BUTTONS` desactivada.

Las teclas que se usarán en el juego pueden definirse fácilmente para cualquiera de las dos configuraciones dando valores a la estructura "keys" y a dos variables que se definen justo después:

```
#ifndef USE_TWO_BUTTONS
    // Define here if you selected the TWO BUTTONS configuration

    struct sp_UDK keys = {
        0x047f, // .fire
        0x04fd, // .right
        0x01fd, // .left
        0x02fd, // .down
        0x02fb  // .up
    };

    int key_jump = 0x087f;
    int key_fire = 0x047f;
#else
    // Define here if you selected the NORMAL configuration

    struct sp_UDK keys = {
        0x017f, // .fire
        0x01df, // .right
        0x02df, // .left
        0x01fd, // .down
        0x01fb  // .up
    };
#endif
```

De "fábrica", las teclas son **W A S D** para la "cruzeta" y **N M** para los "botones" en el modo de dos botones, y **O P Q A** y **SPACE** en el modo normal.

Si quieres otras teclas en la configuración que vayas a usar, el tema se trata de cambiar esos numeritos, lo cual no es nada complicado porque hemos incluido esta tabla justo arriba para que la tengas enfrente a la hora de cambiarlos. La tabla es algo así:

B\A	01	02	04	08	10
f7	1	2	3	4	5
fb	Q	W	E	R	T
fd	A	S	D	F	G
fe	CS	Z	X	C	V

B\A	01	02	04	08	10
ef	0	9	8	7	6
df	P	O	I	U	Y

B\A	01	02	04	08	10
bf	EN	L	K	J	H
7f	SP	SS	M	N	B

Pensemos en los números como en `0xAABB`. Lo que tenemos que hacer es buscar la tecla que queremos en la tabla (que si te fijas es como las dos partes del teclado de un Spectrum original), y mirar el número `AA` de la columna y el número `BB` de la fila correspondientes, y con ellos formar el `0xAABB` que necesitamos. Por ejemplo, la tecla "8" está en la columna `04` y la fila `ef`, por lo que el número que la representaría sería `0x04ef`.

Para la configuración de un botón hay que sustituir la estructura de abajo, en orden *disparo, derecha, izquierda, abajo, arriba* (como queda indicado en los comentarios). Para la configuración de los dos botones habrá que sustituir la estructura de arriba, en el mismo orden, y adicionalmente rellenar el valor de las teclas *salto* y *disparo*.

Configuración de la pantalla

En esta sección colocamos todos los elementos en la pantalla. ¿Recordáis cuando estábamos diseñando el marco? Pues es aquí donde vamos a poner todos los valores que dejamos apuntados. Estas directivas son de las que se pueden invalidar con el valor 99. Normalmente el motor no tratará de mostrar marcadores que no va a usar, como por ejemplo el de munición si el juego no activa `MAX_AMMO`. Sin embargo, hay ocasiones en la que necesitamos tener activado uno de los motores pero que no se muestre el marcador correspondiente. Para lograrlo, ponemos sus coordenadas a 99 y los desactivaremos.

```
#define VIEWPORT_X      1      //
#define VIEWPORT_Y      0      // Viewport character coordinates
```

Definen la posición (siempre en coordenadas de carácter) del área de juego. Nuestro área de juego empezará en (0, 1), y esos son los valores que le damos a `VIEWPORT_X` y `VIEWPORT_Y`.

```
#define LIFE_X          22      //
#define LIFE_Y          21      // Life gauge counter character coordinates
```

Definen la posición del marcador de las vidas (del numerico, vaya).

```
#define OBJECTS_X       17      //
#define OBJECTS_Y       21      // Objects counter character coordinates
```

Definen la posición del contador de objetos, si usamos objetos (la posición del numerico). Recordad que en **Dogmole**, por configuración, en estas coordenadas se mostrará el valor del flag que usaremos

para llevar el recuento de los objetos, y no el contador interno de objetos.

```
#define OBJECTS_ICON_X      15      //  
#define OBJECTS_ICON_Y      21      // Objects icon character coordinates (use wit
```

Estas dos se utilizan con `ONLY_ONE_OBJECT` : Cuando “llevemos” el objeto encima, el motor nos lo indicará haciendo parpadear el icono del objeto en el marcador. En `OBJECTS_ICON_X` e `Y` indicamos dónde aparece dicho icono (el tile con el dibujito de la caja). Como verás, esto obliga a que estemos usando el icono en el marcador, y no un texto u otra cosa. Sí, es una limitación.

```
#define KEYS_X              27      //  
#define KEYS_Y              21      // Keys counter character coordinates
```

Define la posición del contador de llaves.

```
#define KILLED_X            12      //  
#define KILLED_Y            21      // Kills counter character coordinates
```

La cuenta de enemigos matados o moridos.

```
#define AMMO_X              99      //  
#define AMMO_Y              99      // Ammo counter character coordinates  
#define TIMER_X             99      //  
#define TIMER_Y             99      // Timer counter coordinates
```

Estas últimas son para colocar un marcador de munición y mostrar el valor del timer, pero como no las vamos a usar, las dejamos a `99` .

La linea de texto

MTE_MK1 permite imprimir una sencilla linea de texto desde el script. Para activar esta funcionalidad, define `LINE_OF_TEXT` con la coordenada `Y` de la linea de texto. Adicionalmente, `LINE_OF_TEXT_X` sirve para definir su coordenada `x` (0 o 1, normalmente) y `LINE_OF_TEXT_ATTR` el color del texto, en atributos de Spectrum (recuerda, el número que salga de aplicar la fórmula `INK + 8*PAPER + 64*BRIGHT + 128*FLASH`).

Efectos gráficos

En esta sección se definen diversos efectos gráficos (muy básicos) que podemos activar y que controlarán la forma en la que se muestra el güego. Básicamente podemos configurar **MTE MK1** para

que pinte sombras o no a la hora de construir el escenario, y definir un par de cosas relacionadas con los sprites y tal.

```
// #define USE_AUTO_SHADOWS // Automatic shadows made of darker attributes
// #define USE_AUTO_TILE_SHADOWS // Automatic shadows using specially defined t
```

Estas dos se encargan de las sombras de los tiles, y podemos activar sólo una de ellas, o ninguna. Si recuerdas, en el capítulo del tileset hablamos de sombras automáticas. Si activamos `USE_AUTO_SHADOWS`, los tiles obstáculo dibujan sombras sobre los tiles traspasables usando sólo atributos (a veces queda resultón, pero no siempre).

`USE_AUTO_TILE_SHADOWS` emplea versiones sombreadas de los tiles de fondo para hacer las sombras, tal y como explicamos. Desactivando ambas no se dibujarán sombras.

En **Dogmole** no usaremos sombras de ningún tipo, porque las de atributos no nos gustan y las de tiles no nos las podemos permitir porque usaremos esos tiles para embellecer algunas pantallas imprimiendo gráficos por medio del script.

```
// #define UNPACKED_MAP // Full, uncompressed maps. Shadows settings a
```

Si definimos `UNPACKED_MAP` estaremos diciéndole al motor que nuestro mapa es de 48 tiles.

```
// #define NO_MASKS // Sprites are rendered using OR instead of ma
// #define MASKED_BULLETS // If needed
```

Cosas de máscaras

`NO_MASKS` hace que los sprites del jugador y los enemigos se dibujen sin máscaras, con un sencillo OR. Esto funciona bien si tus fondos son planos o muy poco complejos porque el render es más rápido y ganamos algunos ciclos, lo que significa más faps de esos.

`MASKED_BULLETS` hace que los sprites de las balas usen máscaras para más bonitor general si usas fondos complejos.

Tiles animados

MTE MK1 te da la opción de usar tiles animados de una forma muy sencilluna. Básicamente un tile animado es una pareja de tiles correlativos en el tileset. Tú colocas (mediante mapa o mediante scripting) el primero de ellos en pantalla y el motor lo animará alternando entre éste y su pareja cada cierto tiempo un poco al azar (básicamente elige uno de entre todos los que hay cada frame y lo cambia). En **MTE MK1** los tiles animados tienen que estar al final del tileset todos seguidos, con lo

que realmente sólo tienen utilidad si tu mapa es UNPACKED (de 48 tiles) o si vas a poner decoraciones con el script.

```
// #define ENABLE_TILANIMS 32 // If defined, animated tiles are enabled.
// the value specifies first animated tile pa
```

El valor configurado es el del primer par de tiles animados, si ENABLE_TILANIMS está activo. Por ejemplo, si pones 44 , tus tiles animados serán la pareja 44, 45 y la pareja 46, 47 . Siempre que pongas un tile 44 o 45 (en tu mapa unpacked, o desde el script si tu tileset para el mapa es de 16 tiles), se animará.

Color del marcador

```
#define HUD_INK 7 // Use this attribute for digits in the hud
```

Define de qué color se pintan los números en el marcador del juego.

Animación custom

```
// #define PLAYER_CUSTOM_ANIMATION // Code your own animation in my/custom_animat
```

Si definimos PLAYER_CUSTOM_ANIMATION tendremos que ocuparnos de escribir el código que seleccionará el gráfico del sprite del jugador en cada frame de juego basándonos en el estado actual del jugador y escribirlo en p_next_frame . Esto se hace en my/custom_animation.h . Para facilitarte el trabajo, estas son algunas de las variables que puedes consultar:

Variables	Contenido
gpx , gpy	Posición (x, y) en píxels del jugador.
p_vx , p_vy	Velocidad (horizontal, vertical) del jugador.
p_facing	Dirección a la que mira el jugador (*)
possee	1 si el jugador está sobre una plataforma fija.
p_gotten	1 si el jugador está sobre una plataforma movil.
player_cells	Es un array con los 8 frames del sprite.

(*) p_facing será 0 (derecha) y 1 (izquierda) en vista lateral, o tomará los valores FACING_LEFT , FACING_RIGHT , FACING_UP y FACING_DOWN en vista genital.

El código que escribas debe terminar asignando un componente de `player_cells` al puntero `p_next_frame`. Obviamente, no necesitas escribir un código de animación personalizado para el jugador si no estás haciendo cosas muy raras.

Pausa y abortar

```
//#define PAUSE_ABORT
```

```
// Add h=PAUSE, y=ABORT
```

Si se define, la tecla *H* pausará el juego y la tecla *Y* volverá a la pantalla de título.

Get X More

```
//#define GET_X_MORE
```

```
// Shows "get X more" when getting an object
```



Si se define, se mostrará un cuadro con los objetos que te quedan por coger cada vez que cojas uno.

Configuración del movimiento del personaje principal

En esta sección configuraremos como se moverá nuestro jugador. Es muy probable que los valores que pongas aquí tengas que irlos ajustando por medio del método de prueba y error. Si tienes unas nociones básicas de física, te vendrán muy bien para saber cómo afecta cada parámetro.

Básicamente, tendremos lo que se conoce como *Movimiento Rectilíneo Uniformemente Acelerado* (MRUA) en cada eje: el horizontal y el vertical. Básicamente tendremos una posición (digamos *X*) que se verá afectada por una velocidad (digamos *VX*), que a su vez se verá afectada por una aceleración (o sea, *AX*).

Los parámetros siguientes sirven para especificar diversos valores relacionados con el movimiento en cada eje en juegos de vista lateral. En los de vista genital, se tomarán los valores del eje horizontal también para el vertical.

Para obtener la suavidad de los movimientos sin usar valores en coma flotante (que son muy costosos de manejar), usamos aritmética de punto fijo. Básicamente los valores se expresan en 1/64 de pixel. Esto significa que el valor empleado se divide entre 64 a la hora de mover los sprites reales a la pantalla. Eso nos da una precisión de 1/64 pixels en ambos ejes, lo que se traduce en mayor suavidad de movimientos. En palabras techis, estamos usando 10 bits para la parte entera y 6 para la parte "decimal", así que decimos que nuestro punto fijo es **10.6**.

Somos conscientes en mojonía de que este apartado es especialmente densote, así que no te preocupes demasiado si, de entrada, te pierdes un poco. Experimenta con los valores hasta que encuentres la combinación ideal para tu juego.

Eje vertical en güegos de vista lateral

Al movimiento vertical le afecta la gravedad. La velocidad vertical será incrementada por la gravedad hasta que el personaje aterrice sobre una plataforma u obstáculo. Además, a la hora de saltar, habrá un impulso inicial y una aceleración del salto, que también definiremos aquí. Estos son los valores para Dogmole; acto seguido detallaremos cada uno de ellos:

```
#define PLAYER_MAX_VY_CAYENDO    512    // Max falling speed
#define PLAYER_G                  48     // Gravity acceleration

#define PLAYER_VY_INICIAL_SALTO  96     // Initial jump velocity
#define PLAYER_MAX_VY_SALTANDO   312    // Max jump velocity
#define PLAYER_INCR_SALTO        48     // acceleration while JUMP is pressed

#define PLAYER_INCR_JETPAC       32     // Vertical jetpac gauge
#define PLAYER_MAX_VY_JETPAC    256    // Max vertical jetpac speed
```

Caída libre

La velocidad del jugador, que medimos en pixels/frame será incrementada en $\text{PLAYER_G} / 64$ pixels/frame hasta que llegue al máximo especificado por $\text{PLAYER_MAX_CAYENDO} / 64$. Con los valores que hemos elegido para **Dogmole**, la velocidad vertical en caída libre será incrementada en $48 / 64 = 0,75$ pixels/frame hasta que llegue a un valor de $512 / 64 = 8$ píxels/frame. O sea, Dogmole caerá más y más rápido hasta que llegue a la velocidad máxima de 8 píxels por frame.

Incrementar `PLAYER_G` hará que se alcance la velocidad máxima mucho antes (porque la aceleración es mayor). Estos valores afectan al salto: a mayor gravedad, menos saltaremos y menos nos durará el impulso inicial. Modificando `PLAYER_MAX_CAYENDO` podremos conseguir que la velocidad máxima, que se alcanzará antes o después dependiendo del valor de `PLAYER_G`, sea mayor o menor. Usando valores pequeños podemos simular entornos de poca gravedad como el espacio exterior, la superficie de la luna, o el fondo del mar. El valor de 512 (equivalente a 8 píxels por frame) podemos considerarlo el máximo, ya que valores superiores y caídas muy largas podrían resultar en glitches y cosas raras.

Salto

Los saltos se controlan usando tres parámetros. El primero, `PLAYER_VY_INICIAL_SALTO`, será el valor del impulso inicial: cuando el jugador pulse la tecla de salto, la velocidad vertical tomará automáticamente el valor especificado en dirección hacia arriba. Mientras se esté pulsando salto, y durante unos ocho frames, la velocidad se seguirá incrementando en el valor especificado por `PLAYER_INCR_SALTO` hasta que lleguemos al valor `PLAYER_MAX_VY_SALTANDO`. Esto se hace para que podamos controlar la fuerza del salto pulsando la tecla de salto más o menos tiempo. El período de aceleración, que dura 8 frames, es fijo y no se puede cambiar (para cambiarlo habría que tocar el motor), pero podemos conseguir saltos más bruscos subiendo el valor de `PLAYER_INCR_SALTO` y `PLAYER_MAX_VY_SALTANDO`.

Normalmente encontrar los valores ideales exige un poco de prueba y error. Hay que tener en cuenta que al saltar horizontalmente de una plataforma a otra también entran en juego los valores del movimiento horizontal, por lo que si has decidido que en tu güego el prota debería poder sortear distancias de X tiles tendrás que encontrar la combinación óptima jugando con todos los parámetros.

Los dos valores que quedan no se usan en **Dogmole** porque tienen que ver con el jetpac. El primero es la aceleración que se produce mientras pulsamos la tecla arriba y el segundo el valor máximo que puede alcanzarse. Si tu juego no usa jetpac estos valores no se utilizan para nada.

Eje horizontal en güegos de vista lateral / comportamiento general en vista genital

El siguiente set de parámetros describen el comportamiento del movimiento en el eje horizontal si tu güego es en vista lateral, o los de ambos ejes si tu güego es en vista genital. Estos parámetros son mucho más sencillos:

```
#define PLAYER_MAX_VX      256      // Max velocity
#define PLAYER_AX          48        // Acceleration
#define PLAYER_RX          64        // Friction
```

1. `PLAYER_MAX_VX` indica la velocidad máxima a la que el personaje se desplazará horizontalmente (o en cualquier dirección si el güego es en vista genital). Cuanto mayor sea este número, más correrá, y más lejos llegará cuando salte (horizontalmente). Un valor de 256 significa que el personaje correrá a un máximo de $256/64 = 4$ píxels por frame.
2. `PLAYER_AX` controla la aceleración que sufre el personaje mientras el jugador pulsa una tecla de dirección. Cuando mayor sea el valor, antes se alcanzará la velocidad máxima. Valores pequeños hacen que "cueste arrancar". Un valor de 48 significa que se tardará aproximadamente 6 frames ($256/48$) en alcanzar la velocidad máxima.
3. `PLAYER_RX` es el valor de fricción o rozamiento. Cuando el jugador deja de pulsar la tecla de movimiento, se aplica esta aceleración en dirección contraria al movimiento. Cuanto mayor sea el valor, antes se detendrá el personaje. Un valor de 64 significa que tardará 4 frames en detenerse si iba al máximo de velocidad.

Usar valores pequeños de `PLAYER_AX` y `PLAYER_RX` harán que el personaje parezca resbalar. Es lo que ocurre en güegos como, por ejemplo, **Viaje al Centro de la Napia**. Salvo excepciones misteriosas, casi siempre "se güega mejor" si el valor de `PLAYER_AX` es mayor que el de `PLAYER_RX`.

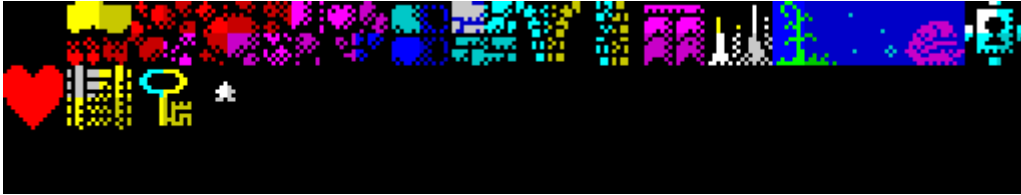
Comportamiento de los tiles

¿Recordáis que explicamos como cada tile tiene un tipo de comportamiento asociado? Tiles que matan, que son obstáculos, plataformas, o interactivables. En esta sección (la última, por fin) de

my/config.h definimos el comportamiento de cada uno de los 48 tiles del tileset completo **aunque estemos usando mapas *packed* de 16 tiles**.

Tendremos que definir los comportamientos de los 48 tiles sin importar que nuestro güego utilice un tileset de 16 porque los tiles extra que no van en el mapa podrían llegar a aparecer por obra del script o de inyección de código. En Dogmole no vamos a poner ningún obstáculo "fuera de tileset", pero güegos como los de la saga de **Ramiro el Vampiro** sí que tienen obstáculos de este tipo.

Para poner los valores, simplemente abrimos el tileset y nos fijamos, están en el mismo orden en el que aparecen los tiles en el tileset:



```
unsigned char behs [] = {  
    0, 8, 8, 8, 8, 8, 0, 8, 0, 0, 8, 1, 0, 0, 0,10,  
    0, 0,0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,  
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0  
};
```

Como vemos, tenemos el primer tile vacío (tipo 0), luego tenemos cinco tiles de roca que son obstáculos (tipo 8), otro tile de fondo (la columnita, tipo 0), el tile de ladrillos (tipo 8), los dos tiles que forman un arco (tipo 4, plataforma, y tipo 0, traspasable), las tejas (tipo 8), unos pinchos que matan (tipo 1), tres tiles de fondo (tipo 0), y el tile cerrojo (tipo 10, interactuable).

Cuando modifiques estos valores ten cuidado con las comas y que no se te baile ningún número.

Recuerda, además, que los tipos de tile son combinables *hasta donde tenga sentido* sumando los valores: por ejemplo, destructible y que mata sería $16+1 = 17$, etc.

¡Arf, arf, arf!

Ah, ¿aún estás ahí? Pensaba que ya había conseguido aburrirte y quitarte todas las ganas de seguir con esto. Bien, veo que eres un tipo constante y con mucho podewwwr. Sí, ya hemos terminado de configurar nuestro güego. Ahora viene cuando lo compilamos.

A estas alturas del cuento ya tendríamos que tener todo en su sitio y listo (salvo el script) para ser compilado. Incluso `compile.bat` debería tener ya el par de modificaciones necesarias (el nombre del juego y el tamaño del mapa en el conversor).

Echando un vistazo a `compile.bat`

Como no nos gustan las cajas negras vamos a echar un vistazo a `compile.bat` para que todos le veamos bien el totete.

```
@echo off

set game=dogmole

echo Compilando script
cd ..\script
msc %game.spt msc.h 25 > nul
copy *.h ..\dev > nul
cd ..\dev

echo Convirtiendo mapa
..\utils\mapcnv.exe ..\map\mapa.map 8 3 15 10 15 packed > nul
cd ..\dev

echo Convirtiendo enemigos/hotspots
..\utils\ene2h.exe ..\enems\enems.ene enems.h

echo Importando GFX
..\utils\ts2bin.exe ..\gfx\font.png ..\gfx\work.png tileset.bin forcezero >nul

..\utils\sprcnv.exe ..\gfx\sprites.png sprites.h > nul

..\utils\sprcnvbin.exe ..\gfx\sprites_extra.png sprites_extra.bin 1 > nul
..\utils\sprcnvbin8.exe ..\gfx\sprites_bullet.png sprites_bullet.bin 1 > nul

..\utils\png2scr.exe ..\gfx\title.png ..\gfx\title.scr > nul
..\utils\png2scr.exe ..\gfx\marco.png ..\gfx\marco.scr > nul
..\utils\png2scr.exe ..\gfx\ending.png ..\gfx\ending.scr > nul
..\utils\png2scr.exe ..\gfx\loading.png loading.bin > nul
..\utils\apack.exe ..\gfx\title.scr title.bin > nul
..\utils\apack.exe ..\gfx\marco.scr marco.bin > nul
..\utils\apack.exe ..\gfx\ending.scr ending.bin > nul

echo Compilando juego
zcc +zx -vn mk1.c -o %game%.bin -lsplib2_mk2.lib -zorg=24000 > nul
..\utils\printsize.exe %game%.bin

echo Construyendo cinta
rem cambia LOADER por el nombre que quieres que salga en Program:
..\utils\bas2tap -a10 -sLOADER loader\loader.bas loader.tap > nul
..\utils\bin2tap -o screen.tap -a 16384 loading.bin > nul
..\utils\bin2tap -o main.tap -a 24000 %game%.bin > nul
copy /b loader.tap + screen.tap + main.tap %game%.tap > nul

echo Limpiando
del loader.tap > nul
del screen.tap > nul
del main.tap > nul
del ..\gfx\*.scr > nul
del *.bin > nul
```


echo Hecho!

En él vemos que se realiza toda una serie de fullerías dedicadas a convertir automáticamente todos los archivos implicados en la construcción de nuestro juego: script, mapa, enemigos y gráficos. Acto seguido **compila el juego** y posteriormente **construye la cinta**.

Preparando el entorno

El script `compile.bat` necesita saber donde está **z88dk**. Nosotros sabemos que está en `C:\z88dk10` porque lo descomprimimos ahí en el paso 1, pero cada vez que abramos una ventana de línea de comandos nueva habrá que recordarle al entorno donde está. Para eso ejecutamos `setenv.bat` que debería mostrar la versión de `zcc` si todo fue bien, algo así:

```
$ setenv.bat
zcc - Frontend for the z88dk Cross-C Compiler
v2.59 (C) 16.4.2010 D.J.Morris
```

Una vez hecho esto no tenemos más que ejecutar `compile.bat` y, tras un mágico proceso que podremos ver suceder ante nuestros atónitos óculos, obtendremos un archivo `tap` que se llame igual que nuestro juego (el valor de `game` en `compile.bat`). En nuestro caso `dogmole.tap`. Nos vamos rápido y lo cargamos en el emu para ver que todo se mueve y está en su sitio.

Y como ya nos duele la cabeza a tí y a mí, pues lo dejamos hasta el capítulo que viene. Si decidiste empezar con algo sencillo tipo **Lala Prologue** o **Sir Ababol**, ya has terminado. Si no, aún hay muchas cosas por hacer.

Capítulo 8: empezando con el scripting

Antes que nada, bájate el paquete de materiales correspondiente a este capítulo pulsando en este enlace:

[Material del capítulo 8](#)

¡Hombre, por fin!

Sí, ya. Pero ahora te vas a cagar. Porque esto puede ser tan denso y chungo como tú quieras. Nah, en serio, no es nada. Vamos a hacerlo además con vaselina. En este primer capítulo explicaremos cómo está montado el sistema, para que entiendas qué hace, para qué sirve, y cómo funciona, y terminaremos viendo ejemplos super sencillos de nivel 1, fácil fácil. El siguiente capítulo terminaremos de hacer el script de Dogmole Tuppowski y, a partir de ahí, exploraremos, parte por parte, lo que se puede hacer con el script. Porque se puede hacer mucho y variado.

¡Vamos a ello, pues!

Vale. ¿Qué es un script de *MTE MK1*? Pues no es más que un conjunto de comprobaciones con acciones asociadas, organizadas en secciones, que sirven para definir el gameplay de tu güego. O sea, las cosas que pasan, y las cosas que tienen que pasar para que todo vaya guay, o para que todo vaya mal.

A ver, sin script tienes un gameplay básico. Coger X objetos para terminar, matar X bichos... Ir más allá de eso precisa comprobaciones y acciones relacionadas: si estamos en tal sitio y hemos hecho tal cosa, abrir la puerta del castillo. Si entramos en la pantalla tal y hablamos con el personaje cual, que salga el texto "OLA K ASE" y suene un ruidito. A eso nos referimos.

El script te sirve desde para colocar un tile bonito en la pantalla 4 y un texto que ponga "ESTÁS EN TU CASA" hasta para reaccionar a lo que hagas en una pantalla, comprobar que has hecho otras cosas, ver que has empujado tal tile, y entonces encender el temporizador o cambiar el escenario o lo que sea.

En cuanto sepas las herramientas de las que dispones seguro que se te ocurren mil cosas que hacer. Muchas veces descubrimos aplicaciones que ni siquiera sabíamos que eran posibles cuando nos pusimos a diseñar el sistema, así que ya ves.

¡Esto es lo realmente divertido!

Pero es programar.

Claro, cojones, pero una cosa es tener que programar un gameplay en C y meterlo en el motor y otra cosa es tener un lenguaje específicamente diseñado para describir un gameplay y que es tan sencillo de aprender y dominar. Porque estoy seguro de que a muchos o va a sonar cómo está montado esto, sobre todo si algún día habéis hecho una aventura con el PAWS o el GAC o habéis trasteado con algún game maker. Porque a los Mojones nos encanta eso de reinventar ruedas, y resulta que el super sistema que ideamos es el más usado para estos menesteres de todos cuantos existen. Ya lo verás.

Vale, cuéntame como va eso.

De acuerdo. A ver si lo puedo decir de un tirón, y luego ya lo vamos desglosando: un script se compone de secciones. Cada sección no es más que un conjunto de cláusulas. Cada cláusula está formada por una lista de comprobaciones y una lista de comandos. Si se cumplen todas y cada una de las comprobaciones de la lista, se ejecutarán, en orden, todos y cada uno de los comandos. El motor del güego llamará al motor de scripting en determinadas ocasiones, ejecutando una de esas secciones. Ejecutar una sección significa ir cláusula por cláusula haciendo las comprobaciones de su lista de comprobaciones y, si se cumplen, ejecutar en orden los comandos de su lista de comandos. Ese es el concepto importante que hay que tener claro.

Para saber en qué momentos el motor del güego llama al motor de scripting, tenemos que entender qué son las secciones y qué tipos de sección hay:

1. `ENTERING SCREEN n` : con `n` siendo un número de pantalla, se ejecutan justo al entrar en una nueva pantalla, una vez dibujado el escenario, inicializados los enemigos, y colocados los hotspots. Las podemos utilizar para modificar el escenario o inicializar variables. Por ejemplo, asociada a la pantalla 3, podemos colocar un script que compruebe si hemos matado a todos los enemigos y, si no, que pinte un obstáculo para que no podamos pasar.
2. `ENTERING ANY` : es una sección especial que se ejecuta para TODAS las pantallas, justo antes de `ENTERING SCREEN n` . O sea, cuando tú entras en la pantalla 3, se ejecutará primero la sección `ENTERING ANY` (si existe en el script), y justo después se ejecutará la sección `ENTERING SCREEN 3` (si existe en el script).
3. `ENTERING GAME` : se ejecuta una sola vez al empezar el juego. Es lo primero que se ejecuta. Lo puedes usar para inicializar el valor de variables, por ejemplo. Ya veremos esto luego.
4. `PRESS_FIRE AT SCREEN n` : con `n` siendo un número de pantalla, se ejecuta en varios supuestos estando en la pantalla `n` : si el jugador pulsa el botón de acción, al empujar un bloque si hemos activado la directiva `PUSHING_ACTION` , o al entrar en una zona especial definida desde scripting llamada "fire zone" (que ya explicaremos) si hemos activado la directiva `ENABLE_FIRE_ZONE` . Normalmente usaremos estas secciones para reaccionar a las acciones del jugador.
5. `PRESS_FIRE AT ANY` : se ejecuta en todos los supuestos anteriores, para cualquier pantalla, justo antes de `PRESS_FIRE AT SCREEN n` . O sea, si pulsamos acción en la pantalla 7, se ejecutarán las cláusulas de `PRESS_FIRE AT ANY` y luego las de `PRESS_FIRE AT SCREEN 7` .

6. `ON_TIMER_OFF` : se ejecuta cuando el temporizador llegue a cero, si tenemos activado el temporizador y hemos configurado que ocurra esto con la directiva `TIMER_SCRIPT_0` .

7. `PLAYER_KILLS_ENEMY` : Se ejecuta cuando matamos a un malo.

Por cierto, que no es obligatorio escribir todas las secciones posibles. El motor ejecutará una sección sólo si existe. Por ejemplo, si no hay nada que hacer en la pantalla 8, pues no habrá que escribir ninguna sección para la pantalla 8. Si no hay ninguna acción común al entrar en todas las pantallas, no habrá sección `ENTERING ANY` . Y así. Si no hay nada que ejecutar, el motor no ejecuta nada y ya.

A ver, recapitulemos: ¿para qué tanto pifostio de secciones y cacafuti? Muy sencillo: por un lado porque, por lo general, las comprobaciones y acciones serán específicas de una pantalla. Esto es de cajón. Pero lo más importante es que estamos en un micro de 8 bits y no nos podemos permitir estar continuamente haciendo todas las comprobaciones. No tenemos tiempo de frame, por lo que hay que dejarlas para momentos aislados: nadie se va a coscar si se tardan unos milisegundos más al cambiar de pantalla o si la acción se detiene brevemente al pulsar la tecla de acción.

Guardando valores: las flags

Antes de que podamos seguir, tenemos que explicar otro concepto más: **los flags**, que no son más que **variables** donde podemos almacenar valores que posteriormente podremos consultar o modificar desde el script, y que además **nos sirven en algunos casos para comunicarnos con el motor**, como habrás podido discernir si te empapaste bien del capítulo anterior.

Muchas veces necesitaremos recordar que hemos hecho algo, o contabilizar cosas. Para ello tendremos que almacenar valores, y para ello tenemos las flags. En principio, tenemos 32 flags, numeradas del 0 a al 31, aunque este número puede modificarse fácilmente cambiando el valor de `MAX_FLAGS` en `my/config.h` .

Cada flag puede almacenar **un valor de 0 a 127**, lo cual nos da de sobra para un montón de cosas. La mayoría del tiempo sólo estaremos almacenando un valor booleano (0 o 1).

Valores numéricos y flags

En el script, la mayoría de las comprobaciones y comandos toman valores numéricos. Por ejemplo, `IF PLAYER_TOUCHES 4, 5` evaluará a "cierto" si el jugador está tocando la casilla de coordenadas (4, 5). Si antepone un `#` al número, estaremos **referenciando el valor del flag correspondiente**, de forma que `IF PLAYER_TOUCHES #4, #5` evaluará a "cierto" si el jugador está tocando la casilla de coordenadas almacenadas en los flags 4 y 5, sea cual sea este valor.

Este nivel de indirección (apréndete esa palabra para decirla en la discoteca: las nenas caen fulminadas ante los programadores que conocen este concepto) es realmente útil porque así podrás ahorrar mucho código. Por ejemplo, es lo que permite, en **Cadàveriön**, que el control del número de estatuas colocadas o de eliminar la cancela que bloquea la salida de cada pantalla puedan hacerse

desde una única sección común: todas las coordenadas están almacenadas en flags y usamos el operador # para acceder a sus valores en las comprobaciones.

Afúf.

¿Mucha información? Soy consciente de ello. Pero en cuanto lo veas en movimiento seguro que lo pillas del tirón. Vamos a empezar con los ejemplos más sencillos de scripting viendo algunas de las secciones que necesitamos para nuestro **Dogmole**, que iremos construyendo poco a poco. En esto dedicaremos este capítulo y el siguiente. Luego le llegará el turno a **el manual de MSC3**, donde es explica **todo** lo que puedes hacer con el script de una forma más directa.

¿Cómo activo el scripting? ¿Dónde se meten los comandos?

Para activar el scripting tendremos activarlo y configurarlo en `my/config.h`. Recordemos que las directivas relacionadas con la activación y configuración del scripting son estas:

```
#define ACTIVATE_SCRIPTING           // Activates msc scripting and flag related st
#define MAX_FLAGS                    32
#define SCRIPTING_DOWN              // Use DOWN as the action key.
// #define SCRIPTING_KEY_M          // Use M as the action key instead.
// #define SCRIPTING_KEY_FIRE       // User FIRE as the action key instead.
// #define ENABLE_EXTERN_CODE       // Enables custom code to be run from the scri
// #define ENABLE_FIRE_ZONE         // Allows to define a zone which auto-triggers
```

La primera directiva, `ACTIVATE_SCRIPTING`, es la que activará el motor de scripting, y añadirá el código necesario para que se ejecute la sección correcta del script en el momento preciso. Como hemos mencionado, `MAX_FLAGS` controla el número de flags disponible.

De las tres siguientes, tendremos que activar solo una, y sirven para configurar qué tecla será la tecla de acción, la que lance los scripts `PRESS_FIRE AT ANY` y `PRESS_FIRE AT SCREEN n`, o sea, **la tecla de acción**. La primera, `SCRIPTING_DOWN`, configura la tecla “abajo”. Esta es perfecta para gñuegos de perspectiva lateral, ya que esta tecla no se usa para nada más. La segunda, `SCRIPTING_KEY_M` habilita la tecla “M” para lanzar el script. La tercera, `SCRIPTING_KEY_FIRE`, configura la tecla de disparo (o el botón del joystick) para tal menester. Obviamente, si tu juego incluye disparos, no puedes usar esta configuración. Bueno, si puedes, pero allá tú.

La siguiente directiva, `ENABLE_EXTERN_CODE`, sirve para ejecutar código C que escribas tú desde el script. Hay un comando de script especial, `EXTERN n`, donde `n` es un número de 0 a 255, que lo que hace es llamar a una función de C situada en el archivo `my/extern.h` pasándole ese número. En esta función puedes añadir el código C que te de la gana y que necesites para hacer cosas divertidas. Si no vas a necesitar programar tus propios comportamientos en C, déjala desactivada y ahorra unos bytes.

Por último, `ENABLE_FIRE_ZONE` sirve para que podamos definir un rectángulo especial dentro del area de juego de la pantalla en curso. Normalmente, usaremos en `ENTERING_SCREEN n` para definir el rectángulo usando el comando `SET_FIRE_ZONE x1, y1, x2, y2` (en pixels) o `SET_FIRE_ZONE_TILES x1, y1, x2, y2` (en coordenadas de tiles, más fáciles de manejar). Cuando el jugador esté dentro de este rectángulo especial, se ejecutarán los scripts `PRESS_FIRE AT ANY` y `PRESS_FIRE AT_SCREEN n` de la pantalla actual. Esto viene realmente bien para poder ejecutar acciones sin que el jugador tenga que pulsar la tecla de acción. Es lo que usamos en **Sgt. Helmet** para poner las bombas en la pantalla final o mostrar el mensaje `VENDO MOTO SEMINUEVA`. Si crees que vas a necesitar esto, activa esta directiva. Si no, déjala sin activar y ahorra unos bytes. No te preocupes que ya explicaremos esto más despacio.

El compilador de scripts `msc3_mk1`

Lo siguiente que hay que hacer es modificar levemente una línea de `compile.bat`, ya que el compilador de scripts, `msc3_mk1`, **necesita saber el número de pantallas que hay en tu mapa**, esto es, **el valor de `MAP_W * MAP_H`**. En el caso de **Dogmole**, que tenemos un mapa de 8x3 pantallas, este valor será **24**.

`msc3_mk1`, el compilador de scripts, recibe los siguientes parámetros, que nos chiva el propio programa al ejecutarlo desde la ventana de línea de comandos:

```
$ ..\utils\msc3_mk1.exe
MojonTwins Scripts Compiler v3.97 20191202
MT Engine MK1 v5.00+

usage: msc3_mk1 input n_rooms [rampage]

input    - script file
n_pants  - # of screens
rampage  - If included, generate code to read script from ram N
```

Vemos que necesita saber el archivo de entrada que contiene el script (incluyendo la ruta si es necesaria), el número total de habitaciones, y que luego admite un parámetro opcional "rampage" que sirve para generar scripts que puedan almacenarse en la RAM extra de los Spectrum de 128K y que por ahora obviaremos.

Examinemos pues la línea que nos interesa en `compile.bat`. La encontrarás justo al principio. La ajustamos con el número correcto de pantallas para **Dogmole**, que es 24:

```
..\utils\msc3_mk1.exe %game%.spt 24
```

Que el archivo de entrada esté referenciado con `%game%.spt` significa que tienes que llamarlo justo así: con el identificador que has puesto al principio de este archivo en la línea `set game=dogmole`, o sea, `dogmole.spt`. El archivo deberá estar en `/script`.

Puedes crear un archivo `dogmole.spt` vacío o **renombrar** el archivo base que hay, `script.spt` . Hagamos esto último. Le cambiamos el nombre a `script.spt` por `dogmole.spt` . Si lo abres verás este sencillo script mínimo.

```
# Script mínimo que no hace nada
# MTE MK1 v5.0

# flags:
# 1 -

ENTERING GAME
  IF TRUE
    THEN
      SET FLAG 1 = 0
    END
  END
END
```

Es aquí donde vamos a escribir nuestro script. Fíjate el aspecto que tiene: eso que hay ahí es la sección `ENTERING GAME` , que se ejecuta nada más empezar la partida. Dentro de esta sección hay una única cláusula. Esta cláusula sólo tiene una comprobación: `IF TRUE` , que siempre será cierto. Luego hay un `THEN` y justo ahí, y hasta el `END` , empieza la **lista de comandos**. En este caso hay un único comando: `SET FLAG 1 = 0` , que sencillamente pone el flag 1 a 0.

Este script no sirve absolutamente para nada. Además de no hacer nada, resulta que el sistema pone todos los flags a 0 al principio, así que no hace falta inicializarlos a cero. ¿Por qué está ahí? No se, joder, para que hubiera algo. De hecho, lo primero que vas a hacer es BORRARLO.

Es interesante modificar esa cabecera. Las líneas que empiezan por `#` (no tiene por qué ser `#`, puedes usar `;`, por ejemplo, o `'`, o `//`, o lo que quieras) son comentarios. Acostúmbrate de poner los comentarios en su propia línea. No pongas comentarios al final de una comprobación o un comando, que el compilador es vago y puede interpretar lo que no es. Y, sobre todo, acostúmbrate a poner comentarios. Así podrás entender qué leches hiciste hace tres días, antes de la borrachera y ese affair con el moreno de recepción.

Como por ahora las variables (flags) se identifican con un número (a partir de cierta versión también se pudo empezar definir alias, que asignan un identificador a un flag, pero eso lo dejaremos para más adelante) es buena idea hacerse una lista de qué hace cada una para luego no liarnos. Yo siempre lo hago, mira:

```
# Cadàveriön
# Copyleft 2013 Mojon Twins
# Churrera 3.99.2
# flags:
# 1 - Tile pisado por el bloque que se empuja
# 2, 3 - coordenadas X e Y
# 4, 5 - coordenadas X e Y del tile «retry»
# 6, 7 - coordenadas X e Y del tile puerta
```



```
# 8 - número de pantallas finalizadas
# 9 - número de estatuas que hay que colocar
# 10 - número de estatuas colocadas
# 11 - Ya hemos quitado la cancela
# 12 - Pantalla a la que volvemos al agotarse el tiempo
# 13, 14 - Coordenadas a las que volvemos... bla
# 15 - Piso
# 0 - valor de 8 almacenado
# 16 - Vendo moto seminueva.
```

¿Ves? Eso viene genial para saber dónde tienes que tocar.

Tus primeras cláusulas chispas

Ya que sabemos dónde tocar, vamos a empezar con nuestro script. Veamos la sintaxis básica. Esto de aquí es la pinta que tiene un script:

```
SECCION
  COMPROBACIONES
  THEN
    COMANDOS
  END
...
END
...
```

Como vemos, cada sección empieza con el nombre de la sección y termina con `END`. Entre el nombre de la sección y el `END` están las cláusulas que la componen, que puede ser una, o pueden ser varias. Cada cláusula empieza con una lista de comprobaciones, cada una en una línea, la palabra `THEN`, seguida de una lista de comandos, cada uno en una línea, y la palabra `END`. Luego pueden venir más cláusulas.

Recuerda el funcionamiento: ejecutar una sección es ejecutar cada una de sus cláusulas, en orden, de arriba a abajo. Ejecutar una cláusula es realizar todas las comprobaciones de la lista de comprobaciones. Si no falla ninguna, o sea, **todas son ciertas**, se ejecutará todos los comandos de la lista.

Para verlo, vamos a crear un script sencillo que introduzca adornos en algunas pantallas. Vamos a extender el tileset de **Dogmole**, incluyendo nuevos tiles que no colocaremos desde el mapa (porque ya hemos usado los 16 que tenemos como máximo), sino que colocaremos desde el script. Este es nuestro nuevo tileset ampliado con adornos o, como los llamamos en el argot mojono, **decoraciones**:



(La magia de esta versión de **MTE MK1** es que cuando cambias el tileset no tienes más que dejarlo en `gfx/work.png`; los usuarios del motor de toda la vida recordarán que antes había que hacerle encaje de bolillos, incluido un paso por el programa SevenUP. ¿Veis? Para hacer este tipo de cosas [utilizo todo el café](#) que me dais).

Aquí hay un montón de tiles decorativos que vamos a colocar desde el scripting. El sitio para hacerlo son las secciones `ENTERING SCREEN n` de las pantallas que queramos adornar, ya que se ejecutan cuando todo lo demás está en su sitio: el fondo ya estará dibujado, por lo que podremos pintar encima. Vamos a empezar decorando la pantalla número 0, que es donde hay que ir a llevar las cajas. Tendremos que colocar el pedestal, formado por los tiles 22 y 23, y además pondremos más adornos: unas vasijas (29), unas cuantas estanterías (20 y 21), unas cuantas cajas (27 y 28), una armadura (32 y 33) y una bombilla colgando de un cable (30 y 31). Empezamos creando la sección `ENTERING SCREEN 0` en nuestro script `dogmole.spt`:

```
# Vestíbulo de la universidad
ENTERING SCREEN 0

END
```

El pintado de los tiles extra se hace desde la lista de comandos de una cláusula. Como queremos que la cláusula se ejecute siempre, emplearemos la condición más sencilla que existe: la que siempre evalúa a cierto y ya vimos más arriba (como verás, utilizo tabulaciones para ayudarme a distinguir mejor la estructura del script. Esto se llama *indentar* y deberías hacerlo tú también. `msc3_mk1` ignora los espacios en blanco así que los usamos simplemente como guía visual humana):

```
# Vestíbulo de la universidad
ENTERING SCREEN 0
    # Decoración y pedestal
    IF TRUE
    THEN

    END
END
```

Esto significa que siempre que entremos en la pantalla 0, se ejecutarán los comandos de la lista de comandos de esa cláusula, ya que su única condición SIEMPRE evalúa a cierto.

El comando para pintar un tile en la pantalla tiene esta forma:

```
SET TILE (x, y) = t
```

Donde `(x, y)` es la coordenada (recuerda, tenemos 15×10 tiles en la pantalla, por lo que `x` podrá ir de 0 a 14 e `y` de 0 a 9) y `t` es el número del tile que queremos pintar. Es aquí donde Ponedor vuelve a ser muy útil: recuerda que si pasas el ratón por el área de edición **te va chivando las coordenadas de**

las casillas. Así pues, con el Ponedor delante para chivar casillas y ver donde tenemos que pintar las cosas, vamos colocando primero el pedestal y luego todas las decoraciones:

```
# Vestíbulo de la universidad
ENTERING SCREEN 0
  # Decoración y pedestal
  IF TRUE
  THEN
    # Pedestal
    SET TILE (3, 7) = 22
    SET TILE (4, 7) = 23

    # Decoración
    SET TILE (1, 5) = 29
    SET TILE (1, 6) = 20
    SET TILE (1, 7) = 21
    SET TILE (6, 6) = 20
    SET TILE (6, 7) = 21
    SET TILE (7, 7) = 28
    SET TILE (1, 2) = 27
    SET TILE (1, 3) = 28
    SET TILE (2, 2) = 29
    SET TILE (2, 3) = 27
    SET TILE (3, 2) = 32
    SET TILE (3, 3) = 33
    SET TILE (9, 1) = 30
    SET TILE (9, 2) = 30
    SET TILE (9, 3) = 31
  END
END
```

Vale, has escrito tu primera cláusula. No ha sido tan complicado ¿verdad?. Supongo que para que la satisfacción sea completa querrás verlo. Bien: vamos a añadir un poco de código que luego eliminaremos de la versión definitiva y que usaremos para irnos a la pantalla que queramos al empezar el juego y probar así que lo estamos haciendo todo bien.

Si recuerdas, una de las posibles secciones que podemos añadir al script es la que se ejecuta justo al principio del juego: `ENTERING GAME`, que es la que venía vacía al principio y hemos borrado porque no nos servía para nada. Bien, pues vamos a hacer una `ENTERING GAME` que nos servirá para irnos directamente a la pantalla 0 al principio y comprobar que hemos colocado guay todos los tiles en los comandos de la cláusula de la sección `ENTERING SCREEN 0`. Añadimos, por tanto, este código (puedes añadirlo donde quieras, pero yo suelo dejarlo al principio. Da igual donde lo pongas, pero siempre mola seguir un orden).

```
ENTERING GAME
  IF TRUE
  THEN
    WARP_TO 0, 12, 2
```

END
END

¿Qué hace esto? Pues hará que, al empezar el juego, se ejecute esa lista de cláusulas formada por una única cláusula, que siempre se ejecutará (porque tiene IF TRUE) y que lo que hace es trasladarnos a la coordenada (12, 2) de la pantalla 0, porque eso es lo que hace el comando WARP:

WARP_TO n, x, y Nos traslada a la pantalla n, y nos hace aparecer en las coordenadas (x, y).

Cuando el jugador empiece la partida se ejecutará la sección ENTERING GAME . Esta sección lo único que hace es trasladar al jugador a la posición (12, 2) y cambiar a la pantalla 0. Entonces, al entrar en la pantalla 0, se ejecutará la sección ENTERING SCREEN 0 , que nos pintará los tiles extra. ¡Vamos a probarlo! Compila el güego y ejecútalo. Si todo va bien, deberíamos aparecer en nuestra pantalla 0 decorada:



¡Hala! Jodó, qué bien. Más, más. Vamos a pintar más tiles para decorar otras pantallas. Exactamente de la misma forma que hemos decorado la pantalla 0, vamos a decorar también la pantalla 1, colocando el cartel de la Universidad de Miskatonic (tiles 24, 25, y 26) y unas armaduras (tiles 32 y 33):

```
# Pasillo de la universidad
ENTERING SCREEN 1
  # Cartel de miskatonic, etc.
  IF TRUE
  THEN
    SET TILE (7, 2) = 24
    SET TILE (8, 2) = 25
```

```

SET TILE (9, 2) = 26
SET TILE (1, 6) = 32
SET TILE (1, 7) = 33
SET TILE (13, 6) = 32
SET TILE (13, 7) = 33
END
END

```

¡Vamos a verlo! Cambia `ENTERING_GAME` para saltar a la pantalla 1 en vez de la pantalla 0:

```

ENTERING_GAME
  IF TRUE
    THEN
      WARP_TO 1, 12, 2
    END
  END
END

```

Compila, ejecuta... et voie-la!



De la misma manera añadimos código para poner más decoraciones en el mapa. La verdad es que nos aburriríamos pronto y sólo hay mandanga en la pantalla 6 (una lámpara) y la pantalla 18 (un ancla en la playa). ¿Por qué no aprovechas tú y pones más? Estas son las que vienen en el juego original:

```

ENTERING_SCREEN 6
  IF TRUE
    THEN
      SET TILE (10, 1) = 30
      SET TILE (10, 2) = 31
    END
  END

```

```

        SET TILE (10, 4) = 35
    END
END
ENTERING SCREEN 18
IF TRUE
    THEN
        SET TILE (4, 8) = 34
    END
END
END

```

Decoraciones más mejor (y menos peor)

Cuando tienes que pintar más de tres tiles tenemos un atajo que hace que el script sea más rápido de escribir y que además ocupe menos bytes. Se trata de emplear el comando `DECORATIONS`, al que deberá seguir una lista de posiciones de tile y números de tile, uno por línea, terminados con un `END`. De este modo, el script que habíamos introducido para las pantallas 0 y 1 se convierte en:

```

# Vestíbulo de la universidad
ENTERING SCREEN 0
    # Decoración y pedestal
    IF TRUE
    THEN
        DECORATIONS
            # Pedestal
            3, 7, 22
            4, 7, 23

            # Decoración
            1, 5, 29
            1, 6, 20
            1, 7, 21
            6, 6, 20
            6, 7, 21
            7, 7, 28
            1, 2, 27
            1, 3, 28
            2, 2, 29
            2, 3, 27
            3, 2, 32
            3, 3, 33
            9, 1, 30
            9, 2, 30
            9, 3, 31
        END
    END
END

# Pasillo de la universidad
ENTERING SCREEN 1
    # Cartel de miskatonic, etc.
    IF TRUE
    THEN

```

```
DECORATIONS
    7, 2, 24
    8, 2, 25
    9, 2, 26
    1, 6, 32
    1, 7, 33
    13, 6, 32
    13, 7, 33
END
END
END
```

Semánticamente esto es equivalente 100% a lo que habíamos escrito, pero hace que el script ocupe menos y se ejecute más rápido, ya que al entrar en "modo ristra de impresiones de tiles" el intérprete tiene que hacer menos trabajo. Y es una de las cosas nuevas de msc3 que hemos inyectado directamente de MK2 para la v5 de MK1.

Creo que lo voy pillando

Pues es el momento para dejarlo. Intenta absorber bien estos conocimientos, empápatelos bien. Si no has pillado algo de aquí, no tengas prisa y espérate a que sigamos, que seguramente lo terminarás entendiendo. Y, como siempre, lo que quieras preguntar ¡pregúntalo!

En el siguiente capítulo meteremos la chicha del gameplay: detectaremos que se han muerto todos los hechiceros para quitar el piedro de la entrada de la universidad, y programaremos la lógica para ir dejando las cajas en el vestíbulo.

Hasta entonces, cacharrea con esto. En el archivo con el material de este capítulo tienes el script de **Dogmole** a medio hacer con las cosas que hemos visto en este capítulo y el `work.png` con el tileset completo.

Capítulo 9: Scripting básico

Aquí te he dejado el paquete de materiales de este capítulo, que en realidad consiste en el script de **Dogmole** terminado.

[Material del capítulo 9](#)

¿Scripting básico?

Eso mismo pone. En este capítulo vamos a definir el gameplay de **Dogmole Tuppowski** y vamos a aprender algunas nociones básicas de scripting.

El sistema de scripting de **MTE MK1** es muy sencillo y parece bastante limitado, pero te puedes empeñar bastante y conseguir cosas medianamente complejas con algo de maña. De hecho, el scripting a partir de v5.0 es bastante más potente ya que se ha integrado la versión más reciente de MSC3, el sistema empleado en MK2, sustituyendo al MSC original, por lo que si practicas puedes lograr en tu juegos diseños de gameplay bastante complejos. Las reglas de **Dogmole** son sencillas a propósito, y nos servirán para ilustrar un comportamiento sencillo y aprender. Luego veremos el script de diferentes juegos mojonos para que veáis cómo hemos conseguido hacer las cosas.

El sistema de scripting tiene muchos comandos y comprobaciones diferentes. Como no quiero convertir este curso en una referencia con una lista de cosas interminables. Para ello puedes consultar la [referencia completa de MSC3](#). No está de más echarle un ojaldre cuando tengas las nociones básicas.

Refresquemos un poco

Recordemos que el script está formado por secciones, y que cada sección incluye cláusulas. Cada cláusula no es más que una lista de comprobaciones y una lista de comandos: si todas las comprobaciones son ciertas, se ejecutarán los comandos.

Controlamos qué cláusulas se ejecutarán colocándolas en una u otra sección. Echa un vistazo de nuevo al capítulo 8 para recordar qué secciones había disponibles. Hasta ahora sólo hemos usado `ENTERING_GAME` y `ENTERING_SCREEN n`, pero ahora veremos algunas más.

¡Vamos a ello!

Antes de empezar vamos a recapitular, porque es importante que sepamos qué estamos haciendo. Recordemos, pues, cuál era el diseño de gameplay de nuestro **Dogmole Tuppowsky**:

En primer lugar, **la puerta de la universidad está cerrada**, y para abrirla **hay que matar a todos los monjes**. Hay **20 monjes** puestos por todo el mapa, en la parte de abajo (las dos filas inferiores) y hay

que cargárselos a todos. Cuando estén todos muertos, habrá que **quitar el piedro** que colocamos en el mapa a la entrada de la universidad.

Luego hay que programar la lógica del pedestal, dentro de la universidad. Si **tocamos el pedestal llevando un objeto**, lo **perdemos** y se **incrementará un flag** contando el número de objetos que hemos depositado. Cuando este número **llegue a 10**, habremos **ganado** el juego.

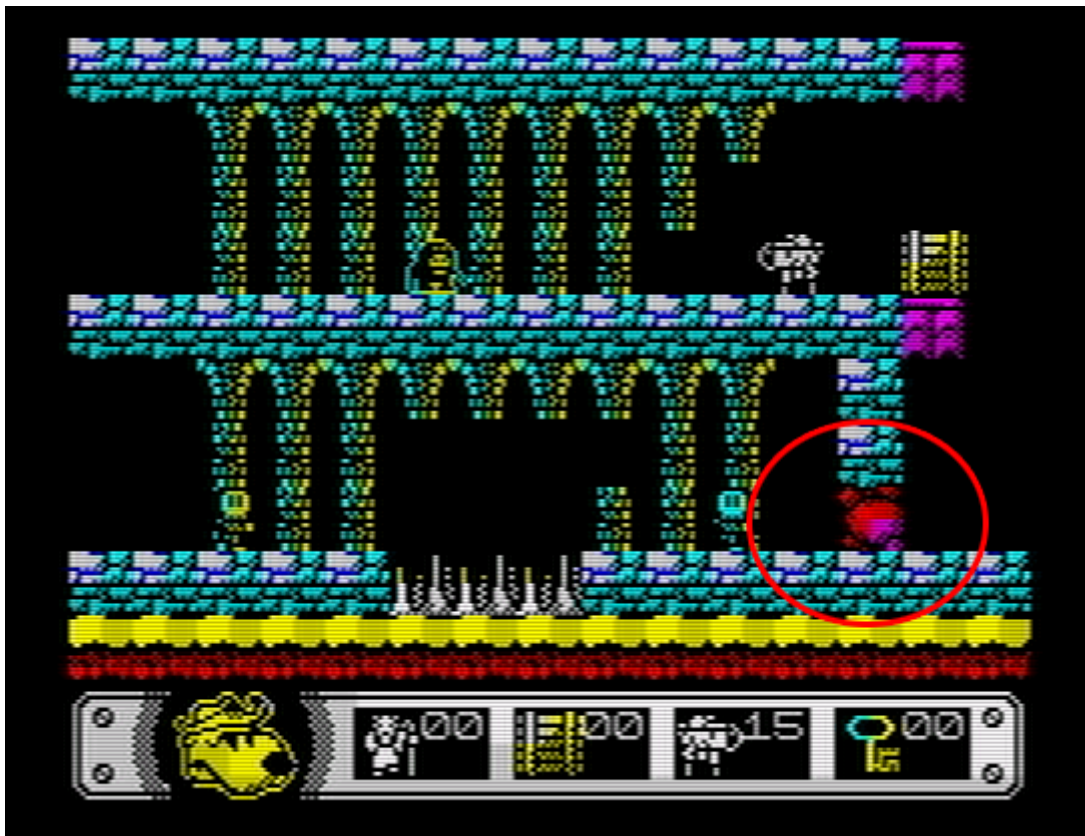
El script de este juego va a ser muy sencillo. Lo primero que tenemos que mirar es qué vamos a necesitar almacenar para destinar algunas flags para ello. En nuestro caso, como el motor ya se encarga de contar los monjes que hemos matado, sólo necesitaremos ir contando las cajas que vamos depositando y además necesitaremos recordar si hemos quitado el piedro o no. Vamos a usar dos flags: la 1 y la 3. ¿Por qué estas dos y no otras? Pues porque sí. En realidad, da igual.

Recordemos que mencionamos en el anterior capítulo que era interesante apuntar qué hacía cada flag al principio de nuestro script:

```
# flags:
# 1 - cuenta general de objetos.
# 3 - 1 = puerta de la universidad abierta.
```

Contando monjes muertos

No, no es el título del último disco del grupo de Death Metal Yitan Vayin. Lo primero que vamos a ver es cómo contar los monjes muertos para quitar el piedro de la pantalla 2. Antes que nada, tenemos que ver qué es lo que vamos a quitar. La pantalla 2 es esta, y he marcado el piedro que tenemos que quitar por scripting:



Si contamos un poquito, nos damos cuenta de que el piedra ocupa las coordenadas (12, 7). Las apuntamos. Hemos dicho que vamos a usar el flag 3 para almacenar si ya hemos matado a todos los monjes o no. Si el flag 3 vale 1, significará que hemos matado a todos los monjes, y en ese caso habría que modificar esa pantalla para borrar el piedra de la posición que tenemos anotada. ¿Por qué no empezar por ahí? Creemos, pues, una cláusula para cuando entremos en la pantalla 2:

```
# Entrada de la universidad
ENTERING SCREEN 2
    # Control de la puerta de la universidad.
    IF FLAG 3 = 1
    THEN
        SET TILE (12, 7) = 0
    END
END
```

Poco hay de nuevo en esta primera cláusula de gameplay que hemos escrito: se trata de la comprobación del valor de un flag. En vez de el IF TRUE que habíamos usado hasta ahora, escribimos IF FLAG 3 = 1 que sólo evaluará a cierto si el valor de nuestro flag 3 es, precisamente, 1. En este caso, se ejecutará el cuerpo de la cláusula: SET TILE (12, 7) = 0 escribirá el tile vacío sobre el espacio que ocupa el piedra, eliminándolo. Por tanto, cuando entremos en esta pantalla con el flag 3 a 1, se borrará el piedra y no habrá obstáculo. ¿Se pilla el concepto?

Sigamos, entonces. Hemos dicho que el flag 3 a 1 significa que hemos matado a todos los enemigos, pero el flag 3 no se va a poner a 1 automáticamente. Necesitamos crear una regla que, efectivamente, lo ponga a 1.

Como en la pantalla donde aparece el piedro no hay monjes, nunca se dará la situación de que matemos al último monje en la pantalla del piedro. Esto es: siempre estaremos en otra pantalla cuando matemos al último monje. Esta es una de esas decisiones de diseño que se toman para poder simplificar mucho el tema y ahorrar por todos los lados. Te vas a tener que hartar de hacer cosas así si vas a dedicarte a esto de hacer güegos para 8 bits.

Un buen sitio para comprobar que hemos matado a todos los monjes es en esa sección del script que se ejecutará cada vez que matemos uno, o sea, en nuestra sección `PLAYER_KILLS_ENEMY`. Ahí comprobaremos que el número de enemigos eliminados vale 20 y, si se da el caso, pondremos el flag 3 a 1:

```
# Abrir la universidad
PLAYER_KILLS_ENEMY
    IF ENEMIES_KILLED_EQUALS 20
    THEN
        SET FLAG 3 = 1
    END
END
```

Con esto conseguimos justo lo que queremos. Fíjate que hay una nueva comprobación: `IF ENEMIES_KILLED_EQUALS 20` será cierta si el número de enemigos eliminados (o monjes) vale exactamente 20. Si eso es cierto, acto seguido comprobamos el valor del flag 3 para ver que vale 0. Con esto lo que hacemos es asegurarnos de que esta cláusula sólo se ejecutará una vez, o de lo contrario se ejecutaría al entrar en cada pantalla.

Si todo se ha cumplido, pondremos el flag 3 a 1 (que es lo que queríamos) además de soltar una serie de pitidos pochos. Sí, el comando `SOUND n` toca el sonido `n`. Se trata de los sonidos del engine. Puedes mirar a qué corresponde cada número en el archivo `beeper.h`, al final.

Con esto tendremos lista la primera parte de nuestro gameplay: si todos los enemigos están muertos, colocamos el flag 3 a 1. En la pantalla 2, si el flag 3 vale 1, quitamos el piedro.

Por cierto: todo esto funcionará sólo si realmente hay 20 monjes en el mapa y están accesibles. Lo primero se puede comprobar muy fácilmente sin tener que andar contando en el Ponedor simplemente compilando y examinando el archivo `enems.h` generado. Buscamos una `N_ENEMS_TYPE_3` y su valor deberá ser 20.

Optimizando

Probablemente te hayas dado cuenta de que podemos ahorrar un flag y código de script si hacemos esto de otra forma equivalente: en vez de comprobar que hemos matado 20 enemigos al matar para poner un flag y luego usar el valor de ese flag al entrar en la pantalla 2, podríamos hacer directamente la comprobación en `ENTERING_SCREEN 2` y fumarnos `PLAYER_KILLS_ENEMY`, así:

```
# Entrada de la universidad
ENTERING_SCREEN 2
    # Control de la puerta de la universidad.
    IF ENEMIES_KILLED_EQUALS 20
    THEN
        SET_TILE (12, 7) = 0
    END
END
```

¡Y tienes razón! Pero hemos elegido hacerlo de la otra forma porque luego vamos a meter más mierdas y necesitamos la detección donde está. Pronto lo veremos. Vamos ahora al otro punto básico del diseño de gameplay:

Lógica de las cajas

Ahora sólo nos queda definir la segunda parte del gameplay. Si recordáis, tenemos configurado el motor con `ONLY_ONE_OBJECT`. Eso significa que el máximo de objetos que podemos recoger es uno, o sea, que sólo podemos llevar una caja, y que no se podrá coger otra hasta que *liberemos* el objeto, y que una de las formas de hacerlo es con el script.

El objetivo del juego es llevar 10 cajas al mostrador de la Universidad, por tanto tendremos que programar en el script la lógica que haga que, si llevamos un objeto y activamos el mostrador, se nos reste ese objeto y se incremente el contador de objetos entregados, que hemos dicho que será el flag 1.

El mostrador está en la pantalla 0, si recordáis: lo hemos pintado con `SET_TILE` desde nuestro script en la sección `ENTERING_SCREEN 0`. El pedestal ocupa las posiciones (3, 7) y (4, 7).

Vamos a escribir ahora un trozo de script que, si pulsamos la tecla de acción en la pantalla 0, comprueba que estamos tocando el pedestal y que llevamos un objeto, para eliminar ese objeto e incrementar en uno la cuenta.

Lo primero que tenemos que resolver es la detección de que estamos tocando el pedestal. Si el pedestal ocupase un sólo tile en (x, y), sería muy sencillo:

```
IF PLAYER_TOUCHES x, y
```

Si cualquier pixel del jugador toca el tile (x, y), esa condición evalúa a cierto. El problema es que nuestro pedestal ocupa dos tiles. Una solución sería escribir dos cláusulas idénticas, una con un `PLAYER_TOUCHES 3, 7` y la otra con un `PLAYER_TOUCHES 4, 7`, pero eso no será necesario ya que tenemos otras herramientas.

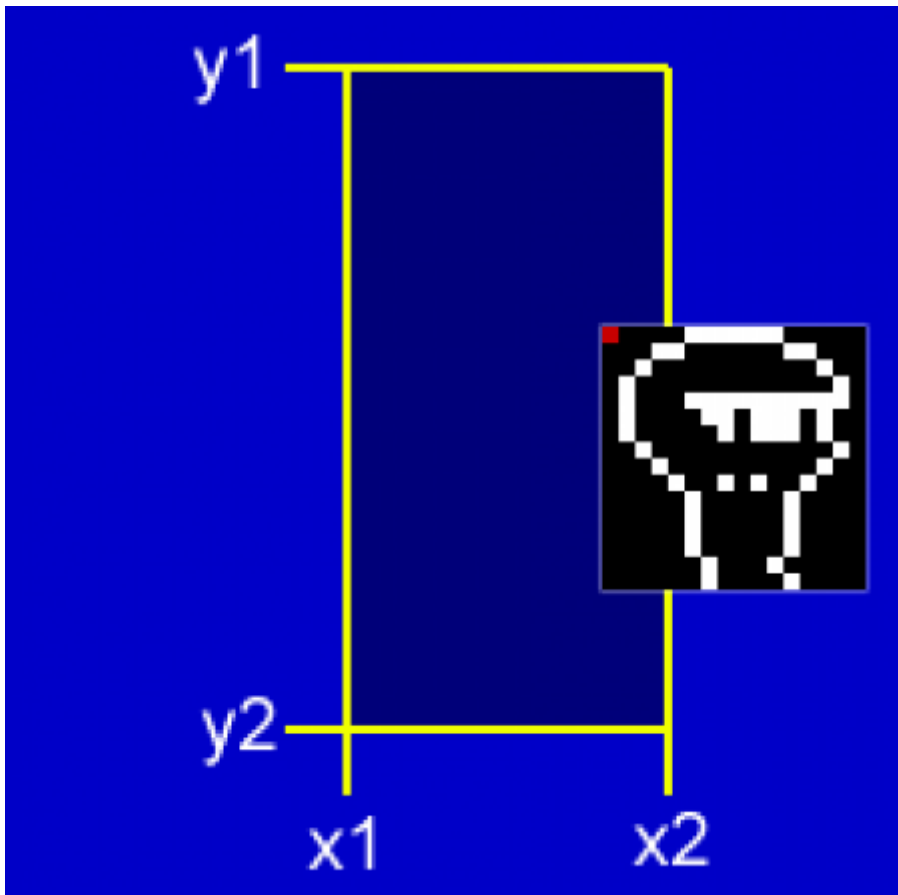
Hay dos formas de hacerlo. Primer veremos la *antigua*, la que se usaba en el viejo MSC de **MTE MK1**, porque nos servirá para entender algunos conceptos importantes. Luego veremos una mucho más cómoda.

Para comprobar que estamos dentro de un area tenemos dos comprobaciones especiales:

```
IF PLAYER_IN_X x1, x2  
IF PLAYER_IN_Y y1, y2
```

La primera evaluará a cierto si la coordenada x, en píxels, de la esquina superior izquierda del cuadro del sprite de nuestro personaje está entre x1 y x2. La segunda lo hará si la coordenada y, en píxels, de la esquina superior izquierda del cuadro del sprite de nuestro personaje está entre y1 e y2.

Veámoslo con un dibujo. Aquí vemos un área delimitada por x1, x2 y por y1, y2. El jugador estará "dentro" de ese área si el píxel marcado en rojo (el de la esquina superior izquierda del sprite) está "dentro" de ese área.



Cuando queremos comprobar que nuestro personaje esté dentro del área rectangular que ocupa un conjunto de tiles, tendremos que seguir la siguiente fórmula para calcular los valores de x1, x2, y1 e y2. Si (tx1,ty1) son las coordenadas (en tiles) del tile superior izquierdo del rectángulo y (tx2, ty2) son las coordenadas (también en tiles) del tile inferior derecho, esto es:



Con el área definida aquí, los valores de x1, x2 e y1, y2 que tendremos que usar en el script son los que se obtienen con las siguientes fórmulas:

```

x1 = tx1 * 16 - 15
y1 = ty1 * 16 - 15 +---+
      |   |
      |   |
+---+ x2 = tx2 * 16 + 15
      y2 = ty2 * 16 + 15
  
```

Para verlo, de nuevo, un dibujito. Fíjate que he superpuesto un sprite para que veáis que para que “toque” los tiles debe estar en el rectángulo definido por las coordenadas (x1, y1) y (x2, y2):



Sí, si no estáis acostumbrados a hacer números programando esto es un lío de cojones, pero en realidad no lo es tanto cuando memorizáis la fórmula, o, mejor, si la comprendéis. Se multiplica por 16 para pasar de coordenadas de tiles a coordenadas de pixels porque los tiles miden 16×16 pixels. La suma y resta de 15 es para hacer "colisión por caja" con el sprite.

Para terminar de verlo, trasladémonos a nuestro caso y hagamos las operaciones necesarias utilizando los valores de nuestro juego. Aquí, el rectángulo está formado únicamente por dos tiles en las coordenadas (3, 7) y (4, 7). Los tiles de las esquinas son esos dos tiles, precisamente, por lo que tx1 valdra 3, ty1 valdrá 7, tx2 valdrá 4 y ty2 valdrá también 7. De ese modo, siguiendo las fórmulas:

$$\begin{aligned}x1 &= 3 * 16 - 15 = 33 \\y1 &= 7 * 16 - 15 = 97 \\x2 &= 4 * 16 + 15 = 79 \\y2 &= 7 * 16 + 15 = 127\end{aligned}$$

O sea, que para tocar el mostrador, el sprite debe estar entre 33 y 79 en la coordenada X y entre 97 y 127 en la coordenada Y. Veámoslo gráficamente con un gráfico lioso: fíjate como para que el sprite esté tocando el mostrador, el píxel superior izquierdo del cuadrado de su sprite (marcado en rojo) debe estar dentro del área que hemos definido:



Además, tendremos que comprobar que llevemos una caja en el inventario. Sería algo así:

```
PRESS_FIRE AT SCREEN 0
# Detectar pedestal.
# Lo detectamos definiendo un rectángulo de píxels.
# Luego comprobamos si el jugador ha cogido un objeto.
# Si todo se cumple, decrementamos el número de objetos e incrementamos FLAG 1
IF PLAYER_IN_X 33, 79
```

```

IF PLAYER_IN_Y 97, 127
IF PLAYER_HAS_OBJECTS
THEN
    INC FLAG 1, 1
    DEC OBJECTS 1
    SOUND 7
END
END

```

Ahí está todo lo que hemos visto: primeramente, comprobamos la posición de Dogmole con `IF PLAYER_IN_X` e `IF_PLAYER_IN_Y`. Si todo se cumple, comprobamos que tengamos un objeto recogido con `IF PLAYER_HAS_OBJECTS`. Si se cumple todo haremos tres cosas: primero, incrementaremos en 1 el flag 1 mediante `INC FLAG 1, 1`. Luego decrementaremos en 1 en número de objetos recogidos (con lo que volverá a ser 0, y podremos volver a recoger otra caja) con `DEC OBJECTS 1`. Finalmente, tocaremos el sonido número 7.

¡Cálculos manuales! ¡Como en los 80!

Sí. Pero los tiempos cambian y pronto nos dimos cuenta que con un sencillo apaño a nivel del compilador de scripts podíamos ofrecer una interfaz más amable para el programador que hiciera esos cálculos automáticamente. Para cuando haya que comprobar rangos de coordenadas asociadas a tile tenemos estas dos comprobaciones:

```

IF PLAYER_IN_X_TILES tx1, tx2
IF PLAYER_IN_Y_TILES ty1, ty2

```

Que comprobará que el jugador esté entre las coordenadas **de tile** indicadas, ambos límites inclusive. Se pueden comprobar filas o columnas de un solo tile repitiendo el valor (`PLAYER_IN_X_TILES 3, 3` se cumplirá cuando el player esté tocando la columna del tile 3).

De este modo nuestro script quedaría así, que es mucho más legible y, desde luego, mucho más fácil de modificar:

```

PRESS_FIRE AT SCREEN 0
# Detectar pedestal.
# Lo detectamos definiendo un rectángulo de píxels.
# Luego comprobamos si el jugador ha cogido un objeto.
# Si todo se cumple, decrementamos el número de objetos e incrementamos FLAG 1
IF PLAYER_IN_X_TILES 3, 4
IF PLAYER_IN_Y_TILES 7, 7
IF PLAYER_HAS_OBJECTS
THEN
    INC FLAG 1, 1
    DEC OBJECTS 1
    SOUND 7
END
END

```

Más comprobaciones.

Hecho esto, sólo nos queda una cosa que hacer: comprobar que hemos llevado las 10 cajas. Un buen sitio para hacerlo es justo después de la anterior cláusula. Como todas las cláusulas de una sección se ejecutan en orden, justo después de contabilizar colocaremos la comprobación de que ya hemos puesto 10 para terminar el juego. Ampliamos, por tanto, la sección `PRESS_FIRE AT SCREEN 0` con la nueva cláusula. Quedaría así:

```
PRESS_FIRE AT SCREEN 0
    # Detectar pedestal.
    # Lo detectamos definiendo un rectángulo de píxels.
    # Luego comprobamos si el jugador ha cogido un objeto.
    # Si todo se cumple, decrementamos el número de objetos e incrementamos FLAG 1
    IF PLAYER_IN_X_TILES 3, 4
    IF PLAYER_IN_Y_TILES 7, 7
    IF PLAYER_HAS_OBJECTS
    THEN
        INC FLAG 1, 1
        DEC OBJECTS 1
        SOUND 7
    END

    # Fin del juego
    # Si llevamos 10 cajas, ¡hemos ganado!
    IF PLAYER_IN_X 48, 79
    IF PLAYER_IN_Y 112, 127
    IF FLAG 1 = 10
    THEN
        WIN GAME
    END IF
END
```

De nuevo, muy sencillo: si llevamos 10 cajas (o sea, si el flag1 vale 10), habremos ganado. El comando `WIN GAME` hace que el juego termine con éxito y se muestre la pantalla del final.

¿Ves que no ha sido tanto?

Mejora interesante

Tal y como hemos configurado nuestro güego, el jugador tiene que pulsar acción para activar el mostrador y depositar un objeto. No es un problema, pero molaría más que el jugador no tuviese que pulsar nada. Precisamente para eso introdujimos en el motor lo que hemos llamado “la zona de fuego”, o *fire zone*. Esta zona de fuego no es más que un rectángulo en pantalla, especificado en píxels. Si el jugador entra en el rectángulo, el motor se comporta como si hubiese pulsado acción. La zona de fuego se desactiva automáticamente al cambiar de pantalla, por lo que si la definimos en un `ENTERING SCREEN n`, sólo estará activo mientras estemos en esa pantalla.

Esto viene divinamente para nuestros propósitos: si al entrar en la pantalla 0 definimos una zona de fuego alrededor del mostrador, en cuanto el jugador lo toque se ejecutará la lógica que hemos programado en el script para dejar el objeto que lleve e incrementar el contador.

La zona de fuego se define con el comando `SET_FIRE_ZONE`, que recibe las coordenadas `x1`, `y1`, `x2`, e `y2` del rectángulo que queramos usar como zona de fuego. Si queremos hacer coincidir la zona de fuego con un rectángulo formado por tiles, como es nuestro caso, se aplican las mismas fórmulas que explicamos antes. O sea, que vamos a usar exactamente los mismos valores.

Lo primer que tenemos que hacer es decirle al motor que vamos a usar zonas de fuego. Para ello, tenemos que activar la directiva correspondiente en nuestro `my/config.h`:

```
#define ENABLE_FIRE_ZONE // Allows to define a zone which auto-triggers
```

Hecho esto, sólo tendremos que modificar la sección `ENTERING SCREEN 0` añadiendo el comando `SET_FIRE_ZONE x1, y1, x2, y2` al final del todo:

```
# Vestíbulo de la universidad
ENTERING SCREEN 0
  # Decoración y pedestal
  IF TRUE
  THEN
    DECORATIONS
      # Pedestal
      3, 7, 22
      4, 7, 23

      # Decoración
      1, 5, 29
      1, 6, 20
      1, 7, 21
      6, 6, 20
      6, 7, 21
      7, 7, 28
      1, 2, 27
      1, 3, 28
      2, 2, 29
      2, 3, 27
      3, 2, 32
      3, 3, 33
      9, 1, 30
      9, 2, 30
      9, 3, 31
    END

    SET_FIRE_ZONE 33, 97, 79, 127
  END
END
```

Como a lo mejor has adivinado, en **MSC3** `SET_FIRE_ZONE` también tiene una versión que emplea coordenadas de casillas de tiles, que nos viene super genial cuando el *fire zone* corresponde con un rectángulo de tiles, como es el caso: `SET_FIRE_ZONE_TILES` . El script quedaría así:

```
# Vestíbulo de la universidad
ENTERING SCREEN 0
  # Decoración y pedestal
  IF TRUE
  THEN
    DECORATIONS
      # Pedestal
      3, 7, 22
      4, 7, 23

      # Decoración
      1, 5, 29
      1, 6, 20
      1, 7, 21
      6, 6, 20
      6, 7, 21
      7, 7, 28
      1, 2, 27
      1, 3, 28
      2, 2, 29
      2, 3, 27
      3, 2, 32
      3, 3, 33
      9, 1, 30
      9, 2, 30
      9, 3, 31
    END

    SET_FIRE_ZONE_TILES 3, 7, 4, 7
  END
END
```

¡Vamos a usar EXTERN!

En el capítulo 7 vimos que había una directiva `ENABLE_EXTERN_CODE` que dijimos que era para poder ejecutar código C desde nuestro script. En concreto, lo que se hace es llamar a la función `do_extern_action (unsigned char n)` que está en el archivo `my/extern.h` pasándole el numerito que pongamos en nuestro script. Esto viene cojonudo para hacer cosas personalizadas, y lo veremos con un ejemplo muy sencillo.

Cuando hayas matado a todos los monjes se retirará el piedro, pero no será algo que veamos. Esto queda muy poco intuitivo, y como ya no estamos en los 80, vamos a hacer algo para remediarlo: vamos a avisar al jugador imprimiendo un cartelito. Es por esto por lo que decidimos no hacer la optimización para ahorrarnos el flag 3 y eliminar `PLAYER_KILLS_ENEMY` .

Lo primero es añadir el código C que vamos a ejecutar a la función `do_extern_action`. Obviamente para poder hacer esto os vendría bien conocer cosas de las tripas del motor de **MTE MK1**, pero por ahora no te preocupes si te suena a checoslovaco. Ya buscaré tiempo para documentarle bien el totete al motor. Sólo quiero que veamos cómo funciona la integración:

```
unsigned char *my_spacer = "          ";
unsigned char *my_message = " PUERTA ABIERTA ";

void do_extern_action (unsigned char n) {
    // Add custom code here.

    // Discard n, we don't need it. There's only one action to perform

    // Print message
    _t = 79;
    _x = 8; _y = 10; _gp_gen = my_spacer; print_str ();
    _x = 8; _y = 12;                print_str ();
    _x = 8; _y = 11; _gp_gen = my_message; print_str ();

    sp_UpdateNowEx (0);

    // Wait
    espera_activa (150);

    // Force reenter
    o_pant = 99;
}
```

Como vemos, al tener sólo una acción *externa* posible, pasamos completamente del parámetro `n`. Poner `EXTERN 0` o `EXTERN 100` en el script dará lo mismo, por tanto.

Lo siguiente será modificar el script para que ejecute el `EXTERN` una vez que hayamos eliminado a los 20 monjes. Por tanto toqueteamos de nuevo la sección `PLAYER_KILLS_ENEMY`, que se quedaría así:

```
# Abrir la universidad
PLAYER_KILLS_ENEMY
    IF ENEMIES_KILLED_EQUALS 20
    THEN
        SET FLAG 3 = 1
        EXTERN 0
    END
END
```

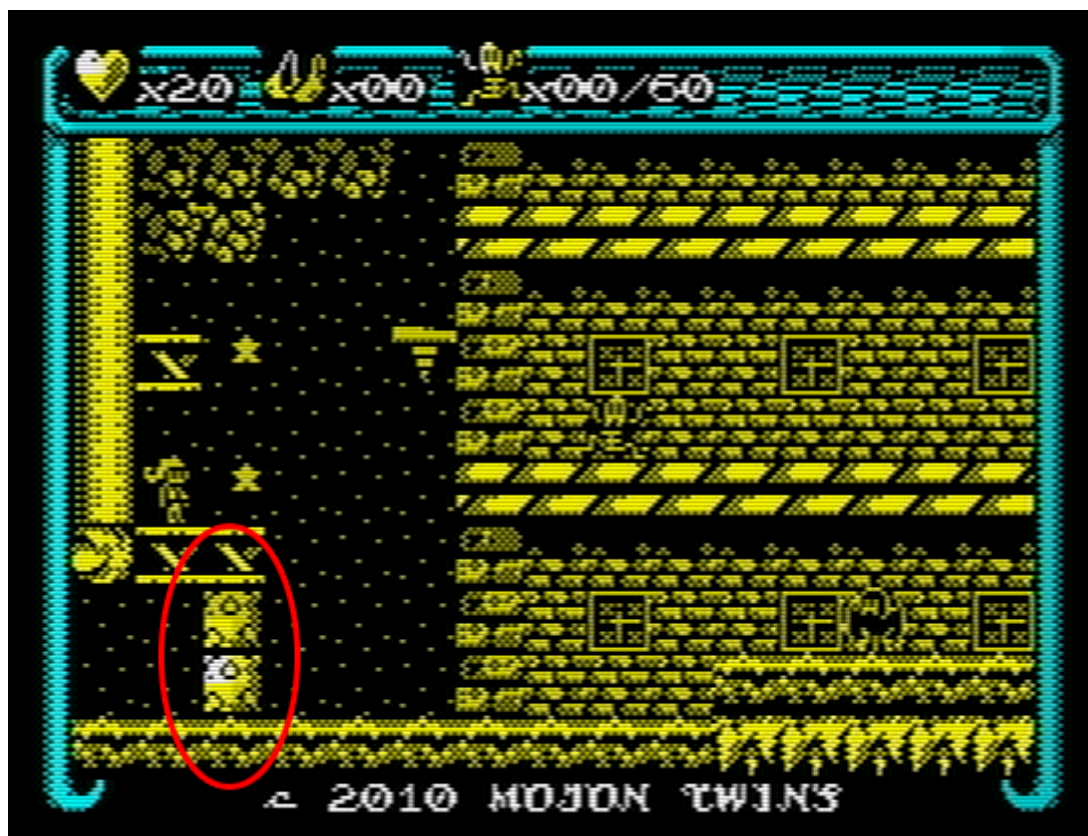
Tío, estoy un poco perdido

Me hago cargo. Hay que hacerse un poco el coco al funcionamiento del script. Creo que es ideal empezar por algo muy sencillo, incluso más sencillo que el **Dogmole** que hemos visto, e ir progresando.

Se me ocurre algo genial: íbamos a terminar aquí el capítulo de scripting básico, pero creo que nos vendría muy bien ver juntos los scripts de algunos de nuestros juegos. Voy a elegir unos cuantos juegos con un script sencillo, y lo iremos explicando paso por paso. Sería interesante que, mientras tanto, fueses jugando al juego para ver cómo afectan las diferentes cláusulas.

Cheril Perils

El güego con el que estrenamos el motor de scripting fue **Cheril Perils**. Entonces todo estaba en pañales y era muy sencillo. El script de **Cheril Perils** es el script más sencillo de todos nuestros güegos con Script. Aquí sólo se hace una cosa: que hayamos matado a todos los enemigos, en cuyo caso quitaremos los pinchos de la primera pantalla. Estos pinchos:



En principio se parece mucho a parte de lo que hemos hecho en **Dogmole**: al entrar en la pantalla, comprobamos que hemos matado a todos los enemigos (hay 60 en total). Si se da el caso, imprimimos el tile vacío sobre los pinchos:

```
ENTERING SCREEN 20
  IF ENEMIES_KILLED_EQUALS 60
    THEN
      SET TILE (2, 7) = 0
      SET TILE (2, 8) = 0
    END
  END
```

Pero aquí ocurre algo que no nos pasaba en el Dogmole: hay enemigos en la pantalla donde hay que quitar los pinchos. No nos vale con detectar esto al entrar, ya que si matamos al último bicho en esta

pantalla (puede suceder) necesitaríamos salir y volver a entrar en la pantalla para que el motor se coscase. Necesitamos más código que detecte que hemos matado al último bicho y que se ejecute cuando lo matemos. Ni cortos ni perezosos, añadimos esto otro:

```
PLAYER_KILLS_ENEMY
  IF ENEMIES_KILLED_EQUALS 60
  THEN
    SET TILE (2, 7) = 0
    SET TILE (2, 8) = 0
  END
END
```

Ahora vamos a detectar que “salimos”. Si no hiciésemos nada, saliendo de la pantalla 20 por la izquierda nos encajamos en la pantalla 19 (cosas de como funciona internamente el mapa)... Que además está en la otra punta del mapa. En el juego original, esto estaba solucionado con un hack guarro, pero con la versión actual de **MTE MK1** puede hacerse bien.

Lo primero será definir una zona de fuego que cubra la parte izquierda de la pantalla, de forma que se ejecute la sección `PRESS_FIRE AT SCREEN 20` cuando nos acerquemos a ella. Añadimos, pues, la definición de la zona de fuego en la sección `ENTERING SCREEN 20`. Esta sección, por tanto, queda así (actualizado a MSC3):

```
ENTERING SCREEN 20
  IF ENEMIES_KILLED_EQUALS 60
  THEN
    SET TILE (2, 7) = 0
    SET TILE (2, 8) = 0
  END

  IF TRUE
  THEN
    SET_FIRE_ZONE_TILES 0, 7, 0, 8
  END
END
```

¿Queda claro? Al entrar en la pantalla 20 pasan dos cosas: primero se comprueba si el número de enemigos vale 60, en cuyo caso se eliminan los tiles-pincho que bloquean la salida. Luego, en cualquier caso (`IF TRUE`) se define una zona de fuego que cubre toda una tira pegada a la izquierda de 15 pixels de ancho. En cuanto el jugador entre en esta zona (no podrá hacerlo si no se ha eliminado la barrera: simplemente no puede pasar), se ejecutará la sección `PRESS_FIRE AT SCREEN 20`. Ahora tendremos que añadir código en la sección `PRESS_FIRE AT SCREEN 20` para detectar que el jugador está intentando salir por la izquierda y, en ese caso, terminar el juego con éxito. Quedaría así:

```
PRESS_FIRE AT SCREEN 20
  IF PLAYER_IN_X_TILES 0, 0
  THEN
    WIN
```

END
END

Recapitulemos para que quede bien claro. Veamos lo que pasaría, paso por paso. Imaginemos que llegamos a la pantalla 20 después de haber matado a todos los malos. Esta es la secuencia de acontecimientos:

1. Al entrar en la pantalla 20, después de dibujarla y tal, se ejecuta la sección `ENTERING_SCREEN_20`. En ella, se comprueba que llevamos 60 enemigos matados, cosa que es cierta, y se elimina la barrera. Además, se define una zona de fuego de que ocupa los tiles (0,7) y (0,8), que es básicamente la zona "por la que se saldría del mapa".
2. Se ejecuta el bucle principal del juego. El jugador juega y tal y cual, ve la barrera abierta, y se dirige a la izquierda.
3. Cuando el jugador entra en la zona de fuego tocando los tiles (0,7) o (0,8), se ejecuta la sección `PRESS_FIRE_AT_SCREEN_20`. En ella se comprueba que la coordenada X del jugador, en la columna 0 de la pantalla, cosa que es cierta (ya que hemos entrado en esta sección por haber entrado en la zona de fuego, que está definida justo en ese area), por lo que ejecuta `WIN` y se nos muestra el final del juego.

Sgt. Helmet Training Day

Vamos a ver ahora un script un poco más largo, pero igualmente sencillo. En este juego la misión es recoger las cinco bombas, llevarlas a la pantalla del ordenador (pantalla 0) para depositarlas, y luego volver al principio (pantalla 24).

Hay muchas formas de hacer esto. La que usamos nosotros para montarlos es bastante sencilla:

Podemos contar el número de objetos que llevamos desde el script, por lo que las bombas serán objetos normales y corrientes del motor. Las colocamos con el colocador como hotspot de tipo 1.

Cuando lleguemos a la pantalla del ordenador, haremos una animación chula colocando las bombas alrededor. Usamos el colocador porque mola para saber las coordenadas de cada casilla (si pones el ratón sobre una casilla salen las coordenadas arriba del todo) y apuntamos en un papel donde las vamos a pintar.

Usaremos el flag 1 para comprobar que hemos colocado las bombas. Al principio del juego valdrá 0, y lo pondremos a 1 cuando coloquemos las bombas.

Cuando entremos en la pantalla 24, que es la pantalla principal, comprobaremos el valor del flag 1, y si vale 1, terminará el juego.

Además, iremos imprimiendo textos en la pantalla con lo que vamos haciendo. Recordemos que en `my/config.h` había tres directivas que mencionamos por encima hace algunos capítulos:

```
#define LINE_OF_TEXT      0      // If defined, scripts can show text @ Y = #
#define LINE_OF_TEXT_X    1      // X coordinate.
#define LINE_OF_TEXT_ATTR 71     // Attribute
```

Sirven para configurar donde sale una linea de texto que podremos escribir desde el script con el comando TEXT. Para ello dejamos sitio libre en el marco: fíjate como hay sitio en la fila de arriba, ya que hemos configurado la linea de texto en las coordenadas (x, y) = (1, 0).



Lo primero que hará nuestro script, por tanto, será definir un par de mensajes que aparecerán por defecto al entrar en cada pantalla, dependiendo de valor del flag 1. Esto lo hacemos en la sección ENTERING ANY . Esta sección, recordemos, se ejecuta al entrar en cada pantalla, justo antes de la sección ENTERING SCREEN n correspondiente. Atención a esto: nos permitirá definir un texto general que podamos sobrescribir fácilmente si hace falta para alguna pantalla en concreto, ya que si ponemos texto en ENTERING SCREEN n sobrescribirá el que pusimos en ENTERING ANY al ejecutarse después.

Para imprimir texto en la linea de texto definida, usamos el comando TEXT . El texto que le sigue va sin comillas. Usaremos el carácter de subrayado _ para representar los espacios. Es conveniente, además, rellenar con espacios para que, si hay que sobrescribir un texto largo con uno corto, se borre entero.

La longitud máxima de los textos dependerá de tu marco de juego y de cómo hayas definido su posición. En nuestro caso la hemos colocado en (x, y) = (1, 0) porque tenemos borde a la izquierda y a la derecha, con lo que la longitud máxima será de 30 caracteres.

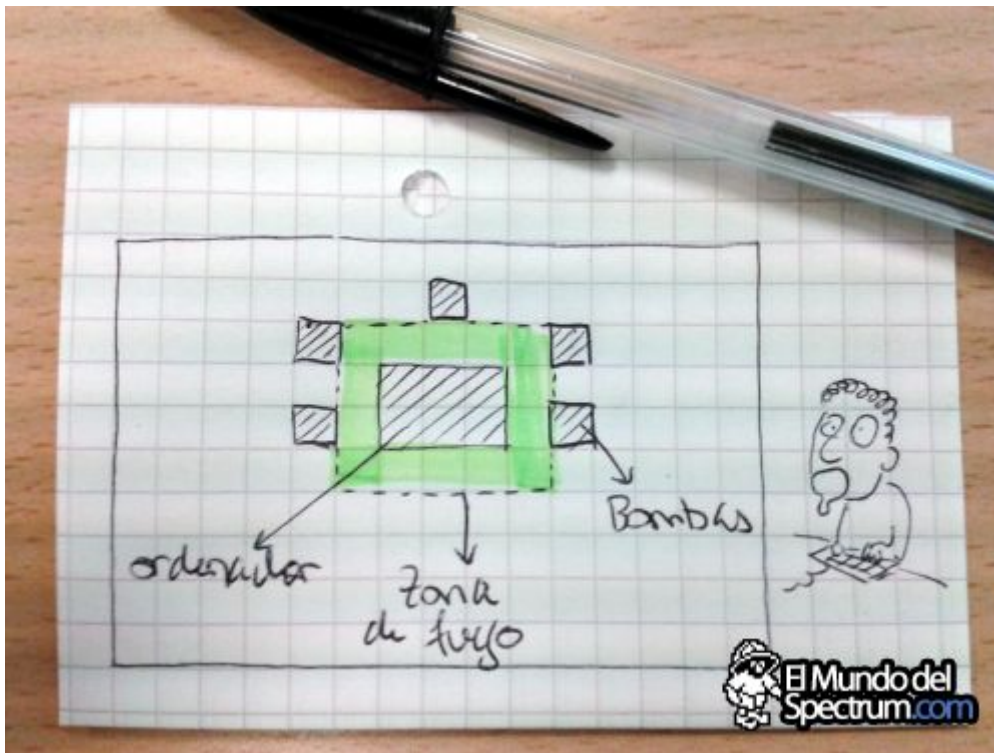
Escribamos nuestra sección ENTERING ANY, pues. Hemos dicho que imprimiremos un texto u otro dependiendo del valor del flag 1:

```
ENTERING ANY
  IF FLAG 1 = 0
  THEN
    TEXT BUSCA_5_BOMBAS_Y_EL_ORDENADOR!
  END

  IF FLAG 1 = 1
  THEN
    TEXT MISION_CUPLIDA!_VUELVE_A_BASE
  END
END
```

No tiene misterio ¿verdad? Si el flag 1 vale 0, o sea, la situación del principio del juego (todos los flags de ponen a 0 al empezar), al entrar en cada pantalla aparecerá el texto BUSCA 5 BOMBAS Y EL ORDENADOR en la zona del marco definida para la línea de texto. Si el flag 1 vale 1, cosa que ocurrirá cuando coloquemos las bombas en el ordenador, el texto por defecto que aparecerá al entrar en las pantallas será MISION CUMPLIDA! VUELVE A BASE .

Vamos a hacernos un croquis mierder de la pantalla para ver donde va el ordenador y las bombas:



Vamos ahora con la chicha. Lo primero que haremos será escribir las condiciones para la pantalla del ordenador, que es la pantalla 0. En esta pantalla tenemos que hacer varias cosas. Tengámoslas claras antes de empezar:

Siempre que entremos tendremos que pintar el ordenador, que está compuesto por los tiles 32 a 38 del tileset. Lo haremos como hemos visto, con SET TILE o, mejor, con DECORATIONS .

Además, tendremos que definir un area de fuego alrededor del ordenador para que el juego detecte automáticamente cuando nos acercamos a él.

Si volvemos a entrar en la pantalla después de haber colocado las bombas (puede pasar), tendremos que coscarlos de ello y pintar también las bombas.

Si entramos por primera vez (no hemos puesto las bombas) escribiremos un mensajito de ayuda que diga PON LAS CINCO BOMBAS Y CORRE

Si nos acercamos al ordenador, habrá que hacer la animación chula de poner las bombas, y además colocar el flag 1 a 1.

Ahora que tenemos algo de experiencia, nos daremos cuenta que las cuatro primeras cosas se hacen al entrar en la pantalla, y la última al pulsar acción (o entrar en la zona de fuego). Vayamos una por una. Empecemos por las cosas que hay que hacer al entrar en esta pantalla. Me gusta empezar por las cosas que hay que hacer siempre: pintar el ordenador y definir la zona de fuego:

```
ENTERING SCREEN 0
```

```
# Siempre: pintar el ordenador.  
IF TRUE  
THEN  
  DECORATIONS  
    6, 3, 32  
    7, 3, 33  
    8, 3, 34  
    6, 4, 36  
    7, 4, 37  
    8, 4, 38  
  END  
  SET_FIRE_ZONE_TILES 5, 2, 9, 5  
END
```

El área que lanzará el script equivale al rectángulo formado desde el tile (x, y) = (5, 2) hasta el (9, 5). O sea, un reborde de un tile alrededor de los seis tiles que ocupa el ordenador. Coge un papel de cuadritos y te lías menos.

Seguimos: si entramos luego de haber colocado las bombas (algo que puede pasar) tendremos que coscarlos y pintar las bombas. Nada más sencillo:

```
# Si ya hemos puesto las bombas: pintarlas.  
IF FLAG 1 = 1  
THEN  
  DECORATIONS  
    4, 4, 17  
    4, 2, 17  
    7, 1, 17  
    10, 2, 17  
    10, 4, 17
```

```
END
END
```

Nos hemos fijado en el croquis, por supuesto, para saber las posiciones de las bombas. La bomba está en el tile 17, que es el tile que se usa para pintar los objetos, si recordáis.

Ahora solo queda poner un texto de ayuda si no hemos colocado aún las bombas. Fíjate que ocurre lo que dijimos: como esta sección se ejecuta después de `ENTERING ANY`, el texto que imprimamos aquí sobrescribirá el que ya hubiese. Es por eso, además, que usamos espacios en blanco alrededor: centrarán el texto y eliminarán los caracteres del texto anterior, que es más largo:

```
# Si no, mensajito.
IF FLAG 1 = 0
THEN
    TEXT _PON_LAS_CINCO_BOMBAS_Y_CORRE_
END
END
```

Listo. Ahora sólo queda reaccionar a la zona de fuego, en la sección `PRESS_FIRE AT SCREEN 0`. Haremos algunas comprobaciones y luego haremos la animación:

```
PRESS_FIRE AT SCREEN 0
    IF PLAYER_IN_X_TILES 5, 9
    IF PLAYER_IN_Y_TILES 2, 5
    IF OBJECT_COUNT = 5
    IF FLAG 1 = 0
    THEN
        SET FLAG 1 = 1
        SET TILE (4, 4) = 17
        SHOW
        SOUND 0
        SET TILE (4, 2) = 17
        SHOW
        SOUND 0
        SET TILE (7, 1) = 17
        SHOW
        SOUND 0
        SET TILE (10, 2) = 17
        SHOW
        SOUND 0
        SET TILE (10, 4) = 17
        SHOW
        SOUND 0
        TEXT ____AHORA_VUELVE_A_LA_BASE____
    END
END
```

Veámoslo poco a poco, porque hay cosas nuevas:

Lo primero es comprobar que estamos donde tenemos que estar (el jugador siempre puede pulsar la tecla acción en vez de entrar en la zona de fuego, y no mola ejecutarlo si el jugador está en cualquier sitio). Eso lo hacemos como ya hemos visto: con `PLAYER_IN_X_TILES` y `PLAYER_IN_Y_TILES` y las mismas coordenadas de la zona de fuego.

Lo siguiente es comprobar que tenemos las cinco bombas, o lo que es lo mismo, que tenemos cinco objetos. Esto se hace con `OBJECT_COUNT`, que representa el número de objetos que el jugador lleva recogidos.

Por último, muy importante, hay que comprobar que aún no hemos dejado las bombas, o cosas divertidas podrían pasar.

Si se cumplen todas estas condiciones, pondremos el flag 1 a 1 (ya hemos puesto las bombas) y hacemos la animación, que consiste en ir pintando una a una las bombas y tocando un sonido. Ves ahí el comando `SHOW`, necesario porque los cambios que hagamos en la pantalla no serán visibles hasta que se actualice, cosa que pasa normalmente al volver al bucle principal, pero no en medio de la ejecución de una cláusula. Como queremos que se vea cada bomba justo después de pintarla, ejecutamos `SHOW`. Cada sonido, además, parará la ejecución durante unos instantes (estamos en modo 48K), lo que nos viene genial. Por último, imprimiremos un texto de ayuda, de nuevo con espacios a los lados para completar los 30 caracteres máximos y borrar lo que hubiese del texto anterior.

Y con esto hemos terminado todo lo que había que hacer en la pantalla 0.

Si seguimos con nuestro guión, lo próximo que había que hacer era volver a la pantalla inicial, que es la 24. Lo que queda por hacer es bastante sencillo: consiste en comprobar, al entrar en la pantalla 23, que el flag 1 vale 1. Esto sólo pasará si anteriormente hemos colocado las bombas, por lo que no necesitamos más... Simplemente comprobamos eso y, si se cumple, terminamos el juego con éxito... Nada más sencillo que hacer esto:

```
ENTERING SCREEN 23
  IF FLAG 1 = 1
    THEN
      WIN
    END
  END
END
```

¡Ea! Ya tenemos el juego programado. En el script de **Sgt. Helmet** hay un detalle más: nuestro habitual “vendo moto seminueva”. Pero eso lo dejo ya, no tiene nada de especial: imprimir tiles, definir zona de fuego, detectar posición, y escribir un texto. Todo eso lo sabes hacer ya.

Arf, arf.

Podría seguir, pero mejor lo dejamos por ahora. En el próximo capítulo seguiremos viendo ejemplos paso por paso, pero ya con scripts más complejos como, por ejemplo, el de **Cadàveriön**. Luego ya

seguiremos viendo cosas interesantes, como juegos de 128K, cambiar la música y los efectos, fases comprimidas... Uf, no vamos a terminar nunca.

Capítulo 10: Música y FX (48K)

En este capítulo veremos como cambiar la música y los efectos de sonido (de 48K) en nuestros juegos. Cambiar la música es trivial. Cambiar los sonidos puede ser muy fácil o muy laborioso. Vamos al rock.

Descárgate los materiales de este capítulo (básicamente, el proyecto Beepola con la música de Dogmole):

[Material del capítulo 10](#)

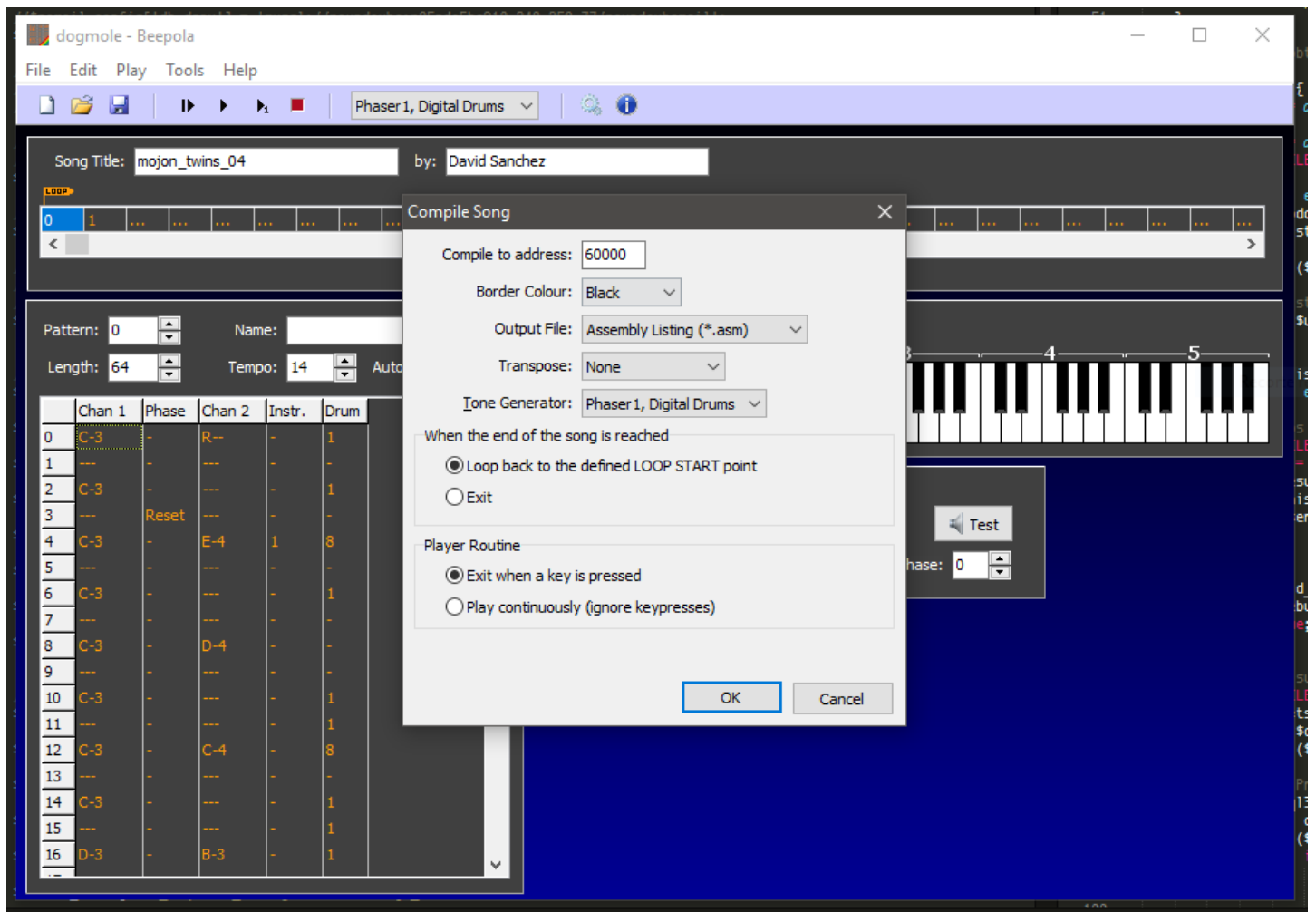
Cambiando la música

Para poder cambiar la música de un juego necesitamos disponer de la misma y de su player en formato de código fuente. Puede hacerse de otras maneras (como en los inicios, cuando usábamos *Wham! The Music Box*) pero es un dolor. En la actualidad existen varias formas de componer música 1 bit para Spectrum en programas que se ejecutan en nuestros PC y exportan el código del player y la música en un bonito archivo .asm, como Beepola.

Beepola ([link](#)) es un tracker de toda la vida que es capaz de manejar varios players. Puedes elegir el que quieras **siempre que no necesite cosas extrañas como estar ubicado en un sitio concreto o necesitar un manejador de interrupciones**. Por ejemplo, nosotros solemos usar **Phaser - Synth Drums** que suena muy bien, permite algunas chiribitas, y tiene una percusión potente y que no ocupa un montón. Nosotros vamos a seguir el procedimiento para este player precisamente, pero al final daremos pautas para adaptar otro tipo de players.

No vamos a explicar aquí cómo se maneja Beepola. Si eres músico te harás rápido con el programa, y si no lo eres, seguro que tu amigo el músico sabe usarlo. Solo nos detendremos en la parte que nos interesa: la **exportación**. Si quieres practicar ya puedes probar a cargar el archivo `dogmole.bbsong` que incluimos con este capítulo y seguir las indicaciones a partir de aquí.

En el menú principal de la aplicación seleccionamos `Tools` → `Compile Song` .



Es importante que elijamos la opción `assembly listing (asm)` en el desplegable `output file`. Si no viene por defecto, habrá que especificar que cuando la canción termine vuelva al principio (marcando `Loop back to the defined LOOP START point`) y que se puede interrumpir pulsando una tecla (marcando `Exit when a key is pressed`). Finalmente pulsamos `OK` y lo grabamos. En nuestro caso hemos salvado `dogmole_music_phaser.asm` en el directorio `/mus` de nuestro proyecto.

Esto habrá generado un archivo con el código en ensamble del player *Phaser - Synth Drums* y una sección de datos con nuestra canción. Sin embargo, no podemos usar ese archivo directamente en nuestro proyecto, ya que `z88dk` es especialito. Por suerte hemos añadido una pequeña utilidad que puede tomar el archivo en ensamble que echa Beepola y transformarlo en un `.h` listo para usar: `asm2z88dk`. Si lo ejecutamos nos chiva cosas:

```
D:\git\MK1\src\mus>..\utils\asm2z88dk.exe
$ asm2z88dk.exe in.asm out.h [mk1]
```

`in.asm` - standard assembly (pasmo)
`out.h` - output filename

This program:

1. Changes labels: to `.labels`
2. Adds `#asm / #endasm`
3. If `mk1`, removes `DI/EI` and stuff

Si hemos guardado `dogmole_music_phaser.asm` en `/mus`, podemos irnos a `/dev` y ejecutar así el conversor:

```
$ ..\utils\asm2z88dk.exe ..\mus\dogmole_music_phaser.asm music.h mk1
```

Y con esto habremos terminado.

Otros players

Podemos usar otros players siempre que cumplan las siguientes restricciones:

1.- No necesitan ubicarse en una posición específica de RAM (de todos modos, el player acabará fuera de la memoria en contienda cuando se compile el juego, por lo que si el requisito es que estén fuera de la memoria en contienda nos valdría).

2.- No necesita una rutina de servicio de interrupción (ISR) específica.

3.- Se ejecuta llamando a un punto de entrada y sale al pulsar una tecla.

Si todo eso se cumple habrá que averiguar cuál es el punto de entrada del player. Para eso hay que mirar el código en ensamble o quizá consultar la documentación del player concreto y ver cuál es, identificar adónde hay que llamar para que se ponga a tocar la música. Por ejemplo, para Phaser, este punto es la etiqueta `musicstart`.

Una vez identificado, tendremos que modificar la función `select_joyfunc` en `engine.h`, que es donde se hace la llamada al player:

```
#asm
    ; Music generated by Beepola
    call musicstart
#endasm
```

Cambiamos el `musicstart` con la etiqueta de entrada al player y listo.

Cambiando los FX

Cambiar los efectos de sonido puede ser más problemático. El set que incluye **MTE MK1** fue creado a mano por Shiru hace eones. Si quieres usar otros sonidos tendrás que buscarte la forma de generarlos.

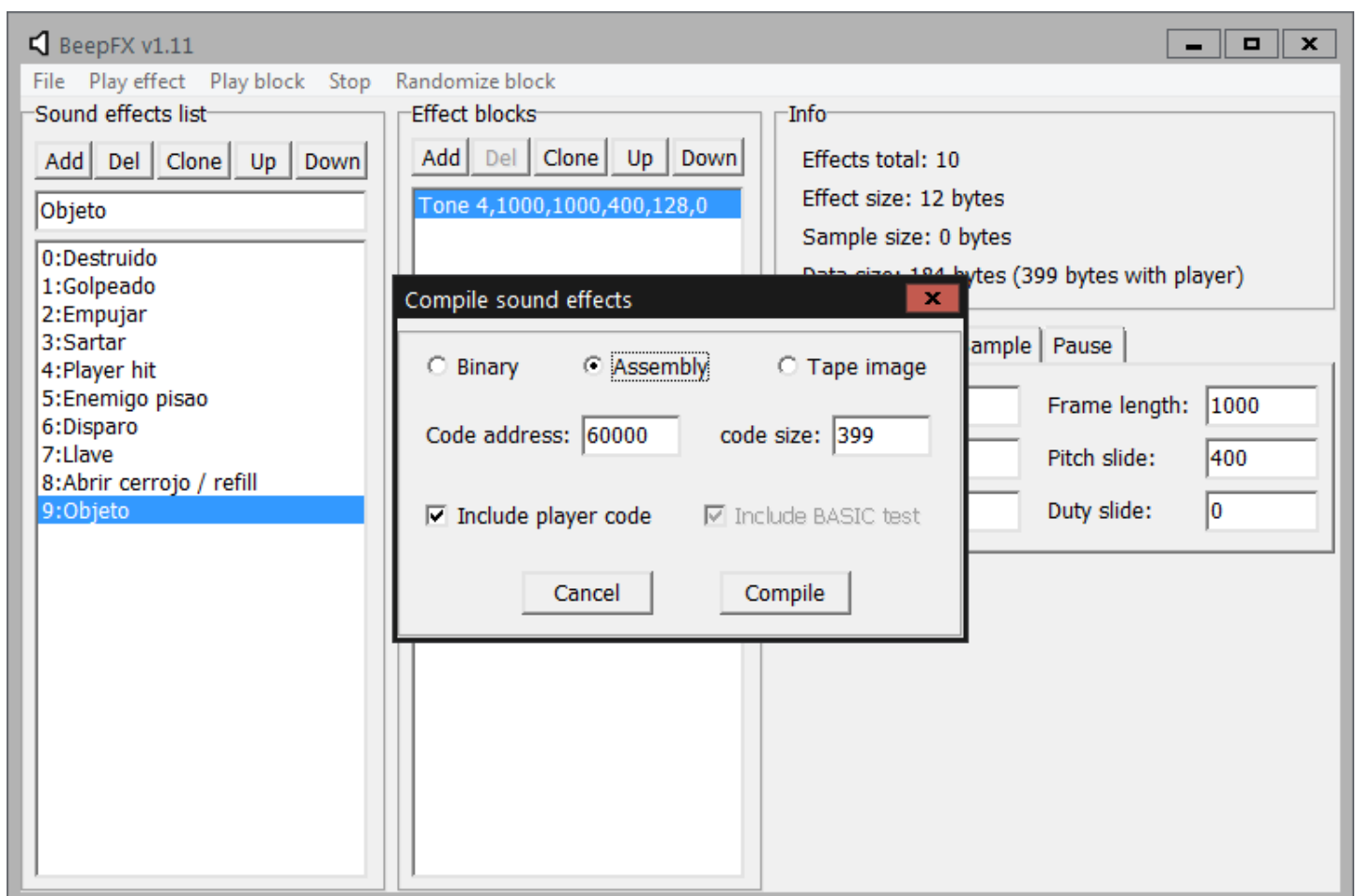
BeepFx ([link](#)), del propio Shiru, puede servirnos. En este tutorial describiremos cómo integrar el player de sonidos de BeepFX con **MTE MK1**. Para integrar otros generadores de sonidos habrá que hacer algo muy parecido.

El objetivo es construir 10 sonidos que sirvan para estas cosas:

#	Cosa
0	Enemigo / tile destruido
1	Enemigo / tile golpeado
2	Empujar bloque
3	Salto
4	Jugador golpeado
5	Enemigo pisado
6	Disparo
7	Coger llave / Recarga de tiempo
8	Abrir cerrojo / Coger refill
9	Coger objeto / Munición

Tienen que aparecer en ese orden. Podemos usar cualquier tipo de rutina que ofrezca BeepFX, teniendo en cuenta que cuantas más uséis, más código necesitará. Una vez que las tengamos todas tendremos que exportar el código:

En el menú principal de la aplicación seleccionamos `File → Compile`.



Marcamos `Assembly` e `Include player code` y pulsamos `Compile`. Grabamos el archivo como, por ejemplo, `mus/dogmole_fx.asm`.

Al igual que pasaba con la música de Beepola, el código que exporta BeepFX no nos sirve directamente, sino que habrá que pasarlo por la turmix. Suponiendo que tengamos el archivo guardado como `mus/dogmole_fx.asm`, nos vamos a `/dev` y ejecutamos así el conversor:

```
$ ..\utils\asm2z88dk.exe ..\mus\dogmole_fx.asm beeper.h mk1
```

Pero todavía no es suficiente, ya que necesitamos una interfaz con **MTE MK1**, una función en C que reciba un parámetro, lo cocine, y llame al player de efectos de BeepFx. Editamos `beeper.h` y añadimos este código al final, que hace precisamente eso:

```
void beep_fx (unsigned char n) {  
    // Cargar en A el valor de n  
    asm_int = n;  
    #asm  
        push ix  
        push iy  
        ld a, (_asm_int)  
        call sound_play  
        pop ix  
        pop iy  
    #endasm  
}
```

Y listo. Si quieres que suenen más tipos de sonido o remapear los que hay me temo que te tendrás que poner a toquetear código.

Capítulo 11: Code Injection Points

Tras este nombre tan rimbombante se esconde una funcionalidad muy sencilla pero que, una vez que nos vayamos empapando hasta el tuétano de **MTE MK1**, puede darnos alas. Se trata de una serie de archivos en los que podemos añadir código C que son incluidos en partes clave del código. De esa forma podremos ampliar la funcionalidad del motor de formas muy interesantes o programar el gameplay igual que hacíamos mediante scripting, pero con mejor kung fu.

Hay una buena colección de puntos de inyección de código que están documentados en la [documentación](#) (!), así que no nos vamos a parar en describirlos todos. Lo que haremos será tomar nuestro **Dogmole**, desactivar el script, y replicar toda la funcionalidad usando código C usando algunos de estos.

Tocando el totete

Exacto: programar en C usando puntos de inyección de código significa tocarle directamente el totete al motor de **MTE MK1**. Eso significa que tendremos que conocer hasta cierto punto dónde tocar, qué tocar y cómo: qué puntos de inyección hay, las funciones de la API, variables globales importantes... Todo eso podréis mirarlo en la [documentación de los CIP](#) y en la [API de MTE MK1](#). Sin embargo, aunque todo está ahí, trataremos de ir describiendo como es debido todo lo que usemos a medida que vayamos metiendo cosas.

Recordando el diseño de gameplay

En el juego hay dos misiones: primero hay que encargarse de eliminar a todos los monjes. Esto abrirá el paso a la Universidad de Miskatonic (eliminando un piedra de la pantalla 2). Una vez abierta, tendremos que ir buscando las cajas y llevándolas al mostrador en la pantalla 0. Cuando estén las 10, terminaremos el juego.

Empezando

Para empezar hemos copiado el proyecto en una carpeta nueva (podéis encontrarlo en `examples/dogmole_ci` del repositorio). Seguidamente, editamos `compile.bat` para cambiar el nombre del proyecto a `dogmole_ci` y posteriormente editamos `my/config.h` para comentar `ACTIVATE_SCRIPTING`.

Si recompilamos en este punto deberemos ver claramente que el script ocupa 0 bytes:

```
Compilando script
Convirtiendo mapa
Convirtiendo enemigos/hotspots
Importando GFX
```

```
Compilando juego
dogmole_ci.bin: 26713 bytes
scripts.bin: 0 bytes
Construyendo cinta
Limpiando
Hecho!
```

Inicializando el juego

Vamos a usar los flags para almacenar valores (en concreto los flags 1 y 3, como en el script original). Sin embargo, al haber desactivado el script, no habrá nada que los inicialice al empezar cada partida, así que tendremos que hacerlo nosotros. Para ello usamos `my/ci/entering_game.h` (que equivale a la sección `ENTERING GAME` del script) y ponemos los dos flags que necesitamos a 0:

```
// my/ci/entering_game.h

flags [1] = 0;
flags [3] = 0;
```

Decoraciones

Necesitamos imprimir decoraciones en cuatro pantallas. En el motor de **MTE MK1** tenemos la función `draw_decorations` que espera que `_gp_gen` apunte a una colección de decoraciones terminada en `0xff` y se encarga precisamente de dibujarlas. Cada decoración es un par de bytes `0xXY` y `0xTT`, el primero con las coordenadas (X, Y) y el segundo con el número de tile.

Tenemos decoraciones en las pantallas 0, 1, 6 y 18. La forma más sencilla de usar `draw_decorations` es definir arrays con nuestras decoraciones y apuntar `_gp_gen` a ellas.

Para definir nuestras propias variables usamos `my/ci/extra_vars.h`:

```
// my/ci/extra_vars.h

const unsigned char decos_0 [] = { 0x37, 22, 0x47, 23, 0x15, 29, 0x16, 20, 0x17, 21, 0x66,
const unsigned char decos_1 [] = { 0x72, 24, 0x82, 25, 0x92, 26, 0x16, 32, 0x17, 33, 0xD6,
const unsigned char decos_6 [] = { 0xA1, 30, 0xA2, 31, 0xA4, 35, 0xff};
const unsigned char decos_18 [] = { 0x48, 34, 0xff };
```

El momento de presentar las decoraciones es el de entrar la pantalla. El punto de inserción de código `my/ci/entering_screen.h` se ejecuta, igual que la sección `ENTERING GAME`, justo en ese momento, después de dibujarla y prepararlo todo, y antes de enviar los resultados a la memoria de video. Es el momento perfecto para modificar la pantalla sin que se note el cambio, por tanto.

```
// my/ci/entering_screen.h

_gp_gen = 0;

switch (n_pant) {
    case 0:
        _gp_gen = decos_0; break;
    case 1:
        _gp_gen = decos_1; break;
    case 6:
        _gp_gen = decos_6; break;
    case 18:
        _gp_gen = decos_18; break;
}

if (_gp_gen) draw_decorations ();
```

Contando monjes

El motor se encarga de contar los enemigos que vamos eliminados en la variable `p_killed`. Como los únicos enemigos que pueden matarse son los monjes, en cuanto `p_killed` valga 20 sabremos que hemos eliminado a todos. Para rizar el rizo, el conversor `ene2h` cuenta cada tipo de enemigo de forma que nuestro código será super robusto si comparamos `p_killed` con el número total de enemigos de tipo 3 (que son los monjes). Las macros `N_ENEMS_TYPE_n` contienen el número exactos de enemigo de tipo `n`, por lo tanto tendremos que comparar con `N_ENEMS_TYPE_3`.

El sitio perfecto para hacer esta comprobación es el punto de inyección de código `my/ci/on_enems_killed.h`, que se ejecutará cada vez que eliminemos un enemigo. Para funcionar como en el script, levantaremos el flag 3 cuando hayamos eliminado los 20 monjes.

Abriremos `my/ci/on_enems_killed.h` y añadiremos el siguiente trozo de código:

```
// my/ci/on_enems_killed.h

if (p_killed == N_ENEMS_TYPE_3) {
    flags [3] = 1;
}
```

Otra cosa que hacíamos era usar `EXTERN` para ejecutar el código que mostraba el cartel de que la puerta estaba abierta. Como la función con nuestro código `extern` no está disponible al haber desactivado el scripting tendremos que añadir el código directamente en el `if` que acabamos de meter. Queda así:

```
// my/ci/on_enems_killed.h

if (p_killed == N_ENEMS_TYPE_3) {
```



```

    flags [3] = 1;

    // Print message
    _t = 79;
    _x = 8; _y = 10; _gp_gen = my_spacer; print_str ();
    _x = 8; _y = 12; print_str ();
    _x = 8; _y = 11; _gp_gen = my_message; print_str ();

    sp_UpdateNowEx (0);

    // Wait
    espera_activa (150);

    // Force reenter
    o_pant = 99;
}

```

Este código necesita dos variables con las cadenas que se imprimen, `my_spacer` y `my_message`. Las añadimos a `my/ci/extra_vars.h`:

```

// my/ci/extra_vars.h

const unsigned char decos_0 [] = { 0x37, 22, 0x47, 23, 0x15, 29, 0x16, 20, 0x17, 21, 0x66,
const unsigned char decos_1 [] = { 0x72, 24, 0x82, 25, 0x92, 26, 0x16, 32, 0x17, 33, 0xD6,
const unsigned char decos_6 [] = { 0xA1, 30, 0xA2, 31, 0xA4, 35, 0xff};
const unsigned char decos_18 [] = { 0x48, 34, 0xff };

unsigned char *my_spacer = "          ";
unsigned char *my_message = " PUERTA ABIERTA ";

```

En el trozo de código que hemos escrito vemos varias cosas:

Imprimir un string

Para imprimir un string empleamos la función `print_str`. La función, como la mayoría del motor de **MTE MK1**, no recibe ningún parámetro, pero necesita que demos valores a las globales generales `_x` e `_y` con las coordenadas, `_t` con el atributo, y `_gp_gen` apuntando a la cadena.

Enviar el buffer a la pantalla

La impresión se hace en el buffer de **splib2**. Para que sea visible hay que decirle a la biblioteca que nos lo envíe a la pantalla. Para ello empleamos la función custom que hemos añadido en **mojonia** a **splib2** `sp_UpdateNowEx` que se encarga de volcar a la pantalla los cuadros de 8x8 (rejilla de caracteres/atributos) que han cambiado. La función recibe un parámetro que puede ser 0 o 1 dependiendo de si queremos que se actualicen los sprites en los cuadros que cambian.

Esperar un rato

La función `espera_activa` del motor de **MTE MK1** espera un tiempo (!) y puede interrumpirse pulsando una tecla.

Forzar reentrada

Llamaos *reentrada* a recargar completamente la pantalla. En este caso, esto es necesario porque hemos impreso un cartel que ha tapado parte de la misma. El motor de **MTE MK1** llama a `draw_scr`, que se encarga de dibujar e inicializar una nueva pantalla, cada vez que la variable `n_pant` (número de pantalla actual) y `o_pant` (número de pantalla anterior) son diferentes. Una forma fácil de redibujar la pantalla sin cambiar de idem es poner en `o_pant` a un valor no válido, como puede ser 99.

Quitar el piedro

Usaremos de nuevo `my/ci/entering_screen.h`. Simplemente detectamos que acabamos de entrar en la pantalla 2 usando `n_pant`, consultamos el valor de `flag [3]` y usamos `update_tile` para modificar el tile:

```
// my/ci/entering_screen.h

_gp_gen = 0;

switch (n_pant) {
    case 0:
        _gp_gen = decos_0; break;
    case 1:
        _gp_gen = decos_1; break;
    case 6:
        _gp_gen = decos_6; break;
    case 18:
        _gp_gen = decos_18; break;
}

if (_gp_gen) draw_decorations ();

if (n_pant == 2) {
    if (flags [3]) {
        _x = 12; _y = 7; _t = _n = 0; update_tile ();
    }
}
```

En qué pantalla estamos:

`n_pant` contiene el número de la pantalla actual.

Actualizando tiles:

La función `update_tile` del motor de **MTE MK1** sirve para modificar tiles en pantalla. Además de dibujar el tile necesario se encargan de modificar los buffers necesarios para que ese tile sea interactuable, por lo que será la función que haya que llamar para hacer una modificación completa. Si sólo quisiésemos modificar gráficamente la pantalla sin modificar los buffers tendríamos que llamar a `draw_invalidate_coloured_tile_gamearea`.

La función `update_tile` no recibe parámetros, pero espera que las coordenadas donde hay que imprimir el tile estén en las variables globales `_x` e `_y` (en coordenadas de tile), el número de tile en `_t` y el comportamiento deseado en `_n`.

La interacción con el altar

Para detectar la interacción con el altar utilizaremos la el punto de inserción de código `my/ci/extra_routines.h`, que se ejecuta al final de cada vuelta del bucle principal. Lo primero que haremos será detectar que estamos en la pantalla correcta, la 0, para posteriormente ver que estamos tocando el altar.

En todo momento, `p_tx` y `p_ty` contienen las coordenadas de la casilla de tile que está tocando el centro del sprite del jugador, por lo que vienen estupendamente para lo que tenemos que hacer: detectar que estemos tocando el altar, que ocupa las casillas (3, 7) y (4, 7).

Cuando todo esto se cumpla tendremos que "liberar" el objeto y contar uno más en el flag 1. Recordad que en modo `ONLY_ONE_OBJECT`, `p_objs` vale 1 cuando llevamos un objeto y 0 cuando no. Aprovecharemos también para detectar que hemos recogido 10:

```
// my/ci/extra_routines.h

if (n_pant == 0) {
    if (p_objs && p_ty == 7 && (p_tx == 3 || p_tx == 4)) {
        p_objs = 0;      // Liberamos el objeto
        ++ flags [1];    // Contamos uno más.

        if (flags [1] == 10) {
            // Terminar el juego "bien"
            success = 1;
            playing = 0;
        }
    }
}
```

Detectando dónde está el jugador nivel fácil

Como hemos visto, una forma muy sencilla de ver dónde está el jugador, o, mejor dicho, qué tile está tocando, es mirar `p_tx` y `p_ty`.

Conteo interno de objetos

El conteo interno de objetos se lleva a cabo en `p_objs`. Si hemos definido `ONLY_ONE_OBJECT`, entonces `p_objs` valdrá 0 al empezar el juego y se pondrá a 1 al coger un objeto... Y entonces ya no podremos coger otro más. Que `p_objs` valga 1 significa que llevamos un objeto. Para "liberar" el objeto tendremos que ponerlo a 0.

Conteo custom de objetos

Si definimos `OBJECT_COUNT`, en modo normal, `p_objs` se copia a `flags [OBJECT_COUNT]` siempre que cogemos un objeto, y además el marcador muestra `flags [OBJECT_COUNT]` en lugar de `p_objs`. Sin embargo, en modo `ONLY_ONE_OBJECT` `p_objs` NO se copia en `flags [OBJECT_COUNT]` aunque lo que se muestre en el marcador sea `flags [OBJECT_COUNT]`. Esto es así para que esto sirva para algo: que `p_objs` sirva únicamente como indicador de si *llevamos un objeto o no*, y usemos `flags [OBJECT_COUNT]` por nuestra cuenta para llevar la cuenta real.

Terminar el juego

Para terminar el juego hay que poner `playing` a 0. El valor de `success` determinará si ganamos el nivel o se muestra **Game Over**.

Y ya está.

Me hago cargo de que este capítulo es bastante árido, pero espero que sirva como ilustración del uso de los puntos de inyección de código (que, como dije al principio, están [debidamente documentados](#)).

Tenemos el scripting para hacer muchas cosas, pero se puede lograr mucho más mediante puntos de inyección de código (por ejemplo, añadir nuevos tipos de enemigos o incluso nuevos motores de movimiento para el jugador). Obviamente, es más complicado usar los puntos de inyección de código, sobre todo si no se conoce bien cómo funciona el motor por dentro.

Otra ventaja de los puntos de inyección de código es que, para juegos con scripts pequeños, generalmente ahorran bastante espacio. Quédate con este dato: [en el momento de elaborar este tutorial] usando scripting, dogmole ocupa 28189 bytes, y usando inyección de código ocupa 26972, ¡que son 1218 bytes menos!

Te recomiendo que te pases por los diferentes ejemplos incluidos con **MTE MK1** y revises los *postmortem* donde se explica cómo se ha resuelto la implementación de los güegos.

Capítulo 12: Juegos multinivel (48K)

Este capítulo es muy interesante pero no ponemos paquetito de materiales, ya que ¿qué mejor paquete que el del Sgt Helmet? Entre los ejemplos de **MTE MK1** tienes la nueva versión de uno de los juegos de Mojon Twins que más nos gusta a los Mojon Twins, esta vez con tres fases. Puedes mirar las cosas de las que hablamos aquí en marcha en ese juego.

Por cierto, este capítulo es denso como su puta madre. Pero así es la vida. Si quieres multinivel tendrás que empaparte muy bien de como funciona el motor. Así que al turrón, chino Cudeiro.

En qué se basa el multiniveleo

Básicamente el multinivel en **MTE MK1** (¡y en **MK2**!) se basa un poco en engañar al chamán. Básicamente el motor es como el abuelo y tú vas y le cambias las pastillas de sitio y le pones las verdes en vez de las rojas.

Para entendernos, **MTE MK1** funcionará (salvo por un par de detalles) como si estuviese ejecutando un juego normal de sólo un nivel como los de la churrera de toda la vida, solo que habrá un agente que se encargará de, antes de que empiece el bucle de juego, de *descomprimir* un nuevo set formado por mapa, cerrojos, tileset, enemigos, hotspots, comportamientos y opcionalmente sprites sobre los actuales, efectivamente *cambiando de fase*.

Lo que se hace para construir un juego de **MTE MK1** multinivel es configurar una especie de nivel "dummy" o "vacío" para hacer "sitio": un mapa a 0, cerrojos vacíos, un tileset con todos los tiles negros, todas las pantallas sin enemigos ni hotspots, y un montón de comportamientos a cero. Obviamente, a esto no se puede jugar. Pero además cargamos una serie de binarios comprimidos con mapas, tilesets, enemigos... y antes de empezar el juego descomprimos los que necesitamos sobre los espacios "vacíos" que hemos reservado.

Activando en `config.h`

En principio, para activar el multinivel sólo tendremos que tocar un par de cosas en `my/config.h`. Obviamente, luego habrá que tunear por varios sitios, pero lo más básico es configurar esto:

```
#define COMPRESSED_LEVELS           // use levels.h instead of mapa.h and enems.h
#define MAX_LEVELS                  3    // # of compressed levels
```

Esto activará los manejos necesarios para el multiniveleo y además indicará que el número de niveles de nuestra secuencia será 3. O sea, que tendremos tres niveles.

Modificando `compile.bat`

Como hemos dicho, los recursos comprimidos se descomprimirán sobre "espacios" especiales para cada tipo de datos que maneja el motor. Por lo general, esto implica que no tendremos que generar enemigos o mapas desde `compile.bat`. Lo único que vamos a dejar es la generación de un tileset "vacío" (porque necesitamos la fuente), un spriteset y las pantallas fijas. Del hacer hueco para el resto ya se encarga **MTE MK1** con la configuración que le hemos hecho. Por tanto abrimos en `compile.bat` y:

1. Nos fumamos la conversión del mapa (`mapcnv`) y la importación de los enemigos (`ene2h`).
2. Modificamos la conversión del tileset (`ts2bin`) para que genere uno usando sólo la fuente, sin incluir tiles. Para ello empleamos la cadena `notiles` en lugar de una ruta al un archivo de tileset:

```
..\..\..\src\utils\ts2bin.exe ..\gfx\font.png notiles font.bin 7 >nul
```

3. Si fuéramos a cambiar el spriteset en cada fase tendríamos que eliminar también la llamada a `sprcnv`.

Preparando los assets

Los *assets* o los *recursos* son las *cosas* con las que vamos a construir nuestros niveles. Como hemos mencionado, un *nivel* de **MTE MK1** se corresponde de (voy a resumirte cosas que ya sabes):

1. Un mapa, en formato PACKED (75 bytes por pantalla, 2 tiles por byte, 16 tiles) o UNPACKED (150 bytes por pantalla, 1 tile por byte, 48 tiles).
2. Un set de cerrojos. Cada cerrojo representa eso, un cerrojo. Se almacena en qué pantalla está y en qué coordenadas, y si está abierto o cerrado.
3. Un tileset, o 192 caracteres o "patrones" y los atributos (o colores) con los que se pintan.
4. Un set de enemigos, con tres por pantalla. Cada enemigo ocupa 10 bytes.
5. Un set de hotspots, uno por pantalla. Cada hotspot ocupa 3 bytes.
6. Un set de comportamientos de tile.
7. Opcionalmente (en 48K, en 128K no es opcional) un spriteset de 16 tiles de 144 bytes más una "cabecera" de 16 bytes (2320 bytes).

Nosotros tendremos que generar nuestro conjunto de recursos, comprimirlos en formato aplib, y luego combinarlos para formar los niveles. El primer paso, por tanto (aparte de, ejem, *hacerlos*) será generar binarios comprimidos de cada uno de ellos, en el formato de **MTE MK1**.

Personalmente, a mi me gusta crearme un archivo `.bat` aparte de `compile.bat` que se encargue de convertir y comprimir todos los assets. En este archivo llamaremos a los conversores para cada

recurso y luego comprimiremos el resultado. Puedes guardarlo todo en `/bin` y así no lo tienes por medio.

Si eres una persona avispada estarás pensando que no paro de hablar de binarios y algunos de los conversores echan archivos `.h` con los arrays en formato código. Y así es, ¡cinco puntos para Gryffindor, señorita Granger!. Por suerte, hay versiones de esos conversores pensadas para multinivel que escupen binarios. Y en esta sección lo vamos a ver usando la generación de recursos de **Helmet** como ejemplo.

Tilesets sin fuente

¿Para qué vamos a guardar la fuente varias veces? Si recordáis, cuando hablamos de `ts2bin` mencionamos que se le podía decir que no pillara fuente. Si en vez de la ruta a nuestro archivo `font.png` ponemos `nofont`, el binario generado sólo contendrá los 192 patrones que forman los tiles (desde el 64 al 255) y los atributos, algo así:

```
..\..\..\src\utils\ts2bin.exe nofont ..\gfx\work0.png ..\bin\tileset0.bin 6 >nul
```

En **Helmet** tenemos tres fases y usamos tres tilesets diferentes, por lo que verás tres líneas de `ts2bin` diferentes en `build_assets.bat`. Recuerda que el numerito que se pasa como cuarto parámetro es la tinta por defecto que se debe usar si en un patrón sólo hay un color. Fijáos que en la fase 2, donde el suelo es mayormente cyan, pasamos un "5".

Fíjate también como colocamos la salida de la conversión en `/bin` especificando la ruta relativa desde `/dev` directamente en el tercer parámetro. Si esto de rutas relativas te suena a chino deberías leer [algún tutorial sobre el tema](#) y reforzar tus conocimientos de la ventana de línea de comandos.

El siguiente paso es comprimirlo todo. Puedes usar `apack` de toda la vida o `apultra`. `Apultra` es [un nuevo compresor de Emmanuel Marty](#) que comprime un poco más que el viejo `apack`. Por contra es bastante más lento, pero si tu PC no es como el mío y pertenece a la actualidad no lo notarás.

Por convención, la versión comprimida de cada binario se llamará igual que el binario pero con una "c" al final, e igualmente la dejaremos en `/bin`. Personalmente me encargo además de borrar los archivos sin comprimir por el tema del orden y la limpieza. Esta sección, para **Helmet**, queda así:

```
echo Making tilesets
..\..\..\src\utils\ts2bin.exe nofont ..\gfx\work0.png ..\bin\tileset0.bin 6 >nul
..\..\..\src\utils\ts2bin.exe nofont ..\gfx\work1.png ..\bin\tileset1.bin 5 >nul
..\..\..\src\utils\ts2bin.exe nofont ..\gfx\work2.png ..\bin\tileset2.bin 6 >nul

..\..\..\src\utils\apultra.exe ..\bin\tileset0.bin ..\bin\tileset0c.bin >nul
..\..\..\src\utils\apultra.exe ..\bin\tileset1.bin ..\bin\tileset1c.bin >nul
..\..\..\src\utils\apultra.exe ..\bin\tileset2.bin ..\bin\tileset2c.bin >nul

del ..\bin\tileset?.bin > nul
```

Tomamos nota de los archivos generados: `tileset0c.bin`, `tileset1c.bin` y `tileset2c.bin`.

Importando mapas y cerrojos binariamente

El viejo `mapcnv` que venimos usando desde tiempos inmemoriales sacaba un archivo de código con dos arrays, uno para el mapa y otro para los cerrojos, por lo que no nos sirve. Por suerte un día se reprodujo por gemación y en la carpeta apareció su análogo binario, el señor `mapcnvbin`, que toma los mismos parámetros (y nos los chiva igualmente al ejecutar simplemente):

```
$ ../utils/mapcnvbin.exe
** USO **
MapCnvBin archivo.map archivo.h ancho_mapa alto_mapa ancho_pantalla alto_pantalla tile_

- archivo.map : Archivo de entrada exportado con mappy en formato raw.
- archivo.h : Archivo de salida
- ancho_mapa : Ancho del mapa en pantallas.
- alto_mapa : Alto del mapa en pantallas.
- ancho_pantalla : Ancho de la pantalla en tiles.
- alto_pantalla : Alto de la pantalla en tiles.
- tile_cerrojo : Nº del tile que representa el cerrojo.
- packed : Escribe esta opción para mapas de la churrera de 16 tiles.
- fixmappy : Escribe esta opción para arreglar lo del tile 0 no negro
```

Por ejemplo, para un mapa de 6x5 pantallas para MTE MK1:

```
MapCnvBin mapa.map mapa.bin 6 5 15 10 15 packed
```

Output will contain the map, and then the bolts

`mapcnvbin` genera un binario con la misma información que `mapcnv`: el mapa en el formato indicado y seguidamente los cerrojos. En la llamada añadiremos `packed` si el mapa es de 16 tiles y `fixmappy` si el primer tile del tileset no era negro completamente y Mappy nos hizo la fullería de desplazar todo el tileset:

```
../utils/mapcnvbin.exe ../map/mapa0.map ../bin/mapa_bolts0.bin 1 24 15 10 15 pac
```

De nuevo ponemos la salida en `/bin`.

Igual que con los tilesets, el siguiente paso será comprimir los binarios. Seguimos la misma convención para los nombres de los archivos y también limpiamos los archivos originales sin comprimir. En **Helmet**:

```
echo Converting maps
../utils/mapcnvbin.exe ../map/mapa0.map ../bin/mapa_bolts0.bin 1 24 15 10 15 pac
../utils/mapcnvbin.exe ../map/mapa1.map ../bin/mapa_bolts1.bin 1 24 15 10 15 pac
../utils/mapcnvbin.exe ../map/mapa2.map ../bin/mapa_bolts2.bin 1 24 15 10 15 pac
```



```

..\..\..\src\utils\apultra.exe ..\bin\mapa_bolts0c.bin ..\bin\mapa_bolts0c.bin >nul
..\..\..\src\utils\apultra.exe ..\bin\mapa_bolts1c.bin ..\bin\mapa_bolts1c.bin >nul
..\..\..\src\utils\apultra.exe ..\bin\mapa_bolts2c.bin ..\bin\mapa_bolts2c.bin >nul

del ..\bin\mapa_bolts?.bin >nul

```

Tomamos nota de los archivos generados: `mapa_bolts0c.bin` , `mapa_bolts1c.bin` y `mapa_bolts2c.bin` . Como veis, intento que los nombres de archivo sean lo más descriptivos posible.

Importando los enemigos y hotspots binariamente

De la misma forma que pasaba con los mapas, normalmente usábamos un conversor que generaba un archivo de código C (`ene2h`) que ahora no nos sirve. La utilidad que emplearemos será `ene2bin_mk1` . Ojal, que el formato es diferente al de **MK2** y no nos vale el `ene2bin` de MK2. Si lo ejecutamos sin parámetros vemos qué espera de nosotros:

```

$ ..\utils\ene2bin_mk1.exe
$ ene2bin_mk1.exe enems.ene enems+hotspots.bin life_gauge [2bytes]

```

The 2bytes parameter is **for** really old .ene files which stored the hotspots 2 bytes each instead of 3 bytes.
 As a rule of thumb:
 .ene file created with `ponedor.exe` -> 3 bytes.
 .ene file created with `colocador.exe` **for** MK1 -> 2 bytes.

El funcionamiento es análogo, aunque deberemos fijarnos muy bien en el parámetro `life_gauge` . Para ahorrar código, como este valor va en la estructura y se descomprime cada vez que empezamos el nivel, si va ya puesto en el binario nos ahorramos el código que lo inicializa. En **helmet** este valor es 2.

Seguidamente comprimimos y borramos y bla bla. Nos queda así:

```

echo Converting enems
..\..\..\src\utils\ene2bin_mk1.exe ..\enems\enems0.ene ..\bin\enems_hotspots0c.bin 2 >nul
..\..\..\src\utils\ene2bin_mk1.exe ..\enems\enems1.ene ..\bin\enems_hotspots1c.bin 2 >nul
..\..\..\src\utils\ene2bin_mk1.exe ..\enems\enems2.ene ..\bin\enems_hotspots2c.bin 2 >nul

..\..\..\src\utils\apultra.exe ..\bin\enems_hotspots0c.bin ..\bin\enems_hotspots0c.bin >nul
..\..\..\src\utils\apultra.exe ..\bin\enems_hotspots1c.bin ..\bin\enems_hotspots1c.bin >nul
..\..\..\src\utils\apultra.exe ..\bin\enems_hotspots2c.bin ..\bin\enems_hotspots2c.bin >nul

del ..\bin\enems_hotspots?.bin

```

Tomamos nota de los archivos generados: `enems_hotspots0c.bin` , `enems_hotspots1c.bin` y `enems_hotspots2c.bin` .

Los comportamientos (behs)

El array `behs` de `my/config.h` ya no se utilizará. En su lugar tendremos un binario de 48 bytes comprimido con los sets de comportamientos. Para generar estos binarios de forma sencilla tenemos `behs2bin` , que toma una lista de 48 valores separada por comas y la convierte en un binario de 48 bytes.

Lo primero por tanto será crear los archivos con las listas de comportamientos. Nosotros en **helmet** usamos dos, ya que los dos primeros tilesets son equivalentes y podemos reaprovechar los comportamientos.

Así se ve un archivo de texto con comportamientos. Nada muy excitante:

```
0,0,8,8,8,8,8,8,17,8,8,8,8,8,10,10,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,8,8,8,8,8,8,8,8,8,0,0
```



Esto equivale a este tileset:



Nosotros hemos tenido a bien guardarlos en `/gfx` junto a los tilesets que representan, pero tú organízate como mejor veas. Tenemos dos: `behs0_1.txt` para las dos primeras fases (recuerda que los verdaderos programadores empieza a contar por 0) y `behs_2.txt` para la tercera.

`behs2bin` únicamente toma dos parámetros: fichero de entrada con la lista en modo texto, y fichero de salida con los 48 bytes en binario. Y luego se comprimen y se borran los originales y bla bla:

```
del ..\bin\enems_hotspots?.bin

echo Converting behs
..\..\..\src\utils\behs2bin.exe ..\gfx\behs0_1.txt ..\bin\behs0_1.bin >nul
..\..\..\src\utils\behs2bin.exe ..\gfx\behs2.txt ..\bin\behs2.bin >nul

..\..\..\src\utils\apultra.exe ..\bin\behs0_1.bin ..\bin\behs0_1c.bin >nul
..\..\..\src\utils\apultra.exe ..\bin\behs2.bin ..\bin\behs2c.bin >nul

del ..\bin\behs0_1.bin
del ..\bin\behs2.bin
```

Toma nota de los últimos archivos generados: `behs0_1c.bin` y `behs2c.bin` . Ya lo tenemos todo.

El levelset

El levelset es un array que contiene información sobre cada nivel de tu juego. Se define en `my/levelset.h` y sus miembros son elementos con esta estructura:

```
// 48K format:
typedef struct {
    unsigned char map_w, map_h;
    unsigned char scr_ini, ini_x, ini_y;
    unsigned char max_objs;
    unsigned char *c_map_bolts;
    unsigned char *c_tileset;
    unsigned char *c_enems_hotspots;
    unsigned char *c_behs;
#ifdef PER_LEVEL_SPRITESET
    unsigned char *c_sprites;
#endif
#ifdef ACTIVATE_SCRIPTING
    unsigned int script_offset;
#endif
} LEVEL;
```

(en modo 48K; en modo 128K es más sencillo pero ya lo veremos en otro capítulo, más adelante). El tema está en crear un array que referencie qué recursos necesitamos para cada nivel y algunos valores relevantes. Pero antes necesitamos que los recursos *estén disponibles*.

La forma de hacerlo es crear una referencia externa a ellos y posteriormente incluirlos desde ensamble en línea con la directiva `BINARY`. Super top todo. Soy consciente de que esto se complica, pero puedes tirar de receta.

Básicamente primero se define una referencia externa para poder añadir cosas a nuestro array:

```
extern unsigned char my_extern_array [0];
```

Esto sencillamente le dice al compilador que hay "algo" fuera que se llama `my_extern_array`. (Probablemente haya mejores formas de hacer esto, pero eso es lo que se traga `z88dk` y si no está roto, no lo arregles).

Luego se mete una sección de ensamble en línea y se mete lo que antes hemos referenciado, tal que así:

```
#asm
    .my_extern_array
        BINARY "my_extern_binary.bin"
#endasm
```

Ojal como la etiqueta del ensamble equivale al nombre del array externo referenciado arriba, pero con un subrayado delante. Porque cuando se compila el C y se genera ensamble, los identificadores de C se convierten en etiquetas de ensamble con el subrayado delante. Y cosas.

El `BINARY` lo que hace es incluir como datos lo que se encuentre en el archivo cuya ruta recibe como parámetro. Y esto es lo que nos hace el truco.

De esta forma estamos metiendo muy fácilmente los binarios que necesitamos en nuestro código, y será la técnica que emplearemos para meter todos los recursos comprimidos. Los importaremos e incluiremos de esta manera en `my/levelset.h` y posteriormente los referenciaremos en el array del *levelset*.

Ahora es cuando coges el papel donde habías apuntado todos los binarios que tenías. Yo he creado estas referencias externas para incluirlos todos. Fíjate como se incluye la ruta relativa adónde está el archivo `mk1.c`, o sea, `../bin/`:

```
// In 48K mode, include here your compressed binaries:
```

```
extern unsigned char map_bolts_0 [0];
extern unsigned char map_bolts_1 [0];
extern unsigned char map_bolts_2 [0];
extern unsigned char tileset_0 [0];
extern unsigned char tileset_1 [0];
extern unsigned char tileset_2 [0];
extern unsigned char enems_hotspots_0 [0];
extern unsigned char enems_hotspots_1 [0];
extern unsigned char enems_hotspots_2 [0];
extern unsigned char behs_0_1 [0];
extern unsigned char behs_2 [0];
```

```
#asm
```

```
._map_bolts_0
    BINARY "../bin/mapa_bolts0c.bin"
._map_bolts_1
    BINARY "../bin/mapa_bolts1c.bin"
._map_bolts_2
    BINARY "../bin/mapa_bolts2c.bin"
._tileset_0
    BINARY "../bin/tileset0c.bin"
._tileset_1
    BINARY "../bin/tileset1c.bin"
._tileset_2
    BINARY "../bin/tileset2c.bin"
._enems_hotspots_0
    BINARY "../bin/enems_hotspots0c.bin"
._enems_hotspots_1
    BINARY "../bin/enems_hotspots1c.bin"
._enems_hotspots_2
    BINARY "../bin/enems_hotspots2c.bin"
._behs_0_1
    BINARY "../bin/behs0_1c.bin"
```

```

        ._behs_2
        BINARY "../bin/behs2c.bin"
    #endasm

```

Con esto de arriba tendremos de gratis un puntero al inicio de cada uno de los recursos de los niveles de nuestro juego, así que ya podemos crear el array con el *levelset*. Este array debe llamarse `levels` e incluir toda la información para cada nivel. Vamos a echar un vistazo al de **Helmet** y vamos comentando:

```

// Define your level sequence array here:
// map_w, map_h, scr_ini, ini_x, ini_y, max_objs, c_map_bolts, c_tileset, c_enems_hotspots
LEVEL levels [] = {
    { 1, 24, 23, 12, 7, 99, map_bolts_0, tileset_0, enems_hotspots_0, behs_0_1 },
    { 1, 24, 23, 12, 7, 99, map_bolts_1, tileset_1, enems_hotspots_1, behs_0_1 },
    { 1, 24, 23, 6, 8, 99, map_bolts_2, tileset_2, enems_hotspots_2, behs_2 }
};

```

Como son muchos campos siempre me hago una chuleta en un comentario. Porque uno es listo, pero no tanto. Vamos a verlos en orden (que es el mismo orden con el que se han definido en el `struct` de más arriba):

1. **Ancho del nivel**, o `map_w` (ver la nota más adelante sobre niveles de diferentes tamaños).
2. **Alto del nivel**, o `map_h`.
3. **Pantalla de inicio**, o `scr_ini`.
4. **Coordenada X de inicio**, o `ini_x`, es coordenadas de tiles.
5. **Coordenada Y de inicio**, o `ini_y`, es coordenadas de tiles.
6. **Máximo de objetos para terminar el nivel**, o `max_objs`, o 99 si esto no debe ser tomado en cuenta, ya que 99, en este contexto, es un valor fuera de rango (ver la siguiente sección).
7. Puntero al recurso con el **mapa y los cerrojos**, o `c_map_bolts`.
8. Puntero al recurso con el **tileset** o `c_tileset`.
9. Puntero al recurso con los **enemigos y hotspots**, o `c_enems_hotspots`.
10. Puntero al recurso con los **comportamientos**, o `c_behs`.
11. Si vamos a usar un *spriteset* diferente por cada fase (en **Helmet** no), un puntero al recurso con el **spriteset**, o `(c_sprites)`.
12. Si el *scripting* está activado (en **Helmet** no), el **offset dentro del script para este nivel**, o `script_offset`. Estos offsets se generan en `my/msc_config.h` al compilar un script multi-nivel, y son historias de `msc3` que aún no hemos visto y que dejaremos para otro momento.

Con esto tendríamos definido nuestro *levelset*. Pero aún hay que tunear un poquito.

Controlando la condición de final de cada nivel

Esto puede complicarse hasta niveles estratosféricos dependiendo de como sea tu juego, sobre todo si va a tener variedad. En realidad todas estas historias terminan siendo muy sencillas cuando empiezas a manejarte realmente bien con el motor y los puntos de inyección de código son tus amigos y tus amantes. Voy a intentar dar aquí un porrón de información y luego ya ves tú qué haces con ella.

Número de objetos

De fábrica, **MTE MK1** terminará la fase actual si `p_objs` vale `PLAYER_NUM_OBJETOS` y `PLAYER_NUM_OBJETOS` fue definida (si no se define, el código de la comprobación no se incluye). Esto va muy bien si ponemos el mismo número de objetos para coleccionar en todas las fases pero ¿y si queremos poner un número diferente en cada fase?

Para lograrlo, *engañaremos al chamán* (TM). Este es el código C que hace la comprobación de que tenemos todos los objetos (en `mainloop/game_loop.h`):

```
#ifdef PLAYER_NUM_OBJETOS
    || p_objs == PLAYER_NUM_OBJETOS
#endif
```

Los `#define` sirven para que el preprocesador de C haga sustituciones textuales, por lo que si queremos que el número sea diferente para cada fase no tendremos más que definir `PLAYER_NUM_OBJETOS` para que no sea un literal numérico, sino el acceso a una variable.

En nuestro *levelset* tenemos información sobre el número de objetos de la fase: el miembro `num_objs` de la estructura `LEVEL`. En **MTE MK1** el nivel actual se almacena en la variable `level`, por lo que el número de objetos definido para el nivel actual es `levels [level].num_objs`. Por tanto, si nos vamos a `my/config.h` y definimos así `PLAYER_NUM_OBJS` lo tenemos:

```
#define PLAYER_NUM_OBJETOS (levels [level].num_objs)
```

Llegar a un sitio concreto

Aquí estamos en una situación igual. *Engañaremos al Chamán* (TM) de igual manera. Este es el código que comprueba que hemos llegado al sitio concreto:

```
#ifdef SCR_FIN
    || (n_pant == SCR_FIN
    #if defined PLAYER_FIN_X && defined PLAYER_FIN_Y
        && ((gpx + 8) >> 4) == PLAYER_FIN_X && ((gpy + 8) >> 4) == PLAYER_FIN_Y
    #endif
    )
#endif
```

Como se ve, si sólo queremos que se detecte que llegamos a la pantalla final, `SCR_FIN` debe estar definida pero `PLAYER_FIN_X` y `PLAYER_FIN_Y` no; si todas están definidas se comprobará también la posición.

Como esto lo hemos usado muy poco no tuve a bien meterlo en la estructura `LEVEL`, pero podemos controlarlo nosotros igualmente con un poco de inyección de código (ver el [capítulo 11](#)). Veamos el caso en que queramos comprobar las tres cosas.

En primer lugar creamos tres arrays con los valores para cada fase de nuestro juego en `my/ci/extra_vars.h`. Vamos a poner que tenemos tres fases en nuestro juego ficticio que deben acabarse en las pantallas 5, 17 y 12, y coordenadas (7,8), (10,6) y (1,1), respectivamente.

```
// my/ci/extra_vars.h
unsigned char scr_fin [MAX_LEVELS]      = { 5, 17, 12 };
unsigned char player_fin_x [MAX_LEVELS] = { 7, 10,  1 };
unsigned char player_fin_y [MAX_LEVELS] = { 8,  6,  1 };
```

Y con esto, no tenemos más que hacer las definiciones en base a estos arrays y la variable `level` en `my/config.h`.

```
#define SCR_FIN                scr_fin [level]
#define PLAYER_FIN_X          player_fin_x [level]
#define PLAYER_FIN_Y          player_fin_y [level]
```

Scripting

Terminar el nivel mediante scripting es muy sencillo. Al igual que para juegos de un solo nivel, si comentamos la definición de las macros `MAX_OBJECTS` y `SCR_FIN` y activamos el scripting, la única forma de terminar la fase será ejecutando

```
WIN GAME
```

desde el script. Además, en multilevel tenemos otra:

```
GAME ENDING
```

Que se salta todas las fases que quedan y muestra directamente el final del juego.

Inyección de código custom

Con la inyección de código es como más fullerías se pueden hacer (ya hemos visto una). Tienes puntos de inyección de códigos muy interesantes para modificar el comportamiento general y aburrido del motor (una fase después de la otra, desde la 0 a la última). Puedes hacer cosas muy

interesantes en `my/ci/before_game`, `my/ci/entering_game.h`, `my/ci/extra_routines.h` y `my/ci/after_game_loop.h` para personalizar hasta el infinito, pero hay algunas cosas automáticas que puedes manejar con muy poco código, generalmente en los puntos de inyección que se ejecutan durante el juego (es buen momento para mirar [la documentación](#)):

Para interrumpir el bucle del juego hay que poner la variable `playing` a 0. Entonces, el motor va a actuar de una u otra forma según el valor un conjunto de variables:

1. Si `success` vale 1, significa que hemos terminado el nivel "BIEN". Si no hacemos nada más, el motor saltará al nivel siguiente o mostrará el final si llegamos a `MAX_LEVELS`. Si, en cambio, vale 0, mostrará la pantalla de Game Over.
2. Si queremos mostrar el final del juego directamente, lo mejor es poner `success` a 1 y establecer `level` a un valor fuera de rango, mayor que `MAX_LEVELS`. Por convención usaremos `0xff`. Hacer esto hace que se interrumpa el bucle de juego y se muestre el final:

```
success = 1; level = 0xff;
```

3. Si tenemos el scripting activado podemos lograr lo mismo poniendo `script_result` a 4.
4. Si tenemos el scripting desactivado y ponemos `warp_to_level` a 1, o si lo tenemos activado y ponemos `script_result` a 3, se volverá a ejecutar el bucle de juego sin modificar `level`, `n_pant`, `p_x`, `gpx`, `p_y` ni `gpy`, por lo que podemos usar para saltar de un nivel a un punto concreto de otro:

```
level = 2;
n_pant = 20;
gpx = 5 << 4; p_x = gpx << FIXBITS;
gpy = 7 << 4; p_y = gpy << FIXBITS;
warp_to_level = success = 1;
```

Eso terminará el bucle de juego y saltará al nivel 2, pantalla 20, y pondrá al jugador en las coordenadas (de tile) (5, 7).

Niveles de diferentes tamaños

Aquí estoy, verborrea loca en Groenlandia, aquí estoy soltando panoja según se me va ocurriendo. Es que son muchas cosas. Quizá algún día venga alguien y ordene esto mejor.

Sí, puedes hacer niveles de diferentes tamaños, de eso te has tenido que dar cuenta porque tienes `map_w` y `map_h` en la estructura `LEVEL`. A lo mejor también te has dado cuenta de la clave de todo esto: como el sistema funciona descomprimiendo un binario comprimido en un espacio en el que hemos hecho sitio, y este sitio se hace obviamente haciendo caso de las variables `MAP_W` y `MAP_H`, entonces ha de cumplirse que el número de pantallas del nivel que más tenga tiene que ser menor o

igual que `MAP_W * MAP_H` . Si te habías dado cuenta de este detalle, otros diez puntos para Gryffindor, señorita Granger, y una estrellita.

En efecto, aunque la geometría del nivel de igual, tiene que caber en el buffer. Al final la geometría da lo mismo, porque un mapa o un conjunto de enemigos y hotspots no deja de estar ordenados como una tira de pantallas en memoria.

Por tanto, lo que hay que hacer es ver cuál es tu nivel más grande (el que más pantallas tenga) y hacer que los valores de `MAP_W` y `MAP_H` en `my/config.h` coincidan con sus dimensiones. Con esto ya lo tienes.

Y listo

Como ves, ya me detengo menos en decir tonterías y más en vomitar información, pero a estas alturas del cuento asumo que las bases están bien interiorizadas y además nos estamos moviendo ya a niveles más avanzados. Así que agárrate, que vienen más curvas.

Capítulo 13: Un juego de 128K

Si quieres saber como hacer un juego para 128K de un sólo nivel (pero con música AY y más sitio para tus cosas en RAM baja) puedes echarle un vistazo al [postmortem del port a v5 de Godkiller 2](#). Con todo lo que sabes y lo que pone ahí deberías tener suficiente.

¡Vaya! Si has conseguido llegar hasta aquí sin mandarme a freir castañas es que debes ser todo un experto en **MTE MK1**, o quizá viste el título del capítulo por separado y has venido directamente aquí porque a lo que tú aspiras es a hacer un juego de verdaz, nada de mierdecillas.

Sea como sea, voy a suponer que has entendido muy bien cómo funciona el multinivel en **MTE MK1** y no me entretendré en explicaciones sobre el tema. También vamos a ver alguna que otra inyección de código, que voy a suponer que sabes como va.

Aquí vamos a construir un juego multinivel de 128K, con cutscenes, casinos y furcias. Vamos a construir

Goku Mal

Goku Mal fue el primer juego para 128K de escrito con **MTE MK1** (que, con él, evolucionó a la versión 3.99.3, si mal no recuerdo). También fue nuestro último juego con el motor, ya que para el siguiente ya nos inventamos MK2.

Se trata de un juego muy sencillo que tiene cinco fases en las que básicamente avanzamos y disparamos, salvo en la cuarta, en la que dejamos un momento aparte el desarrollo puramente lineal y nos entretendremos recogiendo rosas antes de volver a la acción.

La configuración del motor será de vista lateral, con saltos y disparos en 8 direcciones. Este juego levantó una gran polémica entre los super tacañones, ya que estaba pensado para jugarse con dos botones de acción y no incluimos una opción para los que están acostumbrado a agarrarse de un joystick de un solo botón (esto no es una referencia fálica). En esta revisión corregiremos eso.

Tienes el juego completo en el directorio `/examples`. En este tutorial, además de muchas explicaciones, seguiremos paso por paso el mismo proceso que he seguido yo para montarlo en **MTE MK1 v5**.

Los juegos multinivel de 128K

Si recordaréis, un juego multinivel de 48K de **MTE MK1** se parece mucho a un juego normal, solo que antes de empezar a jugar descomprimos nuevos datos sobre el espacio donde el motor espera encontrarlos.

En 128K es básicamente lo mismo, pero con la diferencia de que tendremos casi toda la RAM extra como almacén de datos contenidos. En concreto, de las cinco páginas que no forman los 48K base del Spectrum de 128K, usaremos una para la música y tendremos cuatro más para datos comprimidos. ¡Y en 64K cabe mogollón de cosas!

Para que te hagas una idea, las tres fases de Helmet ocupaban menos de 7Kb.

Aunque al principio el tema acojone un poco, lo cierto es que hacer un juego de 128K es más sencillo que un multinivel de 48K, porque aunque tenga más trabajo (normalmente meteremos más contenido y además tendremos que diseñar una banda sonora), los procesos son mucho más sencillos, como veremos muy pronto.

Empezamos

Lo primero que haremos será reunir todos los recursos. Posteriormente veremos cómo importarlos, qué es un *level bundle* y cómo funciona **librarian**.

Todos los tiestos.

He puesto *recursos* o *assets* porque sonaba más profesional y tal, pero la verdad es que en mojonía siempre llamamos a estas cosas *tiestos*. Vamos a ver qué tiestos lleva **Goku Mal**.

Todos los tiestos irán en la RAM extra. Generaremos un script `build_assets.bat` para construir los binarios que se cargarán en la RAM extra, incluyendo la banda sonora.

En primer lugar tendremos un buen porrón de pantallas comprimidas. Además de las tres estándar de **MTE MK1** (título, marco y final) tendremos tres splash screens que se muestran tras cargar el juego y varias con imágenes para las *cutscenes*. Esta es una lista completa:

1. `title.png`, `marco.png` y `ending.png` ya sabes para qué son. Este juego, como ves, tiene marco separado de la pantalla de título.
2. `logo.png` tiene un logo de los Mojon Twins y el título del juego y se muestra nada más finalizar la carga.
3. `dedicado.png` es una dedicatoria, y se muestra también una única vez, al terminar la carga.
4. `controls.png` la hicimos para que los super tacañones no se quejaran mucho por no usar OPQA (¡ni QAOP!), pero no funcionó para nada porque igualmente nos pusieron a parir.
5. `intro1.png`, `intro2.png`, `intro3.png` e `intro4.png` se muestran al iniciar una nueva partida y contienen la introducción del juego.
6. `intro5.png` se muestra antes de la fase 4 y sirve para ilustrar una escena del argumento del juego.
7. `intro6.png` e `intro7.png` forman parte de la secuencia final.
8. `zoneA.png` y `zoneB.png` se muestran antes de empezar cada nivel y muestran la cara de mala hostia de Goku Mal y el número de la fase que toca.

9. Finalmente tendremos nuestra pantalla de carga, `loading.png` , que tendremos que convertir pero no comprimir. Dejaremos esta tarea para `compile.bat` más adelante.

Sin salirnos de `gfx/` , tendremos además los gráficos para construir los niveles:

1. `work0.png` , `work1.png` , `work2.png` , `work3.png` y `work4.png` contienen los tilesets de las cinco fases. Es importante notar que todos estos tilesets tienen un tile 0 que no es del todo negro, por lo que habrá que tener en cuenta esto más adelante (ya que los `.MAP` tendrán los valores desplazados debido al glitch de mappy que ya conocemos bien). Tenemos aquí sus archivos de comportamientos asociados, `behs0.txt` a `behs4.txt` .
2. `sprites0.png` , `sprites1.png` , `sprites2.png` , `sprites3.png` y `sprites4.png` contienen los cinco spritesets correspondientes a cada una de las cinco fases.
3. `font.png` contiene la fuente base que se usará durante el juego.
4. `level_screen_ts.png` es un charset completo que se empleará para las pantallas de nuevo nivel especiales que tiene este juego.

En `map/` tenemos los mapas de los cinco niveles, `mapa0.fmp` a `mapa4.fmp` y sus exports en formato MAP : `mapa0.MAP` a `mapa5.MAP` .

En `enems/` tenemos los archivos de colocación de cosas para Ponedor, `enems0.ene` a `enems4.ene` . También hemos copiado aquí los `work?.png` con los tilesets y los `mapa?.MAP` con los mapas para que Ponedor los tenga bien a mano.

Este juego no tiene scripting, así que por último tenemos la banda sonora y OGT en el directorio `mus/` . Esta OGT fue creada con **Davidian** y tiene un montón de canciones. Ahora nos vamos a centrar más en la importación y construcción del juego pero dedicaré un capítulo para hablaros de cómo montar las OGT con Wyz Player.

Importación: pantallas fijas

Empezaremos a construir nuestro script encargado de crear los binarios con las pantallas fijas. Atención y muy importante: **Para que MTE MK1 funcione en modo 128K, las pantallas fijas básicas title, marco y ending deben llamarse `title.bin` , `marco.bin` y `ending.bin`**

```
@echo off
```

```
if [%1]==[skipscr] goto skipscr
```

```
echo Converting Fixed Screens
```

```
..\..\..\src\utils\png2scr ..\gfx\title.png ..\bin\title.scr > nul
..\..\..\src\utils\png2scr ..\gfx\marco.png ..\bin\marco.scr > nul
..\..\..\src\utils\png2scr ..\gfx\ending.png ..\bin\ending.scr > nul
..\..\..\src\utils\png2scr ..\gfx\logo.png ..\bin\logo.scr > nul
..\..\..\src\utils\png2scr ..\gfx\dedicado.png ..\bin\dedicado.scr > nul
..\..\..\src\utils\png2scr ..\gfx\controls.png ..\bin\controls.scr > nul
..\..\..\src\utils\png2scr ..\gfx\intro1.png ..\bin\intro1.scr > nul
..\..\..\src\utils\png2scr ..\gfx\intro2.png ..\bin\intro2.scr > nul
```

```

..\..\..\src\utils\png2scr ..\gfx\intro3.png ..\bin\intro3.scr > nul
..\..\..\src\utils\png2scr ..\gfx\intro4.png ..\bin\intro4.scr > nul
..\..\..\src\utils\png2scr ..\gfx\intro5.png ..\bin\intro5.scr > nul
..\..\..\src\utils\png2scr ..\gfx\intro6.png ..\bin\intro6.scr > nul
..\..\..\src\utils\png2scr ..\gfx\intro7.png ..\bin\intro7.scr > nul
..\..\..\src\utils\png2scr ..\gfx\zoneA.png ..\bin\zoneA.scr > nul
..\..\..\src\utils\png2scr ..\gfx\zoneB.png ..\bin\zoneB.scr > nul

..\..\..\src\utils\apultra ..\bin\title.scr ..\bin\title.bin > nul
..\..\..\src\utils\apultra ..\bin\marco.scr ..\bin\marco.bin > nul
..\..\..\src\utils\apultra ..\bin\ending.scr ..\bin\ending.bin > nul
..\..\..\src\utils\apultra ..\bin\logo.scr ..\bin\logo.bin > nul
..\..\..\src\utils\apultra ..\bin\dedicado.scr ..\bin\dedicado.bin > nul
..\..\..\src\utils\apultra ..\bin\controls.scr ..\bin\controls.bin > nul
..\..\..\src\utils\apultra ..\bin\intro1.scr ..\bin\intro1.bin > nul
..\..\..\src\utils\apultra ..\bin\intro2.scr ..\bin\intro2.bin > nul
..\..\..\src\utils\apultra ..\bin\intro3.scr ..\bin\intro3.bin > nul
..\..\..\src\utils\apultra ..\bin\intro4.scr ..\bin\intro4.bin > nul
..\..\..\src\utils\apultra ..\bin\intro5.scr ..\bin\intro5.bin > nul
..\..\..\src\utils\apultra ..\bin\intro6.scr ..\bin\intro6.bin > nul
..\..\..\src\utils\apultra ..\bin\intro7.scr ..\bin\intro7.bin > nul
..\..\..\src\utils\apultra ..\bin\zoneA.scr ..\bin\zoneA.bin > nul
..\..\..\src\utils\apultra ..\bin\zoneB.scr ..\bin\zoneB.bin > nul

del ..\bin\*.scr > nul

:skipscr

```

Como normalmente en un desarrollo de este tipo las pantallas fijas son lo que menos va a evolucionar y además tardan un ratillo en comprimirse, añadimos un poco de magia batch. Cuando ejecutemos `build_assets.bat` con el parámetro `skipscr` la ejecución se saltará la conversión de las imágenes, lo que vendrá bien cuando tengamos que regenerar una y otra vez los mapas o la colocación de enemigos (repito, en un desarrollo normal en el que vas construyendo y depurando el juego poco a poco. Aquí ya lo tengo todo hecho).

Importación: Level bundles

En modo 128K quisimos simplificar la importación de niveles creando los **level bundles**, que no son más que gruesos binarios que contienen todos los tiestos que definen un nivel: mapa, cerrojos, tileset, enemigos, hotspots, comportamientos y spriteset. Estos binarios se descomprimen de un plumazo sobre la zona de datos de **MTE MK1** ya que están organizados de la misma forma que ésta.

Para crear estos bundles utilizamos `buildlevels_MK1.exe` que puede recibir y recibe un millón de parámetros. Como siempre podemos ejecutar sin pasar nada y nos los chiva:

```

C:\git\MK1\src\utils>buildlevels_MK1.exe
buildlevel v0.5 20200125
Builds a level bundle for MTE MK1 5.0+

usage

```

```
$ buildlevel.exe output.bin key1=value1 key2=value2 ...
```

output.bin Output file name.

Parameters to buildlevel.exe are specified as key=value, where keys are as follows:

MAP DATA

mapsize	Needed if game contains differently sized levels: MAP_W*MAP:H
mapfile	Especifies the .map file. packed/unpacked autodetected.
map_w	Map width in screens.
map_h	Map height in screens.
decorations	Output filename for decorations. This makes your map packed.
lock	Tile # for locks. (optional)
fixmappy	Fixes mappy's 'no first black tile' behaviour

TILESET/CHARSET DATA

tilesfile	work.png (256x48) containing 48 16x16 tiles.
behsfile	behs.txt containing 48 comma-separated values.
defaultink	Value to use when PAPER=INK.

SPRITESET

spritesfile	sprites.png (256x?) containing N 16x16 sprites + masks.
nsprites	# of sprites in sprites.png. If omitted, defaults to 16.

ENEMIES

enemsfile	enems.ene file
-----------	----------------

HEADER STUFF

scr_ini	Initial screen #
ini_x	Initial x position (tiles)
ini_y	Initial y position (tiles)
max_objs	Max objects (can be omitted If not applicable)
enems_life	Enems life

Como ves, se trata de un increíble batiburrillo de cosas de todos los conversores que hemos visto hasta el momento. Es posible que te estés preguntando por qué narices no se usa algo así también para los juegos normales y nos olvidamos de tanto conversor. Y es una pregunta muy interesante para la que ahora mismo no tengo respuesta :-D

Vamos por partes:

1. output.bin (obligatorio) es el nombre de archivo de salida, donde se escribirá toda la mandanga. El resto de los parámetros se escribe en formato `clave=valor` .

2. `mapsize` (opcional): indica el número total de pantallas que hay en el espacio de memoria reservado para el nivel, esto es, el resultado de multiplicar `MAP_W * MAP_H`. Esto es para juegos en los que tengas niveles de tamaños diferentes (mira el capítulo anterior para refrescar estos conceptos). Es opcional porque si todos tus niveles tienen el mismo número de pantallas te basta con especificar los siguientes parámetros:
3. `map_w` y `map_h` (obligatorios): especifican el ancho y alto en pantallas del nivel. Recuerda que `map_w * map_h` debe ser menor que el resultado de `MAP_W * MAP_H` (las macros de `my/config.h`) y por tanto menor o igual que `mapsize` si lo has especificado.
4. `mapfile` (obligatorio) es una ruta al archivo con el mapa en formato `.MAP`.
5. `decorations` (opcional) es una ruta a un archivo de salida en formato script del motor con decoraciones automáticas y que puedes incluir desde tu script para adornar las pantallas de forma automática. Básicamente el mapa se fuerza a *PACKED* y los tiles fuera de rango se escriben como decoraciones de las que se imprimen en los `ENTERING SCREEN`. Es una característica de `msc3` que aún no hemos comentado.
6. `lock` (opcional) sirve para especificar qué tile hace de cerrojo. Para **MTE MK1** tiene que ser el 15. Puedes omitir el parámetro si no usas cerrojos.
7. `fixmappy` (opcional) para deshacer el desbarajuste que líaa mappy si tu tileset no empieza por uno completamente negro.
8. `tilesfile` (obligatorio) debe contener la ruta al archivo con el tileset de hasta 48 tiles, que debe ser un archivo tipo png de 256x48, como hemos visto.
9. `behsfile` (obligatorio) es la ruta al archivo con los comportamientos, en el formato que vimos en el anterior capítulo: un archivo de texto con los 48 valores separados por comas.
10. `defaultink` (opcional) especifica una tinta por defecto para los patrones que solo tengan un color, igual que en `ts2bin`.
11. `spritesfile` (obligatorio) es la ruta al archivo con el spriteset en el formato de siempre.
12. `nsprites` (opcional) por si tenemos más de 16 sprites y que, por el momento, omitiremos (no soportado por el motor).
13. `enemsfile` (obligatorio) contendrá la ruta al archivo de colocación de enemigos y hotspots `.ene` del Ponedor.
14. Los datos de inicio `scr_ini`, `ini_x` e `ini_y` (obligatorios) sirven para indicar donde se empieza.
15. `max_objs` (opcional) es el número de objetos recogiscibles del nivel. Recuerda que este parámetro no se tomará en cuenta de todos modos a menos que hagamos la configuración

parecida a la que explicamos en el capítulo anterior (en el apartado **Controlando la condición de final de cada nivel**) y que veremos mas tarde.

16. `enems_life` (obligatorio) establece la vida de los enemigos.

Hemos dicho que **Goku Mal** es un juego muy sencillo. En él no ha scripting y originalmente había algo de código custom, pero en esta recreación intentaremos hacerlo funcionar primero con lo más básico de **MTE MK1**. En las cuatro fases de acción en las que sólo hay que avanzar colocaremos un único item al final de la fase, de forma que al tocarlo terminaremos el nivel. En la fase 4, en la que hay que reunir un número de flores, igualmente terminaremos al reunir las todas. Por tanto, la condición de final de todas las fases será sencillamente recoger todos los objetos.

Lo que tenemos que hacer ahora es escribir las cinco llamadas a `buildlevels_MK1` para generar los cinco *level bundles*. Veamos el primero.

El primer nivel se compone de `mapa0.map`, `enems0.ene`, `work0.png`, `behs0.txt` y `sprites0.png`. Se trata de un nivel vertical de 25 pantallas de altura. En este juego todos los niveles tienen 25 pantallas en diferentes geometrías menos el cuarto, que tiene 15, por lo que es necesario emplear el parametro `mapsize`. Configuraremos `MAP_W` y `MAP_H` en `my/config.h` para que multiplicados den 25, y usaremos `mapsize=25` en todas las conversiones.

Como el primer tile de nuestro tileset no estaba vacío y lo hemos usado directamente para crear el mapa en mappy, éste ha desplazado todos los valores una unidad y necesitaremos especificar `fixmappy`.

En el nivel se empieza en el nivel más bajo, o sea, en la pantalla 24, pegados al suelo, en la parte izquierda (coordenadas (2, 8)). En este juego no usamos cerrojos, por lo que omitiremos el parámetro. Tendremos que recordar activar `DEACTIVATE_KEYS`, ya que en el level bundle no se generarán cerrojos y por tanto tenemos que decirle al motor que no reserve sitio para ellos.

Finamente, en esta primera fase los enemigos normales soportarán dos golpes antes de morir.

Por tanto la llamada a `buildlevels_MK1.exe`, pensada para ejecutarse desde `dev/`, sera algo así, para esta fase:

```
$ ../../src/utls/buildlevels_MK1.exe ../bin/level0.bin mapsize=25 mapfile=../map/mapa0
```

Si ejecutas esto directamente verás todo el proceso y todas las partes que, si te fijas, aparecen en el mismo orden que los distintos espacios reservados en `assets/levels.h` para poder plantar el binario justo ahí y que todo funcione.

Añadimos pues una llamada a `buildlevels` para cada uno de nuestros niveles, ajustando los parámetros necesarios según cada nivel. Fíjate en como las dimensiones del mapa van cambiando (aunque siempre son 25 pantallas), en cada nivel se empieza en un sitio diferente, cambia el número de objetos, y la cantidad de vida de los enemigos. Posteriormente comprimimos y limpiamos:


```

echo Converting levels
..\..\..\src\utils\buildlevels_MK1.exe ..\bin\level0.bin mapsize=25 mapfile=..\map\mapa0.M
..\..\..\src\utils\buildlevels_MK1.exe ..\bin\level1.bin mapsize=25 mapfile=..\map\mapa1.M
..\..\..\src\utils\buildlevels_MK1.exe ..\bin\level2.bin mapsize=25 mapfile=..\map\mapa2.M
..\..\..\src\utils\buildlevels_MK1.exe ..\bin\level3.bin mapsize=25 mapfile=..\map\mapa3.M
..\..\..\src\utils\buildlevels_MK1.exe ..\bin\level4.bin mapsize=25 mapfile=..\map\mapa4.M

..\..\..\src\utils\apultra ..\bin\level0.bin ..\bin\level0c.bin > nul
..\..\..\src\utils\apultra ..\bin\level1.bin ..\bin\level1c.bin > nul
..\..\..\src\utils\apultra ..\bin\level2.bin ..\bin\level2c.bin > nul
..\..\..\src\utils\apultra ..\bin\level3.bin ..\bin\level3c.bin > nul
..\..\..\src\utils\apultra ..\bin\level4.bin ..\bin\level4c.bin > nul

del ..\bin\level?.bin > nul

```

Ahora necesitamos una conversión más que usaremos luego en nuestras super pantallas de nuevo nivel super mega hiper chachis. Vamos a importar 192 caracteres que tenemos en level_screen_ts.png que descomprimiremos sobre el area del tileset para mostrar números gordunos. Para esa conversión tenemos chr2bin :

```

C:\git\MK1\src\utils>chr2bin.exe
chr2bin v0.4 20200119 ~ Usage:

$ chr2bin charset.png charset.bin n [defaultink|noattrs]

```

where:

- * charset.png is a 256x64 file with max. 256 chars.
- * charset.bin is the output, 2304/2048 bytes bin file.
- * n how many chars, 1-256
- * defaultink: a number 0-7. Use this colour as 2nd colour if there's only one colour in a 8x8 cell
- * noattrs: just outputs the bitmaps.

Para lo que vamos a usar estos gráficos no necesitamos los atributos así que los convertiremos así; luego los comprimimos y limpiamos:

```

..\..\..\src\utils\chr2bin ..\gfx\level_screen_ts.png ..\bin\level_screen_ts.bin 192 noatt
..\..\..\src\utils\apultra ..\bin\level_screen_ts.bin ..\bin\level_screen_tsc.bin > nul

del ..\bin\level_screen_ts.bin > nul

```

The Librarian

Aquí viene la parte guay. Hemos dicho que todos estos tiestos irán a nuestro almacenamiento en la RAM extra, que son los 64K que hay juntando las RAMs 3, 4, 6 y 7. Aparte de esto, en RAM 1

meteremos la música y el player, y las RAMs 5, 2 y 0 son las que (en ese orden) forman los 48K base que es donde se carga y opera el binario principal de nuestro juego.

Para meter todos estos tiestos en la RAM extra usaremos **The Librarian** en su versión 2, que leerá una lista de binarios y los irá metiendo en hasta cuatro archivos RAM?.bin de salida usando dos algoritmos diferentes:

1. Automático: **The Librarian** usa un algoritmo para encontrar una ordenación de los binarios que minimice el número de RAMs empleadas. Encontrar una ordenación *óptima* es un problema NP-Completo (de complejidad exponencial). **The Librarian** usa un algoritmo que encuentra una ordenación bastante buena, pero no es perfecto y a veces fallará. Por eso tenemos el modo:
2. Manual: **The Librarian** sigue el orden que aparece en el archivo de entrada con la lista.

Aunque no es necesario por ahora para este juego, **The Librarian** además acepta precargar binarios al principio de una o varias RAMs, y empezará a acomodar el resto de tus binarios detrás. Esto sirve por ejemplo para colocar el script de tu juego en RAM extra, por ejemplo en RAM6. En un caso así, se renombraría `scripts.bin` como `preload6.bin` y **The Librarian** lo colocaría automáticamente al principio de RAM6 y acomodaría el resto de los binarios en las otras RAMs y tras los scripts en RAM6. Así le podríamos decir al motor que los scripts están al principio de RAM6 y todo funcionaría. Pero como **Goku Mal** no tiene scripting, pues no necesitaremos esta característica.

Finalmente, lo otro que hace **The Librarian** es generar una estructura con información sobre dónde encontrar cada binario y macros para identificarlos, que necesitaremos más adelante cuando estemos construyendo nuestro *levelset*.

Antes de que te abrumes demasiado vamos a verlo en acción. **The Librarian** usa unos cuantos parámetros que nos chivará, como es costumbre, si lo ejecutamos a pelo:

```
C:\git\MK1\src\utils>librarian2.exe
librarian v2.0 20200202 ~ **ERROR** list is Missing!

usage:

$ librarian2 list=list.txt index=output.h [bins_prefix=bins_prefix]
           [rams_prefix=rams_prefix] [manual]

* list.txt contains the list of binaries to store
* output.h is the output file with the index
* bins_prefix will be prepended to input preload?.bin & bin files
* rams_prefix will be prepended to output RAMn.bin files
* use manual for manual order.
```

Los parámetros hacen lo siguiente:

1. `list` (obligatorio) indica la ruta al archivo con la lista de binarios. A mí me gusta dejarla en `/bin` y llamarla `list.txt`.

2. `index` (obligatorio) indica la ruta al archivo de código que contendrá el índice de binarios (esto es: dónde encontrarlos). Para **MTE MK1** esta ruta debe ser `/dev/assets/librarian.h`.
3. `bins_prefix` (opcional) sirve para proporcionar una ruta para los archivos binarios que aparecen en la lista. Si ejecutamos **The Librarian** desde `dev/`, desde nuestro script, y los binarios en la lista aparecen simplemente nombrados pero están en `bin/`, como será nuestro caso, tendremos que emplear este parámetro con el valor `../bin/` para que **The Librarian** pueda encontrarlos.
4. `rams_prefix` (opcional) funciona parecido pero con los archivos `RAM?.bin` de salida. Si no ponemos nada los creará en `dev/`, pero queremos que estén en `bin/`, por lo que tendremos que especificar `../bin/` también como valor de este parámetro.
5. Por último, si añadimos `manual` a la línea de comando, se empleará el orden natural. Nosotros confiaremos en el algoritmo para este juego, que funcionará genialmente y ubicará los 35K de binarios comprimidos en tenemos en tres páginas de RAM: RAM3, RAM4 y RAM6.

Por lo tanto, lo siguiente será añadir una nueva línea a nuestro `build_assets.bat`: la llamada a **The Librarian**, que será:

```
echo Running The Librarian
..\..\..\src\utils\librarian2.exe list=..\bin\list.txt index=assets\librarian.h bins_prefi
```

Y crear un archivo `list.txt` en `bin/` con la lista de todos los binarios, uno por línea, o sea:

```
title.bin
marco.bin
ending.bin
dedicado.bin
controls.bin
logo.bin
zoneA.bin
zoneB.bin
intro1.bin
intro2.bin
intro3.bin
intro4.bin
intro5.bin
intro6.bin
intro7.bin
level_screen_tsc.bin
level0c.bin
level1c.bin
level2c.bin
level3c.bin
level4c.bin
```

Como hemos dicho, no nos detendremos en cómo montar una OGT aquí, sino que lo dejaremos para otro capítulo. Aquí sólo veremos cómo generar RAM1 a partir de una OGT ya hecha.

Pondremos en `mus/` todo lo necesario, a saber:

1. `WYZproPlay47aZXc.ASM` , el código del player, ya modificado con la lista de canciones, e incluyendo nuestros instrumentos y efectos de sonido.
2. `instrumentos.asm` , con los instrumentos según exporta Wyz Player.
3. `efectos.asm` , con los efectos según los exporta Wyz Player.
4. `*.mus` , todas las canciones.

Con todo en su sitio, sólo tendremos que llamar al ensamblador `pasmo` , incluido en `/src/utils/` , para generar `RAM1.bin` y colocarla en `bin/` . Añadimos una última línea a nuestro `build_assets.bat` :

```
echo Making music
..\..\..\src\utils\pasmo ..\mus\WYZproPlay47aZXc.ASM ..\bin\RAM1.bin
```

Finalmente me gusta chivar lo que ocupa cada archivo:

```
echo DONE
..\..\..\src\utils\printsize ..\bin\RAM1.bin
..\..\..\src\utils\printsize ..\bin\RAM3.bin
..\..\..\src\utils\printsize ..\bin\RAM4.bin
..\..\..\src\utils\printsize ..\bin\RAM6.bin
```

¡Y ya lo tenemos todo! Es el momento de irse a `dev/` y ejecutar `build_assets.bat` y ver como se obtienen los diferentes `RAM?.bin` en `bin/` . El resultado de la ejecución debería ser algo parecido a esto:

```
$ build_assets.bat
Converting Fixed Screens
Converting levels
Converting more stuff
Running The Librarian
Making music
DONE
..\bin\RAM1.bin: 14729 bytes
..\bin\RAM3.bin: 16136 bytes
..\bin\RAM4.bin: 16348 bytes
..\bin\RAM6.bin: 2640 bytes
```

Modificando `compile.bat`

El siguiente paso es modificar `compile.bat` para quitar toda la importación de tiestos de un juego normal mononivel y posteriormente construir una cinta de 128K.

Nos fumaremos completamente las secciones donde se convierten mapas, enemigos, sprites y pantallas fijas (menos la pantalla de carga), pero tendremos que dejar la generación de los sprites extra y la importación de la fuente. La importación de tios en `compile.bat` se queda, pues, en la mínima expresión:

```
echo Importando GFX
..\..\..\src\utils\ts2bin.exe ..\gfx\font.png notiles font.bin 7 >nul

..\..\..\src\utils\sprcnvbin.exe ..\gfx\sprites_extra.png sprites_extra.bin 1 > nul
..\..\..\src\utils\sprcnvbin8.exe ..\gfx\sprites_bullet.png sprites_bullet.bin 1 > nul

if [%1]==[justassets] goto :end
```

Generando la cinta de 128K

La carga de un juego de 128K, desde BASIC, se basa en ir cargando primero cada RAM en `$8000` y luego ejecutando una pequeña rutina en ASM que pagina la RAM correcta y posteriormente copia el bloque a `$C000`, para finalmente volver a poner la configuración de páginas como estaba para seguir cargando el siguiente bloque. Finalmente se carga el bloque principal y se ejecuta.

Así era y ha sido siempre hasta que haciendo **Ninjajar!** nos vimos en la tesitura de que estábamos llenando también RAM7 completamente y descubrimos de mala manera que el 128 BASIC corrompe RAM7. Por eso, haciendo uso de la infinita sabiduría de **Antonio Villena**, construimos un sencillo cargador en ensamble.

El cargador de marras es `dev/loader/loaderzx.asm-orig`. Este archivo tendrá que ser preprocesado para sustituir unas macros que tiene por la longitud real de los diferentes archivos que tiene que cargar y posteriormente compilado por pasmo, pero lo primero que tendremos que hacer es adaptar el archivo a nuestro proyecto, ya que de fábrica viene preparado para cargar RAM1, RAM3, RAM4, RAM6 y RAM7.

En Goku Mal no tenemos RAM7, así que tendremos que editar el archivo y quitar el bloque que carga esta ram, o sea, eliminar todo esto (línea 68 a la 79):

```
; RAM7
    ld  a, $17      ; ROM 1, RAM 7
    ld  bc, $7ffd
    out (C), a

    scf
    ld  a, $ff
    ld  ix, $C000
    ld  de, %ram7_length%%
    call $0556
    di
```

Hecho esto nos tendremos que ir a `compile.bat` y sustituir toda la parte de generación de la cinta de 48K, o sea, esto:

```
echo Construyendo cinta
rem cambia LOADER por el nombre que quieres que salga en Program:
..\..\..\src\utils\bas2tap -a10 -sDOGMOLE loader\loader.bas loader.tap > nul
..\..\..\src\utils\bin2tap -o screen.tap -a 16384 loading.bin > nul
..\..\..\src\utils\bin2tap -o main.tap -a 24000 %game%.bin > nul
copy /b loader.tap + screen.tap + main.tap %game%.tap > nul
```

Por esto otro:

```
echo Construyendo cinta
..\..\..\src\utils\imanol.exe ^
    in=loader\loaderzx.asm-orig ^
    out=loader\loader.asm ^
    ram1_length=?..\bin\RAM1.bin ^
    ram3_length=?..\bin\RAM3.bin ^
    ram4_length=?..\bin\RAM4.bin ^
    ram6_length=?..\bin\RAM6.bin ^
    mb_length=?%game%.bin > nul

..\..\..\src\utils\pasmox.exe loader\loader.asm ..\bin\loader.bin loader.txt

..\..\..\src\utils\GenTape.exe %game%.tap ^
    basic 'GOKU_MAL' 10 ..\bin\loader.bin ^
    data          loading.bin ^
    data          ..\bin\RAM1.bin ^
    data          ..\bin\RAM3.bin ^
    data          ..\bin\RAM4.bin ^
    data          ..\bin\RAM6.bin ^
    data          %game%.bin
```

Atención que aquí hay un poco de magias. Empezamos ejecutando `imanol.exe`, que es un programa sencillo para hacer sustituciones textuales por cosas calculadas, como la longitud de un archivo. Aquí simplemente estamos preprocesando `loaderzx.asm-orig` para generar un `loader.asm` con las longitudes correctas de cada bloque.

En la siguiente línea llamamos a `pasmox` para que lo ensamble y genere un `../bin/loader.bin`.

Por último usamos `GenTape` del mismo **Antonio Villena** para construir la cinta con todos los bloques necesarios. Fíjate como va primero la pantalla de carga seguida de los cuatro bloques para las RAMs extra que usamos (RAM1, RAM3, RAM4 y RAM6) y finalmente el binario principal. Si tu juego tiene un número de RAMs diferentes tendrás que ajustar esto también.

Qué ha hecho The Librarian

The Librarian ha hecho algo genial: nos ha creado un archivo `assets/librarian.h` que contiene información sobre la ubicación de cada uno de nuestros tiestos, y además ha creado una macro para cada tiesto para que los podamos referenciar cómodamente en el *levelset* o donde necesitemos. Si abres el archivo verás primero la estructura `resources` con la página y el offset de cada uno de nuestros recursos:

```
RESOURCE resources [] = {
    { 3, 0xC000 },
    { 3, 0xC9D7 },
    { 3, 0xD357 },
    { 3, 0xDBE8 },
    { 3, 0xE470 },
    { 3, 0xECE9 },
    [...]
};
```

Y luego una ristra de macros:

```
#define LEVEL0C_BIN      0
#define LEVEL4C_BIN      1
#define LEVEL2C_BIN      2
#define LEVEL1C_BIN      3
#define ZONEA_BIN        4
#define LEVEL3C_BIN      5
#define ZONEB_BIN        6
#define INTRO07_BIN      7
#define INTRO05_BIN      8
#define INTRO06_BIN      9
#define INTRO02_BIN     10
#define INTRO04_BIN     11
#define INTRO03_BIN     12
#define ENDING_BIN       13
#define INTRO01_BIN     14
#define DEDICADO_BIN     15
#define TITLE_BIN        16
#define LOGO_BIN         17
#define CONTROLS_BIN     18
#define LEVEL_SCREEN_TSC_BIN 19
#define MARCO_BIN        20
```

Cuando se activa el modo 128K (como veremos más adelante), está disponible la función `get_resource`, que sirve para descomprimir un recurso situado en RAM extra en la dirección de memoria `destination`. `n` es el número de recurso y, obviamente, podemos usar estas macros que crea **The Librarian**.

Si te fijas, las macros equivalen al nombre de archivo origen, en mayúsculas, y con `.` sustituido por `_`. Así nuestro `title.bin` puede referenciarse con la macro `TITLE_BIN`.

El levelset

Es el momento de crear nuestro *levelset*. En modo 128K, como usamos *level bundles*, el *levelset* será mucho más sencillo. Al igual que en modo 48K, se configura en el archivo `my/levelset.h`. En modo 128K la estructura que describe un nivel es mucho más sencilla ya que la mayoría de los parámetros forman parte de la cabecera del *level bundle*:

```
// 128K format:
typedef struct {
    unsigned char resource_id;
    unsigned char music_id;
#ifdef ACTIVATE_SCRIPTING
    unsigned int script_offset;
#endif
} LEVEL;
```

Para cada nivel, por tanto, sólo tendremos que referenciar el recurso con el *level bundle* correspondiente al nivel, el número de la música de fondo en la OGT y, si tenemos activado el scripting, un offset al inicio del script correspondiente al nivel, cosa que, como ya hemos mencionado varias veces en este capítulo, dejaremos para otra ocasión.

Como en modo 48K, el *levelset* se define en el array `levels`. Como no tenemos scripting en **Goku Mal**, nuestro *levelset* es tal que así:

```
// Define your level sequence array here:
// Resource_id, music_id, script_offset
LEVEL levels [] = {
    { LEVEL0C_BIN, 3 },
    { LEVEL1C_BIN, 4 },
    { LEVEL2C_BIN, 5 },
    { LEVEL3C_BIN, 7 },
    { LEVEL4C_BIN, 3 },
};
```

Los números 3, 4, 5, 7, 3 referencian músicas de la OGT que, como hemos dicho también, ya explicaremos como montar.

La estructura level_data

Cuando se descomprime un *level bundle*, los primeros 16 bytes del binario corresponden con una cabecera que está accesible al motor mediante la estructura `level_data`, que vemos en el cuadro siguiente. La mayoría del espacio está desocupado, en provisión de ampliaciones futuras.

```
typedef struct {
    unsigned char map_w, map_h;
```



```

unsigned char scr_ini, ini_x, ini_y;
unsigned char max_objs;
unsigned char enems_life;
unsigned char d01; // Reserved
unsigned char d02;
unsigned char d03;
unsigned char d04;
unsigned char d05;
unsigned char d06;
unsigned char d07;
unsigned char d08;
unsigned char d09;

} LEVELHEADER;

[...]

extern LEVELHEADER level_data [0];
#asm
    ._level_data defs 16
#endasm

```

Accediendo a los datos en `level_data` podremos leer las dimensiones del nivel actual o donde vamos a empezar, el número máximo de objetos o la vida de los enemigos. Como `level_data` es un puntero a una estructura `extern`, tendremos que usar `->` para acceder a los valores de sus campos, por ejemplo:

Configurando el motor

Vamos a ver qué manejos tendríamos que hacer en `config.h` para activar el modo 128K multinivel y un par de engaños al chamán, además de dejar todo el motor listo para **Goku Mal** activando algunas cosas chulas que no hemos visto en el tutorial.

Configuración del modo 128K multinivel

La primera es fácil. Tendremos que tocar tres directivas:

Con esto tenemos el motor configurado en modo 128K y esperando un levelset en la memoria extra. Además estamos diciéndole al manejador principal que hay cinco niveles.

La configuración de Goku Mal

Para no hacer este capítulo enorme, vamos a poner sólo lo que activamos y configuramos, entendiendo que todo lo demás va comentado y desactivado.

```
#define MAP_W          5          //
#define MAP_H          5          // Map dimensions in screens
#define SCR_INICIO     99         // Initial screen
#define PLAYER_INI_X   99         //
#define PLAYER_INI_Y   99         // Initial tile coordinates
//#define SCR_FIN       99         // Last screen. 99 = deactivated.
//#define PLAYER_FIN_X  99         //
//#define PLAYER_FIN_Y  99         // Player tile coordinates to finish game
#define PLAYER_NUM_OBJETOS level_lata->max_objs // Objects to get to finish ga
#define PLAYER_LIFE     6         // Max and starting life gauge.
#define PLAYER_REFILL   1         // Life recharge
#define COMPRESSED_LEVELS // use levels.h instead of mapa.h and enems.h
#define MAX_LEVELS     5          // # of compressed levels
#define REFILL_ME       // If defined, refill player on each level
```

En la configuración principal lo único reseñable es `PLAYER_NUM_OBJETOS`. En modo 128K, el array del *levelset* `levels` es la mínima expresión, como hemos visto, ya que la información de cada nivel va en la cabecera del *level bundle*. Por tanto, para hacer que el número de objetos que hay que conseguir en cada fase coincida con el que hemos configurado al crear los bundles, habrá que *apuntar* `PLAYER_NUM_OBJETOS` al campo `max_objs` de la estructura `level_data`.

```
#define BOUNDING_BOX_8_BOTTOM // 8x8 aligned to bottom center in 16x16
#define SMALL_COLLISION      // 8x8 centered collision instead of 12x12
```

En el apartado de colisiones, usamos las *benévolas*, lo que hará que el gameplay de este juego, que es un *shooter*, sea mucho más mejor y menos peor.

```
#define PLAYER_CHECK_MAP_BOUNDARIES // If defined, you can't exit the map.
#define DEACTIVATE_KEYS             // If defined, keys are not present.
#define FULL_BOUNCE                 // If defined, evil tile bounces equal MAX_VX,
#define PLAYER_FLICKERS             // If defined, collisions make player flicker
```

En el apartado de configuraciones generales configuramos para que se compruebe que no nos salimos del mapa (porque los niveles no tienen "paredes" externas para que haya más espacio para moverse), quitamos las llaves, ponemos rebote completo para con los pinchos y decimos que el jugador parpadée un rato cuando le alcancen.

```
#define ENABLE_FANTIES                // If defined, Fanties are enabled!
#define FANTIES_BASE_CELL              2    // Base sprite cell (0, 1, 2 or 3)
#define FANTIES_SIGHT_DISTANCE        104   // Used in our type 6 enemies.
#define FANTIES_MAX_V                 256   // Flying enemies max speed (also for custom t
#define FANTIES_A                     16    // Flying enemies acceleration.
#define FANTIES_LIFE_GAUGE             5     // Amount of shots needed to kill flying enemi
#define FANTIES_TYPE_HOMING           // Unset for simple fanties.
```

En este juego necesitamos *fanties* de tipo *homing*. De hecho, fueron introducidos en el motor justo en este juego. Los configuramos para que usen siempre el gráfico 2 y les ponemos los valores que se ven. Habrá que soltarles 5 hostias para matarlos.

```
#define PLAYER_CAN_FIRE                // If defined, shooting engine is enabled.
#define PLAYER_BULLET_SPEED           8     // Pixels/frame.
#define MAX_BULLETS                   4     // Max number of bullets on screen. Be careful
#define PLAYER_BULLET_Y_OFFSET        6     // vertical offset from the player's top.
#define PLAYER_BULLET_X_OFFSET        0     // vertical offset from the player's left/right
#define ENEMIES_LIFE_GAUGE             leveldata->enems_life // Amount of shots needed to k
#define RESPAWN_ON_ENTER              // Enemies respawn when entering screen
#define CAN_FIRE_UP                   // If defined, player can fire upwards and dia
```

Aquí vemos otra fullería para adaptar un valor aparentemente fijo a algo variable usando el poder de las macros en C: Si configuramos `ENEMIES_LIFE_GAUGE` como se ve, lograremos que el valor de vida con el que se reactiva a los enemigos a entrar en una pantalla (hemos activado `RESPAWN_ON_ENTER`) coincida con el que se define en la cabecera de cada nivel.

Ojal también al `CAN_FIRE_UP`, que permitirá que disparemos en diagonal.

```
#define TIMER_ENABLE                  // Enable timer
#define TIMER_INITIAL                 99    // For unscripted games, initial value.
#define TIMER_REFILL                  50    // Timer refill, using tile 21 (hotspot #5)
#define TIMER_LAPSE                   40    // # of frames between decrements
#define TIMER_START                   // If defined, start timer from the beginning
#define TIMER_GAMEOVER_0              // If defined, timer = 0 causes "game over"
#define SHOW_TIMER_OVER               // If defined, "TIME OVER" shows when time is
```

En esta sección configuramos el *timer*. Tendrá un valor inicial de 99, empezará con el nivel, se recargará con 50 unidades y cuando se acabe se mostrará "TIME'S UP!" y se acabará el juego.

```
#define PLAYER_HAS_JUMP                // If defined, player is able to jump.
```

Obviamente.

```
#define USE_TWO_BUTTONS
```

```
// Alternate keyboard scheme for two-buttons games
```

Este juego usa dos botones de acción: uno para saltar y otro para disparar. Como ya he comentado, por eso nos pusieron a parir los unsolobotoners, los OPQArS y los QAOPers. Unos porque no querían cambiar las teclas que usaban siempre y otros, con más razón, porque jugar con joystick era muy raro.

No hay solución buena a jugar con joystick, pero al menos la que implementa **MTE MK1** v5 es mejor que nada: cuando seleccionas joystick en un juego con `USE_TWO_BUTTONS` se remapean los controles y *arriba* significa a la vez *arriba* y *botón de salto*. No será tan controlable como teniendo dos botones, pero al menos se podrá jugar.

```
#define VIEWPORT_X      1      //
#define VIEWPORT_Y      2      // Viewport character coordinates
#define LIFE_X          3      //
#define LIFE_Y          0      // Life gauge counter character coordinates
#define OBJECTS_X       99      //
#define OBJECTS_Y       99      // Objects counter character coordinates
#define OBJECTS_ICON_X  99      //
#define OBJECTS_ICON_Y  99      // Objects icon character coordinates (use wit
#define KEYS_X          99      //
#define KEYS_Y          99      // Keys counter character coordinates
#define KILLED_X        99      //
#define KILLED_Y        99      // Kills counter character coordinates
#define AMMO_X          99      //
#define AMMO_Y          99      // Ammo counter character coordinates
#define TIMER_X         29      //
#define TIMER_Y         0      // Timer counter coordinates
```

Sobre la colocación de los elementos no diremos nada, que la tenemos muy vista.

```
#define USE_AUTO_TILE_SHADOWS      // Automatic shadows using specially defined t
#define MASKED_BULLETS            // If needed
#define PAUSE_ABORT                // Add h=PAUSE, y=ABORT
#define GET_X_MORE                 // Shows "get X more" when getting an object
#define HUD_INK                    7  // Use this attribute for digits in the hud
```

Este juego usa sombreado por tiles extra, como habrás podido descubrir si has mirado los tilesets. Queremos máscaras para que las balas se vean bien, necesitamos una pantalla de pausa, y que se muestre `GET x MORE` cuando cojamos objetos.

```
#define PLAYER_MAX_VY_CAYENDO  512  // Max falling speed
#define PLAYER_G                48  // Gravity acceleration
```

```

#define PLAYER_VY_INICIAL_SALTO    96      // Initial jump velocity
#define PLAYER_MAX_VY_SALTANDO    360     // Max jump velocity
#define PLAYER_INCR_SALTO         64      // acceleration while JUMP is pressed

#define PLAYER_INCR_JETPAC        32      // Vertical jetpac gauge
#define PLAYER_MAX_VY_JETPAC     256     // Max vertical jetpac speed

// IV.2. Horizontal (side view) or general (top view) movement.

#define PLAYER_MAX_VX            192     // Max velocity
#define PLAYER_AX                48      // Acceleration
#define PLAYER_RX                32      // Friction

```

Las constantes que definen el movimiento permiten saltos muy altos, resbalones, y no demasiada velocidad horizontal.

¡Y configuración lista!

Valores importantes diferentes en cada nivel

Antes hemos visto la fullería que hemos hecho con `PLAYER_NUM_OBJETOS` para poder controlar el final de cada fase con un número diferente de objetos, que era parecida a la que describimos para el modo 48K en el capítulo anterior, pero levemente diferente. Aprovechamos para resumir aquí cómo se implementarían otros casos en los que querramos tener valores diferentes para cosas importantes en cada nivel.

Número de objetos

Como ya hemos visto, se trata de configurar `PLAYER_NUM_OBJETOS` al valor de la cabecera `level_data->max_objs` :

```

#define PLAYER_NUM_OBJETOS        (level_lata->max_objs)

```

Llegar a un sitio concreto

Se hace exactamente igual que en modo 48K: creando arrays en `my/ci/extra_vars.h` y configurando `SCR_FIN` y opcionalmente también `PLAYER_FIN_X` y `PLAYER_FIN_Y` para que apunten a esos arrays usando la variable `level` .

Vida de los malos

Para que el número de disparos que haya que meterles a los enemigos normales sea diferente para cada fase habrá que hacer que la macro `ENEMIES_LIFE_GAUGE` resuelva al valor correcto de la cabecera:

```
#define ENEMIES_LIFE_GAUGE
```

```
(leveldata->enems_life)
```

Primera compilación

Llegados a este punto nos gusta generar y compilar todo por primera vez para ver que todo está en su sitio. Con el juego en este estado podrás jugar a todas las fases en orden:

```
na_th_an@MOJONIA Z:\MK1\examples\goku_mal\dev
$ build_assets.bat
Converting Fixed Screens
Converting levels
Converting more stuff
Running The Librarian
Making music
DONE
..\bin\RAM1.bin: 14729 bytes
..\bin\RAM3.bin: 16136 bytes
..\bin\RAM4.bin: 16348 bytes
..\bin\RAM6.bin: 2640 bytes
```

```
na_th_an@MOJONIA Z:\MK1\examples\goku_mal\dev
$ compile.bat
Compilando script
Importando GFX
Compilando guego
goku_mal.bin: 26873 bytes
scripts.bin: 0 bytes
Construyendo cinta
```

```
File goku_mal.tap generated successfully
Limpiando
Hecho!
```

Añadidos

Ahora vamos a poner las cosas especiales que hacen tu juego más pulido y molón. Usaremos inyección de código para todo.

Splash screens

Las splash screens son una serie de pantallas que se muestran nada más cargar el juego. Como todo el resto de los tiestos se sacan con `get_resource`. Llamaremos también a `espera_activa` para esperar unos 10 segundos entre cada pantalla (interrumpibles pulsando una tecla).

Si abrimos `assets/librarian.h` veremos que las macros que representan los recursos que necesitamos para `get_resource` son estos, que habrá que descomprimir directamente sobre la

pantalla (dirección 16384):

```
#define LOGO_BIN 17
#define CONTROLS_BIN 18
#define DEDICADO_BIN 15
```

El punto de inyección de código que necesitamos tocar es `my/ci/after_load.h`. Para ahorrar cortapegs vamos a crear un array en `my/ci/extra_vars.h` con la secuencia de recursos:

```
// extra_vars.h
const unsigned char splash_screens [] = { LOGO_BIN, CONTROLS_BIN, DEDICADO_BIN };

// after_load.h

sp_UpdateNow (); // Clear buffer

for (gpit = 0; gpit < 3; ++ gpit) {
    get_resource (splash_screens [gpit], 16384);
    espera_activa (500);
    wyz_play_sound (SFX_START);
    cortina ();
    blackout ();
}
```

Selección de idioma

La selección de idioma sirve para mostrar el texto de las cutscenes en español o inglés. Lo primero que necesitamos es, por tanto, un sitio donde almacenar la selección. Crearemos una variable `lang` en `my/ci/extra_vars.h`:

```
// extra_vars.h
unsigned char lang = 99;
```

El código para seleccionar el idioma lo colocaremos también en `my/ci/after_load.h` justo detrás de las *splash screens*:

```
// after_load.h

_x = 11; _y = 11; _t = 71; _gp_gen = "1. ENGLISH"; print_str ();
_x = 11; _y = 12; _t = 71; _gp_gen = "2. SPANISH"; print_str ();
sp_UpdateNow ();

while (lang == 99) {
    gpjt = sp_GetKey ();
    if (gpjt == '1' || gpjt == '2') lang = gpjt - '1';
}
```

```

}
cortina ();

```

Este código hará que lang valga 0 para inglés o 1 para español.

El menú y los passwords

El juego es capaz de saltar a la fase que sea usando una serie de passwords textuales que no almacenan nada (el caso más sencillo). Vamos a modificar la pantalla de título por defecto para mostrar la opción de jugar o meter un password tras elegir el control. La pantalla de título se puede modificar fácilmente porque está en `my/title_screen.h`. Añadiremos nuestro código tras el que ya existe, que nos vale. Queda así:

```

{
    #ifdef MODE_128K
        get_resource (TITLE_BIN, 16384);
    #else
        #asm
            ld hl, _s_title
            ld de, 16384
            call depack
        #endasm
    #endif

    wyz_play_music (0);

    select_joyfunc ();

    _x = 11; _y = 16; _t = 7; _gp_gen = lang ? "-- MENU =" : "--SELECT=";
    print_str ();

    _x = 11; _y = 17;          _gp_gen = lang ? "1 JUGAR   " : "1 PLAY   ";
    print_str ();

    _x = 11; _y = 18;          _gp_gen =          "2 PASSWORD";
    print_str ();

    sp_UpdateNow ();
    level = 99; while (level == 99) {
        gpjt = sp_GetKey ();
        switch (gpjt) {
            case '1': level = 0; break;
            case '2': level = check_password ();
        }
    }

    wyz_stop_sound ();
    wyz_play_sound (SFX_START);
}

```


Imprimimos unas cosas y luego esperamos que el usuarios pulse 1 o 2. En el caso que pulse 1, se pone `level` a 0 y empezamos. Si pulsa 2, `level` valdrá el resultado de `check_password`, que es una función que tendremos que implementar. ¿Dónde? Pues en el sitio de implementar funciones, que es `my/ci/extra_functions.h`.

La función es muy tonta pero la puedes aprovechar para tus propios passwords con alguna que otra modificación:

```
// extra_vars.h

#define PASSWORD_LENGTH 6
#define MENU_Y          16
#define MENU_X          11
extern unsigned char passwords [0];
#asm
    .passwords
    defm "TETICA"
    defm "PICHON"
    defm "CULETE"
    defm "BUTACA"
#endasm

unsigned char *password = "***** ";

// extra_functions.h

unsigned char check_password (void) {
    wyz_play_sound (SFX_START);

    _x = MENU_X; _y = MENU_Y      ; _t = 7; _gp_gen = " PASSWORD "; print_str ();
    _x = MENU_X; _y = MENU_Y + 1;      ; _gp_gen = "          "; print_str ();
    _x = MENU_X; _y = MENU_Y + 2;      ;                print_str ();

    for (gpit = 0; gpit < PASSWORD_LENGTH; ++ gpit) password [gpit] = '.';
    password [PASSWORD_LENGTH] = ' ';

    gpit = 0;
    sp_WaitForNoKey ();
    _y = MENU_Y + 2; _t = 71; _gp_gen = password;
    while (1) {
        password [gpit] = '*';
        _x = 16 - PASSWORD_LENGTH / 2; print_str ();
        sp_UpdateNow ();

        do {
            gpjt = sp_GetKey ();
        } while (gpjt == 0);

        if (gpjt == 12 && gpit > 0) {
            password [gpit] = gpit == PASSWORD_LENGTH ? ' ' : '.';
            -- gpit;
        }
    }
}
```

```

    } else if (gpjt == 13) break;
    else if (gpjt > 'Z') gpjt -=32;

    if (gpjt >= 'A' && gpjt <= 'Z' && gpit < PASSWORD_LENGTH) {
        password [gpit] = gpjt; ++gpit;
    }

    wyz_play_sound (1);
    sp_WaitForNoKey ();
}

sp_WaitForNoKey ();

// Check password
_gp_gen = passwords;

for (gpit = 0; gpit < MAX_LEVELS - 1; ++ gpit) {
    rda = 1; for (gpjt = 0; gpjt < PASSWORD_LENGTH; ++ gpjt) {
        if (password [gpjt] != *_gp_gen++) rda = 0;
    }

    if (rda) return (1 + gpit);
}

return 0;
}

```

Con esto y tal y como llevamos la cosa, si compilas tendrás el sistema de passwords totalmente funcional y podrás jugar a la fase que quieras poniendo el password correspondiente. Pero aún quedan más cosas que añadir.

Las cutscenes

El sistema de cutscenes que implementaremos en una función `do_cutscene` es muy sencillo. Para cada idioma tendremos 7 cadenas de texto, una por cada imagen. La función recibirá dos números del 1 al 7, y mostrará las imagenes y cadenas de texto entre ambos números, ambos inclusive. También recibirá un tercer parámetro con la música que debe tocar.

Hay tres puntos en los que hay que montar cutscenes: el primero es al empezar un juego nuevo; se mostrarán las cadenas 0 a 3. El segundo es antes de empezar la cuarta fase (`level == 3`); mostraremos la cadena 4. Y el tercero será al terminar la quinta fase; mostraremos las cadenas 5 y 6.

Montados todo este sistema en `my/ci/extra_routines.h` igualmente, a continuación del código que ya habíamos introducido. Puedes verlo en la carpeta del juego.

Lo que sí vamos a ver aquí son los *enganches*, o sea, las llamadas a `do_cutscene` , que habrá que pincharlas en dos puntos diferentes: Si te das cuenta, las dos primeras secuencias se muestran antes de mostrar la pantalla de "NIVEL XX", y la tercera antes de mostrar el final del juego. Por tanto,

meteremos las llamadas a `do_cutscene` al principio de `my/level_screen.h` y al principio de la función `game_ending` en `my/fixed_screens.h`.

```
// level_screen.h
{
    // Show cutscenes at the beginning of levels 0 and 3

    if (level == 0) do_cutscene (0, 3, 1);
    else if (level == 3) do_cutscene (4, 4, 1);

    // Show new level screen (customized)
    [...]
}

// fixed_screens.h
[...]

void game_ending (void) {
    // Show final cutscene
    do_cutscene (5, 6, 9);

    // On to the normal ending
    [...]
}

[...]
```

Las pantallas de "nuevo nivel"

Las pantallas de "nuevo nivel" de Goku Mal muestran una imagen de fondo: `zoneA.png` en la primera fase y `zoneB.png` en las siguientes. A partir de la segunda fase se muestra el password correspondiente. Además, se pinta un número bien gordo usando caracteres de un set gráfico especial que tendremos que descomprimir sobre la zona de tiles. Los números gordos se pintan atendiendo a un array lleno de números que tuve a bien de teclear en tiempos, algo que ni se me ocurriría hacer en los tiempos que corren, porque en el rato que tardo en picar números me hago un conversor automático, que es más divertido. Pero bueno, el mal ya está hecho.

Ponemos nuestras ristras en `my/ci/extra_vars.h` a continuación de todo lo que ya tenemos.

```
// extra_vars.h

unsigned char levelnumbers [] = {
    64, 65, 66, 66, 66, 66, 66, 0, 0,
    67, 68, 66, 66, 66, 66, 66, 0, 0,
    0, 0, 66, 66, 66, 66, 66, 0, 0,
    0, 0, 66, 66, 66, 66, 66, 0, 0,
    0, 0, 66, 66, 66, 66, 66, 0, 0,
    69, 66, 66, 66, 66, 66, 66, 66, 70,
```

```

71, 66, 66, 66, 66, 66, 66, 66, 72,
66, 66, 73, 74, 74, 74, 75, 66, 66,
0, 0, 76, 77, 78, 79, 66, 66, 80,
81, 79, 66, 66, 82, 83, 84, 85, 0,
66, 66, 86, 85, 0, 0, 0, 66, 66,
66, 66, 66, 66, 66, 66, 66, 66,

66, 66, 66, 66, 66, 66, 66, 66,
66, 66, 87, 88, 89, 90, 66, 91, 92,
0, 93, 94, 95, 66, 66, 66, 96, 97,
98, 68, 68, 68, 68, 68, 99, 66, 66,
66, 66,114, 0, 0, 0,115, 66, 66,
100, 66, 66, 66, 66, 66, 66, 66,101,

0, 0,102,103, 66,104,105, 0, 0,
102,103, 66,104,105, 0, 0, 0, 0,
66,104,105, 0,106,106, 0, 0, 0,
66, 66, 66, 66, 66, 66, 66, 66, 66,
66, 66, 66, 66, 66, 66, 66, 66,
0, 0, 0, 0, 66, 66, 0, 0, 0,

66, 66, 66, 66, 66, 66, 66, 66, 66,
66, 66, 0, 0, 0, 0, 0, 0, 0,
66, 66,107,108, 66, 66, 66,109,110,
68, 68,111,112,112,112,113, 66, 66,
66, 66,114, 0, 0, 0,115, 66, 66,
100, 66, 66, 66, 66, 66, 66, 66,101
};

```

El código que saca la pantalla, toca la música de nuevo nivel y saca el numerón lo meteremos sustituyendo el que hay por defecto en `my/level_screen.h` , detrás de las llamadas a `do_cutscene` que pusimos antes. Tiene este aspecto:

```

// Show new level screen (customized)

// Unpack tileset
get_resource (LEVEL_SCREEN_TSC_BIN, (unsigned int) (tileset));

// Show zone screen
sp_UpdateNow ();
blackout ();
get_resource (level ? ZONEB_BIN : ZONEA_BIN, 16384);

// Show password
if (level) {
    _x = 7; _y = 18; _t = 70; _gp_gen = " PASSWORD "; print_str ();
    _gp_gen = passwords + ((level - 1) * PASSWORD_LENGTH);
    gpx = 9; for (gpit = 0; gpit < PASSWORD_LENGTH; ++ gpit) {
        #asm
            ld  h1, (__gp_gen)
            ld  a, (h1)

```

```

        inc hl
        ld  (__gp_gen), hl
        ld  e, a

        ld  d, 70

        ld  a, (__gpx)
        ld  c, a
        inc a
        ld  (__gpx), a

        ld  a, 19

        call SPPrintAtInv
    #endasm
}
}

// Show big number
map_pointer = level * 54 + levelnumbers;
for (gpy = 16; gpy < 22; ++ gpy) {
    for (gpx = 22; gpx < 31; ++ gpx) {
        #asm
            ld  hl, (_map_pointer)
            ld  a, (hl)
            inc hl
            ld  (__map_pointer), hl
            ld  e, a

            ld  d, 71

            ld  a, (__gpx)
            ld  c, a

            ld  a, (__gpy)

            call SPPrintAtInv
        #endasm
    }
}

sp_UpdateNow ();
wyz_play_music (2);
espera_activa (250);
wyz_stop_sound ();

```

Boost al subir de pantalla

Una cosa que suele ayudar mucho al gameplay, sobre todo cuando el juego es más de avanzar que de explorar, es dar un pequeño boost al jugador al subir a la pantalla de arriba. Vamos a detectar que estemos cambiando de pantalla hacia arriba para dar a la VY el máximo valor negativo, lo que hará más fácil dirigir a Goku Mal a una plataforma en lugar de fallar y volver a la pantalla de abajo.

Si recordamos, `my/ci/before_entering_screen.h` se inyecta justo al ir a cambiar pantalla, cuando `n_pant` y `o_pant` son distintas. Justo ahí podremos detectar que `n_pant == (o_pant - level_data->map_w)` y en ese caso aplicar el boost:

```
// before_entering_screen.h

if (n_pant == (o_pant - level_data->map_w)) p_vy = -PLAYER_MAX_VY_SALTANDO;
```

Fin!

Otro capítulo super denso, pero que creo que cubre todo lo que necesitabas saber pero nunca te atreviste a preguntar sobre juegos de 128K multinivel.

Capítulo 14: Sonido 128K

En este capítulo veremos cómo montar una banda sonora u OGT usando WYZ Tracker 1.5.2 de Augusto Ruiz y WYZ Player 4.7 de WYZ.

Introducción y preparativos

Lo primero que necesitamos hacer es descomprimir y preparar WYZ Tracker. Para eso:

1. descomprimiremos el archivo `env/WYZTracker-0.5.0.2.zip` donde estemos instalando las utilidades.
2. Entraremos en el directorio de WYZ Tracker y ejecutaremos `oalinst.exe` para instalar OpenAL. Dale a todo si a todo guay.



3. Ya podremos ejecutar WYZ Tracker mediante su ejecutable `WYZTracker.exe` (¡no tiene pérdida!). Si tienes problemas para ejecutarlo, prueba a ejecutarlo como Administrador. Mi ratio entre problemas y no problemas es 50% en mis equipos, pero de una forma u otra termina funcionando.

Música

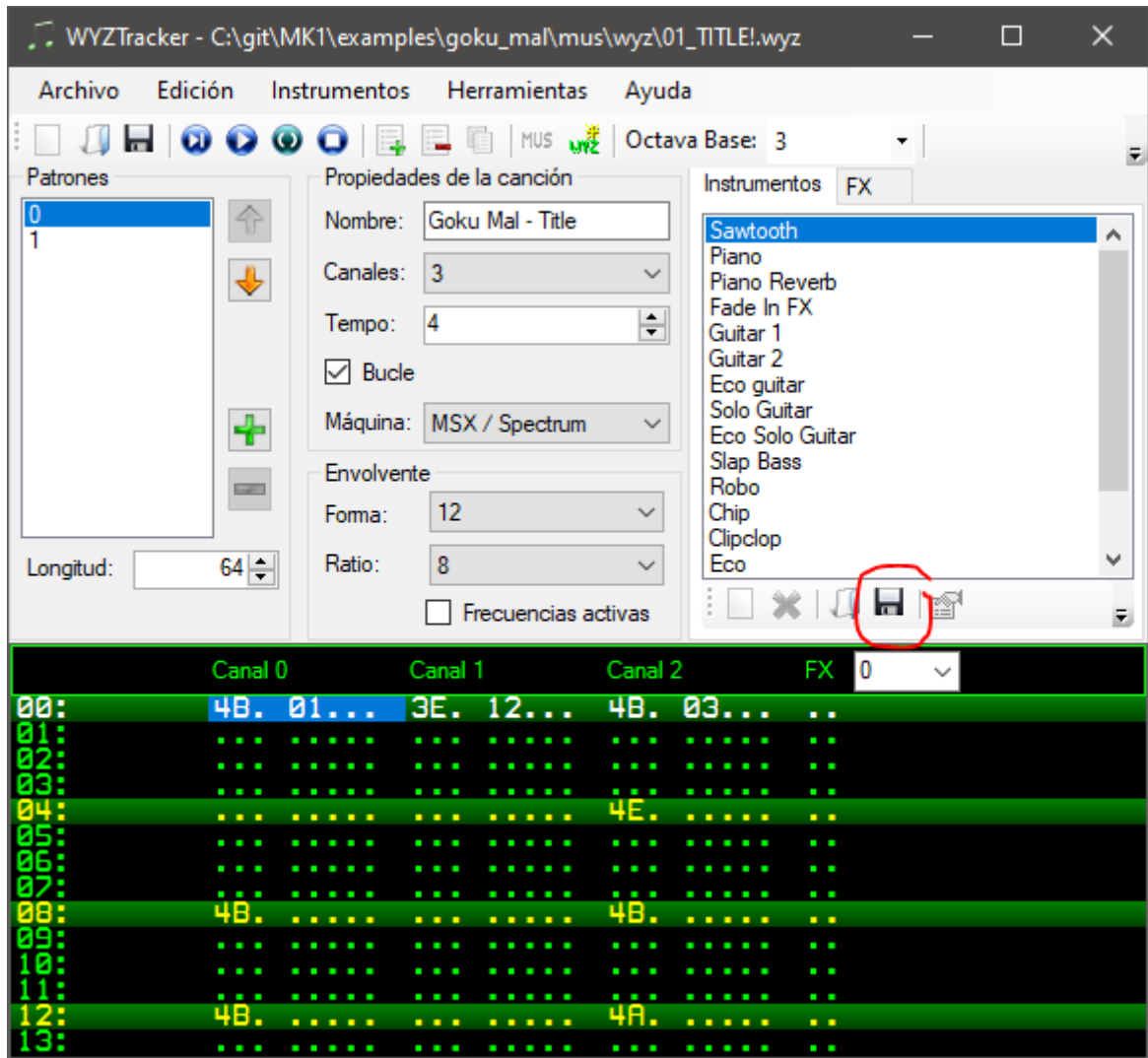
La música de tu juego se compone de canciones que se comprimen de forma individual en formato aplitb y se descomprimen a un buffer antes de tocarlas.

El sistema de sonido emplea RAM1. Tiene 16K para el player, la música comprimida, y el buffer de descompresión. Recuerda que tu binario no debe ocupar más de 16K menos lo que ocupe la canción más grande, o no se podrá descomprimir para tocarla.

Instrumentos y efectos

Las canciones están compuestas por una serie de eventos ordenados en el tiempo: notas musicales o percusiones. Las notas suenan a través de los *instrumentos*, que definen más o menos el timbre (añade muchas comillas, tú ya me entiendes), y la percusión mediante los *efectos*.

Para hacer una OGT para **MTE MK1** la restricción es que todas las canciones tienen que usar el mismo set de instrumentos y el mismo set de efectos. No nos vamos a poner aquí a decir cómo tienes que manejarte haciendo música, pero la técnica suele ser hacer la primera canción e ir creando los instrumentos y efectos según te van haciendo falta, grabarlos, y cuando empieces la siguiente canción cargar los mismos sets, y, si necesitas algún instrumento o efecto más, añadirlo. Trabajar de forma incremental, vaya.



Exportando

Cuando se está trabajando en la OGT usamos el propio formato de WYZ Tracker (archivos `.wyz`) para almacenar las canciones. Sin embargo, y como suele ocurrir, eso no nos sirve directamente.

El formato que toca **WYZ Player** es el formato `mus` . Desde el menú `Archivo` → `Exportar` podremos exportar la canción que esté cargada en el tracker.

Esto generará dos archivos:

1. Archivo `.mus` , con la canción en sí.
2. Archivo `.mus.asm` , con los instrumentos y efectos de la canción.

Cuando tengamos exportadas todas las canciones, podremos descartar todos los archivos `.mus.asm` menos el que esté más completo. Este lo renombraremos como `instrumentos.asm` . Este archivo y todos los `.mus` los colocaremos en el directorio `/mus` junto con el código del player, `WYZproPlay47aZXc.ASM` .

El siguiente paso será comprimir todos los archivos `.mus` . Te recomiendo que te crees un archivo `.bat` para hacer esto de forma automática porque siempre hay algún retoquecillo que te obliga a reconstruirlo todo. Puedes tomar como ejemplo este archivo que he creado para la OGT de **Goku Mal** y que he puesto en `/mus` para ejecutar a mano cuando lo necesite:

```
@echo off
..\..\..\src\utils\apultra.exe 01_TITLE!.mus 01_TITLE!.mus.bin
..\..\..\src\utils\apultra.exe 02_INTRO!!!.mus 02_INTRO!!!.mus.bin
..\..\..\src\utils\apultra.exe 03_ZONE123!.mus 03_ZONE123!.mus.bin
..\..\..\src\utils\apultra.exe 04_FASE1.mus 04_FASE1.mus.bin
..\..\..\src\utils\apultra.exe 05_FASE2.mus 05_FASE2.mus.bin
..\..\..\src\utils\apultra.exe 06_FASE3.mus 06_FASE3.mus.bin
..\..\..\src\utils\apultra.exe 07_ZONE5!.mus 07_ZONE5!.mus.bin
..\..\..\src\utils\apultra.exe 08_FASE5.mus 08_FASE5.mus.bin
..\..\..\src\utils\apultra.exe 10_GAME_OVER!.mus 10_GAME_OVER!.mus.bin
..\..\..\src\utils\apultra.exe 09_ENDING!.mus 09_ENDING!.mus.bin
```

Efectos de sonido

El siguiente paso será crear lo efectos de sonido. Quiero dar las gracias a **GreenWebSevilla** y a **thEpOpE** por sus contribuciones en este apartado: el primero por la creación de un [tutorial](#) con el proceso y el segundo por escribir la herramienta de conversión `WyzFx2Asm.exe` y las modificaciones necesarias a **WYZ Player** para que los efectos puedan usar el canal de ruido.

Aunque obviamente todo es posible y te puedes poner a refinar esto muchísimo más, por defecto **MTE MK1** tiene la siguiente lista de sonidos numerados, como siempre, a partir de cero:

#	Khe
0	Efecto "START"
1	Tile rompiscible golpeado
2	Tile rompiscible destruído

#	Khe
3	Empujar una caja / abrir un cerrojo
4	Disparar
5	Coger un objeto coleccionable
6	Matar a un enemigo
7	Golpear a un enemigo
8	Modo "un sólo objeto", recoger
9	Modo "un sólo objeto", ya tengo
10	Coger una llave
11	Coger cualquier tipo de refill
12	Sartar
13	Pincharse
14	Ser golpeado por enemigo

La lista está compuesta por una serie de macros definidos en el archivo `assets/ay_fx_numbers.h` , por si alguna vez tienes la necesidad de cambiarlo o ampliarlo.

Para cambiar los efectos de sonido tendremos que crear una lista de *efectos* equivalente en **WYZ Player**. El primer efecto se convertirá en el primer efecto de sonido (el 0, "START"), el segundo efecto en el segundo efecto de sonido, etcétera. Puedes encontrar más detalles sobre el tema en [el tutorial de GreenWebSevilla](#) y en el [manual de WYZ Tracker](#)

Una vez que tengamos la lista creada, la grabaremos en formato `.fx` . El siguiente paso es convertirla a código ASM, y para ello emplearemos la herramienta de conversión `WyzFx2Asm.exe` de **thEpOpE** que encontraremos en `/src/utils/` .

Esta herramienta funciona de forma interactiva. Al ejecutarla, nos pedirá que ubiquemos primero el archivo `.fx` de entrada, y seguidamente, que seleccionemos una ubicación para el archivo de salida `efectos.asm` . Elegiremos el directorio `mus/` de nuestro proyecto.

Montando el player

Lo siguiente será montar nuestro playlist en el player. Para eso tenemos que editar el código de **WYZ Player**, o sea, el archivo `WYZproPlay47aZX.ASM` que esta en `mus/` .

Lo primero que hay que hacer es incluir todas las canciones comprimidas de nuestra OGT en orden, usando una etiqueta `SONG_n` , con `n` el número de orden empezando en 0, para cada una.

Encontrarás donde hacerlo porque en el archivo originalmente hay un *stub* `SONG_0` que deberás eliminar para poner tu lista. Por ejemplo, para la OGT de **Goku Mal** nos quedaria así:

```
;; Las canciones tienen que estar comprimidas con aplib
```

```
SONG_0:
    INCBIN "01_TITLE!.mus.bin"
SONG_1:
    INCBIN "02_INTRO!!!.mus.bin"
SONG_2:
    INCBIN "03_ZONE123!.mus.bin"
SONG_3:
    INCBIN "04_FASE1.mus.bin"
SONG_4:
    INCBIN "05_FASE2.mus.bin"
SONG_5:
    INCBIN "06_FASE3.mus.bin"
SONG_6:
    INCBIN "07_ZONE5!.mus.bin"
SONG_7:
    INCBIN "08_FASE5.mus.bin"
SONG_8:
    INCBIN "10_GAME_OVER!.mus.bin"
SONG_9:
    INCBIN "09_ENDING!.mus.bin"
```

Ahora hay que hacer un array con esas etiquetas para que el motor las pueda usar. Un poco más abajo verás la etiqueta `TABLA_SONG`. Ahí, tras el `DW`, deberás referenciar las etiquetas de todas tus canciones. Para **Goku Mal** queda así:

```
;; Añadir entradas para cada canción
```

```
TABLA_SONG:    DW    SONG_0, SONG_1, SONG_2, SONG_3
               DW    SONG_4, SONG_5, SONG_6, SONG_7
               DW    SONG_8, SONG_9
```

Si no hemos alterado la lista de efectos de sonido, no tendremos que tocar nada más. Si sí que lo hemos hecho, habra que modificar la lista etiquetada como `TABLA_EFECTOS` para que aparezcan los efectos en orden. Por defecto sale así:

```
;; Añadir entradas para cada efecto
```

```
TABLA_EFECTOS: DW    EFECT00, EFECT01, EFECT02, EFECT03
               DW    EFECT04, EFECT05, EFECT06, EFECT07
               DW    EFECT08, EFECT09, EFECT010, EFECT011
               DW    EFECT012, EFECT013, EFECT014
```

¡Y listo! Todo debería estar en su sitio. Si no lo está, revisalo todo y hazlo más despacito.

Capítulo 15: Custom Vertical Engine

O *vEng* personalizado. Cómo me gusta inventarme términos y que luego la gente los use. En mojonina nos encanta que digáis "perspectiva genital". Pero ¿qué es un deso? Pues muy sencillo: una forma fácil de introducir tu propio código para controlar el movimiento en el eje vertical.

Programar un nuevo eje vertical suele generar una necesidad: definir una nueva forma de organizar los 8 cells de animación del sprite del jugador para adecuarse al nuevo tipo de movimiento. Por suerte esto también lo tenemos cubierto: es fácil definir tu propia animación para el personaje.

¿Por qué sólo el vertical? Porque suele ser suficiente para hacer de todo. ¿Y por qué no también el horizontal? Bueno, ahí estamos esperando vuestros *pull request*. Recordad que mayormente **MTE MK1** tiene características que hemos necesitado para nuestros juegos o que nos habéis pedido y hemos considerado factibles.

Para añadir tu propio eje vertical deberás poner código en `my/ci/custom_veng.h`.

El pifostio de los movimientos verticales.

A ver, voy a intentar explicar cómo está montado el eje vertical así a modo de culturilla general y así podremos entender mejor el tema.

Como sabrás, desde el día 1 **MTE MK1** soporta juegos en vista lateral y en vista genital. Además de otras historias, la principal diferencia radica en cómo se maneja el eje vertical, ya que el eje horizontal se controla exactamente igual para ambas vistas.

- En vista genital, el eje vertical lo controla el teclado. Si pulsas arriba se aplica aceleración hacia arriba, y si pulsas abajo se aplica aceleración hacia abajo. Si no pulsas nada se aplica fricción en el sentido contrario al movimiento.
- En vista lateral, el eje vertical lo controla la gravedad, de forma que en todo momento se aplica aceleración hacia abajo. Aparte de esto, hay un par de formas de conseguir velocidad negativa (hacia arriba): mediante los diversos tipos de salto o el jetpac.

Desde v5, **MTE MK1** permite varias cosas interesantes:

- Activar varias características del eje vertical *a la vez*. Si, en modo de vista lateral, definimos `VENG_SELECTOR` en `my/config.h`, podremos marcar a la vez varios motores de eje vertical (salto, jetpac, bootee, teclado (como en vista genital)) y elegir cuál está activo en cada momento dándole valores a la variable `veng_selector` (hay macros creadas en `definitions.h` para las selecciones: `VENG_JUMP`, `VENG_JETPAC`, `VENG_BOOTEE` y `VENG_KEYS`, respectivamente).
- Desactivar la gravedad en modo de vista lateral.

- Añadir tu código personalizado para manejar el eje vertical.

Esto permite muchas combinaciones. Lo más básico (que será lo que haremos en este capítulo), es dejar la gravedad tal y como está, desactivar todo lo demás (salto, jetpac, bootee y teclado), y añadir código para gestionar el movimiento del choco. Dejaremos comentado `VENG_SELECTOR` y desactivaremos todas las macros que controlan el movimiento vertical en modo lateral:

```
// my/config.h

// #define PLAYER_HAS_JUMP           // If defined, player is able to jump.
// #define PLAYER_HAS_JETPAC        // If defined, player can thrust a vertical je
// #define PLAYER_BOOTEE            // Always jumping engine.
// #define PLAYER_VKEYS             // Use with VENG_SELECTOR. Advanced.
// #define PLAYER_DISABLE_GRAVITY   // Disable gravity. Advanced.
```

Pero si os dais cuenta, con estas características las posibilidades son infinitas. Por ejemplo sé podría añadir el movimiento de nadar de **Ninjajar** en `my/ci/custom_veng.h`, activar `VENG_SELECTOR`, y cambiar entre salto o nadar con diferentes valores de `PLAYER_G` dependiendo de si estamos o no en el agua.

Otra cosa que me gustaría mencionar una *feature*: desde siempre (desde la versión 3.0 de 2011), si tu juego no usa disparos, puedes activar a la vez (y sin `VENG_SELECTOR`) el jetpac y los saltos normales, ya que el primero emplea la tecla *arriba* y los segundos la tecla *fire*.

El movimiento choco

Desde hace mucho tiempo en Mojonía habíamos querido hacer un juego protagonizado por un choco, Blip Blep, que hiciese cosas chocosas en el espacio, cerca de la galaxia de Choconia, y dijera ¡Blip, Blep!. Para eso el choco tendría que moverse como un choco, por lo que diseñamos este movimiento que describo aquí:

- Gravedad baja, para simular espacio o debajo del agua.
- Velocidad horizontal y aceleración/fricción bajas, para idem, y sobre todo porque la mayor parte del tiempo el choco no está posado en plataformas y queda mejor.
- Cuando pulsas FIRE, el choco se comprime *haciendo fuerza*.
- Al soltar FIRE, el choco sale propulsado hacia arriba con la fuerza que había hecho.

Cuanto más tiempo pulses FIRE, con más fuerza se impulsará el choco al soltarlo, por lo que necesitamos un contador, un valor de incremento, y un valor máximo.

Con esta especificación podemos programar el movimiento básico del chico en sólo unas cuantas líneas de código. Necesitaremos antes definir algunas variables y macros:

```
// extra_vars.h

// Custom vEng

unsigned char fire_pressed;
signed int p_thrust;

#define P_THRUST_ADD    16
#define P_THRUST_MAX    384
```

Usaremos `fire_pressed` como bandera que nos servirá detectar cuando hemos "despulsado" la tecla de disparo. Si ponemos `fire_pressed` a 1 cuando detectamos que está pulsada y a 0 cuando detectamos que no, si vemos que NO se está pulsando la tecla de disparo pero `fire_pressed` vale 1 eso significará que se acaba de dejar de pulsar.

`p_thrust` será donde iremos acumulando "fuerza". Al notar que se libera la tecla de disparo se asignará esta "fuerza" a `p_vy` en negativo (hacia arriba). `P_THRUST_MAX` permite configurar la velocidad con la que se acumula la "fuerza", y `P_THRUST_MAX` el valor máximo que se alcanzará. Los valores de más arriba significa que la velocidad máxima que se transferirá a `p_vy` será de 384, y que esta se alcanzará tras $384/16 = 24$ frames pulsando la tecla de disparo.

Implementémoslo:

```
if ((pad0 & sp_FIRE) == 0) {
    fire_pressed = 1;
    p_thrust -= P_THRUST_ADD;
    if (p_thrust < -P_THRUST_MAX) p_thrust = -P_THRUST_MAX;
    pad0 = 0xff;
} else {
    if (fire_pressed) {
        p_vy = p_thrust;
        p_thrust = 0;
    }
    fire_pressed = 0;
}
```

Como no estamos usando varios ejes verticales excluyentes no necesitamos activar `VENG_SELECTOR` ni emplear la variable `veng_selector`.

La animación choco

Todo esto no queda nada bien si no cambiamos la animación. Hemos hecho este *spriteset* para el choco con 8 cells de animación:



Los dos primeros muestran al choco en su animación "idle", cuando se está pulsando nada ni ascendiendo (sólo dejando que la gravedad y la inercia muevan al choco).

Los cuatro siguientes muestran al choco desplazándose hacia la izquierda y hacia la derecha. Hay dos cells de animación para cada dirección. Los iremos alternando si el choco "nada" hacia esa dirección. Si el choco está subiendo tras "lanzarse" (lo de pulsar la tecla de disparo para "hacer fuerza") no se animarán y se usará sólo el primer cell.

El siguiente cell muestra al choco "haciendo fuerza", y el último al choco ascendiendo verticalmente tras soltar el botón de disparo.

Habrà otras implementaciones posibles, pero esta me parece muy legible. Me gusta precalcular el número de frame en una variable temporal (rda aquí) y luego emplear el array `player_cells` que contiene 8 punteros a los 8 cells:

```
// custom-animation.h

if (fire_pressed) {
    rda = 6;
} else {
    if (gpx == gpox) {
        if (p_vy < 0) rda = 7;
        else rda = (maincounter >> 4) & 1;
    } else {
        rda = 4 - (p_facing << 1);
        if (p_vy >= 0) rda += ((gpx >> 3) & 1);
    }
}

p_next_frame = player_cells [rda];
```

No hay que olvidarse de activar `PLAYER_CUSTOM_ANIMATION` en `my/config.h`.

Más cosas para el choco

Queremos que el choco pueda romper bloques rompiscibles cuando se haya lanzado hacia arriba con la suficiente fuerza. Para eso tendremos que ampliar nuestro código en `my/ci/custom_veng.h`. Detectaremos la colisión en el punto central de la fila superior del cuadro de 16x16 que contiene al sprite. Como esto sólo va a detectarse con el choco a determinada velocidad negativa, en este píxel hay cabeza de choco porque el choco estará estirado y quedará bien.

Pensando en que (misteriosamente) queramos reutilizar este movimiento en otro juego con otras características, vamos a hacerlo bien y usar `#ifdef` como mandan los cánones. Añadimos:

```
// custom_veng.h

[...]
```

```

#ifdef BREAKABLE_WALLS
    // Detect head
    if (p_vy < -P_BREAK_VELOCITY_OFFSET) {
        cx1 = ptx1; cx2 = ptx2;
        cy1 = cy2 = (gpy - 1) >> 4;
        cm_two_points ();

        if (at1 & 16) {
            _x = cx1; _y = cy1;
            break_wall ();
            p_vy = P_BREAK_VELOCITY_OFFSET;
        }

        if (cx1 != cx2 && (at2 & 16)) {
            _x = cx2; _y = cy2;
            break_wall ();
            p_vy = P_BREAK_VELOCITY_OFFSET;
        }
    }
#endif

```

`P_BREAK_VELOCITY_OFFSET` es el valor mínimo de velocidad que hay que llevar para romper un piedra, y lo definimos con las demás macros en `my/ci/extra_vars.h`:

```

// extra_vars.h

#define P_BREAK_VELOCITY_OFFSET 128

```

Fricción del agua

El problema de esto es que nos hemos puesto en la tesitura de que, al rebotar hacia abajo, `p_vy` puede valer más que `PLAYER_MAX_VY_CAYENDO` y esto no quedaría bien, ya que la implementación de la gravedad limita la velocidad hacia abajo a este valor. Lo que vamos a hacer es desactivar la gravedad y usar nuestra propia implementación.

Primero descomentamos `PLAYER_DISABLE_GRAVITY` en `my/config.h` y luego añadimos:

```

// custom_veng.h

[...]

// Water gravity & friction
if (p_vy < PLAYER_MAX_VY_CAYENDO) {
    p_vy += PLAYER_G;
    if (p_vy > PLAYER_MAX_VY_CAYENDO) p_vy = PLAYER_MAX_VY_CAYENDO;
} else {
    p_vy -= P_WATER_FRICTION;
}

```



```

    if (p_vy < PLAYER_MAX_VY_CAYENDO) p_vy = PLAYER_MAX_VY_CAYENDO;
}

```

La macro que nos queda, `P_WATER_FRICTION`, la definimos igualmente en `my/ci/extra_vars.h`:

```

// extra_vars.h

#define P_WATER_FRICTION          16

```

Otro ejemplo

Aquí tenéis un motor de movimiento parecido a subaquatic (que no era **MTE MK1**, ni se le parecía), basado en este spriteset:



```

// config.h
//#define PLAYER_HAS_JUMP                // If defined, player is able to jump.
//#define PLAYER_HAS_JETPAC              // If defined, player can thrust a vertical je
//#define PLAYER_BOOTEER                 // Always jumping engine. Don't forget to disa
//#define PLAYER_VKEYS                   // Use with VENG_SELECTOR. Advanced.
#define PLAYER_DISABLE_GRAVITY           // Disable gravity. Advanced.

[...]

#define PLAYER_MAX_VX                    128    // Max velocity
#define PLAYER_AX                        32      // Acceleration
#define PLAYER_RX                        64      // Friction

```

```

// extra_vars.h

#define P_MAX_VY_FLOATING                128
#define P_AY_FLOATING                    16
#define P_MAX_VY_DIVING                  128
#define P_AY_DIVING                      32

// custom_veng.h

if ((pad0 & sp_DOWN) == 0) {
    // Dive
    p_vy += P_AY_DIVING;
    if (p_vy > P_MAX_VY_DIVING) p_vy = P_MAX_VY_DIVING;
} else {
    // Float

```

```
p_vy -= P_AY_FLOATING;  
if (p_vy < -P_MAX_VY_FLOATING) p_vy = -P_MAX_VY_FLOATING;  
}
```

```
// custom_animation.h
```

```
rda = p_facing ? 0 : 4  
if (p_vy > 0 || p_vx) rda += (maincounter >> 2) & 1;
```

Y esto es todo.

Ya has aprendido (o no) cómo añadir tu propio manejador del eje vertical del movimiento, que además rima con pimienta.

API.md

Eventualmente. Dame un respiro D-: Esto va a ser un WIP largo. Dejarse venir con los PDF }:-D

Antes de empezar

¿Cuánto puede ocupar mi güego? Pues en 48K la pila se coloca bajando desde 61697. No suele ocupar mucho, y $61697 - 24000 = 37697$. Pongamos que **37500 bytes** es una buena marca.

Variables globales

Aquí las variables globales que te pueden interesar:

Estado

Estas variables controlan el estado del juego. En qué nivel estamos, en qué pantalla, etcétera.

- `pad0` : entrada del joystick / teclado. Consultar la documentación de `splib2` para ver cómo leer la información del teclado (contiene el resultado devuelto por `sp_JoyKeyboard`, `sp_joyKempston` o `sp_JoySinclair1`).
- `flags [MAX_FLAGS]` : Banderas que pueden usarse desde el scripting o desde inyección de código. Son valores que pueden valer de 0 a 127 (en scripting) o de 0 a 255 (en inyección de código).
- `level` : Número de nivel actual.
- `warp_to_level` : **Si no está activo el scripting**, podemos poner a 1 esta variable mediante inyección de código, establecer `level`, `n_pant`, `p_x`, `p_y`, `gpx` y `gpy` y poner `playing` a 0 para saltar a cualquier punto de otro nivel.
- `n_pant` : Número de la pantalla actual.
- `o_pant` : Copia del número de la pantalla actual. Se utiliza entre otras cosas para detectar el cambio: si `n_pant != o_pant` hay que cambiar de pantalla. Cada vez que se cambia de pantalla se vuelve a hacer `o_pant = n_pant`.
- `x_pant`, `y_pant` : coordenadas de la pantalla dentro del mapa, sólo si `PLAYER_CHECK_MAP_BOUNDARIES`.
- `is_rendering` : está a 1 mientras se está entrando en una nueva pantalla. Se utiliza por las funciones de `printer.h` para saber que no deben invalidar los tiles que dibujen ya que se invalidarán todos de un plumazo al terminar (ahorrando tiempo).
- `maincounter` se va incrementando con cada frame. Como es de tipo `unsigned char` cuando llegue a 255 volverá a 0.

Generales

- `map_attr [150]` buffer con los comportamientos de todos los tiles de la pantalla actual. Para direccionar este array puede usarse le macro `COORDS(x,y)` que admite coordenadas a nivel de tiles.
- `map_buff [150]` buffer con los valores de todos los tiles de la pantalla actual. Para direccionar este array puede usarse le macro `COORDS(x,y)` que admite coordenadas a nivel de tiles.
- `brk_buff [150]` 'vida' de cada tile de la pantalla actual si es rompible (si activamos `BREAKABLE_WALLS`).

Player

- `p_x` , `p_y` : coordenadas del jugador en punto fijo 10.6, 1/64 de píxel.
- `gpx` , `gpy` : coordenadas del jugador en píxels (`gp? = p_? / 64`).
- `p_tx` , `p_ty` : coordenadas de la casilla (coordenadas de tile) sobre la que está el punto central del sprite del jugador.
- `p_vx` , `p_vy` : velocidad del jugador en cada eje, en 1/64 de píxel por frame.
- `ptgmx` , `ptgmy` : velocidad a la que una plataforma móvil está desplazando al jugador (si `p_gotten` vale 1).
- `p_saltando` : el jugador está saltando.
- `p_cont_salto` : contador de frames de salto.
- `p_jetpac_on` : el jetpac está propulsando (estamos pulsando *arriba*).
- `p_facing` : orientación del jugador. En vista lateral, valdrá 0 si mira a la derecha y 1 si mira a la izquierda. En vista genital, tomará un valor `FACING_RIGHT` , `FACING_LEFT` , `FACING_UP` y `FACING_DOWN`
- `p_estado` , `p_ct_estado` : Estados alterables (por ahora, `EST_NORMAL` o ' `EST_PARP` (parpadeando)).
- `possee` : en vista lateral, el jugador está sobre una plataforma fija.
- `p_gotten` : en vista lateral, el jugador está moviéndose con una plataforma móvil.
- `p_life` : vida del jugador.
- `p_objs` : cuenta de items coleccionables recogidos.
- `p_keys` : cuenta de llaves que tiene el jugador.
- `p_fuel` : cantidad de combustible para el jet pac.
- `p_killed` : número de enemigos eliminados por el jugador.
- `p_ammo` : munición disponible.
- `p_disparando` : el jugador pulsó el botón de disparo y aún no lo soltó.
- `p_facing_v` y `p_facing_h` : se emplean en la vista genital para almacenar valores temporales que luego se resuelven en `p_facing` dependiendo de si está o no definida `TOP_OVER_SIDE` .
- `p_killme` : el jugador fue alcanzado por un enemigo o colisionó con un tile que mata y debe morir este frame (se consume en el bucle principal a cada vuelta).
- `p_kill_amt` : cuánta vida se restará cuando `p_killme` se pone a cierto. Se resetea a 1 al principio de cada frame.

Enemies

Hay un array principal: `malotes` viene del archivo `.ene` convertido y es una estructura en RAM conteniendo valores de todos los enemigos de forma que estos son persistentes entre pantallas; los arrays `en_an_*` contienen toda la información de los tres enemigos que hay en la pantalla actual:

- `enoffs` : Offset dentro del array `malotes` de los enemigos de la pantalla actual (vale `n_pant * 3`).
- `malotes` : Información general (persistente) de todos los enemigos del nivel actual. Se trata de un array de structs `MALOTE` con esta definición:

```
typedef struct {
    unsigned char x, y;
    unsigned char x1, y1, x2, y2;
    char mx, my;
    char t;
#ifdef PLAYER_CAN_FIRE
    unsigned char life;
#endif
} MALOTE;
```

- `enit` : Se utiliza siempre para iterar enemigos. En los puntos de inyección de código que se incluyen dentro de bucles de enemigos (`enems_load.h`, `enems_move.h`, `on_enems_killed`), siempre apunta al enemigo actual que se está procesando.
- `en_an_base_frame [3]` : Frame base (0, 2, 4 o 6) de los enemigos. Sirven para construir índices de la tabla `enem_cells`, que contiene punteros a los 8 gráficos de los enemigos:

```
const unsigned char *enem_cells [] = {
    sprite_9_a, sprite_10_a, sprite_11_a, sprite_12_a,
    sprite_13_a, sprite_14_a, sprite_15_a, sprite_16_a
};
```

- `en_an_frame [3]`, `en_an_count` se utilizan para animar a los enemigos normales. Cada 4 frames (contadas por `en_an_count`), el valor de cada `en_an_frame` alterna entre 0 y 1. `en_an_count` También se utiliza como contador de frames mientras se muestra la explosión en modo 128K.
- `en_an_next_frame [3]` : contiene un puntero al gráfico con el que se actualizará el sprite. Los sprites de los enemigos se animan cambiando estos valores.
- `en_an_current_frame [3]` : contiene un puntero al gráfico que muestra el sprite. Se emplea internamente junto con `en_an_next_frame` pues las funciones de `splib2` necesitan un desplazamiento en cada cuadro de juego para saber si cambiar el gráfico.
- `en_an_state [3]` : estado actual de cada enemigo. Se emplea para señalar que un sprite está en su animación de morir (en modo 128K) o para contener el estado actual de los enemigos tipo *fanties* si se han configurado de tipo *homing*.

Fanties

- `en_an_x [3]` , `en_an_y [3]` contienen coordenadas de punto fijo (1/64 de pixel) de los enemigos tipo fanties.
- `en_an_vx [3]` , `en_an_vy [3]` contienen velocidades en cada eje en 1/64 de píxel por frame de los enemigos tipo fanties.

EIJs (perseguidores)

- `en_an_alive [3]` el enemigo está "muerto" (sin aparecer, valor 0), "apareciendo" (valor 1), o "vivo" (ha aparecido, está moviéndose y es matable, valor 2).
- `en_an_dead_row [3]` cuenta de frames que el enemigo debe permanecer en el estado 0.
- `en_an_rawv [3]` es el valor absoluto de la velocidad con la que se mueve el sprite y valdrá 1, 2 o 4.

Copias temporales

Para ahorrar tiempo y memoria (y hacer que el posible paso a ensamble se más sencillo) a cada frame y para cada enemigo se copian temporalmente los valores de `malotes` en el siguiente conjunto de variables temporales (nótese que hay una por cada componente del struct). En los puntos de inyección de código `enems_move.h` y `on_enems_killed` contendrán los valores del enemigo actual y **pueden modificarse**, ya que al final de cada vuelta del bucle de actualización se actualiza `malotes` con los nuevos valores.

- `_en_x` , `_en_y` : Coordenadas (en pixels) del enemigo actual.
- `_en_x1` , `_en_x2` , `_en_y1` , `_en_y2` : Límites de trayectoria del enemigo actual. `_en_x1` , `_en_y1` se utilizan además en los fanties y en los EIJs para señalar el punto de origen donde fueron definidos los enemigos en Ponedor. Puedes reutilizar estas variables en tus tipos custom de enemigos como quieras.
- `_en_mx` , `_en_my` : Píxels en cada eje que deberá moverse el enemigo en el frame actual.
- `_en_t` : Tipo del enemigo.
- `_en_life` : Vida del enemigo.

Balas

- `bullets_estado [MAX_BULLETS]` vale 1 si la bala está activa.
- `bullets_x [MAX_BULLETS]` y `bullets_y [MAX_BULLETS]` contienen las coordenadas en pixels de la bala.
- `bullets_mx [MAX_BULLETS]` y `bullets_my [MAX_BULLETS]` contienen cuántos píxels debe moverse cada bala en cada eje cada frame mientras esté activa.
- `bullets_life [MAX_BULLETS]` "duración" de la bala, que se decrementa cada frame hasta llegar a 0, momento en el que la bala desaparece, si se activa `LIMITED_BULLETS` .

Hotspots

Los hotspots del nivel actual se almacenan en un array `hotspots` de estructuras `HOTSPOT` que tiene estos miembros:

```
typedef struct {
    unsigned char xy, tipo, act;
} HOTSPOT;
```

Si el hotspot de la pantalla actual está activo (no se ha recogido y es distinto de 0), entonces:

- `hotspot_x`, `hotspot_y` coordenadas (en pixels) del hotspot. El jugador colisionará con el cuadrado que va desde este punto hasta (`hotspot_x + 15`, `hotspot_y + 15`).
- `orig_tile` es el tile que había originalmente en lugar del hotspot.

Timer

- `timer_on` - El temporizador está activo.
- `timer_t` - Valor del temporizador.
- `timer_frames` - El valor de `timer_t` se decrementa cada `timer_frames` cuadros de juego.
- `timer_count` - Contador de cuadros. Cuando llega a `timer_frames` se decrementa `timer_t`.
- `timer_zero` - Se pone a 1 cada vez que `timer_t` llega a 0.

Scripting

- `f_zone_ac` vale 1 si `ENABLE_FIRE_ZONE` está activo y hemos definido desde el script una *fire zone*.
- `fzx1`, `fzy1`, `fzx2`, `fzy2` son los límites del rectángulo definido como *fire zone*.

Miscelánea

- `pushed_any` se activa al empujar con el botón de disparo si `FIRE_TO_PUSH` y sirve como flag interno para no soltar un disparo.
- `rdx`, `rdy`, `rda`, `rdb`, `rdc`, `rdd`, `rdn` son variables para hacer tiestos temporales.
- `gen_pt` puntero de propósito general
- `_x`, `_y`, `_n`, `_t`, `_gp_gen` se emplean como "pseudoparámetros" para muchas funciones de la API.
- `cx1`, `cy1`, `cx2`, `cy2`, `at1`, `at2` son "pseudoparámetros" de entrada y salida para funciones de colisión.
- `x0`, `y0`, `x1`, `y1` se emplean como "pseudoparámetros" en funciones de mover y destruir tiles.
- `ptx1`, `pty1`, `ptx2`, `pty2` se utilizan para definir el *bounding box* para la colisión del jugador con el escenario.

Cells de animación

- `player_cells []` es un array que contiene 8 punteros a los 8 cells de animación para el jugador. Puedes usarlo si defines `PLAYER_CUSTOM_ANIMATION` e implementas tu propio sistema de animación.
- `enem_cells []` es un array que contiene 8 punteros a los 8 cells de animación para los enemigos.

Funciones

Esta referencia no cubre todas y cada una de las funciones del motor, ya que hay mucho de tramolla interna que poco uso va a darte a la hora de escribir código custom en los puntos de inyección de código. Estas son, pues, las **funciones interesantes**, divididas en módulos.

Puedes ver una lista de todas las funciones de **MTE MK1** en el archivo `dev/prototypes.h`.

Hay muchas funciones que no reciben parámetros pero sí *pseudoparámetros* en forma de valores a variables globales generales. Estos suelen ser las variables `_x`, `_y`, `_n`, `_t`, y `_gp_gen`.

Printer

`void draw_coloured_tile (void)`

Dibuja un tile del tileset **sin invalidarlo** en cualquier punto de la pantalla (incluso fuera del área de juego o *viewport*). La función espera estos *pseudoparámetros*:

- `_x`, `_y` : coordenadas de **caracter** (`x = 0-30`; `y = 0-22`) donde imprimir el tile.
- `_t` : Número de tile (0-47).

Para que el tile se actualizado en la próxima llamada a `sp_UpdateNow` o `sp_UpdateNowEx` habrá que llamar a la siguiente función, `invalidate_tile`, sobre las mismas coordenadas.

`void invalidate_tile (void)`

Invalida un área del buffer de 2x2 caracteres. La función espera estos *pseudoparámetros*:

- `_x`, `_y` : coordenadas de **caracter** (`x = 0-30`; `y = 0-22`) donde imprimir el tile.

`void invalidate_viewport (void)`

Invalida todo el área de juego (30x20 caracteres a partir de (`VIEWPORT_X`, `VIEWPORT_Y`)). Los cambios serán visibles en la próxima llamada a `sp_UpdateNow` o `sp_UpdateNowEx`.

`draw_coloured_tile_gamearea (void)`

Dibuja un tile del tileset en la rejilla del área de juego **sin invalidarlo**. Generalmente no llamaremos a esta función directamente. La función espera estos *pseudoparámetros*:

- `_x` , `_y` : coordenadas de **rejilla de tile** ($x = 0-14$; $y = 0-9$) donde imprimir el tile.
- `_t` : Número de tile (0-47).

draw_invalidate_coloured_tile_gamearea (void)

Dibuja un tile del tileset en la rejilla del área de juego y lo invalida (internamente hace una llamada a `draw_coloured_tile_gamearea` y otra a `invalidate_tile`). La función espera estos *pseudoparámetros*:

- `_x` , `_y` : coordenadas de **rejilla de tile** ($x = 0-14$; $y = 0-9$) donde imprimir el tile.
- `_t` : Número de tile (0-47).

update_tile (void)

Dibuja un tile del tileset en la rejilla del área de juego, lo invalida, y actualiza los buffers `map_buff` y `map_attr`, haciendo el tile interactuable (internamente hace una llamada a `draw_invalidate_coloured_tile_gamearea` y posteriormente modifica los buffers). La función espera estos *pseudoparámetros*:

- `_x` , `_y` : coordenadas de **rejilla de tile** ($x = 0-14$; $y = 0-9$) donde imprimir el tile.
- `_t` : Número de tile (0-47).
- `_n` : Valor para `beh`, que puede ser `behs [_t]` o cualquier otro valor.

void draw_decorations (void)

Sirve para pintar ristas de tiles (decoraciones). s. `draw_decorations` espera que `_gp_gen` apunte a una colección de decoraciones terminada en `0xff`. Podemos crear arrays con decoraciones en `my/ci/extra_vars.h`.

void print_number2 (void)

Imprime un número decimal de dos cifras en el color `HUD_INK` (definido en `my/config.h`). Internamente se utiliza para los marcadores. La función espera estos *pseudoparámetros*:

- `_x` , `_y` : coordenadas de **caracter** ($x = 0-30$; $y = 0-22$) donde imprimir el valor.
- `_t` : Valor (0-99).

void print_str (void)

Imprime el contenido de una cadena o cualquier buffer terminado en 0 en pantalla. La función espera estos *pseudoparámetros*:

- `_x` , `_y` : coordenadas de **caracter** ($x = 0-30$; $y = 0-22$) donde imprimir la cadena.
- `_t` : atributo (color).
- `_gp_gen` : debe apuntar al inicio de la cadena o del buffer terminado en 0.

Recordemos que podemos hacer algo así en C y que `z88dk` el quisquilloso se lo come:

```
_x = 2; _y = 2; _t = 6;  
_gp_gen = "OLA K ASE";  
print_str ();
```

void blackout_area (void)

Pinta de negro el área de juego.

void clear_sprites (void)

Saca los sprites de la pantalla. El resultado será visible en el próximo `sp_UpdateNow ()`.

Contenido de los buffers y colisión

unsigned char attr (char x, char y)

Devuelve el comportamiento del tile situado en las coordenadas (`x` , `y`) de la rejilla de la pantalla actual. Equivale a `map_attr [COORDS (x,y)]`.

unsigned char qtile (unsigned char x, unsigned char y)

Devuelve el número del tile situado en las coordenadas (`x` , `y`) de la rejilla de la pantalla actual. Equivale a `map_buff [COORDS (x,y)]`.

unsigned char collide (void)

Sirve para saber si el player colisiona con un ente de 16x16 pixels situado en las coordenadas (`cx2` , `cy2`). Si se define `SMALL_COLLISION` la caja de colisión es de 8x8. Si no, es de 13x13. Devuelve `1` si hay colisión.

unsigned char cm_two_points (void)

Se emplea para hacer colisiones por caja cuando un objeto avanza en cierta dirección. Devuelve los comportamientos de los tiles que tocan los puntos (`cx1` , `cy1`) y (`cx2` , `cy2`) en las variables `at1` y `at2` , respectivamente.

Enemigos

void enems_draw_current (void)

Actualiza el sprite del enemigo `enit` a la posición `_en_x` , `_en_y` y cambia su gráfico si `en_an_current_frame` ha cambiado.

void enems_kill (void)

Mata al enemigo `enit` (levanta el bit 4 de `_en_t`). Nótese que si se llama desde fuera el bucle principal de enemigos habrá que dar un valor correcto a `_en_t` antes de llamar y actualizar el array `malotes` después:

```
_en_t = malotes [enoffs + enit].t;
enems_kill ();
malotes [enoffs + enit].t = _en_t;
```

Jugador

```
void player_deplete (void)
```

Resta a la vida del jugador `p_kill_amt` unidades que, si no lo hemos modificado durante el frame actual, valdrá 1.

Sonido (48K)

```
void beep_fx (unsigned char n)
```

Toca el efecto de sonido `n` (0-9). Esta es la tabla de sonidos:

#	Cosa
0	Enemigo / tile destruido
1	Enemigo / tile golpeado
2	Empujar bloque
3	Salto
4	Jugador golpeado
5	Enemigo pisado
6	Disparo
7	Coger llave / Recarga de tiempo
8	Abrir cerrojo / Coger refill
9	Coger objeto / Munición

Sonido (128K)

```
void wyz_play_sound (unsigned char fx_number)
```

```
void wyz_play_music (unsigned char song_number)
```

```
void wyz_stop_sound (void)
```

Miscelanea

```
unsigned char rand (void)
```

Devuelve un número pseudo-aleatorio entre 0 y 255: `rda = rand ();` .

```
unsigned int abs (int n)
```

Devuelve el valor absoluto de un número.

```
void step (void)
```

Hace un sonidito minimal de "paso".

```
void cortina (void)
```

Efecto de borrado de pantalla bonito.

Compresión

```
void get_resource (unsigned char n, unsigned int destination);
```

Está disponible en modo 128K y sirve para descomprimir un recurso situado en RAM extra en la dirección de memoria `destination` . `n` es el número de recurso y, obviamente, podemos usar las constantes que crea **The Librarian**.

```
void unpack (unsigned int from, unsigned int to);
```

Está disponible en modo 48K cuando se activa `COMPRESSED_LEVELS` . Sirve para descomprimir desde la dirección `from` a la dirección `to` . Pueden usarse punteros si les haces un cast a `(unsigned int)` .

Bajo nivel y splib2

Iré apuntando aquí cosas de `splib2` que piense que pueden ser interesantes a la hora de escribir tu propio código en los puntos de inyección. Generalmente usarás sólo la API del motor, pero es posible que haya que afinar en un momento dado.

Descripción muy general de cómo funciona splib2

O al menos la parte gráfica. Y muy a grosso modo.

splib2 divide la pantalla en celdas de carácter, esto es, en una rejilla de 32x24 celdas. En el totete de **splib2** cada celda tiene asignadas cuatro cosas: un número de carácter o *patrón* de fondo, un atributo para ese *patrón*, un puntero a una lista de sprites que lo pisan (que puede ser nulo), y un bit que indica si es *válido*. Todas las operaciones se hacen sobre estas estructuras de datos. Para mostrar los

resultados llamamos a una función que actualiza la pantalla **enviando a la misma únicamente las celdas de la rejilla que hayan sido *invalidadas*** usando precisamente el bit que hemos nombrado antes.

Move un sprite por la pantalla, por ejemplo, invalida celdas: las que pisaba el sprite antes de moverse, y las que pisa después. De esa forma, en la siguiente actualización, se redibujarán todas esas celdas y parecerá que el sprite se ha movido sin borrar el fondo.

A la hora de pintar tiles del motor, para no tener que hacer cálculos para cada patrón del tile, primero se modifican las estructuras y luego se invalida el cuadro de un plumazo. Es más, a la hora de pintar una nueva pantalla se escriben todos los *patrones* y atributos en las estructuras de **splib2** y luego se invalida toda la pantalla de una vez, ganando mucha velocidad. Es por eso por lo que en la lista de funciones has visto muchas en las que se indica que *no se invalida* y hay que llamar a una función de invalidación si se quiere ver los cambios reflejados en pantalla.

splib2 está escrita en ensamble y es bastante potente, pero para que sea fácil de usar desde un programa C viene con una interfaz C que a veces mete bastante overhead (por ejemplo en `sp_moveSprAbs` y otras funciones que tienen muchos parámetros) por lo que para, ganar ciclos y espacio, pasamos de la interfaz C en **MTE MK1** y usamos directamente las rutinas en ensamble. Esto implica siempre preparar los parámetros a mano y meterlos en registros desde ensamble en línea.

Imprimir un caracter

Para imprimir un carácter se empleaba la función `sp_PrintAtInv`, pero se ha tratado de prescindir de la interfaz C de **splib2** para las funciones más llamadas, por lo que para imprimir un carácter o *patrón* tenemos que usar directamente las tripas de *splib2*.

Necesitamos invalidar el carácter impreso

Para ello llamaremos a la rutina `spPrintAtInv` que recibe los siguientes parámetros en registros:

Reg.	Parámetro
A	Fila (0..23)
C	Columna (0..31)
D	Atributo
E	Nº de <i>patrón</i>

Por ejemplo, para imprimir una A color azul sobre cyan en la posición 7, 8:

```
#asm
ld  a, 8
ld  c, 7
ld  d, 5*8+1
ld  e, 33
```

```
call spPrintAtInv
#endasm
```

Vamos a imprimir muchos caracteres juntos

En ese caso lo mejor es atacar directamente a las estructuras de datos de **splib2** y luego invalidar. La estructura de datos que contiene caracteres, atributos y punteros a listas de sprites es una tabla de 32x24 celdas de 4 bytes cada una. Para imprimir un carácter en la tabla habrá que moverse a la celda correcta y escribir primero el atributo y luego el número de patrón, dejando sin tocar los otros dos bytes.

Para encontrar la posición donde tenemos que escribir dadas unas coordenadas podemos llamar a la rutina `SPCompDListAddr`, que recibe estos parámetros:

Reg.	Parámetro
A	Fila (0..23)
C	Columna (0..31)

Que devolverá la dirección de la celda en `HL`. Una vez obtenida, podemos iterar y movernos por el buffer incrementando o decrementando el puntero de forma correcta. Puedes ver las funciones en `printer.h` para ver cómo se usa.

Una vez modificado el buffer no tenemos más que invalidar el rectángulo. Para eso utilizamos la rutina `spInvalidate`, que recibe los siguientes parámetros:

Reg.	Parámetro
B	Fila esquina superior izquierda (0..23)
C	Columna esquina superior izquierda (0..31)
D	Fila esquina inferior derecha (0..23)
E	Columna esquina inferior derecha (0..31)
IY	Puntero al rectángulo de clipping

Para `IY` tenemos definido dos rectángulos: uno a toda la pantalla (`fsClipStruct`) y otro sólo para el área de juego (`vpClipStruct`).

Leer los controles

En cada frame leemos los controles y los escribimos en la variable `pad0`. Los controles pulsados se pueden evaluar mirando los bits que estén a 0. **splib2** define una serie de constantes para hacer esto más sencillo:

Constante	Significado
sp_LEFT	izquierda
sp_RIGHT	derecha
sp_UP	arriba
sp_DOWN	abajo
sp_FIRE	disparo

Por ejemplo, para comprobar si se está pulsando 'abajo':

```
if ((pad0 & sp_DOWN) == 0) {  
    // Estamos pulsando "abajo"  
}
```

Cómo hacer...

Ganar o perder

Para hacer que el nivel termine satisfactoriamente habrá que...

```
success = 1;  
playing = 0;
```

Para producir un *game over* inmediato, simplemente:

```
success = playing = 0;
```

¿Algo más?

Si crees que debería aparecer algo que no está dímelo.

SCRIPTING

Documentación "al menos algo es algo" sobre MSC3 y el sistema de scripting de MK1 y MK2.

Copyright 2010-2015, 2020 the Mojon Twins

Este documento es válido para MK1 v5, MK2 v.0.88c y msc 3.91 o superiores.

DISCLAIMER: msc3 soporta más comprobaciones/comandos que no salen en este documento (pero sí en motor-de-clausulas.txt). No se garantiza que funcionen. Algunos directamente no lo harán (son de la rama 4 de la MK1).

DISCLAIMER 2: Con el scripting se pueden hacer cosas increíbles. Cosas que ni te imaginas. Nosotros hemos hecho cosas que ni nos imaginábamos que fueran posibles. Miles de accidentes afortunados y cosas así. Dale vueltas, probablemente encuentres la solución. Y si no, puedes preguntar. Pero dale vueltas antes. Confiamos en tí. Dale, Fran.

msc3

msc3 significa "Mojon Script Compiler 3".

msc3 es el compilador de scripts. Comenzó siendo una solución sencilla para compilar scripts basados en cláusulas en un bytecode fácilmente interpretable por el motor, pero se ha convertido en un monstruo increíble de múltiples tentáculos y cambiantes formas, como un pequeño Nyarlathotep.

msc3 genera un archivo binario script.bin con el bytecode del script y un par de archivos .h que contienen el intérprete y su configuración (msc.h y msc-config.h).

msc3 necesita poco para funcionar: un script, el número de pantallas en total que tiene el mapa global de nuestro juego (*), y si debe preparar el intérprete para funcionar desde la RAM extra de los modelos de 128K.

msc3 se ejecuta desde línea de comandos y su sintaxis es:

```
$ msc3.exe archivo.spt N [rampage]
```

Donde archivo.spt es el archivo con el script que hay que compilar, N es el número de pantallas de nuestro juego (*), y [rampage] es un parámetro opcional que, si se indica, hace que msc3 genere un intérprete preparado para interpretar script.bin desde una página de RAM extra de los modelos de 128K.

(*) En juegos multi-nivel en el que cada nivel tiene un número de pantallas, se refiere al tamaño máximo en pantallas para el que se reserva memoria en RAM baja, o sea, el resultado de multiplicar

MAP_W por MAP_H .

Motor de cláusulas

Los scripts de MK2 se organizan en secciones. Cada sección se ejecutará en un momento preciso y en una pantalla precisa.

Principalmente tenemos secciones tipo ENTERING, que se ejecutarán al entrar en una pantalla, secciones PRESS_FIRE, que se ejecutarán al pulsar la tecla de acción, y secciones especiales que responderán a diversos eventos. Estos son los tipos de secciones:

ENTERING SCREEN x

Se ejecuta cada vez que el jugador entra en la pantalla x

ENTERING GAME

Se ejecuta al empezar cada partida, y sólo esta vez.

ENTERING ANY

Se ejecuta al entrar en cada pantalla, justo antes que ENTERING SCREEN x

PRESS_FIRE AT SCREEN x

Se ejecuta cuando el jugador pulsa la tecla de acción en la pantalla x.

PRESS_FIRE AT ANY

Se ejecuta cuando el jugador pulsa la tecla de acción en cualquier pantalla, justo antes que PRESS_FIRE AT SCREEN x

ON_TIMER_OFF

Se ejecuta cuando el temporizador llega a cero, siempre que hayamos definido la directiva TIMER_SCRIPT_0 en config.h

PLAYER_GETS_COIN

Se ejecuta cuando el jugador toca un tile `TILE_GET` . Necesita tener activada y configurada la funcionalidad `TILE_GET` en `config.h` así como la directiva `TILE_GET_SCRIPT` . **No disponible en MK1**

`PLAYER_KILLS_ENEMY`

Se ejecuta cuando el jugador mata a un enemigo, siempre que hayamos definido la directiva `RUN_SCRIPT_ON_KILL` en `config.h`

-

Los scripts `PRESS_FIRE` se ejecutarán en más supuestos además de cuando el jugador pulse acción:

- Si tenemos `#define ENABLE_FIRE_ZONE` en `config.h`, hemos definido una zona de acción con `SET_FIRE_ZONE` en nuestro script, y el jugador entra en dicha zona.
- Si empujamos un bloque y tenemos definido en `config.h` las directivas `ENABLE_PUSHED_SCRIPTING` y `PUSHING_ACTION` .
- Cada vez que un "floating object" se ve afectado por la gravedad y cae, si tenemos definido `ENABLE_FO_SCRIPTING` y `FO_GRAVITY` . **No disponible en MK1**

En versiones anteriores de MK2 y la Churrera también se lanzaban al matar un enemigo, pero se ha sustituido por la sección `PLAYER_KILLS_ENEMY` .

Varios niveles

Podemos definir script para varios niveles. El mejor ejemplo actual para verlo en acción es Ninjajar!. En nuestro archivo de script concatenamos los scripts de nuestros niveles uno detrás de otro separados por una línea

`END_OF_LEVEL`

Al compilar, msc3 generará una constante para identificar dónde empieza cada script para cada nivel que podremos usar en nuestro `level_manager` o como mejor nos convenga. La constante será del tipo `SCRIPT_X` con X el número de orden del script dentro del archivo .spt.

Sin embargo, a partir de MK2 0.90 podemos tener más control sobre esto. Si incluimos al principio de una sección de nuestro script (en realidad puede aparecer en cualquier sitio antes de `END_OF_LEVEL` , pero seamos ordenados) una línea

`LEVELID xxxxx`

Con "xxxxx" un literal alfanumérico, este literal será empleado como nombre de la constante generada. En lugar de `SCRIPT_X` será lo que pongamos nosotros, lo que nos dará quizá un poco más de control y legibilidad a la hora de montar nuestro manejador de niveles.

Cláusulas

Todas las secciones descritas arriba contendrán una lista de cláusulas. Cada cláusula se compone de una lista de comprobaciones y de una lista de comandos.

El intérprete recorrerá la lista de comprobaciones en orden, realizando cada comprobación. Si alguna falla, dejará de procesar la cláusula.

Si todas las comprobaciones han resultado ser ciertas, se ejecutará la lista de comandos asociada en orden.

La sintaxis es:

```
IF COMPROBACION
...
THEN
    COMANDO
    ...
END
```

Todas las cláusulas de una sección se ejecutan en orden, sin parar (a menos que así lo indiques con un comando `BREAK` o tras algunos comandos como `WARP_TO`, `WARP_TO_LEVEL`, `REHASH`, `REPOSTN` o `REENTER`).

Muchas veces puedes ahorrar script y evitar usar `BREAK`. La mayoría de las veces el tiempo de ejecución de un script no es crítico y puedes permitírtelo.

En vez de:

```
IF FLAG 1 = 0
THEN
    SET TILE (2, 4) = 2
    SET FLAG 1 = 1

    # Si no ponemos este break se ejecutará la siguiente
    # cláusula sí o sí, ya que FLAG 1 = 1.
    BREAK
END

IF FLAG 1 = 1
THEN
    SET TILE (2, 4) = 3
END
```

Puedes hacer:

```
IF FLAG 1 = 1
THEN
    SET TILE (2, 4) = 3
END

IF FLAG 1 = 0
THEN
    SET TILE (2, 4) = 2
    SET FLAG 1 = 1
END
```

Y te ahorras un BREAK.

Flags

El motor de scripting maneja un conjunto de banderas o flags que pueden contener un valor de 0 a 127 y que se utilizan como variables. Los flags suelen referenciarse como FLAG N con N de 0 a 127 en el script.

En casi todas las comprobaciones y comandos que admiten valores inmediatos se puede utilizar la construcción #N donde N es un número de flag, que significa "el valor del flag N".

Por ejemplo:

```
IF FLAG 5 = #3
```

Será cierta si el valor del flag 5 es igual al valor del flag 3, o:

```
WARP_TO #3, #4, #5
```

Saltará a la pantalla contenida en el FLAG 3, en la posición indicada por los flags 4 y 5. Por el contrario:

```
WARP_TO 3, 4, 5
```

Saltará a la pantalla 3, en la posición (4, 5).

Sin modificaciones, MK2 permite 32 flags, pero este número puede cambiarse fácilmente editando el #define correspondiente en definitions.h:

```
#define MAX_FLAGS 32
```

Alias

Para no tener que recordar tantos números de flags, es posible definir alias. Cada alias representa un número de flag, y comienzan con el carácter "\$". De este modo, podemos asociar por ejemplo el alias \$LLAVE al flag 2 y usar en el script este alias en vez del 2:

```
IF FLAG $LLAVE = 1
```

Los alias también funcionan con la construcción #N, de este modo si escribimos #\$NENEMS nos referiremos al contenido del flag cuyo alias es NENEMS :

```
IF FLAG $TOTAL = #$NENEMS
```

Se cumple si el valor del flag cuyo alias es \$TOTAL coincide con el valor del flag cuyo alias es \$NENEMS .

Los alias deben definirse al principio del script (en realidad pueden definirse en cualquier parte del script, pero sólo serán válidos en la parte del script que viene después) en una sección DEFALIAS que debe tener el siguiente formato:

```
DEFALIAS
    $ALIAS = N
    ...
END
```

Donde \$ALIAS es un alias y N es un número de flag. Definiremos un alias por cada línea. No es necesario definir un alias para cada flag.

A partir de la versión 3.92 de msc3, puedes obviar la palabra FLAG si usas alias. O sea, que el compilador aceptará

```
SET $LLAVE = 1
```

y también

```
IF $LLAVE = 1
```

También podemos definir macros sencillos dentro de DEFALIAS . Los macros empiezan por el carácter %. Todo lo que venga después del nombre del macro se considera el texto de sustitución. Por ejemplo, si defines:

`%CLS` `EXTERN 0`

Cada vez que pongas `%CLS` en tu código msc3 lo sustituirá por `EXTERN 0`.

Decoraciones

Cada script (y con esto quiero decir cada script de nivel incluido en nuestro archivo de script) puede incluir un set de decoraciones tal y como genera map2bin.exe. Para ello habrá que incluir la directiva `INC_DECORATIONS` al principio del script:

```
INC_DECORATIONS archivo.spt
```

Donde `archivo.spt` es el archivo `.spt` que genera `map2bin.exe`

El intérprete dinámico

msc3 genera un intérprete que sólo será capaz de entender las comprobaciones y comandos que hayas introducido en tu script. Esto se hace para ahorrar memoria no generando código que jamás se ejecutará.

A veces hay varias formas de conseguir una cosa en tu script. Si tienes que elegir, no elijas la que produzca un script más sencillo, sino la que haga que tengas que utilizar menos variedad de comprobaciones o de comandos, ya que un poco más de script ocupa muchísimo menos que el código C necesario para ejecutar una comprobación o comando.

Siempre cierto

Hay una comprobación que siempre vale cierto y que se utiliza para ejecutar comandos en cualquier caso:

```
IF TRUE
```

Comprobaciones y comandos relacionados con los flags

Gran parte de tu script estará comprobando valores de los flags y modificando dichos valores. Para ello hay todo un set de comprobaciones y comandos.

Comprobaciones con flags

<code>IF FLAG x = n</code>	Evaluará a CIERTO si el flag "x" vale "n"
<code>IF FLAG x < n</code>	Evaluará a CIERTO si el flag "x" < n
<code>IF FLAG x > n</code>	Evaluará a CIERTO si el flag "x" > n
<code>IF FLAG x <> n</code>	Evaluará a CIERTO si el flag "x" <> n

Estas otras cuatro son "legacy". Están aquí porque era la única forma de comparar dos flags en las primeras versiones del sistema de scripting, pero pueden obviarse - de hecho DEBEN obviarse, ya que así ahorraremos código de intérprete.

<code>IF FLAG x = FLAG y</code>	Evaluará a CIERTO si el flag "x" = flag "y" Equivale a <code>IF FLAG x = #y</code>
<code>IF FLAG x < FLAG y</code>	Evaluará a CIERTO si el flag "x" < flag "y" Equivale a <code>IF FLAG x < #y</code>
<code>IF FLAG x > FLAG y</code>	Evaluará a CIERTO si el flag "x" > flag "y" Equivale a <code>IF FLAG x > #y</code>
<code>IF FLAG x <> FLAG y</code>	Evaluará a CIERTO si el flag "x" <> flag "y" Equivale a <code>IF FLAG x <> #y</code>

Una de las cosas en mi TODO es modificar MSC para traducir automáticamente las construcciones "IF FLAG x FLAG y" a "IF FLAG X #y".

Comandos con flags

<code>SET FLAG x = n</code>	Da el valor N al flag X. Huelga decir que <code>SET FLAG x = #y</code> dará el valor del flag y al flag x. Pero ya lo he dicho.
<code>INC FLAG x, n</code>	Incrementa el valor del flag X en N.
<code>DEC FLAG x, n</code>	Decrementa el valor del flag X en N
<code>FLIPFLOP x</code>	Si x vale 0, valdrá 1. Si vale 1, valdrá 0. Lo que viene a ser un flip-flop, vaya.
<code>SWAP x, y</code>	Intercambia el valor de los flags x e y

Estas dos son legacy ya que pueden simularse con INC/DEC FLAG. Lo mismo de antes, permanecen aquí desde las primeras versiones cuando no existía la construcción #N.

ADD FLAGS x , y

Hace $x = x + y$.
Equivale a INC FLAG x , $\#y$

SUB FLAGS x , y

Hace $x = x - y$.
Equivale a DEC FLAG x , $\#y$

Cosas del motor que modifican flags

Todo este jaleo de flags sería mucho menos útil si el motor no pudiese modificarlos también dando información de estado. Ciertas directivas de config.h harán que ciertos flags se actualicen con valores del motor:

Número de objetos recogidos

Directiva OBJECT_COUNT

Con "objetos" me refiero a los objetos coleccionables automáticos que se colocan usando hotspots de tipo 1 en el Colocador.

Si se define esta directiva, el valor especificado indicará que flag debe actualizar el motor con el número de objetos que el jugador lleva recogidos. Por ejemplo

```
#define OBJECT_COUNT 1
```

Hará que el FLAG 1 contenga en todo momento el número de objetos que lleva el jugador recogidos. De ese modo,

```
IF FLAG 1 = 5
```

Se cumplirá cuando el jugador haya recogido 5 objetos.

Número de enemigos en la pantalla

No disponible en MK1

Directiva COUNT_SCR_ENEMS_ON_FLAG

Si se define esta directiva, el motor contará cuántos enemigos activos hay en la pantalla a la hora de entrar en ella. Por ejemplo

```
#define COUNT_SCR_ENEMS_ON_FLAG 1
```


Hara que el FLAG 1 contenga el número de enemigos que había en la pantalla al entrar.

Número de enemigos eliminados

Directiva BODY_COUNT_ON

Si se define esta directiva, el motor incrementará el flag especificado siempre que el jugador elimine un enemigo. Por ejemplo,

```
#define BODY_COUNT_ON 2
```

Hará que el FLAG 2 se incremente cada vez que el jugador mata un enemigo. Esto puede ser muy util para usarlo en conjunción con COUNT_SCR_ENEMS_ON_FLAG .

Siguiendo nuestro ejemplo (cuenta enemigos en pantalla en el flag 1, e incrementar el flag 2 al matar un enemigo), si al entrar en una nueva pantalla establecemos el flag 2 a 0 (resetear la cuenta de enemigos matados:

```
ENTERING ANY
  IF TRUE
    THEN
      SET FLAG 2 = 0
    END
  END
END
```

Podemos controlar que hemos matado a todos los enemigos de la pantalla muy fácilmente, en la sección PLAYER_KILLS_ENEMY , que se ejecuta siempre que el jugador mata un enemigo:

```
PLAYER_KILLS_ENEMY
  IF FLAG 2 = #1
    THEN
      # ¡Hemos matado a todos los enemigos de la pantalla!
    END
  END
END
```

Contar los TILE_GET

No disponible en MK1

Directiva TILE_GET_FLAG

El motor te permite definir un tile del mapa como "recogible". Por ejemplo, monedas. Puedes colocar monedas en el mapa, o hacerlas aparecer al romper una piedra (ver Ninjajar!). Si te has empapado whatsnew.txt sabrás que la directiva TILE_GET n hace que el tile "n" sea "recogible". Esto se utiliza en

conjunción con la directiva `TILE_GET_FLAG` , de forma que cuando el jugador toque un tile número `TILE_GET` se incremente ese flag.

No se puede usar `TILE_GET` sin `TILE_GET_FLAG` , por supuesto.

Por ejemplo, puedes hacer que tu tile 10 sea un diamante y colocar estos diamantes por todo el mapa. Luego defines `TILE_GET` a 10 y `TILE_GET_FLAG` a 1. Cada vez que el jugador toque un diamante, este desaparecerá y el flag 1 se incrementará.

En Ninjajar se usa de un modo más complejo, en conjunción con la directiva `BREAKABLE_TILE_GET n` . En ninjajar está definido así:

```
#define BREAKABLE_TILE_GET 12
#define TILE_GET 22
#define TILE_GET_FLAG 1
```

Y produce este comportamiento: Si el jugador rompe el tile 12, que es la caja con estrellas (que en sus "behaviours" tiene activado el flag "rompible"), aparecerá el tile `TILE_GET` , que es el 22, la moneda. Si el jugador toca un tile 22 (una moneda), esta desaparecerá y se incrementará el flag 1, que es la cuenta del dinero que llevamos recogido.

Nótese que Ninjajar usa mapas de 16 tiles - las monedas sólo aparecerán al romper bloques estrella.

Todo esto funciona automáticamente, nosotros sólo tendremos que preocuparnos de examinar el valor del FLAG definido en `TILE_GET_FLAG` desde nuestro script.

Bloques que se empujan

Directivas `MOVED_TILE_FLAG` , `MOVED_X_FLAG` y `MOVED_Y_FLAG`

Si hemos activado que el jugador pueda empujar cajas (tile 14 del tileset, con "behaviour" = 10) mediante `#define PLAYER_PUSH_BOXES` , y además activamos la directiva `ENABLE_PUSHED_SCRIPTING` , cada vez que el jugador mueva una caja (o bloque, lo que sea que hayamos puesto en el tile 14 ;-)), ocurrirá esto:

- El número de tile que se "pisa" se copiará en el flag `MOVED_TILE_FLAG` .
- La coordenada X a la que se mueve la caja se copia en el flag `MOVED_X_FLAG` .
- La coordenada Y a la que se mueve la caja se copia en el flag `MOVED_Y_FLAG` .

Si, además, definimos `PUSHING_ACTION` , se lanzarán las secciones `PRESS_FIRE` (la `ANY` y la correspondiente a la pantalla actual) y podremos hacer las comprobaciones inmediatamente.

Este sistema se emplea en Cadaverion (aunque sea un juego de la Churrera, esto funciona exactamente igual) para comprobar que hemos colocado las estatuas en sus pedestales.

Por ejemplo, si queremos abrir una puerta si se empuja una caja a la posición (7, 7) de la pantalla 4, podemos definir:

```
#define PLAYER_PUSH_BOXES
#define ENABLE_PUSHED_SCRIPTING
#define PUSHING_ACTION
#define MOVED_TILE_FLAG      1    // Esto no lo vamos a usar en este ejemplo
#define MOVED_X_FLAG         2
#define MOVED_Y_FLAG         3
```

Y en nuestro script...

```
PRESS_FIRE AT SCREEN 4
  IF FLAG 2 = 7
  IF FLAG 3 = 7
  THEN
    # La caja que hemos movido está en la posición 7,7
    # abrir puerta...
  END
END
```

Si, por ejemplo, queremos que una puerta se abra siempre que se coloque una caja sobre un tile "pulsador", que hemos dibujado en el tile, digamos, 3, (con los mismos defines) tendríamos:

```
PRESS_FIRE AT SCREEN 4
  IF FLAG 1 = 3
  THEN
    # La caja que hemos movido está sobre un tile 3
    # abrir puerta...
  END
END
```

Que se pueda o no disparar

Directiva `PLAYER_CAN_FIRE_FLAG`

Si tenemos nuestro motor configurado para un juego de disparos usando la directiva `PLAYER_CAN_FIRE` y sus compañeras, podemos definir que el jugador sólo pueda disparar si un flag determinado vale 1 dándole el número de ese flag a la directiva `PLAYER_CAN_FIRE_FLAG`. Por ejemplo:

```
#define PLAYER_CAN_FIRE_FLAG 4
```

Al principio del juego podemos hacer:

```
ENTERING GAME
  IF TRUE
  THEN
    SET FLAG 4 = 0
  END
END
```

Con lo que el jugador no podrá disparar. Luego, más adelante, podemos hacer que el jugador encuentre una pistola y la recoja:

```
... (donde sea)
  IF ... (condiciones de coger la pistola)
  THEN
    SET FLAG 4 = 1
    # Ahora el muñaco puede disparar
  END
END
```

TODO: hacer lo mismo para el hitter (puño/espada). Pronto.

Golpeado por una "floating object" lanzable

No disponible en MK1

Directiva `CARRIABLE_BOXES_COUNT_KILLS`

En realidad, morir aplastado por una caja cuenta como incremento en el flag definido en `BODY_COUNT_ON`, pero por alguna misteriosa razón que no recuerdo (probablemente, desorganización mental) lo puse también separado.

Si se activa esta directiva el número de flag indicado contará sólo los enemigos que mueran golpeados por un floating object lanzable.

(Si se define `BODY_COUNT_ON` también incrementarán el flag definido ahí... no sé, a lo mejor te sirve de algo).

"Floating objects" que caen afectados por la gravedad

No disponible en MK1

Directivas `ENABLE_FO_SCRIPTING`, `FO_X_FLAG`, `FO_Y_FLAG` y `FO_T_FLAG`

Si tu juego utiliza "floating objects" (Leovigildo 1, 2 y 3), si activas la gravedad con `FO_GRAVITY` los objetos caeran si no tienen ningún obstáculo debajo.

En Leovigildo 3 nos inventamos un puzzle en el que había que dejar caer una corchoneta de sartar sobre Amador el Domador para estrujarlo. Esto necesitó ampliar el motor para poder comprobar adónde caía un "floating object".

Si activamos `ENABLE_FO_SCRIPTING`, se ejecutarán las secciones `PRESS_FIRE` general y de la pantalla actual cada vez que un "floating object" caiga y descienda una casilla.

Justo antes de llamar al script, la posición a la que ha caído se copiará en los flags indicados en `FO_X_FLAG` y `FO_Y_FLAG`, y el tipo del "floating object" se copiará en `FO_T_FLAG`.

Es interesante saber que el flag `FO_T_FLAG` se pondrá a 0 al entrar en una pantalla.

```
#define ENABLE_FO_SCRIPTING
#define FO_X_FLAG           1
#define FO_Y_FLAG           2
#define FO_T_FLAG           3
```

Aquí ya usamos alias...

```
DEFALIAS
[... ]
$FO_X 1
$FO_Y 2
$FO_T 3
[... ]
END
```

Y en el `PRESS_FIRE` de la pantalla de Amador encontramos...

```
PRESS_FIRE AT SCREEN 19
# Lanzar el FO encima de Amador
# Amador está en X, Y = (8, 7).
# El FO corchoneta es el tile 17
IF FLAG $FO_T = 17
IF FLAG $FO_X = 8
IF FLAG $FO_Y = 7
THEN
    SOUND 0
    EXTERN 31
    SET FLAG $AMADOR = 5
    REENTER
END
...
```

Comprobaciones y comandos relacionados con la posición

También tenemos varias formas de comprobar y modificar la posición - incluso cambiando de pantalla ¡y de nivel!

Comprobaciones sobre la posición

- | | |
|--|--|
| <code>IF PLAYER_TOUCHES x, y</code> | Evaluará a CIERTO si el jugador está tocando el tile (x, y). x e y pueden llevar #. |
| <code>IF PLAYER_IN_X x1, x2</code> | Evaluará a CIERTO si el jugador está horizontalmente entre las coordenadas en pixels x1 y x2. |
| <code>IF PLAYER_IN_Y y1, y2</code> | Evaluará a CIERTO si el jugador está verticalmente entre las coordenadas en pixels y1 e y2. |
| <code>IF PLAYER_IN_X_TILES x1, x2</code> | Evaluará a CIERTO si el jugador está horizontalmente entre los tiles x1 y x2, ambos inclusive. |
| <code>IF PLAYER_IN_Y_TILES y1, y2</code> | Evaluará a CIERTO si el jugador está verticalmente entre los tiles y1 e y2, ambos inclusive. |

Cambiando de posición

Estos comandos sirven para modificar la posición del personaje. Todas se expresan a nivel de tiles (x de 0 a 14, y de 0 a 9).

- | | |
|-------------------------|---|
| <code>SETX x</code> | Colocará al personaje en la coordenada de tile x (modifica solo la posición horizontal) |
| <code>SETY y</code> | Colocará al personaje en la coordenada de tile y (modifica solo la posición vertical) |
| <code>SETXY x, y</code> | Colocará al personaje en la coordenada de tile (x, y) (modifica ambas coordenadas, x e y). |

Comprobaciones sobre la pantalla

Aunque poder definir scripts en ENTERING n y PRESS_FIRE AT n donde n es la pantalla actual y que sólo se ejecuten cuando estamos en dicha pantalla, hay veces en las que es necesario saber en qué pantalla estamos en una de las secciones "generales" (cuando se acaba el TIMER, cuando matamos un enemigo...) Para esos casos tenemos:

<code>IF NPANT n</code>	Evaluará a CIERTO si el jugador <code>está en la</code> pantalla <code>n</code>
<code>IF NPANT_NOT n</code>	Evaluará a CIERTO si el jugador <code>NO está en la</code> pantalla <code>n</code>

Cambiando de pantalla

<code>WARP_TO n, x, y</code>	Mueve al jugador a la posición (<code>x, y</code>) de la pantalla <code>n</code> . <code>x</code> e <code>y</code> a nivel de tiles.
------------------------------	---

Cambiando de nivel

Obviamente, sólo si tu juego tiene varios niveles.

`WARP_TO_LEVEL l, n, x, y, s`

Hace lo siguiente:

- Termina el nivel actual.
- Carga e inicializa el nivel `l`.
- Establece la pantalla activa a `n`.
- Pone al jugador en las coordenadas (`x, y`) (a nivel de tiles).
- Si `s = 1`, no muestra una pantalla de nuevo nivel (*)

(*) Esto es muy relativo y muy custom. En Ninjajar! hay una pantalla antes de empezar cada nivel, que sólo se muestra si la variable `silent_level` vale 0. El valor de `s` se copia a `silent_level`, so...

Si tu manejador de niveles es diferente lo puedes obviar, o usar para otra cosa.

Sí, hacer juegos de muchos niveles y tal es complicado. La vida es así.

Redibujar la pantalla

Es util si haces algo que se cargue la pantalla, como sacar un cuadro de texto con un `EXTERN` (ver más adelante) (todo lo que hemos hecho de Ninjajar en adelante usa `EXTERN` principalmente para sacar cuadros de texto que se cargan la zona de juego). Así vuelves a pintarlo todo. Sólo hay que ejecutar:

`REDRAW`

Ojete: existe un buffer de tamaño pantalla donde cada cosa que se imprime (bien por la rutina que se ejecuta al entrar en una pantalla nueva y que compone el escenario, bien por un `SET TILE (X, Y) = T` del scripting, etc.) se copia ahí. `REDRAW` simplemente vuelca ese buffer a la pantalla. ¡Si has modificado la pantalla con cosas desde el script, `REDRAW` no la va a volver a su estado original!

"Reentrar" en la pantalla

A veces necesitas volver a ejecutar todo el script de `ENTERING ANY` y/o de `ENTERING SCREEN n`, o necesitas reinicializar los enemigos (si les cambias el tipo al vuelo, ver más adelante), o sabe dios qué.

Hay varias formas de reentrar en la pantalla:

<code>REENTER</code>	Vuelve a entrar en la pantalla, exactamente igual que si viniésemos de otra. Lo hace todo: redibuja, inicializa todo, ejecuta los scripts...
<code>REHASH</code>	Lo mismo, pero sin redibujar. Tampoco muestra la pantalla "LEVEL XX" si tu motor está configurado para ello. Pero si inicializa todo y ejecuta los scripts.

Modificar la pantalla

Hay varias formas de modificar la pantalla:

Cambiar tiles del área de juego

Cambiar tiles del área de juego modifica efectivamente el area de juego. Quiero decir que los cambios son persistentes (sobreviven a un `REDRAW`) y además los tiles modificados son interactivables. O sea, si modificas la pantalla eliminando una pared con un tile transparente, el jugador podrá pasar por ahí.

<code>SET TILE (x, y) = t</code>	Pone el tile <code>t</code> en la coordenada <code>(x, y)</code> . Las coordenadas <code>(x, y)</code> están a nivel de tiles.
----------------------------------	---

Por supuesto, y esto es muy útil, tanto `x` como `y` como `t` pueden llevar `#` para indicar el contenido de un flag. Para imprimir en 4, 5 el tile que diga el flag 2, hacemos

```
SET TILE (4, 5) = #2.
```

Para imprimir un tile 7 en las coordendas almacenadas en los flags 2 (`x`) y 3 (`y`), hacemos:


```
SET TILE (#2, #3) = 7
```

Y, recordemos, siempre que referenciemos un flag podemos usar un alias. No lo estoy recordando todo el rato porque confío en tu inteligencia, pero

```
DEFALIAS
    $COORD_X 2
    $COORD_Y 3
END

[...]

SET TILE (#$COORD_X, #$COORD_Y) = 7
```

Vale igual.

También puedes usar listas de decoraciones. Las he mencionado más arriba cuando hablé de INC_DECORATIONS. Las listas de decoraciones las puedes meter en cualquier sección de comandos, son así:

```
DECORATIONS
    x, y, t
    ...
END
```

Donde (x, y) es una coordenada a nivel de tiles, y t es un número de tile.

Por ejemplo:

```
DECORATIONS
    12, 3, 18
    7, 4, 16
    8, 4, 18
    2, 5, 27
    12, 5, 27
    2, 6, 28
    7, 6, 27
    12, 6, 26
    7, 7, 29
    12, 7, 28
    8, 8, 19
    9, 8, 20
    10, 8, 20
    11, 8, 21
END
```

Cambiar sólo el comportamiento

Muy, muy tonto. Si lo necesitas, funciona igual que SET TILE pero sólo sustituye el comportamiento original por el que especifiques:

```
SET BEH (x, y) = b
```

Imprimir tiles en cualquier sitio

Podemos imprimir un tile en cualquier sitio de la pantalla, sea en el area de juego o bien fuera (por ejemplo, en una zona del marcador). Para ello usamos:

```
PRINT_TILE_AT (x, y) = n
Imprime el tile n e (x, y), con (x, y) ¡ojo! en
coordenadas DE CHARACTER (x = 0-30, y = 0-22).
```

Esta función sólo imprime. Aunque el tile que pintemos esté sobre el area de juego no la afectará en absoluto para nada.

Una cosa muy chula para lo que puede servir esto es para hacer pasajes secretos: en tu mapa haces un pasillo, pero luego en el ENTERING SCREEN lo cubres de tiles con PRINT_TILE_AT ... Como estos tiles no afectan al area de juego, parecerá que no se puede pasar por ahí... pero ¡sí que se puede!

Mostrar cambios

Todas las impresiones de tiles en el motor se hacen a un buffer. En cada cuadro de juego, este buffer se dibuja en la pantalla siguiendo un divertido y mágico proceso. Sin embargo, durante la ejecución del script, no se vuelca el buffer a la pantalla.

Si cambiamos algo y queremos que se vea inmediatamente sin tener que esperar a volver al juego (por ejemplo, si estamos haciendo una animación), necesitamos decirle al intérprete de forma explícita que pinte el buffer en la pantalla. Esto se hace con el comando:

```
SHOW
```

Empujar bloques

Ya hemos hablado, pero no habíamos mencionado que:

```
IF JUST_PUSHED
Será cierto si hemos llegado al FIRE_SCRIPT por
haber empujado una caja.
```

Esto es MUY UTIL. Tienes que tener en cuenta que la posición del objeto empujado (almacenada en los flags definidos en `MOVED_X_FLAG` y `MOVED_Y_FLAG`) y el tile que ha sobrescrito (almacenado en el flag definido en `MOVED_TILE_FLAG`) son persistentes - estos flags conservarán su valor hasta que se mueva otro bloque.

En la sección `PRESS_FIRE` se puede entrar de varias formas, por ejemplo pulsando acción. A lo mejor no te conviene que se hagan comprobaciones que impliquen los flags afectados por los bloques empujados si no hemos entrado tras empujar un bloque.

Cadaverion hace uso de esto.

En Cadaverion hay que empujar cierto número de estatuas sobre cierto número de pedestales. La configuración en `config.h` relacionada con esto es:

```
#define ENABLE_PUSHED_SCRIPTING
#define MOVED_TILE_FLAG          1
#define MOVED_X_FLAG            2
#define MOVED_Y_FLAG            3
#define PUSHING_ACTION
```

En el `ENTERING` de cada pantalla, se establece en el flag #9 el número de estatuas/pedestales que hay. O sea, el flag #9 contendrá el número de estatuas que tenemos que colocar en sus pedestales.

En cada pantalla hay una cancela que hay que abrir poniendo las estatuas en sus pedestales. En el `ENTERING` definimos su posición en las flags #6 y #7 (resp., coordenadas X e Y).

En el flag #10 vamos a ir contando las estatuas que colocamos en su sitio. Cuando todas las estatuas estén en su sitio, (esto es, el flag 9 y el flag 10 contengan el mismo valor) vamos a abrir la cancela que permite ir a la siguiente pantalla.

Vamos a usar el flag #11 como bandera. Si vale 0, la cancela está cerrada. Si vale 1, la hemos abierto ya. Entonces, este código es el que abre la cancela:

```
IF FLAG 9 = #10
IF FLAG 11 = 0
THEN
    # Decimos que la cancela está abierta:
    SET FLAG 11 = 1

    # Borramos la cancela poniendo un tile 0 en sus coordenadas:
    SET TILE (#6, #7) = 0

    # Mostramos los cambios inmediatamente.
    SHOW

    # Ruiditos
    SOUND 0
```

```

SOUND 7

# más cosas que no nos interesan.
[...]
END

```

Podríamos haber añadido `JUST_PUSHED` pero da igual. No se cumplirá hasta que no hayamos colocado la última estatua, así que no importa que se compruebe cuando no debe.

Las estatuas corresponden al tile 14, y tiene "behaviour 10", o sea, que las estatuas son el bloque "empujable".

Los pedestales son el tile 13.

En el `PRESS_FIRE AT ANY`, que se lanzará (junto con el `PRESS_FIRE AT SCREEN n` de la pantalla actual) cada vez que movamos un bloque, comprobaremos que hemos pisado un pedestal. O sea, comprobaremos que:

```

IF JUST_PUSHED
IF FLAG 1 = 13
THEN
    INC FLAG 10, 1

```

Esto es: hemos llegado aquí por haber empujado una estatua (`JUST_PUSHED`) y el tile que hemos "pisado" es el 13 (un pedestal). Entonces, sumamos 1 a la cuenta de estatuas colocadas.

¿Qué pasaría si no pusiésemos `JUST_PUSHED` ? Pues imagina: tú coges, mueves una estatua al pedestal. en el `FLAG 1` se copia el tile que acaba de pisar, que es 13. Pero ese valor se queda ahí... Si pulsas ACCIÓN, por ejemplo, el valor seguirá ahí, y contaría como que hemos pisado otro pedestal. No mola.

El código completo de esta cláusula es este, porque se hace otra cosa muy interesante:

```

IF JUST_PUSHED
IF FLAG 1 = 13
THEN
    # Incrementamos la cuenta de estatuas
    INC FLAG 10, 1

    # Sonido
    SOUND 0

    # ¡Cambiamos la estatua por otro tile!
    SET TILE (#2, #3) = 22

    SHOW
    SOUND 0
END

```

Si no cambiásemos la estatua por otro tile, podrías seguir empujándola. Como empujar un bloque normal es destructivo (siempre borra con el tile 0), esto quedaría de la hostia de feo.

Al cambiar el tile que hay en (#2, #3), que es la posición de nuestra estatua según config.h, por otro que no sea el 14 (el 22 es una estatua girada, quedaba chuli), no podrá empujarse más.

Clever, uh?

El timer

El timer es una cosa muy chula que hay en el motor MK2 (también estaba en la Churrera, pero ahora es más mejón). Básicamente es un valor que se decrementa cada cierto número de cuadros de juego. Cuando llega a 0, puede hacerse que el jugador pierda una vida, o que haya un game over... O puede ejecutarse una sección especial del script.

(Hay mucho sobre el timer, que puede funcionar de forma autónoma y hacer muchas cosas, pero eso no nos incumbe. Mira en whatsnew.txt o pregunta).

Si activamos en config.h

```
#define TIMER_SCRIPT_0
```

Cada vez que el timer llegue a 0 se disparará la sección `ON_TIMER_OFF`. Ahí podremos hacer cosas.

Además tenemos ciertas comprobaciones y algunos comandos relacionados con el timer:

Comprobaciones sobre el timer

`IF TIMER >= x` Cierta si el `timer` tiene un valor `>= x`.

`IF TIMER <= x` Cierta si el `timer`... ¡oh, vamos!

Comandos del timer

En realidad, "comando", porque sólo hay este:

`SET_TIMER v, r` Establece el `timer` a un valor `v` con "`rate`" `r`. Significa `que se` decrementará cada `r` cuadros `de` juego. `En` condiciones normales, los juegos van entre 22 y 33 faps por segundo... `de` 25 a 30 son buenos valores si quieres `que tu timer` parezca `que` cuenta `en` segundos. Experimenta.

OJO: SET_TIMER , con "". *Sí, ya sé que hay un SET FLAG sin ""*. Así es, acéptalo. Get over it. Te equivocarás mil veces. Yo me equivoco mil veces. Pero me jodo. En serio, ya lo cambiaré.

TODO: Cambiar esto.

TIMER_START	Enciende el timer.
TIMER_STOP	Apaga el timer.

Se pueden hacer muchas cosas. Por ejemplo, nos vale lo que se hace en Cadaverion...

```
# Esto es lo que pasa cuando se acaba el tiempo.
# En #12 guardamos la pantalla a la que hay que volver al acabarse el tiempo.
# En #13, #14 las coordenadas donde apareceremos cuando esto ocurra.

ON_TIMER_OFF
  IF TRUE
  THEN
    SOUND 0
    SOUND 0
    SOUND 0
    SOUND 0
    SET_TIMER 60, 40
    DEC LIFE 1
    SET FLAG 8 = #0
    WARP_TO #12, #13, #14
  END
END
```

Una cosa que suele hacerse, como acabamos de ver, es reiniciar el timer. En este caso se resta una vida, se resetea el timer, y se cambia al personaje de sitio.

El inventario

No disponible en MK1

Esto empezó de coña, se empepinó en Leovigildo, y ahora es un pepino genial que nos permite hacer muchas cosas con muy poco código. Ains, hubiera venido bien en Ninjajar! - hubiese ahorrado un montón de código de script y de quebraderos de cabeza.

El inventario no es más que un contenedor de objetos. Podemos definir cuántos objetos como máximo vamos a llevar.

El inventario tiene, por tanto, un número N de "slots". Cada slot puede estar vacío o contener un objeto. Los objetos se referencian de la forma más sencilla posible: por su número de tile. "0" no se puede usar porque indica "vacío".

Hay, además, y en todo momento, un "slot seleccionado".

El valor del slot seleccionado y, por comodidad, el valor del objeto que hay en dicho slot, se mantienen en dos flags especiales que podemos elegir.

La posición y configuración del inventario también es bastante configurable y se puede adaptar bastante.

Vamos a hablar primero del inventario "a pelo" y luego hablamos de los "floating objects" de tipo "container", que fueron creados para hacer que hacer juegos con objetos e inventario sea un puto paseo.

Definiendo nuestro inventario.

El inventario se define en un apartado especial a principio del script. En él se definen los diferentes parámetros necesarios para poner en marcha el sistema y tiene esta forma:

```
ITEMSET
  SIZE n
  LOCATION x, y
  DISPOSITION disp, sep
  SELECTOR col, c1, c2
  EMPTY tile_empty
  SLOT_FLAG slot_flag
  ITEM_FLAG item_flag
END
```

Vamos describiendo cada línea una por una (sí, son necesarias todas).

SIZE n	Indica el tamaño de nuestro inventario, esto es, el número de slots que lo compondrán.
LOCATION x, y	Indica la posición (x, y) a nivel de caracteres de la esquina superior del inventario, que se corresponde a dónde aparecerá el primer item.
DISPOSITION disp, sep	"disp" debe valer HORZ o VERT, para indicar si queremos que los slots de nuestro inventario se muestren unos al lado de otros (HORZ) o unos debajo de otros (VERT). "sep" define la separación:

- Si el inventario es HORIZONTAL (HORZ)
 - El primer slot se colocará en (x, y)
 - El siguiente, en (x + sep, y)
 - El siguiente, en (x + sep + sep, y) ...

- Si el inventario es VERTICAL (VERT)
 - El primer slot se colocará en (x, y)
 - El siguiente, en (x, y + sep)
 - El siguiente, en (x, y + sep + sep) ...

Cada slot ocupa 2x2 caracteres (lo que ocupa un tile) pero debe dejarse un espacio de 2x1 caracteres justo debajo para pintar el "selector" que indica cuál es el slot activo.

SELECTOR col, c1, c2	<p>Define el selector. El selector no es más que una flecha o un algo que se coloca justo debajo del slot activo.</p> <p>El selector se pintará de color "col" y se compondrá de los caracteres c1 y c2 del charset, en horizontal.</p> <p>Yo suelo usar dos caracteres de las letras, para no desperdiciar tiles. Mira font.png en cualquiera de los "Leovigildo" y verás la flecha en los caracteres 62 y 63.</p>
EMPTY tile_empty	<p>Contiene el número de tile de nuestro tileset que se empleará para representar un "slot vacío".</p> <p>En Leovigildo pusimos un cuadradito azul muy chulo, puedes verlo en el tileset en la posición 31.</p>
SLOT_FLAG slot_flag	Dice qué flag contiene el slot actual seleccionado
ITEM_FLAG item_flag	Dice qué flag contiene el contenido del slot actual seleccionado.

Para no estorbar, yo suelo definir los flags 30 y 31 (los dos últimos si usamos los 32 que vienen por defecto) para estos dos valores. Así, en todo momento, el flag 30 (SLOT_FLAG 30) contiene qué slot está seleccionado (un número de 0 a n - 1; si el inventario tiene 4 slots, podrá valer 0, 1, 2 o 3 dependiendo de cuál esté seleccionado), y el flag 31 (ITEM_FLAG 31) contiene el objeto que hay en ese slot.

Ejemplos: Leovigildo 1, por ejemplo:

```

ITEMSET
  SIZE 4
  LOCATION 18, 21
  DISPOSITION HORZ, 3
  SELECTOR 66, 62, 63
  EMPTY 31
  SLOT_FLAG 30
  ITEM_FLAG 31
END

```


Tenemos un inventario que podrá contener cuatro objetos. Hemos dejado sitio en el marcador para él, a partir de las coordenadas (18, 21). Se pintará en horizontal, con un nuevo slot cada 3 caracteres. El selector será rojo intenso ($2 + 64 = 66$) y se pintará con los caracteres 62 y 63. El tile que representará un slot vacío es el 31, que en el tileset aparece como un recuadro azul. El flag 30 contendrá el slot seleccionado, y el flag 31 contendrá qué objeto hay en ese slot (0 si no hay ninguno).

En Leovigildo 3 también tenemos un inventario:

```
ITEMSET
  SIZE 3
  LOCATION 21, 21
  DISPOSITION HORZ, 3
  SELECTOR 66, 62, 63
  EMPTY 31
  SLOT_FLAG 30
  ITEM_FLAG 31
END
```

En este caso el inventario es más pequeño, de solo 3 slots. Se colocará a partir de las coordenadas (21, 21), será también horizontal y los slots se dibujarán cada 3 caracteres. El selector será igual, y la configuración de tile vacío y flags es la misma.

Vaya mierda de ejemplos, son muy parecidos. Pero a hoerce.

Acceso directo a cada slot

Podemos acceder directamente a cada slot, recordemos, numerados de 0 a $n - 1$, usando "ITEM", tanto en condiciones como en comandos:

IF ITEM n = t	Evalúa CIERTO si en el slot N está el item T
IF ITEM n <> t	Evalúa CIERTO si en el slot N no está el item T (Dudo que esto sirva para algo... pero bueno)
SET ITEM n = t	Pone el objeto representado por el tile T en el slot N.

SET ITEM nos servirá para inicializar el inventario, por ejemplo, o para hacer aparecer objetos en él por arte potagio. Al principio de cada uego puedes hacer:

```
SET ITEM 0 = 0
SET ITEM 1 = 0
SET ITEM 2 = 0
SET ITEM 3 = 0
```

(para un inventario de 4 slots). Yo lo uso también cuando estoy haciendo debug, para ponerme en el inventario objetos que necesito para probar alguna cosa.

Al igual que ocurría con las impresiones, los cambios no serán visibles hasta que ocurra el siguiente cuadro de juego. Si por alguna razón necesitas modificar el inventario y que se muestre en el medio de un script, puedes usar

```
REDRAW_ITEMS
```

Para refrescar el inventario en pantalla.

Slot seleccionado y objeto seleccionado en el script

Lo de arriba es suficiente para que aparezca el inventario cuando ejecutemos el juego, pero habrá que hacerlo funcionar. Para ello usaremos los flags definidos para tal fin en `SLOT_FLAG` e `ITEM_FLAG`.

Hemos hablado de alias. Podríamos definir estos alias (contando con que estamos siguiendo el ejemplo y hemos definido `SLOT_FLAG` en 30 e `ITEM_FLAG` en 31):

```
ALIAS
    $SLOT_FLAG 30
    $ITEM_FLAG 31
END
```

Incluso podríamos usar los flags 30 y 31 a pelo...

Se puede hacer todo con esto, pero hay ciertas condiciones y ciertos comandos predefinidos que harán tu script más legible... y que serán traducidos de forma transparente a manejos con los flags definidos en `SLOT_FLAG` e `ITEM_FLAG` - por lo que no te costarán "código de intérprete".

<code>IF SEL_ITEM = T</code>	Evalúa CIERTO si el <code>item</code> que hay en el slot <code>seleccionado</code> es el representado por el tile T. (Internamente equivale a <code>IF FLAG \$ITEM_FLAG = T</code>)
<code>IF SEL_ITEM <> T</code>	Evalúa CIERTO si el <code>item</code> que hay en el slot <code>seleccionado</code> NO es el representado por el tile T. (Internamente equivale a <code>IF FLAG \$ITEM_FLAG <> T</code>)

Además se definen el alias automático:

<code>SLOT_SELECTED</code>	Equivale al valor <code>del</code> slot seleccionado.
----------------------------	---

Por tanto, podemos establecer qué ITEM queremos en el slot actual haciendo

```
SET ITEM SLOT_SELECTED = T
```

Esto se utiliza así. Se supone que cuando pulsamos "acción", es para usar el item que tenemos seleccionado en un sitio específico de la pantalla.

Vamos a poner un ejemplo práctico para ver el funcionamiento básico del inventario. Todo esto se simplifica muchísimo con el uso de "floating objects" de tipo contenedor, pero vamos a verlo primero en plan comando.

Imaginad que llegamos a una habitación donde hay un objeto, una hoja de papel. Se trata del tile 22 de nuestro tileset. Lo hemos colocado en la posición (5, 4) al entrar en la habitación en concreto, que es la 6. Tenemos un flag especial, \$PAPEL, que valdrá 0 si aún no lo hemos cogido.

```
ENTERING SCREEN 6
  IF FLAG $PAPEL = 0
  THEN
    SET TILE (5, 4) = 22
  END
END
```

Ahora vamos a permitir que el jugador coja el papel. Para ello, el jugador se irá a por el papel (tocando la posición (5, 4), esto es) y pulsará la tecla de Acción. Esto lanzará el script de PRESS_FIRE correspondiente.

En él vamos a detectar que estamos en el sitio correcto, vamos a eliminar el papel de la pantalla, lo vamos a meter como ITEM en el inventario, y pondremos el flag \$PAPEL a 1 para que no vuelva a aparecer:

```
PRESS_FIRE AT SCREEN 6
  IF FLAG $PAPEL = 0
  IF PLAYER_TOUCHES 5, 4
  THEN
    SET TILE (5, 4) = 0
    SET ITEM SLOT_SELECTED = 22
    SET FLAG $PAPEL = 1
  END
END
```

La línea SET ITEM SLOT_SELECTED = 22 hace que el papel aparezca en el inventario, justo en el slot que el usuario tuviese seleccionado.

Obviamente, si había algo ahí se machacará - para evitarlo habría que montar un pequeño pifostio, pero para eso tenemos los "floating objects" que veremos después. Ahora mismo da igual que machaque, es un ejemplo.

Nos vamos a otra pantalla de nuestro juego, pongamos que es la numero 8. Pongamos que tenemos en (7, 7) a un personaje que espera que le demos un papel para escribir una carta. Cuando se lo demos, se pondrá "contento" y esto hará que pasen cosas. El estado de "contento" lo expresaremos en un flag \$CONTENTO, que valdrá 0 al principio.

El jugador deberá seleccionar el ítem en el inventario y luego irse al sitio correcto y pulsar acción. Eso lanzará nuestro script de PRESS_FIRE :

```
PRESS_FIRE AT SCREEN 8
  IF SEL_ITEM = 22
    THEN
      # Hacer cosas de oh, un papew!
      # Con EXTERN, o lo que sea, sonidos, tal.
      # yasta, ahora...
      SET ITEM SLOT_SELECTED = 0
      SET FLAG $CONTENTO = 1
    END
  END
```

Como véis, hemos quitado el objeto del inventario simplemente diciéndole al juego que ponga un 0 en el slot seleccionado.

¿No tendríamos que haber hecho más comprobaciones? Por ejemplo, que \$PAPEL = 1 o que \$CONTENTO = 0. Se podrían poner y seguiría funcionando, pero no son necesarias: ten en cuenta que, por un lado, el papel sólo se puede coger una vez, y que una vez dado al tío desaparecerá del inventario y, por tanto, del juego. Por tanto, SEL_ITEM nunca podrá volver a ser 22. ¡Ahorro!

"Floating Objects" de tipo "container".

No disponible en MK1

Los "Floating Objects" de tipo "container" no son más que "cajas" donde podemos meter un objeto. Estas cajas se colocan en una pantalla desde el script y se les puede asignar un contenido.

Los "Floating Objects" de tipo "container" (a partir de ahora, les vamos a llamar contenedores) hacen que manejar el inventario sea muy sencillo, ya que, cuando el usuario pulse ACCION tocando uno, harán que el objeto que haya en el slot seleccionado y el objeto contenido en el container se intercambien de forma automática, sin script de por medio. Eso nos sirve para:

- Si el slot seleccionado está vacío, "cogeremos el objeto".
- Si el slot seleccionado está lleno, el objeto que había se intercambiará por el del contenedor, y no se perderá nada.
- Si el contenedor está vacío pero no el slot seleccionado, "dejaremos el objeto".

Para poder usar contenedores, hay que habilitarlos en config.h. Además, hay que hacerle saber al motor qué flag de nuestro script representa el slot seleccionado (tal y como lo definimos en la sección ITEMSET de nuestro script):

```
#define ENABLE_FO_OBJECT_CONTAINERS
```

Con esto activamos los contenedores y le decimos al motor que el slot seleccionado se almacena en el flag 30.

Ah, ¡y que no se nos olvide habilitar los "floating objects", en general!

```
#define ENABLE_FLOATING_OBJECTS
```

Creando un contenedor

Los contenedores pueden crearse desde cualquier cuerpo de cláusula, pero se suelen crear en las secciones ENTERING en un IF TRUE.

A cada contenedor se le asigna un flag. El valor de dicho flag indicará qué objeto hay en el contenedor, y será 0 si está vacío.

Los contenedores se crean con el siguiente comando:

```
ADD_CONTAINER FLAG, X, Y    Crea un contenedor para el flag FLAG en (X, Y)
```

Es buena práctica crear un alias para cada contenedor y ponerles `CONT_`, para distinguirlos fácilmente. Por ejemplo, vamos a crear un contenedor para el papel del ejemplo anterior:

```
DEFALIAS
...
$CONT_PAPEL 16
...
END
```

El papel salía en la pantalla 6, así que vamos a crear ese contenedor en la sección `ENTERING SCREEN` de la pantalla 6. Recordad que antes teníamos que imprimir el tile y tal (ver sección anterior), pero con los contenedores no es necesario. Tampoco vamos a necesitar un flag `$PAPEL` ni pollas.

```
ENTERING SCREEN 6
IF TRUE
THEN
    ADD_CONTAINER $CONT_PAPEL, 5, 4
```

```
END
END
```

¿Pero dónde decimos que en ese contenedor tiene que estar el objeto que se representa por el tile 22 (la hoja de papel) - Recordemos que los contenedores son en realidad abstracciones de flags, así que para que el contenedor que acabamos de crear tenga una hoja de papel, habrá que darle ese valor al flag ¿Cuándo? Al principio del juego:

```
ENTERING_GAME
IF TRUE
THEN
    SET FLAG $CONT_PAPEL = 22
END
END
```

Con esto, en cada partida habrá un contenedor en (5, 4) de la pantalla 6 que tenga una hoja de papel dentro.

Cuando el jugador llegue a la pantalla 6, toque el contenedor (en (5, 4)) y pulse la tecla de acción, el motor intercambiará automáticamente el contenido del slot seleccionado con el contenido del contenedor. Así, si el slot seleccionado estaba vacío, pasará a contener la hoja de papel y el contenedor se quedará vacío. Si teníamos un objeto en ese slot, pasará a estar en la posición (5, 4) y el papel en nuestro inventario.

Y nosotros no tendremos que hacer nada. Ni en el script, ni en nada.

A la hora de irnos a la pantalla 8 a dárselo al jipi, todo sigue igual:

```
PRESS_FIRE_AT_SCREEN_8
IF SEL_ITEM = 22
THEN
    SET ITEM_SLOT_SELECTED = 0
    SET FLAG $CONTENTO = 1
END
END
```

Obviamente, los contenidos de los contenedores no tienen por qué crearse desde el principio del juego. Por ejemplo, en Leovigildo III los objetos para el puzzle final no están disponibles hasta que no hablamos con Nicanor el Aguador en la última pantalla. Inicialmente se ponen a 0 y llegados a ese punto ya se les da valor.

Otros floating objects

El motor soporta otros tipos de floating objects: cajas que se transportan, por ejemplo. Todo esto tiene que ver con la configuración del motor y tal, pero como se colocan desde el script, lo menciono

aquí.

Un floating object está representado por un número de tile y su ubicación inicial. Para crear un floating object tenemos que ejecutar el siguiente comando en el cuerpo de una cláusula (generalmente, en el IF TRUE dentro de una sección ENTERING SCREEN):

```
ADD_FLOATING_OBJECT T, X, Y
```

Crea un floating object con el tile T en (X, Y).

Como decíamos, el comportamiento de FO dependerá de tile que se le asigne y esto a su vez dependerá de cómo tengamos configurado el juego.

Por ejemplo, en Leovigildo las cajas que podemos acarrear y apilar son un floating object y están representadas por el tile 16. Para ello, hemos hecho la siguiente configuración en config.h:

```
#define ENABLE_FLOATING_OBJECTS
#define ENABLE_FO_CARRIABLE_BOXES
#define FT_CARRIABLE_BOXES 16
#define CARRIABLE_BOXES_ALTER_JUMP 180
```

La primera habilita los floating objects, en general. La segunda habilita los floating objects de tipo "CARRIABLE_BOXES" (cajas transportables). La tercera dice que estas cajas se representan por el tile 16. La última tiene que ver con la configuración de esta mierda: si se define, la fuerza del salto se verá alterada cuando llevemos una caja (para saltar menos) y el valor máximo de la velocidad vertical será el definido.

Con esto, añadiremos una de estas bonitas cajas a la pantalla 4 en la posición (5, 5) poniendo esto en el script:

```
ENTERING_SCREEN 4
    IF TRUE
    THEN
        ADD_FLOATING_OBJECT 16, 5, 5
    END
END
```

Comprobaciones y comandos relacionados con los valores del personaje

Existe todo un set de comprobaciones y comandos que tienen que ver con los valores del personaje (por ejemplo, la vida).

Comprobaciones

Estas dos comprobaciones están en desuso porque opino que es mucho más cómodo definir `#define OBJECT_COUNT` y asignarlo a una flag, y operar con dicha flag. Siguen aquí por yo qué sé.

<code>IF PLAYER_HAS_OBJECTS</code>	Evaluará a <code>CIERTO</code> si el jugador tiene objetos.
<code>IF OBJECT_COUNT = n</code>	Evaluará a <code>CIERTO</code> si el jugador tiene N objetos.

Comandos

<code>INC LIFE n</code>	Incrementa el valor de la vida en n
<code>DEC LIFE n</code>	Decrementa el valor de la vida en n
<code>RECHARGE</code>	Recarga toda la vida (la pone al máximo)
<code>FLICKER</code>	Hace que el jugador empiece a parpadear durante un segundo y pico, como cuando te quitan una vida.

Sobre estas dos últimas digo lo mismo que antes: usando `OBJECT_COUNT` y una flag todo es más sencillo, pero ahí siguen:

<code>INC OBJECTS n</code>	Añade n objetos más.
<code>DEC OBJECTS n</code>	Resta n objetos (si <code>objects >= n</code> ; si no <code>objects = 0</code>).

Terminar el juego

Comandos para terminar el juego desde el scripting (es necesario activar, en `config.h`, `#define WIN_CONDITION 2`, en el caso de que queramos GANAR desde el script - para GAME OVER No es necesario).

<code>GAME OVER</code>	Termina el juego con un <code>GAME OVER</code> .
<code>WIN GAME</code>	Termina el juego si no hay varios niveles. En juegos con varios niveles termina el nivel actual (y pasa al siguiente, si tu manejador de niveles funciona de esta manera)
<code>GAME_ENDING</code>	En juegos con varios niveles, termina el juego completamente y le dice al motor que muestre la secuencia final.

[Sólo en MK2] Para que `GAME_ENDING` funcione hay que indicarlo en `config.h` con un bonito `#define SCRIPTED_GAME_ENDING`.

Fire Zone

La "fire zone" de una pantalla es una zona rectangular definida a nivel de pixels que lanzará la sección `PRESS_FIRE` de la pantalla (y `PRESS_FIRE AT ANY`) si el jugador la toca. Nos sirve para lanzar trozos de script cuando el jugador toque algo o entre en algún sitio.

Hay que activarlas en `config.h`

```
#define ENABLE_FIRE_ZONE
```

Para definir el `FIRE_ZONE` activo de una pantalla usamos este comando desde cualquier sección de comandos (normalmente en un `IF TRUE` de `ENTERING SCREEN`)

```
SET_FIRE_ZONE x1, y1, x2, y2
```

Que definirá un rectángulo desde (x1, y1) a (x2, y2), en píxels.

Si quieres desactivar la "fire zone" sólo tienes que poner un rectángulo fuera de rango o vacío:

```
SET_FIRE_ZONE 0, 0, 0, 0
```

Para que sea menos coñazo trabajar, y ya que la mayoría de las veces las fire zones hay que calcularlas en base a un rango de tiles, msc3 entenderá el comando

```
SET_FIRE_ZONE_TILES tx1, ty1, tx2, ty2
```

donde los parámetros definen un rango en coordenadas de tile (ambos límites inclusive) que msc3 traducirá internamente a un `SET_FIRE_ZONE` normal.

Modificando enemigos

La modificación más sencilla es la que permite activar o desactivar enemigos. Los enemigos tienen, internamente, un flag que los activa o desactiva para, por ejemplo, matarlos y que no salgan más. Desde el script podemos modificar ese flag para conseguir varios efectos.

Por ejemplo, en Ninjajar! hacemos aparecer plataformas móviles como resultado de acciones más o menos complejas. Para ello, creamos la plataforma normalmente en esa pantalla, la desactivamos en el `ENTERING SCREEN`, y la activamos posteriormente cuando la acción necesaria ha sido ejecutada.

En las pantallas hay tres enemigos, numerados 0, 1 y 2. Estos números se corresponden con el orden en el que fueron colocados en el Colocador. Hay que tener esto en cuenta, porque necesitamos este número para referenciarlos:

<code>ENEMY n ON</code>	Activa el enemigo "n".
<code>ENEMY n OFF</code>	Desactiva el enemigo "n".

Otra cosa que podemos hacer con los enemigos es cambiar su tipo. En el tipo del enemigo, con el nuevo motor introducido en Leovigildo III, tenemos codificado el tipo de movimiento, si dispara o no, y el número de sprite (todo esto detallado en `whatsnew.txt`, no voy a detenerme aquí), así que con esto tenemos una herramienta bastante potente.

<code>ENEMY n TYPE t</code>	Establece el tipo "t" para el enemigo "n".
-----------------------------	--

A partir de aquí **no disponible en MK1**:

El problema es que estamos modificando un parámetro básico del enemigo. Si nuestro juego maneja varios niveles no importa, ya que siempre tenemos una copia (comprimida) de los valores originales que podemos restaurar cuando nos de la gana.

El problema viene en los juegos de un solo nivel. Si cambiamos el tipo de un enemigo, perderemos el tipo original para siempre. Para poder recuperarlo necesitaremos una "copia de seguridad" de los tipos de todos los enemigos. Esta copia ocupa 3 bytes por pantalla y hay que activarla desde `config.h`:

```
#define ENEMY_BACKUP
```

Con la copia de seguridad activada, podemos restaurar todos los enemigos a su valor original o sólo los de la pantalla actual:

<code>ENEMIES RESTORE</code>	Restablece a sus valores originales los enemigos de la pantalla actual.
<code>ENEMIES RESTORE ALL</code>	Restaura el tipo de TODOS los enemigos del nivel.

Si sólo necesitas restaurarlos al empezar cada partida, puedes pasar de usar `ENEMIES RESTORE ALL` en el script y activar la directiva `RESTORE_ON_INIT` en `config.h`

```
#define RUN_SCRIPT_ON_KILL
```

Ejecuta la sección `PLAYER_KILLS_ENEMY` el script siempre que el jugador mate un enemigo, sea como sea.

Código externo

Hay muchas cosas que no podemos hacer directamente desde el script y por ello el sistema permite ejecutar código externo, que no es más que una función definida en el código de MK2.

Para usar código externo habrá que incluir dicha función activando en config.h la directiva

```
#define ENABLE_EXTERN_CODE
```

Esto hará que se incluya el archivo extern.h en la compilación. Este archivo puede contener el código que nos de la gana siempre que el punto de entrada sea la función

```
void do_extern_action (unsigned char n);
```

En nuestro script disponemos del comando EXTERN:

EXTERN n	Hace una llamada a do_extern_action pasándole "n", donde n es un número de 0 a 255. No se puede usar construcciones #.
----------	--

En el extern.h de casi todos los juegos a partir de Ninjar verás código para imprimir textos comprimidos con textstuffer.exe e incluidos en un text.bin. A nosotros nos parece muy conveniente, pero siempre puedes usar esta característica para lo que te de la gana.

No disponible en MK1: Es posible que 256 posibles acciones externas sean pocas (por ejemplo, si vas a usar mucho texto en tu juego y necesitas hacer más cosas - Ninjar! usa 226 líneas de texto, casi nos quedamos sin valores). En ese caso podemos activar el "extern extendido" en config.h. Además de lo anterior, tienes que definir

```
#define EXTERN_E
```

Ojo, que esto desactiva EXTERN y no incluye el archivo extern.h. En su lugar incluye extern_e.h y activa el comando EXTERN_E. Ahora el punto de entrada, dentro de extern_e.h, debe ser

```
void do_extern_action (unsigned char n, unsigned char m);
```

y en nuestro script deberemos usar el comando:

EXTERN_E n, m	Hace una llamada a do_extern_action pasándole n y m, donde n y m son números de 0 a 255. No se puede usar construcciones #.
---------------	---

Safe Spot

No disponible en MK1

Si defines `#define DIE_AND_RESPAWN` en `config.h`, el jugador, al morir, va a reaparecer en el "último punto seguro". Si `DIE_AND_RESPAWN` está activo, el motor salva la posición (pantalla, x, y) cada vez que nos posamos sobre un tile no traspasable (que no sea un "floating object").

Podemos controlar la definición del "punto seguro" (safe spot) desde nuestro script. Si decidimos hacer esto (por ejemplo, para definir un "checkpoint" de forma manual), es conveniente desactivar que el motor almacene el safe spot de forma automática con:

```
#define DISABLE_AUTO_SAFE_SPOT
```

Hagamos o no hagamos esto, podemos definir el safe spot desde el script con estos dos comandos:

<code>SET SAFE HERE</code>	Establece el "safe spot" a la posición actual del jugador.
----------------------------	--

<code>SET SAFE n, x, y</code>	Establece el "safe spot" a la pantalla n en las coordenadas (de tile) (x, y).
-------------------------------	---

Comandos miscelaneos

No sabía donde meter estos...

<code>SOUND n</code>	Toca el sonido n. Dependerá de qué sonido sea n.
----------------------	--

<code>TEXT texto</code>	Imprime un texto en la línea de textos si la hemos definido en <code>config.h</code> con los <code>#define LINE_OF_TEXT</code> , <code>LINE_OF_TEXT_X</code> , y <code>LINE_OF_TEXT_ATTR</code> . El texto va a pelo, sin comillas, y en vez de espacios tienes que poner <code>"_"</code> . No usamos esto desde hace eones...
-------------------------	---

<code>PAUSE n</code>	Espera n frames, o sea, n = 50 es un segundo. Ojo pelao, que sólo funciona en 128K ya que usa HALT. Los juegos de 48K tienen las interrupciones apagadas por lo que usar PAUSE pausará el juego PARA SIEMPRE.
----------------------	---

<code>MUSIC n</code>	Toca la música n. Obviamente, sólo en juegos de 128K
----------------------	--

Code Injection Points

En este documento veremos para qué sirve cada punto de inyección de código de **MTE MK1** v5. Para obtener información sobre las API de **MTE MK1** y las variables globales consultar [la API](#).

General

extra_vars.h

Se incluye al final de `definitions.h` y permite definir las variables y constantes extra que necesitemos en nuestro código.

after_load.h

Se ejecuta una única vez tras terminar la carga y la inicialización del sistema. Puede emplearse para alguna pantalla tipo *splash screen* o inicializaciones propias.

before_game.h

Se ejecuta antes de empezar el juego, justo tras seleccionar control en la pantalla de título.

entering_game.h

Se ejecuta al empezar el bucle de juego, exactamente igual que la sección `ENTERING GAME` del script. Nótese que `before_game.h` se incluye *antes de empezar el juego* pero `entering_game.h` se incluye *antes de empezar cada nivel*.

before_entering_screen.h

Se ejecuta antes de hacer un cambio de pantalla, justo tras detectar que el jugador va a salir de la misma o, más generalmente, cuando `n_pant` y `o_pant` tienen valores diferentes. En este punto, y en condiciones normales, `o_pant` contendrá el número de la pantalla que estamos abandonando y `n_pant` el de la pantalla a la que nos dirigimos. Justo después de `before_entering_screen.h` se llamará a `draw_scr` y se inicializarán todos los enemigos.

entering_screen.h

Se ejecuta tras el cambio de pantalla, antes de volver al bucle principal, al igual que las secciones `ENTERING SCREEN` del script. En este punto la pantalla ya está en el buffer, los enemigos y hotspots están inicializados, etc.

extra_routines.h

Se ejecuta a cada vuelta del bucle de juego, justo al final, tras haber actualizado todos los actores, dibujado los cambios, etc. Puede usarse para realizar el tipo de tareas que se delegaban a `PRESS_FIRE` en el script pero con más flexibilidad

`after_game_loop.h`

Se ejecuta al salir del bucle principal del juego. Aquí podremos jugar con los valores de `level`, `success` y `script_result` si queremos implementar lógicas de niveles o quizá aprovechar para mostrar mensajes o secuencias de salida. [En 128K] la música está parada y la pantalla borrada.

Hotspots

`hotspots_custom.h`

Sirve para añadir nuevos tipos de hotspots. El código que añadas a este archivo se añadirá al `switch` que comprueba el tipo de hotspot cuando el jugador lo ha tocado, por lo que deberás añadir tu código en forma de nuevos `case` directamente:

```
case 7:
    // Mi implementación del hotspot de tipo 7
    beep_fx (7);
    break;

case 8:
    // Otro hotspot especial que añado
    ++ flags [8];
    break;
```

Recuerda que a partir del número 4 (inclusive) el hotspot N se dibuja con el tile 16+N. Tienes un ejemplo de hotspots personalizados en el postmortem del port a v5 del **Godkiller 2** original [aquí](#).

Módulo de enemigos

`enems_load.h`

Junto con `enems_move.h`, sirve para añadir nuevos tipos de enemigos. El código que añadas a este archivo se añadirá al `switch` que comprueba el tipo de enemigo por si hay que inicializar algún valor especial cada vez que se entra en una pantalla, por lo que deberás añadir tu código en forma de nuevos `case` directamente, por ejemplo:

```
case 5:
    // This enemy type has a fixed base frame:
    en_an_base_frame [enit] = 4;
    break;
```

enems_move.h

Junto con `enems_init.h`, sirve para añadir nuevos tipos de enemigos. El código que añadas a este archivo se añadirá al `switch` que comprueba el tipo de enemigo para saber cómo debe moverlo durante el bucle de juego, por lo que deberás añadir tu código en forma de nuevos `case` directamente, por ejemplo:

```
case 5:
    // static, idle, dummy enemy
    active = 1;
    break;
```

En este punto, `enit` es el número de enemigo, `enoffsmasi` su índice general dentro del array `malotes`, y sus valores estarán copiados en `_en_x`, `_en_y`, `_en_mx`, `_en_my`, `_en_x1`, `_en_y1`, `_en_x2`, `_en_y2`, `_en_t` y `_en_life`. Para que el enemigo sea interactuable (se pueda matar o colisionar) habrá que poner `active` a 1.

enems_extra_mods.h

Se incluye al final del bucle que carga los enemigos, tras haberles puesto todos los valores necesarios para su inicialización, y te permite modificar lo que necesites. Por ejemplo puedes hacer que los enemigos reinicien su posición inicial al entrar en cada pantalla añadiendo este código:

```
malotes [enoffsmasi].x = malotes [enoffsmasi].x1;
malotes [enoffsmasi].y = malotes [enoffsmasi].x2;
```

on_enems_killed.h

Se ejecuta cada vez que el jugador elimina un enemigo, al igual que la sección `PLAYER_KILLS_ENEMY` del script. En este punto, `enit` es el número de enemigo, `enoffsmasi` su índice general dentro del array `malotes`, etc.

Nótese que este CIP se incluye después de haber marcado al enemigo como *muerto*. La marca de *muerto* se hace levantándole el bit 4, o sea, haciéndole `|16`. Es por esto que si quieres comprobar el tipo del enemigo que acaba de matar tendrás que comparar `_en_t` con el número que quieras levantándole el bit 4. Por ejemplo, para ver si has matado al tipo 2:

```
if (_en_t == (2|16)) {
    [...]
}
```

¡No olvides los paréntesis!

enems_extra_actions.h

Se incluye al final del bucle de actualización del enemigo, y se ejecutará sólo si este enemigo tiene `active` a 1. El punto exacto de inclusión es justo tras comprobar la colisión con las balas (si se incluye esta característica). Puedes usarlo por ejemplo para comprobar la colisión con otros agentes custom, como un "hitter".

Módulo de jugador

`custom_veng.h`

Sirve para añadir un manejador custom para el eje vertical de movimiento del jugador. Esto tiene bastantes implicaciones y formas de usar. Se explicará en un tutorial próximamente.

`on_special_tile.h`

Se ejecuta cuando el jugador toca un tile cuyo comportamiento tiene el bit 7 levantado (cumple `& 128`). Cuando esto ocurre, `p_tx` y `p_ty`, que contienen el tile que toca el centro del sprite del jugador, indican precisamente las coordenadas de dicho tile.

Si tienes varios, puedes saber cuál es llamando a `qtile(p_tx, p_ty)` o consultando el valor de `map_buff[COORDS(p_tx, p_ty)]`. Puede servir para mil cosas, por ejemplo para implementar teletransportadores.

Colisiones con el fondo

Se definen cuatro puntos de inserción de código cuando ocurre una colisión contra el escenario que detiene al jugador:

```
bg_collision/obstacle_up.h
bg_collision/obstacle_down.h
bg_collision/obstacle_left.h
bg_collision/obstacle_right.h
```

El código se ejecuta justo ANTES de parar y posicionar bien al jugador. En este punto `p_vx` y `p_vy` aún tienen sus valores originales.

Tiles rompiscibles

En estos puntos de inyección de código el tile está en `(_x, _y)` (coordenadas de tile), y `gpaux` contiene `COORDS(_x, _y)` precalculado. `brk_buff[gpaux]` es el número de golpes que lleva el tile.

`on_wall_hit.h`

Se ejecuta siempre que golpeemos un tile rompible y aún le quede más energía.

on_wall_break.h

Se ejecuta siempre que golpeemos un tile rompible y desaparezca finalmente.

custom_enems_player_collision.h

Se ejecuta tras la detección de las plataformas, y antes de la colisión normal jugador-enemigos. Esto significa que si tu colisión hace cosas que evitan que el jugador se muera, deberás saltarte la comprobación estándar con un bonito:

```
goto player_enem_collision_done;
```

Miscelánea

on_tile_pushed.h

Se ejecuta cada vez que hemos empujado un bloque. En este punto podremos usar estas variables para obtener información sobre el bloque que se ha movido:

- `x0` , `y0` son las coordenadas donde estaba el bloque *antes* de moverse.
- `x1` , `y1` son las coordenadas adonde se ha movido el bloque.
- `rda` contiene el valor del tile que se ha cubierto con el bloque.

on_unlocked_bolt.h

Se ejecuta cada vez que el jugador abra un cerrojo. `p_keys` ya se ha decrementado en este punto. El cerrojo está en las coordenadas (`x0` , `y0`).