PIXEL SQUADRON

Miguel Francisco Quereda Rodríguez
Alejandro Garcia Gil
Ilyas Umalatov

Mecánicas principales

Nuestro juego es un shoot'em up inspirado en el famoso juego space invaders. Es un juego simple en el que el jugador controla una nave con las teclas izquierda y derecha, y dispara balas con el botón A.

Las balas van subiendo hacia arriba a velocidad de 1 píxel cada dos frames.

El objetivo de nuestro juego, es conseguir la máxima puntuación en el marcador eliminando a los enemigos que van apareciendo arriba del mapa antes de que lleguen a tocar el suelo.

Los enemigos aparecerán de tres en tres, y cada conforme se vayan eliminando, irán apareciendo más y más rápidos. Cada vez que eliminemos a un enemigo, obtendremos +10 puntos en el marcador.

Cada vez que se sumen 150 puntos en el marcador, es decir, cada 15 enemigos abatidos, se incrementará la velocidad de descenso de los enemigos. Aumentando así la dificultad del juego conforme va avanzando la partida.

En caso de que el enemigo llegue abajo del todo, el juego se reiniciará inmediatamente volviendo a poner a 0 el contador.

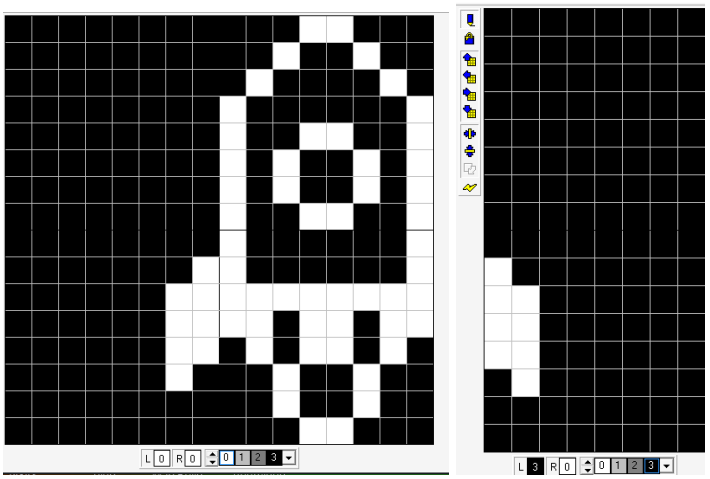
Ahora vamos a explicar parte por parte como hemos implementado este juego y sus distintos tiles creados, mapa, ficheros y características.

Tileset y recursos gráficos:

Todos los tiles que hemos creado para diseñar este juego los hemos creado en la aplicación Tile Designer.

Los tiles que hemos creado para este proyecto, han sido:

- **Nave del player:**

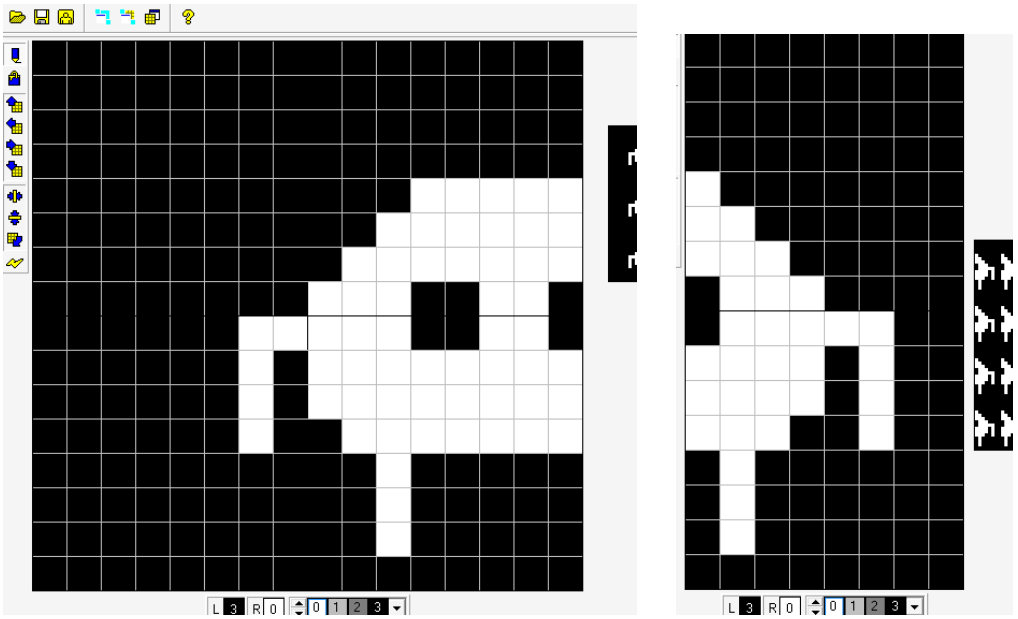


Lo que hemos hecho para crear tanto la nave como el enemigo que ahora mostraremos, ha sido crearla por partes, la nave está compuesta por 6 tiles de 8x8 píxeles, dispuestos en una malla de 3x2, lo que le da un tamaño total de 24x16 px.

La idea de construir la nave 3x2 es para que al crear las balas cuando pulsemos el botón de disparo, las balas se creen justo encima de la nave:

```
ld   a, [$C000]
ld   e, a
ld   a, [$C001]
ld   d, a
inc  e ; centramos en jugador
dec  d ; subimos una fila
bit  7, d
jr   z, .y_valida    "valida":
ld   d, 0
```

- **Tile del enemigo:**

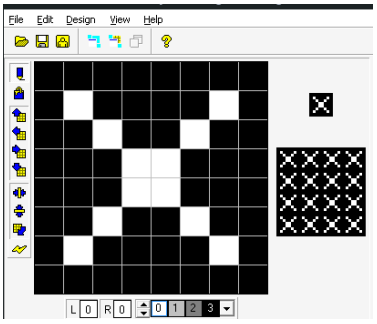


Al igual que con la nave del jugador, hemos creado al enemigo en 6 tiles de 8x8 píxeles, dispuestos en una malla de 3x2, es decir, 24x16 píxeles. Esto es debido a que la idea principal era que disparara balas desde el centro. Aunque más tarde hemos visto más óptimo que el enemigo descendiera hasta chocar abajo, como en el juego Space Invaders.

- **Contador de puntuación:**

Para crear el contador de puntuación hemos creado 10 tiles de 8x8 píxeles, cada uno con un número del 1 al 9.
El contador se dibuja con 4 sprites de 8x8 en OAM , es decir, millares, centenas, decenas y unidades).

- **Estrellas**:



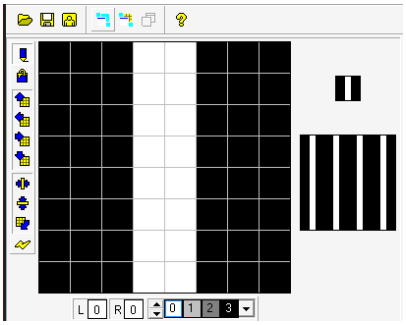
Para ambientar el escenario hemos dibujado un tile de estrella de 8x8 px y lo reutilizamos como sprite OAM en varias posiciones alrededor de la pantalla.

Cada estrella ocupa una entrada de OAM de 4 bytes, con coordenadas, tile y atributos.

La idea de que poner tanto el contador de puntuación como las estrellas se pinten como sprites en la OAM ha sido debido a que las balas se representan escribiendo y borrando tiles en la VRAM. Por tanto si las balas los pisaran desaparecerían.

En la OAM los sprites se renderizan encima del BG y no se ven afectados por las modificaciones del fondo.

- **Bala**



Para crear la bala hemos hecho un tile de 8x8 con una franja vertical en el centro. La bala la copiamos en VRAM al iniciar el juego y aparecerá cuando el jugador dispare, después irá subiendo filas hasta llegar arriba del mapa o chocarse con un enemigo.

La bala no es un sprite, la dibujamos en el BG map con el índice $01, siempre durante V-Blank.

Para mover la bala, el procedimiento que hemos hecho es el siguiente:

Primero se borra la celda anterior del BG escribiendo tile 0 (pintamos de negro)

```
xor  a
ld  [hl+], a
ld  [hl+], a
ld  [hl+], a
ld  [hl],  a
```

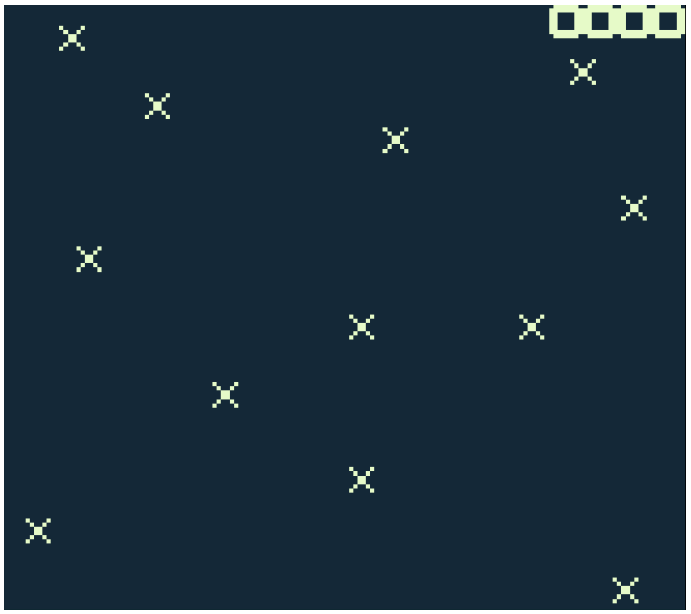
Actualizamos la coordenada de la bala

```
.no_en_borde_1:
    dec  a
    ld   d, a
    ld  [hl], d
    push de
```

Pintamos el tile de la bala en la nueva celda

```
    call wait_vblank    "vbl
    ld   a, TILE_BALA
    ld  [hl], a
    pop  de
    pop  bc
```

- **Mapa**:



Para crear el mapa negro, hemos definido el tile 00 como un tile sólido negro. Así cualquier celda del mapa con valor 0 muestra negro. Esto nos permite borrar escribiendo simplemente 0 en el BG.

Cuando reiniciamos el juego, rellenamos todo el mapa con ceros, dejando el fondo negro.

```
reset_game:
    ; limpia mapa BG
    ld   hl, BG_MAP0
    ld   de, 32*18
    xor  a
.clear_screen:
    call wait_vblank
    ld   b, 32
.clear_row:
    ld   a, $00
    ld  [hl+], a
    dec  b
    jr  nz, .clear_row
    dec  de
    ld   a, d
    or   e
    ld   a, 0
    jr   nz, .clear_screen
```

Sobre el mapa tenemos:

El jugador: la nave compuesta por un bloque de 3x2 tiles dibujada en el BG.
Los enemigos: cada alien es un bloque 3x2 tiles en el BG
Las balas: Un tile que se desplaza hacia arriba por el BG

Y también tenemos las estrellas y el marcador, que al ir en sprites en OAM, no se ven afectados por las escrituras de las balas sobre el BG.

Para pintar y mover todas estas entidades hemos utilizado diversas técnicas:

```
    ld   hl, naveEspacial    "Espacial": Unknown
    ld   de, $8000 + TILE_BYTES*$20
    ld   b,  96
    call copy_tiles

    ld   hl, enemigo    "enemigo": Unknown word.
    ld   de, $8000 + TILE_BYTES*26
    ld   b,  96
    call copy_tiles

    ld   hl, estrella    "estrella": Unknown wor
    ld   de, $8000 + TILE_BYTES*40
    ld   b,  16
    call copy_tiles_invertidoColor    "invertido

    ld   hl, numeros    "numeros": Unknown word.
    ld   de, $8000 + TILE_BYTES*$30
    ld   b,  10*16
    call copy_tiles_invertidoColor    "invertido
```

```
copy_tiles:
    ld   a, [hl+]
    ld  [de], a
    inc  de
    dec  b
    ld   a, b
    cp   0
    jr   nz, copy_tiles
    ret
```

Como podemos ver en este bloque de código hemos cargado todo en la VRAM para luego utilizarlo simplemente cargando los tiles en hl y pasandolos a VRAM con un bucle llamado copy_tiles. Nos hemos encargado de darle un número de tile distinto a cada uno para que no haya solapamiento ni problemas.

Las zonas en la WRAM que usamos en nuestro código para guardar las entidades son:

Para guardar el jugador usamos el primer slot, de la $C000 a la $C003, donde guardamos [X, Y, TILE, ATRIBUTOS], el mismo patrón que en todas las demás entidades.

```
ecs_init_player:
    ; reserva el primer slot para el jugador    "r
    call man_entity_alloc
    ; posición inicial del jugador    "posición":
    ld   a, 9                    ; X
    ld  [hl+], a
    ld   a, 16                   ; Y
    ld  [hl+], a
    ; tile base del jugador    "jugador": Unknown
    ld   a, $20
    ld  [hl+], a
    ; tipo/flags (0)    "tipo": Unknown word.
    xor  a
    ld  [hl], a
    ret
```

Para las balas usamos los 9 slots siguientes, de la $C004 a la $C027.

Usamos un bucle que usa c desde 4 hasta <40 en pasos de 4, y acaba cuando c==40. 40 hace de centinela y se resetea otra vez a 4.

```
1 reference
crear_bala_desde_jugador:    "crear": Unknown wo
    ld   a, [next_free_entity]
    cp   40
    jr   nz, .puntero_ok    "puntero": Unknown w
    ld   a, 4
    ld  [next_free_entity], a
.puntero_ok:    "puntero": Unknown word.
    ld   a, [$C000]
    ld   e, a
    ld   a, [$C001]
    ld   d, a
    inc  e ; centramos en jugador    "centramos"
    dec  d ; subimos una fila    "subimos": Unkn
    bit  7, d
    jr   z, .v_valida    "valida": Unknown word.
```

A los enemigos les hemos reservado de la posición $C050 en WRAM hasta la $C09F, 80 bytes, es decir, 20 slots de 4 bytes para guardar hasta 20 enemigos.

Un slot se considera enemigo activo si ATTR == $01.
Igual que con las balas, hacemos un recorrido en bucle. Iteramos con c desde 80 hasta <160 en saltos de 4 en 4.

```
.loop_slots:
    ld   a, c
    cp   160
    jr   z, .carril_libre
    ld   h, $C0
    ld   l, c
    push hl
    inc  hl
    inc  hl
    inc  hl
    ld   a, [hl]
    cp   ATTR_ENEMIGO
    pop  hl
    jr   nz, .next_slot
    ld   a, [hl]
```

Para dibujar el jugador, copiamos en A los datos de la WRAM.
Después guardamos esos datos en DE y A y utilizamos la función pintar_bloque_3x2

```
1 reference
dibujaJugador:    "dibuja": Unknown word.
    ld   a, [SLOT0_X]
    ld   e, a
    ld   a, [SLOT0_Y]
    ld   d, a
    ld   a, [SLOT0_BASE]
    jp   pintar_bloque_3x2_desde_xy_con_base
63 references
ret
```

Esta función calcula la dirección en BG , escribe los 3 tiles en la fila superior y salta a la fila inferior para escribir los 3 restantes. También hay una función parecida que hace el mismo procedimiento pero encargada de borrar, es decir, pinta $00 (tile negro).

Para el enemigo, primero tenemos un sistema de respawn:

Primero tenemos un timer de spawn que lo inicializamos a 60. Y cuando llega a 0 lo reiniciamos y llamamos a una función que comprueba si faltan enemigos en pantalla. Ya que el máximo que podemos tener juntos son 3.

```
.check_spawn:
    ; gestiona temporizador de spawn y cre
    ld   a, [enemy_spawn_timer]
    dec  a
    ld  [enemy_spawn_timer], a
    jr   nz, .end
    ld   a, 60
    ld  [enemy_spawn_timer], a
    call spawn_enemigo_si_falta
.end:
    ret
```

Esta función no crea enemigos si ya hay 3 activos.
Si hay menos de 3 enemigos, elegimos una columna fija del mapa aleatoria donde respawnear el enemigo y comprobamos si está libre, en ese caso creamos el enemigo en esa columna y lo ponemos en un slot libre.

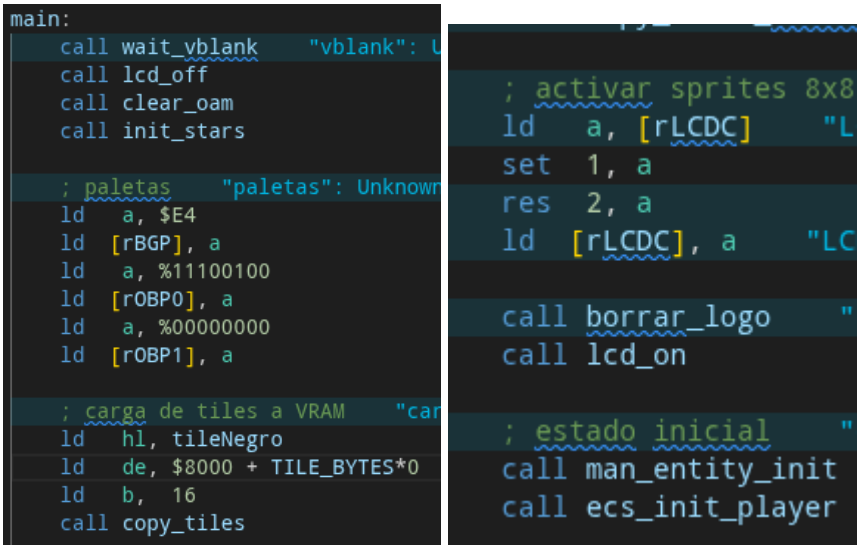
Los slots se van reciclando y cada vez que mueren se ponen a 0 y se pueden volver a utilizar.

FICHEROS .ASM

A continuación vamos a describir cómo hemos organizado el código del juego:

- **main.asm**

El main es el punto de entrada y el orquestador del juego. Primero se apaga el LCD para cargar los gráficos en VRAM y preparar la OAM.



Se configuran paletas, se inician entidades, y llegamos al bucle principal que se repite cada frame.

En cada iteración del bucle principal, se lee y aplica el input del jugador, para moverlo en caso de que presionemos las flechas.
Se actualizan las balas, es decir, se van moviendo para arriba a velocidad de 1 píxel cada 2 frames. Se actualiza la lógica de enemigos, se decrementan los temporizadores, se ajusta la velocidad en función de la puntuación y realiza el respawn del enemigo cuando la verificación da true.

Con el estado ya actualizado dibuja los enemigos en el BG en sus nuevas posiciones.

Comprueba la condición de final de la partida, si algún enemigo alcanza la última fila del mapa. Y en ese caso limpia el fondo BG a negro, reinicia la WRAM, pone el marcador a 0 y vuelve a inicializar el juego.

```
reset_game:
    ; limpia mapa BG
    ld   hl, BG_MAP0
    ld   de, 32*18
    xor  a
.clear_screen:
    call wait_vblank
    ld   b, 32
.clear_row:
    ld   a, $00
    ld  [hl+], a
    dec  b
    jr  nz, .clear_row
    dec  de
    ld   a, d
    or   e
    ld   a, 0
    jr   nz, .clear_screen
```

```
.clear_enemies:
    ld   [hl+], a
    dec  b
    jr   nz, .clear_enemies

    ; marcador a cero
    call init_score

    ; variables de control
    xor  a
    ld  [disparo_cd], a
    ld  [balas_tick], a
    ld  [a_lock], a

    ; vuelve al arranque de escena (ca
    jp inicializar_juego
```

Por último refresca el HUD en OAM actualizados el marcador dígito a dígito

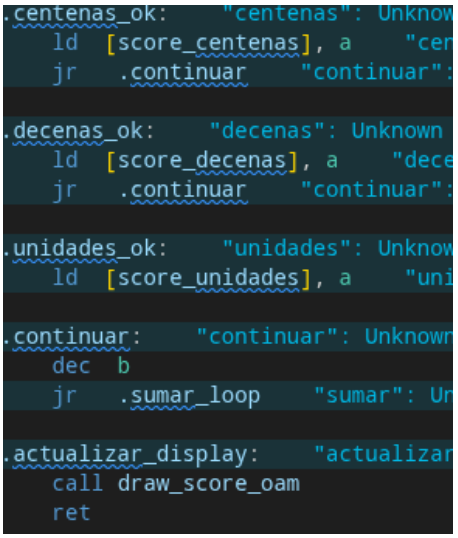
```
call draw_score_oam
```

Ahora explicaremos cómo funciona esta función ya que se encuentra en el fichero score.asm.

- **score.asm**

En este fichero guardamos la lógica para crear el marcador e ir actualizando según se vayan eliminando enemigos.

Primero tenemos la función de iniciar el marcador, donde ponemos a 0 los 4 dígitos. También tenemos la función add_score que nos sirve para incrementar puntos, es decir, recibe en A cuántos puntos debe sumar, que deberían ser 10. E incrementa unidades con acarreo. Al terminar llama a draw_score para refrescar.



draw_score actualiza el HUD escribiendo en OAM los 4 sprites del marcador.
Por cada dígito coloca Y/X fijas y el tile correspondiente, es decir, hace TILE_0 + dígito. Así refresca la puntuación sin provocar parpadeos en pantalla

- **entity_manager.asm**

Para crear este fichero nos hemos basado en implementación que se hizo en la sesión grabada en clase.

Este fichero está compuesto por tres funciones:

man_entity_init: Utiliza la función memset_256 para poner a 0 los 160 bytes de component_sprite y resetea a 0 las variables de siguiente entidad y número de entidades vivas. Se usa tanto para iniciar el juego como para reiniciarlo.

man_entity_alloc: Sirve para devolver el slot que se utilizará y guardar el siguiente slot que estará libre.

ecs_init_player: Llama a man_entity_alloc para reservar el primer slot. Escribe la posición y el tile base del jugador, y iphone los flags a 0.

- **utils.asm**

En este fichero tenemos guardadas unas cuantas funciones útiles y necesarias para el funcionamiento del juego, entre ellas se encuentran:

- wait_vblank: Espera a V-Blank leyendo rLY y comparando con 144. No continúa hasta que la PPU entra en V-Blank y asegurarnos que no dañamos la pantalla.

```
wait_vblank:    "vblan
    ld  a, [rLY]
    cp  VBLANK_LINE
    jr  c, wait_vblank
    ret
```

- leer_botones y leer_dpad: Seleccionamos el DPAD o los botones con 20 o 10 y dejamos el valor en a.

- lcd_off y lcd_on Para apagar y encender el LCD modificando el bit 7 de rLCDC. Para apagar cuando hacemos un gran pintado en VRAM.

- mem_set: Para rellenar memoria desde HL con el byte A, B veces, se suele usar para poner a 0 las entidades.
- memcpy_256: Al contrario que mem_set sirve para copiar B bytes desde HL a DE (destino).

- copy_tiles: Copia B bytes desde HL a DE (VRAM). Lo usamos para cargar tiles del fondo, del jugador, bala, enemigo a la VRAM.
  También hemos creado una función copy_tiles_invertidoColor que tiene el mismo funcionamiento que copy_tiles pero hace CPL a cada byte antes de escribirlo. Nos ha servido para invertir el color facilmente de algunos tiles sin tener que volver a redibujarlos.

```
copy_tiles_invertidoColor:
.cti_loop:
    ld   a,[hl+]
    cpl
    ld  [de],a
    inc  de
    dec  b
    jr   nz,.cti_loop
    ret
```

```
copy_tiles:
    ld   a, [hl+]
    ld  [de], a
    inc  de
    dec  b
    ld   a, b
    cp   0
    jr   nz, copy_tiles
    ret
```

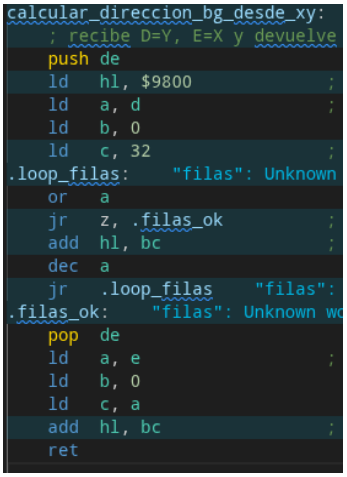
- clear_oam: Limpia toda la OAM poniendo 0 en cada byte

- Funciones explicadas antes de borrar y dibujar bloques

- **funcionesMovimiento.asm**

El objetivo de este fichero es controlar las posiciones y movimientos del jugador.

Tiene varias funciones como calcular_direccion_bg_desde_xy, que nos convierte las coordenadas de tile en una direccion del BG map. Parte de $9800, el inicio del mapa y suma Y filas de 32 celdas y X columnas para dejar HL apuntando a la celda (x,y).



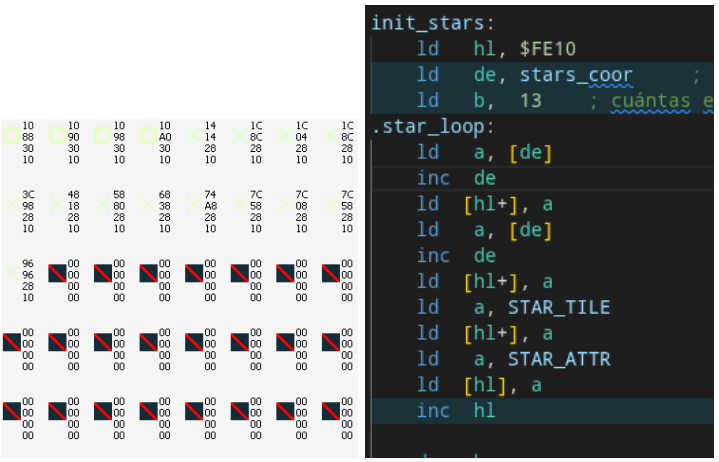
También tiene leer y escribir el slot 0 de la tabla de entidades, funciones que usamos para mover el jugador.

Y por último mover_jugador, que junta toda la lógica. Primero lee la posición con leer_slot0 y borra el bloque 3x2 del jugador. Luego ajusta X, la guarda con escribir_slot0 y finalmente pinta el jugador llamando a la función correspondiente llamada pintar_bloque_3x2_desde_xy_con_base usando como tile base la almacenada en $C002

- **stars.asm**

Stars.asm es un fichero que hemos creado aparte para dibujar las estrellas que complementan al mapa en OAM. Tenemos una función init_stars que recorre la tabla de coordenadas y las va metiendo en OAM mediante un bucle.

Empezamos en $FE10 que es la posición que tenemos libre después de haber puesto el marcador. Se pintan con el tile $28 que es donde hemos guardado el tile en VRAM

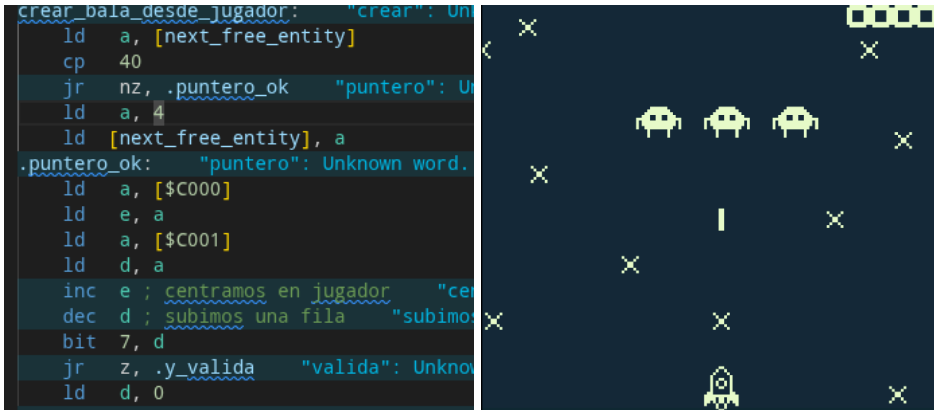


```
init_stars:
    ld   hl, $FE10
    ld   de, stars_coor     ;
    ld   b,  13     ; cuántas e
.star_loop:
    ld   a, [de]
    inc  de
    ld  [hl+], a
    ld   a, [de]
    inc  de
    ld  [hl+], a
    ld   a, STAR_TILE
    ld  [hl+], a
    ld   a, STAR_ATTR
    ld  [hl], a
    inc  hl
```

- **balas.asm**

Aquí hemos implementado todo lo relacionado con la bala, creación, movimiento y borrado en BG.

Primero hemos creado la bala con la función crear_bala_desde_jugador.



```
crear_bala_desde_jugador:    "crear": Un
    ld   a, [next_free_entity]
    cp   40
    jr   nz, .puntero_ok    "puntero": U
    ld   a, 4
    ld  [next_free_entity], a
.puntero_ok:    "puntero": Unknown word.
    ld   a, [$C000]
    ld   e, a
    ld   a, [$C001]
    ld   d, a
    inc  e ; centramos en jugador    "ce
    dec  d ; subimos una fila    "subimo
    bit  7, d
    jr   z, .y_valida    "valida": Unkno
    ld   d, 0
```

Lo que hacemos en este fragmento de código es al pulsar A, tomamos la posición del jugador , incrementamos X en una para que salga en el centro es decir, arriba del medio del player, y subimos una fila decrementando Y con dec d.

Para mover las balas primero llamamos a revisar_colisiones_balas_enemigos. Si no hay balas termina.
Solo mueve balas cuando balas_tick == SPEED_BALAS, Así la bala sube 1 fila cada 2 frames.

Usamos next_free_entity para recorrer las balas activas, para cada bala, tenemos funciones de borrar, mover y pintar.

Borrar: Calculamos la celda de BG donde se encuentra gracias a sus coordenadas X e Y y escribimos el tile 0 (negro)

Mover: si Y==0, significa que la bala ha salido, ponemos sus 4 bytes a 0, si no, restamos 1 a la coordenada de la fila y la pintamos.

Pintar: Con las nuevas coordenadas volvemos a calcular la celda y escribimos el tile de la bala.

- **enemies.asm**

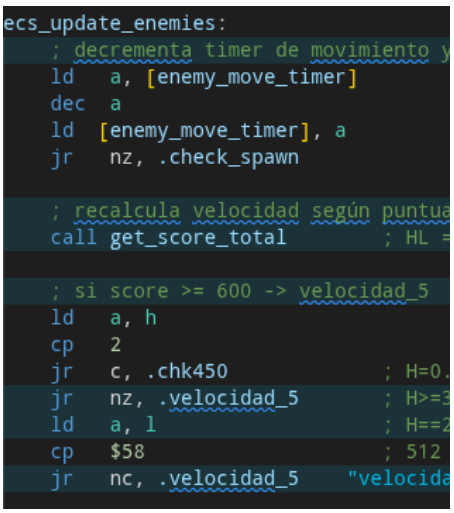
Este fichero contiene toda la lógica del enemigo, la creación, inicialización, actualización y sistema de movimiento.

Creamos un enemigo con la función crear_enemigo_entidad, que escanea desde 80 hasta 160 buscando un slot libre. Cuando lo encuentra escribe las coordenadas, el tile y los atributos.

```
crear_enemigo_entidad:    "crear"
    ; busca un slot libre en [80.
    ld   c, 80
.buscar_slot:    "buscar": Unknow
    ld   a, c
    cp   160
    ret  z                    ;
    ld   h, $C0
    ld   l, c
    ld   a, [hl]              ;
    or   a
    jr   z, .slot_encontrado    "
    ld   a, c
    add  a, 4                 ;
    ld   c, a
    jr   .buscar_slot    "buscar"
```

```
.slot_encontrado:    "encontra
    ld   a, e
    ld  [hl+], a
    ld   a, d
    ld  [hl+], a
    ld   a, ENEMY_TL
    ld  [hl+], a
    ld   a, ATTR_ENEMIGO
    ld  [hl], a
    ret
```

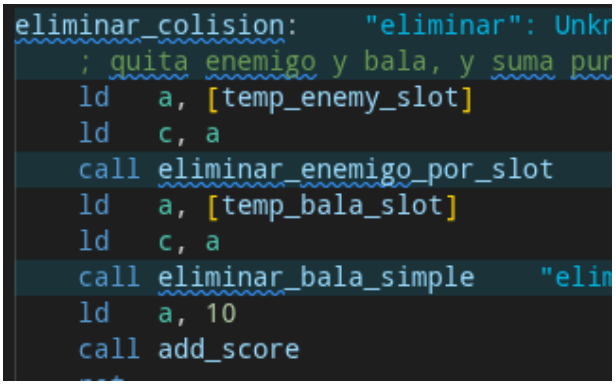
Inicializamos los enemigos y con ecs_update_enemies los vamos actualizando cada frame. Calcula la velocidad según la puntuación total con 5 umbrales, de 150 a 600, cambiando el umbral cada 150 puntos.



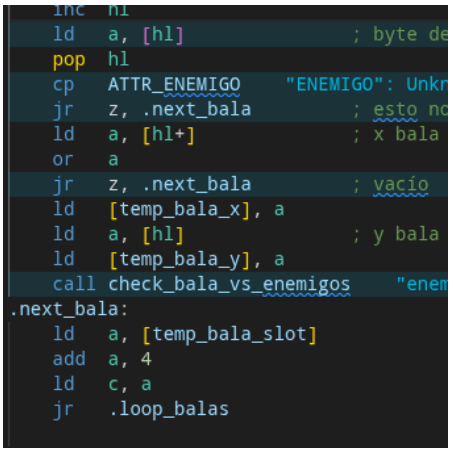
Para mover los enemigos, recorremos todos los slots de enemigos, si encontramos un enemigo activo, es decir ATTR=$01 y X!=0, borramos su bloque, cambiamos las coordenadas, y pintamos el enemigo, el mismo procedimiento que hacemos con el jugador.

- **collisions.asm**

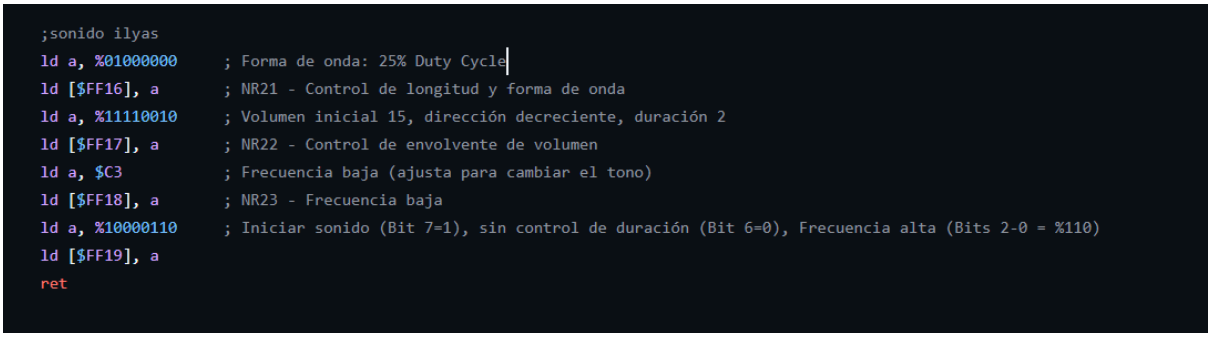
Este fichero maneja las colisiones de la bala con el enemigo. Recorre todas las balas activas, y las compara con todos los enemigos mediante los bucles . Si hay impacto borra ambos y suma 10 puntos al marcador, como podemos ver en el bloque de código.



revisar_colisiones_balas_enemigos recorre todas las balas activas en los slots de la VRAM, del 4 al 27. Para cada slot fliltra que contenga una bala, viendo que no es un enemigo con atributo $01. Guarda las coordenadas de la bala en temporales y comprueba si esa bala colisiona con algún enemigo con la función check_bala_vs_enemigos.



MÚSICA:



El primer bloque de código es el que se encarga de crear el sonido de "disparo" justo cuando se genera una bala. Para hacerlo, primero define la "personalidad" o el timbre del sonido, eligiendo una onda digital específica en el registro $FF16 que le da ese toque "retro" característico.

Inmediatamente después, configura la parte más importante: el efecto "pew" en el registro $FF17. Esto es un truco muy común: le da la orden al chip de sonido de empezar al volumen máximo, pero que, en cuanto suene, empiece a bajar el volumen muy rápidamente.

Al mismo tiempo, se ajusta el tono del sonido en los registros $FF18 y $FF19 para que suene agudo. La última instrucción, que carga el valor %10000110 en $FF19, sirve para que suene ya. El Bit 6 de ese mismo valor le indica que ignore cualquier duración fija y que simplemente deje que el sonido dure lo que tarda en apagarse el volumen. Esto asegura que el sonido tenga un corte limpio y rápido.

En resumen, este primer código configura un tono agudo, le da una forma de disparo que se apaga velozmente y, finalmente, aprieta el gatillo para que suene.

```
eliminar_enemigo_por_slot:
    ld   a, %10000000
    ld   [$FF16], a
    ld   a, %11110011
    ld   [$FF17], a
    ld   a, $64
    ld   [$FF18], a
    ld   a, %10000000
    ld   [$FF19], a
    ret
```

La segunda parte del código se encarga de emitir un sonido al matar a un enemigo, el funcionamiento es el mismo a la primera parte del codigo, la unica diferencia es el sonido emitido, consiguiendo una más grave simulando una explosion

```
play_point_sound:
    push af
    ld   a, %10000000  ; $FF11: Duración (corta) y patrón de onda (12.5%)
    ld   [$FF11], a
    ld   a, %11110011  ; $FF12: Envolvente de volumen (empieza alto, sube, rápido)
    ld   [$FF12], a
    ld   a, $00        ; $FF13: Frecuencia (bits bajos)
    ld   [$FF13], a
    ld   a, %10000110  ; $FF14: Frecuencia (bits altos) y Trigger (bit 7)
    ld   [$FF14], a
    pop  af
    ret
```

```
; centenas++
ld   a, [score_centenas]
inc  a
call play_point_sound
cp   10
jr   c, .centenas_ok
xor  a
ld  [score_centenas], a
```

El tercer bloque de código nos permite emitir un sonido en el momento de obtener 100 puntos, esto se consigue haciendo uso del mismo código pero cambiando la frecuencia para emitir un sonido más agudo. En este caso se creó una subrutina para mantener el código más limpio. También ha sido necesario añadir push af y pop af debido a que en la función ya se hacía uso del registro a.

```
;sonido ilyas

ld a, $FF       ; Activa todos los canales y el volumen máximo
ld [$FF26], a   ; NR52 - Control maestro de sonido (Power ON)
ld a, $77       ; Volumen máximo para ambos altavoces (izquierdo y derecho)
ld [$FF25], a           ;  dibuja al jugador
```

El tercer bloque de código no crea ningún sonido por sí mismo. Su función es preparar el hardware global de la consola para asegurarse de que el disparo que creamos antes se pueda oír.
La primera instrucción, que se escribe en $FF26, actúa como el interruptor de encendido principal de todo el sistema de sonido. Sin esta línea, la consola estaría completamente en silencio, aunque intentáramos reproducir el disparo.
La segunda instrucción, que se escribe en $FF25, funciona como una pequeña mesa de mezclas. Le dice a la consola por dónde debe salir el sonido. El valor $77 que usamos le indica que envíe la señal de nuestro disparo (que está en el Canal 2) a los altavoces al mismo tiempo. Esto asegura que el sonido se escuche centrado y completo, en lugar de sonar solo por un lado de los auriculares.

Por lo tanto, este tercer bloque simplemente enciende el sistema de sonido y conecta los altavoces de forma equilibrada, permitiendo que el efecto de disparo se escuche perfectamente.