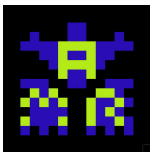
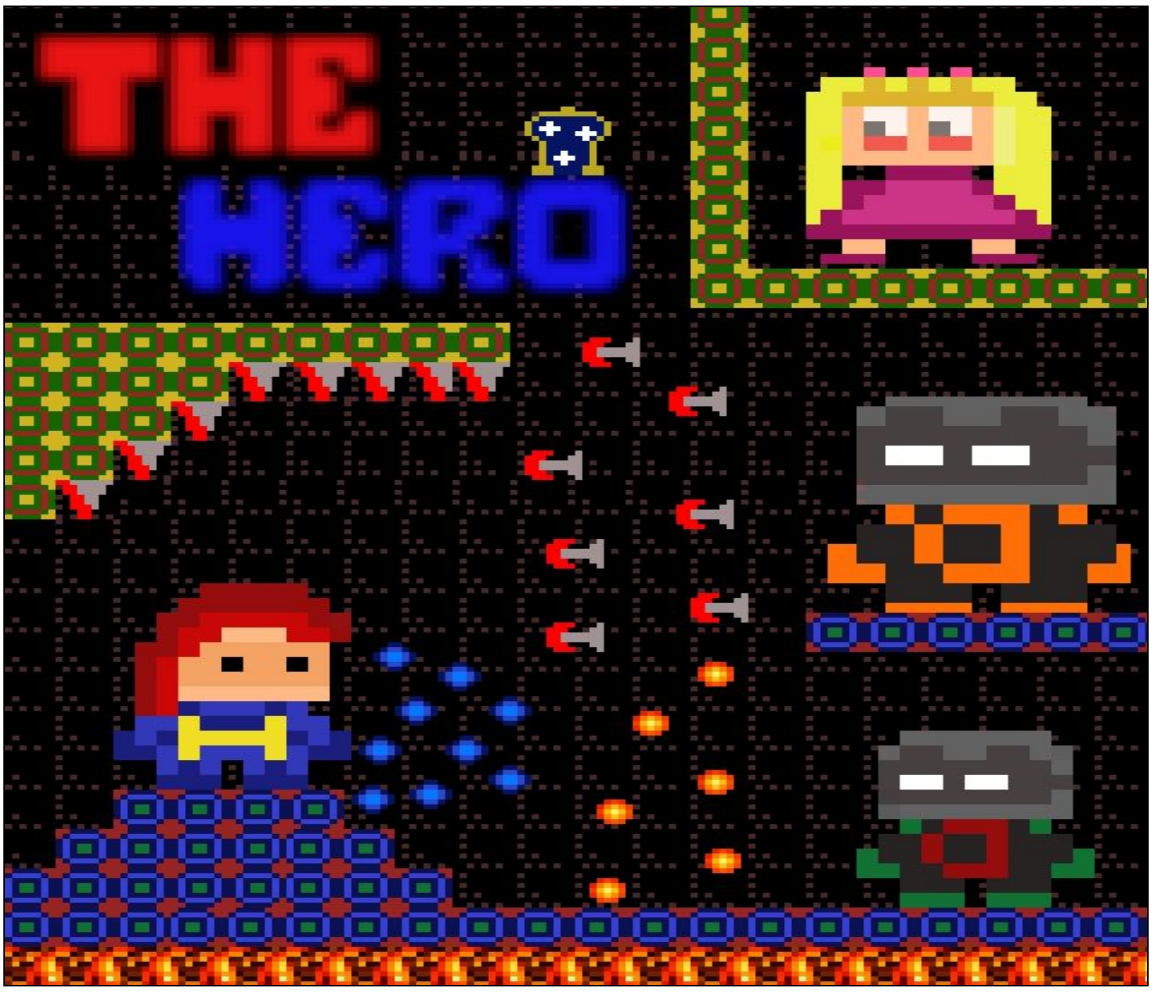
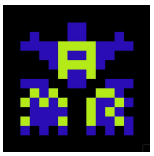


Universitat d'Alacant
Universidad de Alicante

Memoria Videojuego Amstrad CPC 464



Andrés David Rojas Carrillo
Mario Villora Picó
Roberto Ruiz Uclés



Universitat d'Alacant
Universidad de Alicante

Índice

Introducción al videojuego

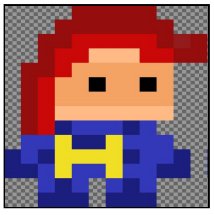
El videojuego "*TheHero*" trata sobre un héroe, el cual debe de rescatar a la princesa que ha sido secuestrada por los ladrones. En su aventura, deberá superar diversos obstáculos tales como pinchos, lava, plataformas, enemigos, flechas, etc. Todo esto mientras recorre un sin fin de niveles con el objetivo de rescatar a la princesa.

Para jugar a este juego, el jugador deberá emplear las siguientes teclas o joysticks:

- Q: Tecla para saltar.
- P: Tecla para desplazarse a la izquierda.
- O: Tecla para desplazarse a la derecha.
- Space: Tecla para disparar.
- Escape: Tecla para acceder al menu pause.
- R: Tecla para reiniciar el juego una vez esté en el menú pause.
- M: Mutear/Desmutear música.
- 1: Start game
- 2: Controls
- 3: Credits

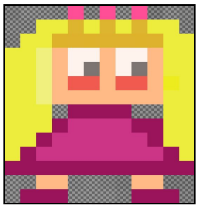
En nuestro videojuego encontrarás los siguientes personajes:

Walker



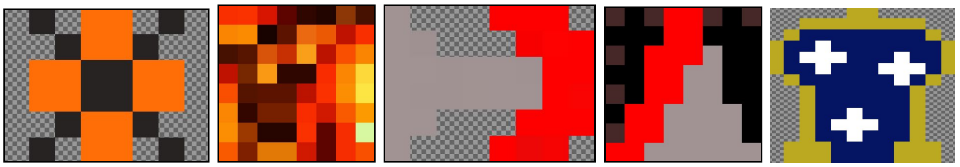
TheHero

Shooter



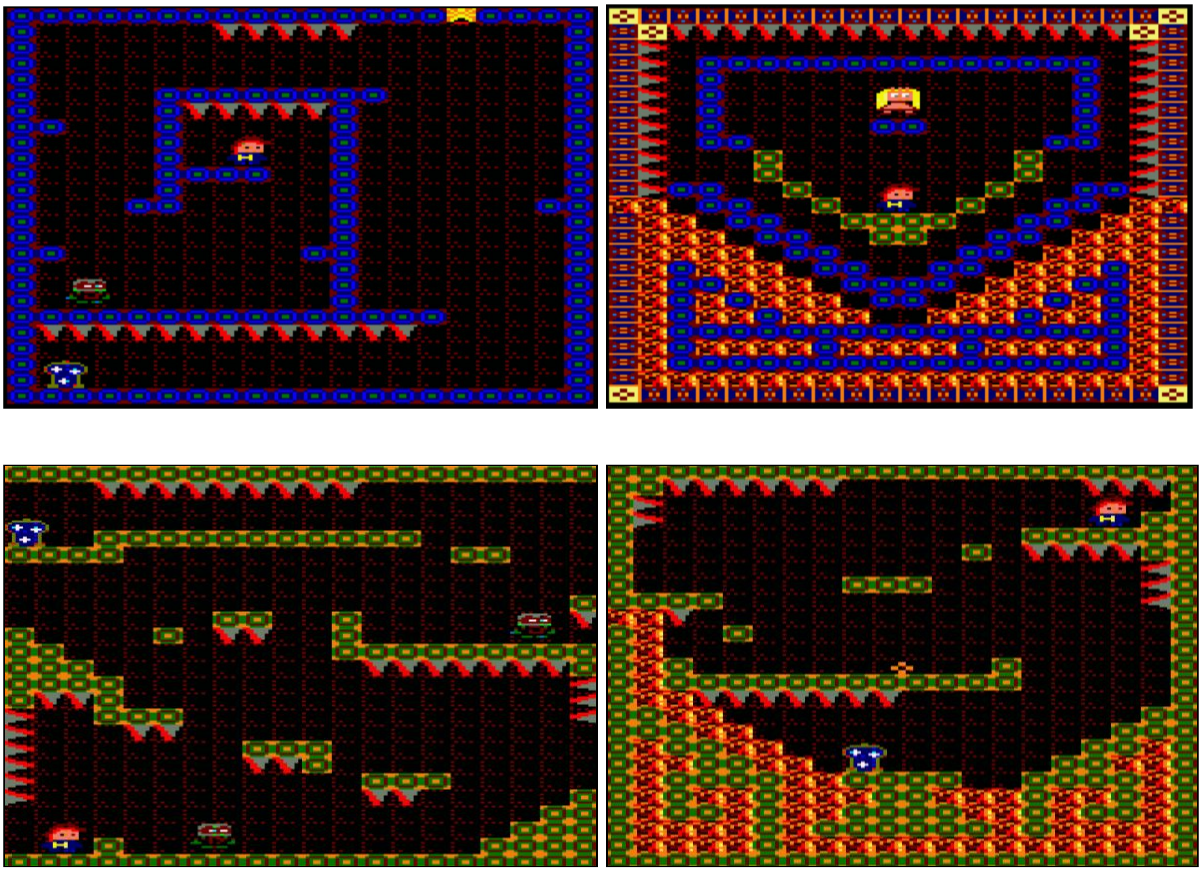
Princess

También encontrarás diversos obstáculos a lo largo del mapa:



Shuriken Lava Flecha Pinchos Portal

Nuestro personaje deberá lidiar con los diferentes recorridos repartidos en diversos mapas tales como los siguientes:



El cometido de este diseño de niveles es el de enseñar poco a poco al jugador a desenvolverse en los mapas mediante la inclusión de pinchos, lava y enemigos conforme va avanzando por el juego.

Explicaciones del desarrollo

Diseño conceptual

Nuestro videojuego pertenece al **género** de videojuegos **plataformas**. Como tal, parte de la premisa de que se necesita un **personaje** principal que deberá **saltar** y **esquivar** obstáculos con el fin de llegar a un objetivo.

Como tal, los objetivos a tratar a la hora de crear este **personaje** son un correcto movimiento y un buen manejo del mismo a la hora de saltar. En nuestro caso, también hemos añadido la opción de disparar con el fin de lograr eliminar los enemigos que aparecerán a lo largo de los niveles.

En relación con los **enemigos**, estos se basan en un movimiento horizontal a lo largo de cierta distancia, hasta que encuentran un obstáculo y cambian de dirección para proseguir su movimiento horizontal, si tocan al jugador, este morirá y se reiniciará el nivel. En el caso de los **enemigos** que también pueden **disparar**, estos dispararán cuando encuentren al jugador dentro de su rango de visión, si algún disparo toca al jugador, se reiniciará el nivel debido a su muerte.

Hablando sobre los **shurikens**, estos son indestructibles ya que su fin es el de provocar el salto del jugador sobre ellos con el fin de evitarlos, si el jugador los toca morirá y se reiniciará el nivel.

Además, en algunos niveles, encontraremos una especie de **flechas** envenenadas que saldrán de una especie de dispensadores de flechas, cuya función es la de matar al jugador; por lo tanto el jugador deberá evitarlas.

En cuanto a los **portales**, como su nombre indica, sirven para teletransportar al jugador de un nivel a otro cuando los toca. Siempre habrá un portal en cada nivel, excepto en el último nivel que será el nivel en el que nos encontremos a la princesa.

Con respecto a la **princesa**, como hemos mencionado anteriormente, esta se encontrará en el último nivel, dando lugar al fin de juego en el momento en el que la toque el jugador, dando a entender que la has rescatado. Le saldrá al jugador una pantalla de victoria cuando lo logre.

Implementación

Este videojuego ha sido desarrollado siguiendo el modelo Entidad-Componente-Sistema (ECS), mediante el cual dividimos el trabajo en 3 principales grupos: entidades que tendrán propiedades a lo largo del juego, componentes que se encargarán de enlazar las entidades con los sistemas, y los propios sistemas que se encargarán de dibujar dichas entidades.

- Generator_system:

```
;;              tipo,            x, y,   ancho, alto, vx, vy
 player: .db  # e type default ,2 , 170 , 6 , 14,  1, 1
;;           sprite
        .dw  # spr player 00
;;            pvX,pvY        ,Salto , Dir  Disparo
        .db 0x00 , 0xC0 ,   -1,   1  , 0
;;           animacion
        .dw 0,0
;;            contRender , nada ,    tipo 2
        .db  0x00 , 0x00 , 00
```

Para crear el personaje principal, creamos una entidad con 19 propiedades que más adelante serán utilizadas tanto por los componentes como por el sistema.

```
enemy: .db  # e type default ,70, 70 , 6 , 12,  1, 0
       .dw  # spr enemy 0
       .db 0x00, 0xC0 , 1 , 1 , 0
       .dw  0,0
       .db  0x00 , 0x00 ,  00
```

Para crear el enemigo que anda, creamos una entidad con 19 propiedades que más adelante serán utilizadas por los componentes como por el sistema,

```
enemy shoter:  .db  # e type default ,70, 70 , 6 , 12,  1, 0
               .dw  # spr enemy shoter
               .db 0x00, 0xC0 , 1 , 0 , 0
               .dw 0, 0
               .db  0x00 , 0x00 ,  00
```

Para crear el enemigo shooter, creamos una entidad con 19 propiedades que más adelante serán utilizadas tanto por los componentes como por el sistema.

```
enemy small:   .db  # e type default ,70, 70 , 4 , 6,  1, 0
               .dw  # spr enemy small
               .db 0x00, 0xC0 , 1 , 0 , 0
               .dw 0 , 0
               .db  0x00 , 0x00 ,  00
```

Para crear el portal, creamos una entidad con 19 propiedades que más adelante serán utilizadas tanto por los componentes como por el sistema.

```
princess: .db  # e_type_default ,2 , 20 , 6 , 14,  0, 0
          .dw  # spr princess
          .db 0x00, 0xC0 , 0 , 1 , 0
          .db 0, 0
          .db  0x00 ,0x00  , #_e_type_default
```

Para crear la princesa, creamos una entidad con 19 propiedades que más adelante serán utilizadas tanto por los componentes como por el sistema.

```
bullet heroe:  .db  # e type default ,50 , 100 , 4 , 6,  2, 0
               .dw  # spr bullet heroe
               .db 0x00, 0xC0 , 0, 1 , 0
               .db 0, 0
               .db 0x00 ,0x00  , 00
```

Para crear el shuriken, creamos una entidad con 19 propiedades que más adelante serán utilizadas tanto por los componentes como por el sistema.

```
flecha:  .db  # e type default ,50 , 100 , 4 , 6,  2, 0
               .dw  # spr arrow
               .db 0x00, 0xC0 , 0, 1 , 0
               .db 0, 0
               .db 0x00 ,0x00  , 00
```

Para crear el flecha, creamos una entidad con 19 propiedades que más adelante serán utilizadas tanto por los componentes como por el sistema.

```
generator Personaje::

 push bc

 ld hl, # player
  call entity man create

 pop bc
 ld 1(ix), b
 ld 2(ix), c

 ld a, # e type movable
 ld b ,# e type render
 or b
 ld b, # e type input
 or b
 ld b, # e type collision
 or b
 ld b, # e type animation
 or b
 ld 0(ix),a;;0110 0111

  ret
```

Para generar cada una de las anteriores entidades, se sigue el esquema de este mismo código (este caso sirve para generar el jugador).

- Entity manager:

```
max_entities == 10
entity_size  == 19

;;; TIPOS DE ENTIDADES
 e_type_invalid ==  0x00 ;;
 e_type_render == 0x01 ;;Entity for render 0000 0001
 e_type_movable == 0x02 ;; Entity for move 0000 0010
 e_type_input == 0x04 ;; Entity for input  0000 0100
 e_type_ia == 0x08 ;; Entity for ia        0000 1000
 e_type_bullet == 0x10 ;; Entity for bullet 0001 0000
 e_type_collision == 0x20 ;; Entity for bullet 0010 0000
 e_type_animation == 0x40 ;; Entity for bullet 0100 0000
 e_type_dead == 0x80 ;;Upper bit signals dead  entity 1000 0000
 e_type_default == 0x7F ;;Default entity 0111 0000
;; TIPO DE ENTIDADES 2
 e_type_princess == 0x01 ;; Entity for princess  0000 0001
 e_type_enemy_small == 0x02;; Entity for enemy small 0000 0010
 e_type_bullet_enemy == 0x04;; Entity for bullet enemy 0000 0100
 e_type_ememy_shoter == 0x08;; Entity for enemy shoter 0000 1000
 e_type_portal == 0x20;; Entity for portal 0001 0000
 e_type_obs_shoter == 0x40;; Entity for Obstaculo Shoter 0001 0000


 num_entities:: .db 0
_last_elem_ptr:: .dw _entity_array
_entity_array:: .ds max_entities*entity_size
.db 0
```

Para crear entidades, lo primero que se hace es definir el número máximo de entidades que tendremos y el espacio de cada entidad; posteriormente definiremos los tipos de entidades y por último crearemos el array de entidades.

```
muerta:    .db  0x00 , 0x00, 0x00 , 0x00 , 0x00,  0x00, 0x00
           .db  0x00, 0x00
           .db  0x00, 0x00 , 0x000 , 0x00 , 0x00
           .db  0x00, 0x00
           .db  0x00 ,0x00  , 0x00

entity_man_getEntitdyArray_IX::
  ld     ix, #_entity_array
   ret

entity_man_getNumEntities_A::
  ld     a, (_num_entities)
   ret
```

Posteriormente crearemos una entidad de tipo muerto para usarla más adelante cuando necesitemos eliminar las existentes; asimismo creamos 2 getters del array de entidades para darle uso más adelante en el código.

```
;;INPUT  HL -> entidad
;;RETURN IX -> puntero a la entidad
entity_man_create::
   call    entity_free_space
```

```
  or      a
   jr       z, fin  ;; si da 0 acabo porque no hay espacio , sino sigo

  ld      ix , ( last_elem_ptr)
  ld      de, ( last_elem_ptr)
  ld      bc, #entity_size
  ldir

  ld      a, ( num_entities)
  inc     a
  ld      ( num_entities), a

  ld      hl, ( last_elem_ptr)
  ld      bc, #entity_size
  add     hl, bc
  ld      ( last_elem_ptr), hl

fin:
   ret
```

Luego, llamaremos a la etiqueta nivel global llamada create que en primer lugar llamara a entity space para comprobar si existe hueco para crear la entidad. Si no existe, se saldrá de la etiqueta porque no hay espacio y si existe creará la entidad.

```
;;INPUT HL el puntero de la función
;;      D el signature
man_entity_forall::
   call    entity_man_getEntityArray_IX  ;; array de entidades  -> IX

 desloop2:

  ld      a, 0(ix) ;; cargo tipo de entidad  0000 0011
  or      a ;; resto los tipos               0000 0011
   ret      z ;;                              0000 0011

  push    de
  push    hl
   call     #man_entity_forall_matching
  pop     hl
  pop     de

   ld       bc,#entity_size ;; carga la cantidad de atributos que tiene
entity  en bc
  add     ix, bc
   jr       desloop2


;;INPUT HL el puntero de la función
;; en D el signature
man_entity_forall_matching::
  ld      b,d  ; 0000 0001
  ld      (to_call), hl


  ld      a, 0(ix) ;0000 0001
  and     b        ;0000 0001 = 0000 0001
  sbc     a,b      ;0000 0001 - 0000 0001 =0000 0000
   ret      nz
```

```
   to_call=.+1
   call    to_call
   ret
```

Más adelante, llamaremos a man_entity_forall y éste a su vez a man_entity_forall_matching con el fin de recorrer el array de entidades y acceder a las propiedades de cada entidad para así lograr renderizar, mover y gestionar dicha entidad.

```
;; INPUT
;; IX posición de puntero
entity set4destroy::
  ld     0(ix),    # e type dead ;;cargo en a el tipo invalido
   ret


;;INPUT IX PUNTERO A DESTRUIR
entity destroy::

    call   entity man getNumEntities A ;; cuantas entidades me quedan a
= 1

  ld     hl, ( last elem ptr)
  ld     bc, #-entity size
  add    hl,bc
  ld     ( last elem ptr),hl

  dec    a
  ld     ( num entities),a

  push   ix
  pop    de
  sbc    hl,de
   jr      z ,actualizar

  ld     hl,( last elem ptr)
  ld     bc,#entity size
  ldir

actualizar:
  ld     hl, ( last elem ptr)
  ld     (hl) ,#0
   ret
```

Posteriormente llamaremos a entity_set4destroy cuando necesitemos marcar una entidad para su posterior eliminación mediante una llamada a entity_destroy para eliminar esa entidad.

```
entity update::
   call    entity man getEntityArray IX  ;; array de entidades  -> IX
    call     entity man getNumEntities A    ;; numero de entidades que
hay en el array vivas  -> A
   jr       desloop
destruir:
   call    entity destroy

 desloop:
  ld     a, 0(ix) ;; cargo tipo de entidad
  or     a ;; si da 0, No hay más entidades
```

```
   ret     z

   call    entity_dead
    jr         z,   next  ;;  si  es  NO  cero  elimino  y  si  no  salto  al
siguiente
   jr      destruir
 next:
  ld       bc,#entity_size  ;; carga la cantidad de atributos que tiene
entity  en bc
  add     ix, bc  ;;
   jr        desloop
;; RETURN en A el valor de free space
entity_free_space::
  ld      a, ( num_entities)
  ld      b,a
  ld      a, #max_entities ;; cargamos en a el max entities
  sbc     a, b ;; max entities -  num entities
   ret
```

Más adelante, llamaremos al entity_update que se encargará de recorrer todas las entidades y actualiza las entidades para ponerlas a valor muerto.

También se encuentra el entity_free_space que nos permite crear espacio para la creación de una entidad.

```
eliminar_entidades::
   call    entity_man_getEntityArray_IX

  ld      a , #max_entities
 eliminar_loop:
  or      a ;; si da 0, No hay más entidades
   jr       z, salto_eliminar

  ld      hl, # muerta
  push    ix
  pop     de
  ld      bc, #entity_size
  ldir

  dec     a
   ld       bc,#entity_size  ;; carga la cantidad de atributos que tiene
entity  en bc
  add     ix, bc  ;;
   jr        eliminar_loop



salto_eliminar:
  ld      a, #0
  ld      ( num_entities), a


   call    entity_man_getEntityArray_IX

  push    ix
  pop     hl

  ld      ( last_elem_ptr), hl

   ret
```

Por último, llamaremos a eliminar_entidades cuando cambiemos de nivel o reiniciemos el mismo, para resetear las que están en dicho nivel.

- Physics_system:

```
physics_update_one_entity::

  ld      b, # e_type_input ; 0100
  ld      a, 0(ix)          ; 0011
  and     b                 ; 0000 = 0000
  sbc     a,b               ; 0000 - 0100 = 00 - 04 = FB -> 1111 1011
   jr       nz,  saltar

   call     physics_check_keyboard

saltar:
   ;; ACTUALIZAR X
  ld      h, 5(ix)
  ld      a, 1(ix)
  add     h
  ld      1(ix), a

   ;;ACTUALIZAR Y
  ld      a, 2(ix) ;; y
  ld      h, 6(ix) ;; vy
  add     h ;; y + vy
  ld      h,a  ;; h -> suma

  ld      a ,2(ix);; y antes de actulizar
  ld      2(ix), h ;; y = suma

  ld      6(ix), #5 ;; vy o gravedad

   ret


physics_update::
  ld      hl, #physics_update_one_entity
  ld      d, # e_type_movable
   call     man_entity_forall
   ret
```

En phsysics_system, llamaremos a psyshics_update el cual se encargará de realizar una llamada a physics_update_one_entity que se encargará de actualizar cada entidad por separado en base al tipo de entidad que sea.

```
IndiceJumpTable:
   .db  0

Jumptable:
   .db  #-7, #-7 , #-7
   .db  #-7, #0 , #0
   .db  #10 ,#10
   .db  #0x80
```

Creamos una tabla de saltos para establecer valores de salto en el personaje principal.

```
;INPUT IX posicion puntero entidad
physics_check_keyboard::
  ld      5(ix), #0 ;; vx

   ;call    cpct_scanKeyboard_f_asm
  ld      hl, #Key_O
   ;;ld hl, #Joy0_Left
   call    cpct_isKeyPressed_asm
   jr      z, o_not_pressed
   ;;jr z, joy_not_left_pressed
```

Posteriormente comprobamos si se está pulsando el teclado para modificar el personaje principal según las teclas que estén pulsadas.

```
o_pressed:

  ld      a, 1(ix) ;; x
  ld      b, #-1
  add     b ;; newx
  or      a ;; comprobacion de que es cero
   jr       z, o_not_pressed

  ld      5(ix), b
  ld      12(ix), #-1

   jp       joy_left_not_pressed

o_not_pressed:
  ld      hl, #Joy0_Left
   call    cpct_isKeyPressed_asm
   jr      z, joy_left_not_pressed

   jp      o_pressed
```

Aquí dependiendo el valor que nos cargue hl , estará presionando la tecla o no. Si presionas la tecla O el personaje irá hacia la izquierda. Sino lo hace, mirará si estámoviéndose el joystick hacia la izquierda

```
;; JOY LEFT

joy_left_not_pressed:
  ld      hl, #Key_P
   call    cpct_isKeyPressed_asm ;; 0 o 1
   jr      z, p_not_pressed
```

Miramos si el joystick está moviendo al personaje a la izquierda. Si no lo está, miramos si está pulsando la tecla P.

```
p_pressed:
  ld      a, 1(ix) ;; x
  ld      b, #1
```

```
  add     b ;; newx
  ld      c , #0x4A
  sbc     a,c ;;
   jr       z, p not pressed
  ld      5(ix), #1
  ld      12(ix), #1

   jp       joy right not pressed

p not pressed:
  ld hl, #Joy0 Right
   call cpct isKeyPressed asm ;; 0 o 1
   jr z, joy right not pressed

   call     p pressed
```

Miramos si la tecla P está pulsada y si lo está el personaje se mueve a la derecha. Si no lo está, miramos si está moviendo el joystick hacia la derecha.

```
joy right not pressed:
  ld      hl, #Key Q
   call     cpct isKeyPressed asm ;; 1
   jr       z, q not pressed
```

Miramos si el joystick se está moviendo a la derecha, si no lo está, miramos si se está pulsando la tecla Q.

```
q pressed:
   ;;-1 = ESTAR SALTANDO | 0 = NO SALTANDO
   ;Comprobar Salto
  ld      a, 11(ix)
  cp      #-1
   jr       z, q not pressed

   ;;Valor actual del Salto
  ld      hl, #Jumptable
  ld      c, a
  ld      b,#0
  add     hl , bc

   ;;Comprobamos si se ha terminado el salto
  ld      a,(hl)
  cp      #0x80
   jr       z, fin salto

   ;; Cargamos el valor de jumptable en vy
  ld      6(ix),a

   ;;Incrementar Indice Jumptable
  ld      a, 11(ix)
  inc     a
  ld      11(ix), a

   jp       joy up not pressed

fin salto:
   ;Actualizar valor de Salto
  ld      a, #-1
```

```
  ld      11(ix), a

q_not_pressed:
  ld      hl, #Joy0_Fire1
   call     cpct_isKeyPressed_asm ;; 0 o 1
   jr       z, joy_up_not_pressed

   call     q_pressed
```

Miramos si la tecla Q está pulsada y si lo está, el personaje saltará siguiendo los valores de la tabla de saltos y actualizándose. Si no lo está, se mira si se está pulsando el botón de salto del joystick.

```
space_pressed:
   ;Comprobar Disparo
  ld      a, 13(ix)
  or      a
   jr       nz, space_not_pressed

   ;Actualizar Disparo
  ld      13(ix), #1

  ld      a, 1(ix)
  ld      b, 2(ix)
  ld      c, 12(ix)
  push ix
   call generator_Balas
  pop ix


   jp       salir


space_not_pressed:

  ld hl, #Joy0_Up
   call cpct_isKeyPressed_asm ;; 0 o 1
   jr z, salir

   jp space_pressed

salir:
   ret
```

Miramos si la tecla Space está pulsada y si lo está, el personaje disparará una bala que podrá colisionar con los enemigos y los matará.

- Pshysics_bala:

```
physics_update_bala::
  ld hl, #physics_update_one_bala
  ld d, # e type bullet
   call man_entity_forall
   ret
```

Utilizaremos physics_update_bala el cual llamará a physics_update_one_bala para actualizar cada entidad bala en base a las condiciones en las que se encuentre en el mapa.

```
physics update one bala::
  ld a, 1(ix) ;; x
  ld b, #2
  add b ;; newx
  ld c , #0x4A
  sbc a,c ;;
   jr nz, no tocar derecha
   call entity set4destroy
  push ix
   call entity man getEntityArray IX
  ld 13(ix), #0
  pop ix
   jp actualizar

no tocar derecha:
  ld a, 1(ix) ;; x
  add #-2
  or a ;; comprobacion de que es cero
   jr z, eliminar

  ld a, 1(ix) ;; x
  add #-1
  or a ;; comprobacion de que es cero
   jr z, eliminar

  ld a, 1(ix) ;; x
  add #0
  or a ;; comprobacion de que es cero
   jr nz, actualizar

eliminar:
   call entity set4destroy
  push ix
   call entity man getEntityArray IX
  ld 13(ix), #0
  pop ix


actualizar:
   ;; ACTUALIZAR X
  ld a, 1(ix)
  ld h, 5(ix)
  add h
  ld 1(ix), a
    ret
```

La bala saldrá del personaje y si toca al enemigo o a un bloque del mapa, esta se eliminará.

- IA_system:

```
ia_update::
  ld a, # e_type_movable
  or a, # e_type_ia
  ld d, a
  ld hl, #ia_update_one_entity
   call man_entity_forall


  ld a, # e_type_ia
  ld d, a
  ld hl, #ia_shoter
   call man_entity_forall

   ret
```

En primer lugar, llamaremos al update de la IA que llamará al one entity para actualizar cada entidad por separado y comprueba si esa entidad es de tipo movimiento y tipo inteligencia artificial.

```
ia_update_one_entity::
  ld a, 1(ix) ;; x
  ld b, #1
  add b ;; newx
  ld c , #0x4A
  sbc a,c ;;
   jr z, toca_borde_der
  ld a, 1(ix) ;; x
  ld b, #-1
  add b ;; newx
  or a ;; comprobacion de que es cero
   jr z, toca_borde_izq

   jp salir

toca_borde_izq:
  ld 5(ix), #0x01
   jp salir

toca_borde_der:
  ld 5(ix), #0xFF

salir:

  call ia_enemy_shoter

   ret
ia_enemy_shoter::


  ld a, 18(ix)
  ld b, #_e_type_ememy_shoter
  and a, b
```

```
   jr z, salir2

  ld b, #-20
  ld c, #20
   call ia_disparo ;; izquierda

  ld 12(ix), #1

  ld a, 13(ix)
  or a
   jr nz, salir2

  ld b, #10
  ld c, #-10
   call ia_disparo ;; derecha

  ld 12(ix), #-1

salir2:
   ret
```

La IA comprueba si está tocando los bordes del mapa y sino si está tocando los bloques que se encuentre a izquierda y derecha, si los toca cambiará de dirección.

```
ia_shoter::
  ld a, 18(ix)
  ld b, # e_type_obs_shoter
  and a, b
   jr z, salir3

   call disparo_flecha

salir3:
   ret


;;INPUT BC = POSICION A COLISIONAR
ia_disparo::
  push   ix

   call entity_man_getEntityArray IX
  push   ix
  pop    iy

  pop    ix

;;IF COMPROBACION COLISION
```

```
  ld      a, 1(ix)
  add     b
  add     3(ix)
  sub     1(iy)
   jr       c, no_disparo

  ld      a, 1(iy)
  add     c
  add     3(iy)
  sub     1(ix)
   jr       c, no_disparo

  ld      a, 2(ix)
  add     4(ix)
  sub     2(iy)
   jr       c, no_disparo

  ld      a, 2(iy)
  add     4(iy)
  sub     2(ix)
   jr       c, no_disparo


   call disparo_enemy

no_disparo:
   ret

disparo_enemy::

  ld a, 13(ix)
  or a
   jr nz, no_disparo2

   ;Actualizar Disparo
  ld 13(ix), #1

  ld a, 1(ix)
  ld b, 2(ix)
  ld c, 12(ix)
  push ix
  pop hl
  push ix
   call generator_Balas_Enemigo
  pop ix

no_disparo2:
   ret
```

En el caso del enemigo shooter, este también es capaz de disparar al jugador si este se encuentra dentro de un rango de detección.

- Render_system:

```
rendersys_init::
   halt
   ld      c, #0
   call    cpct_setVideoMode_asm

   ;CAMBIAR BORDE A NEGRO
```

```
   ld     hl, #0x1410
   call    cpct_setPALColour_asm
;
   ld     de, #16
   ld     hl, #_g_palette
   call    cpct_setPalette_asm
   ret
```

Inicializamos los valores del render.

```
rendersys_update::
   ld     hl, #render_one_entity
  ld     d, #_e_type_render
   call    man_entity_forall
   ret
render_entity::

   ld     e, 9(ix)
   ld     d, 10(ix)
   ld     l ,7(ix) ;; Sprite
   ld     h, 8(ix) ;; Sprite
  ld     b, 3(ix) ;;Width
  ld     c, 4(ix) ;;Height
   call     cpct_drawSpriteBlended_asm ;; necesita de(getScreenPtr) , b
, c ,  hl
   ret

;;INPUT:
;;  IX: Pointer to entity to render
render_one_entity::

   call    entity_dead
   jr      nz , salir

   ld     a, 16(ix)
   cp     #0
   jr      z, salto_sumo
   call    render_xor

salto_sumo:
   ld 16(ix), #1

   ;;Calculate a video memory location
  ld     de, #0xC000
  ld     c, 1(ix) ;;x
  ld     b, 2(ix) ;;y
   call    cpct_getScreenPtr_asm ;; esto devuelve el valor en hl

   ld     9(ix),  l
   ld     10(ix), h

   call    render_entity
salir:
   ret
```

Actualizamos cada una de las entidades de nuestro render y la dibujamos.

```
render_death_entity::
   call    entity_dead
   jr      z, salir2
```

```
   call    render_entity

salir2:
   ret

render_death_update::
   ld     hl, #render_death_entity
  ld     d, # e type dead
   call    man_entity_forall
   ret
```

Si una entidad muere, la dejaremos de borrar mediante render_death_entity.

```
rendersys_pause_update::
   ld     hl, #render_xor
  ld     d, # e type render
   call    man_entity_forall
   ret
```

Redibujamos las entidades después de realizar una pausa.

```
render_tilemap_0::
   push de

   ld bc, #0x1905
   ld de, #0x14
   ld hl, #_tiles_00
   call cpct_etm_setDrawTilemap4x8_ag_asm

   pop de
   ld hl, #0xC000
   call cpct_etm_drawTilemap4x8_ag_asm

   ret

render_tilemap_1::
   push de

   ld bc, #0x1905
   ld de, #0x14
   ld hl, #_tiles_00 ;; imagenes
   call cpct_etm_setDrawTilemap4x8_ag_asm

   pop de  ;; tilemap

   ex de , hl
   ld bc , #0x05
   add hl , bc
   ex de, hl

   ld hl, #0xC014
   call cpct_etm_drawTilemap4x8_ag_asm

   ret

render_tilemap_2::
```

```
   push de

   ld bc, #0x1905
   ld de, #0x14
   ld hl, #_tiles_00 ;; imagenes
   call cpct_etm_setDrawTilemap4x8_ag_asm

   pop de  ;; tilemap

   ex de , hl
   ld bc , #0x0A
   add hl , bc
   ex de, hl

   ld hl, #0xC028
   call cpct_etm_drawTilemap4x8_ag_asm

   ret

render_tilemap_3::
   push de

   ld bc, #0x1905
   ld de, #0x14
   ld hl, #_tiles_00 ;; imagenes
   call cpct_etm_setDrawTilemap4x8_ag_asm

   pop de  ;; tilemap

   ex de , hl
   ld bc , #0x0F
   add hl , bc
   ex de, hl

   ld hl, #0xC03C
   call cpct_etm_drawTilemap4x8_ag_asm

   ret
```

Renderizamos el tilemap por partes, dividimos cada tilemap en 4 partes y las renderizamos.

- Cargador niveles:

```
type player == 0x01
type princess == 0x02
type enemy == 0x04
type enemy small == 0x08
type enemy shoter == 0x10
type portal == 0x20
type obstaculo s = 0x40
```

```
 tile map colision:: .db 0x00, 0x00

 IndiceNiveles:: .db 0

NIVELES:  ;0 1           2 3

   .dw # tilemap 01 , #Tposiciones1
        ;4 5            6 7
   .dw # tilemap 02 , #Tposiciones2
   .dw # tilemap 03 , #Tposiciones3
   .dw # tilemap 04 , #Tposiciones4
   .dw # tilemap 05 , #Tposiciones5
   .dw # tilemap 06 , #Tposiciones6
   .dw # tilemap 07 , #Tposiciones7
   .dw # tilemap 13 , #Tposiciones13
   .dw # tilemap 08 , #Tposiciones8
   .dw # tilemap 09 , #Tposiciones9
   .dw # tilemap 10 , #Tposiciones10
   .dw # tilemap 16 , #Tposiciones16
   .dw # tilemap 11 , #Tposiciones11
   .dw # tilemap 12 , #Tposiciones12
   .db 0x00 , 0x00

IndiceTabla:
   .db 0
```

En esta parte declaramos los tipos de entidades, el índice de los niveles , el índice de la tabla a recorrer y la tabla niveles que se referencia al mapa y a las posiciones donde van aparecer las entidades en dicho mapa.

```
Tposiciones1:
   .db # type player , 7 , 170
   .db # type portal , 5  , 15
   .db # type portal , 5 , 15
   .db 0x00
```

Las posiciones de las entidades relacionadas con cada mapa se cargarán utilizando este esquema para cada tabla.

```
cargador nivel::
   call    eliminar entidades
    ;;Valor actual de las tabla niveles
  ld hl, #NIVELES
  ld a, ( IndiceNiveles) ;0 ; 4
  ld c, a
  ld b,#0
  add hl , bc

   ;;Comprobamos si se ha los niveles
  ld e,(hl)
  inc hl
  ld d,(hl)

  push hl

  ld l, e
  ld h, d
```

```
  ld b, #0x00
  ld c, #0x00
  sbc hl , bc
   jr z, fin_juego

  ld ( tile_map_colision), de
   ;;CAMBIAR RENDER TILE MAP
   call render_tilemap

  pop hl
  inc hl

  ld     e,(hl)
  inc    hl
  ld     d,(hl)

   call recorrer_tabla

   ;;Incrementar Indice NIVELES
  ld a, ( IndiceNiveles) ; 0
  inc a                  ; 1
  inc a                  ; 2
  inc a                  ; 3
  inc a                  ; 4
  ld ( IndiceNiveles), a

   call    loop_juego
   ret

fin_juego:
   call ganar

   ret
```

Mediante cargador_nivel, cagaremos cada nivel por separado cuando se tenga que mostrar por pantalla; se eliminarán las entidades del nivel anterior, se renderizará el tilemap y se actualizará el índice de la tabla.

```
recorrer_tabla::

loop:
  push de
   ;;Valor actual de las tabla
  ld l, e
  ld h, d
  ld a, (IndiceTabla)
  ld c, a
  ld b,#0
  add hl , bc

   ;;Comprobamos si se ha los niveles
  ld a,(hl) ; 0011
  cp #0x00
   jr  z, fin_tablaaa

  ld     b, # type player ; 0001
  and    b               ; 0000
  sbc    a,b             ; 0000 - 0001 = 00 - 04 = FF -> 1111 1011
   jr      nz, no_player

   call    load_posx_y
```

```
   ;; cosas de player
    call    generator_Personaje

  pop     de
   jp      fin_crear

no_player:
  ld      a,(hl)
  ld      b, # type princess ; 0100          0100
  and     b                  ;               0000
  sbc     a,b                ; 0000 - 0100 = 00 - 04 = FB -> 1111 1011
   jr      nz, no_princess

   call   load_posx_y
   ;; cosas de princess
   call   generator_Princesa

  pop     de
   jp      fin_crear
no_princess:
  ld      a,(hl)
  ld      b, # type enemy ; 0100          0100
  and     b               ;               0000
  sbc     a,b             ; 0000 - 0100 = 00 - 04 = FB -> 1111 1011
   jr      nz, no_enemy

   call   load_posx_y
   ;; cosas de enemy
   call    generator_Enemigos

   pop     de
   jp      fin_crear
no_enemy:
  ld      a,(hl)
  ld      b, # type enemy_small ; 0100          0100
  and     b                     ;               0000
  sbc     a,b                   ; 0000 - 0100 = 00 - 04 = FB -> 1111 1011
   jr      nz, no_enemy_small


   call   load_posx_y
   ;; cosas de enemy_small
   call    generator_Enemigos_Pequenyos

   pop     de
   jp      fin_crear
no_enemy_small:
  ld      a,(hl)
  ld      b, # type enemy_shoter ; 0100          0100
  and     b                      ;               0000
  sbc     a,b                    ; 0000 - 0100 = 00 - 04 = FB -> 1111 1011
   jr      nz, no_enemy_shoter

   call   load_posx_y
   ;; cosas de enemy_shoter
   call    generator_Enemigos_Shoter

  pop     de
   jp      fin_crear

no_enemy_shoter:
  ld      a,(hl)
```

```
  ld      b, # type portal ; 0100          0100
  and     b                ;              0000
  sbc     a,b              ; 0000 - 0100 = 00 - 04 = FB -> 1111 1011
   jr       nz, no portal

   call    load posx y
   ;; cosas de enemy shoter
   call     generator Portal

  pop      de
   jp       fin crear

no portal:
  ld      a,(hl)
  ld      b, # type obstaculo s ; 0100          0100
  and     b                ;              0000
  sbc     a,b              ; 0000 - 0100 = 00 - 04 = FB -> 1111 1011
   jr       nz, fin crear

   call    load posx y
   ;; cosas de portal
   call     generator Obstaculo Shoter

  pop     de

fin crear:
  ld a , (IndiceTabla)
  inc a
  inc a
  inc a
  ld (IndiceTabla), a
   jp loop


fin tablaaa:
  pop      de
  ld a, #0
  ld (IndiceTabla), a
   ret
```

En recorrer tabla lo que se hace como su propio nombre indica,recorrer la tabla y cargar los diferentes tipos de entidades en sus respectivos mapas con sus respectivas posiciones.

- Menú:

```
man menu init::

  ld hl, # screenmenu z end
  ld de, #0xFFFF
   call cpct zx7b decrunch s asm
   ret

;; Update del menu
man_menu_update::
```

```
   ;call cpct scanKeyboard f asm
  ld  hl, #Key Return
   call cpct isKeyPressed asm

     ret
```

En el menú, llamamos a menú init para cargar la imagen comprimida y con de crunch descomprimimos la imagen y la mostramos por pantalla.

Posteriormente, llamamos al update de menú que nos permitirá darle al botón de intro para iniciar el juego.

- Música:

```
_IndiceMusica:: .db 0

MUSICAS:  ;0 1
   .dw # song nivel1
   .dw # song nivel2
   .dw # song nivel3
   .dw # song nivel4
   .dw # song nivel5
   .db 0x00 , 0x00

music init::
  ld     de, # song menu
   call    cpct akp musicInit asm ;; USA af af' hl de bc ix iy
   ret

music play::
  push   ix
  push   iy
   call    cpct akp musicPlay asm
  pop    iy
  pop    ix

   ret
```

Creamos un cargador de música en el cual añadimos cada una de las canciones que hemos creado a una tabla y la recorremos mediante un índice. Posteriormente inicializamos cada una de las canciones mediante music_init y las reproducimos mediante music_play.

```
music next song::

  ld hl, #MUSICAS
  ld a, ( IndiceMusica) ;0 ; 4
  ld c, a
  ld b,#0
  add hl , bc

   ;;Comprobamos si se han acabado los niveles
  ld e,(hl)
  inc hl
  ld d,(hl)
```

```
  ld l, e
  ld h, d
  ld b, #0x00
  ld c, #0x00
  sbc hl , bc
   jr z, fin_musica

   ;; en de se carga la nueva cancion
   call    cpct_akp_musicInit_asm ;; USA af af' hl de bc ix iy

   ;;Incrementar Indice NIVELES
  ld a, ( IndiceMusica) ; 0
  inc a                 ; 1
  inc a                 ; 2
  ld ( IndiceMusica), a
   jp salir_musica

fin_musica:
 ld a, #0
 ld ( IndiceMusica), a

salir_musica:
   ret
```

Por último con ayuda de music_next_song avanzamos en la tabla de canciones y cargamos la siguiente canción para reproducirla, una vez se acabe se llama a fin_musica.

- Interrupciones:

```
set_init_handler::
  ;; MODO 1 DE INTERRRUPCIONES
 im   1

  ;; CON ESTO ESPERAMOS A EL WAITVSYNC
  call cpct_waitVSYNC_asm
 halt
 halt
  call cpct_waitVSYNC_asm

  ;;DESHABILITAR INTERRUPCIONES
 di

  ;; CAMBIARMOS 38 Y 39 DE MEMORIA
 ld   a, #0xC3
 ld   (0x38), a
  ;; POENEMOS EL MAN_IR1
 ld   hl, #man_ir1
 ld   (0x39), hl

  ;; HABILITAMOS INTERRUPCIONES
 ei
  ret
```

En cuanto a las interrupciones, creamos un manejador de interrupciones mediante el cual esperaremos a waitVSYNC y posteriormente llamaremos a la primera interrupción (son 6).

```
.macro   setNextManIR dirreccion
 ld   hl, #dirreccion
 ld   (0x39) , hl
.endm

.macro setNumIR number
 ld   a, #number
 ld   (num i), a
.endm

num i:: .db 00

cont_i:: .db 00
```

Seguidamente creamos 2 macros y 2 contadores que nos ayudarán a controlar de forma correcta nuestras interrupciones.

```
man ir1::
  cpctm push af , bc , de , hl
 push ix
 push iy

  setNumIR 1
  setNextManIR man ir2

  cpctm setBorder asm HW BLACK
 ld  a , (cont_i)
 inc a
 ld  (cont_i), a

 pop iy
 pop ix
  cpctm pop  hl , de , bc , af
 ei
 reti
```

Una vez hecho todo lo anterior procedemos a crear 6 interrupciones siguiendo el esquema de la interrupción número 1. Utilizamos un contador para manejar y usar usar 5 interrupciones para renderizar y otras 7 para realizar cálculos.

- Main:

```
_final_muertes: .asciz "HAS MUERTO"
_numero_muertes:: .db 48 , 48 , 48 , 48 , 00
 musica:: .db 0
```

```
main::
 ;;CAMBIAMOS LA PILA DE POSICION
ld   sp, #0xA400

 ;;LLAMAMOS A CAMBIAR INTERRUPCIONES
 call  set_init_handler

 ;;INICIALIZAR MUSICA
 call  music_init

 ;;INICIALIZAR RENDER
 call rendersys_init

 ;; Init systems
 call man_menu_init
```

En lo relativo a main, primero creamos todo lo necesario para manejar el contador de muertes, segundo creamos una variable _musica que nos servirá para gestionar más adelante el estado en el que se encontrará la música. Acto seguido cambiamos la pila de posición para gestionar mejor el espacio, llamamos a las interrupciones, inicializamos la música, inicializamos el render y entraremos en el menú de inicio.

```
loop menuuu:
 call     man menu update
 jr       z,  loop menuuu

ld   hl, #0000
ld   de, #0xC000
ld   bc, #80*200
ldir

 call    cargador nivel

 ret
```

Una vez cargado el menú inicio se le llama y se muestra hasta que no se presione la tecla intro.

```
loop_juego::
 loop:
  call    ia_update
  call    physics_update
  call    physics_update_bala
  call    colision_update
```

Posteriormente se inicia el loop del juego en el cual se llama a la ia, physics y colisiones.

```
esperar:
 ld  a , (cont_i)
 cp #7
  jr nz, esperar
  call    rendersys_update
  call    render_death_update

  call     entity_update
```

A continuación esperamos 7 iteraciones de contador para llamar al render y a las entidades.

```
;MUTE
 ld   hl, #Key M
  call  cpct isKeyPressed asm
  jr z, continuar

 ld  a,  ( musica)
 cp  #0
  jr   nz , poner play

  call cpct akp stop asm
 ld   a, #1
 ld  ( musica), a ;; 1 = mute

  jp continuar
poner play:
 ld   a, #0
 ld  (_musica), a ;; 1 = mute
```

Seguidamente creamos el botón mute que servirá para detener/reproducir la música del juego.

```
continuar:
  ;;PAUSA
 ld  hl, #Key Esc
  call cpct isKeyPressed asm

  jr z, loop

 ld hl, # screenpause z end
 ld de, #0xFFFF
  call cpct zx7b decrunch s asm


 final loop:
 ld  hl, #Key Esc
  call cpct isKeyPressed asm

  jr z, continuamos

  jp volvemos
```

Posteriormente creamos el botón Esc que nos servirá para acceder al menú pause y volver al juego.

```
continuamos:

  ;call cpct scanKeyboard f asm
 ld  hl, #Key R
  call cpct isKeyPressed asm
  jr z,  final loop

  call finalizamos
```

De igual forma creamos el botón reset game para resetear el juego si estamos en el menú pause.

```
push de
 ld ( tile map colision), de
   ;;CAMBIAR RENDER TILE MAP
   call render_tilemap_0

 pop de
 ld ( tile map colision), de
 push de
   ;;CAMBIAR RENDER TILE MAP
  call render_tilemap_1

 pop de
 ld ( tile map colision), de
 push de
   ;;CAMBIAR RENDER TILE MAP
  call render tilemap 2

 pop de
 ld ( tile map colision), de
   ;;CAMBIAR RENDER TILE MAP
  call render tilemap 3

  call rendersys pause update

  jp loop

  ret
```

Llamamos a las renderizaciones de los tilemaps por partes ya que recordamos que dividimos nuestros tilemaps en 4 partes.

```
eliminar todo::

   call  entity man getEntityArray IX  ;; array de entidades  -> IX

 desloop2:

  ld a, 0(ix) ;; cargo tipo de entidad  0000 0011
  or a ;; resto los tipos               0000 0011
    jr z , actualizar entidades;;                                    0000
0011

  ld a, # e type dead
  ld 0(ix), a

   ld      bc,#entity size ;; carga la cantidad de atributos que tiene
entity  en bc
  add    ix, bc
   jr        desloop2

actualizar entidades:
  call entity update

  ret
```

Llamaremos a eliminar_todo una vez nos maten o pasemos de nivel para eliminar las entidades y volver a crearlas con el fin de resetear el mismo nivel o continuar al nivel siguiente.

```
ganar::
```

```
  call eliminar_todo

 ld hl, # screenganar_z_end
 ld de, #0xFFFF
  call cpct_zx7b_decrunch_s_asm

 ld hl , #0x000F
  call cpct_setDrawCharM0_asm

 ld iy , # final_muertes
 ld hl , #0xC0B3
  call my_drawStringM0

  call  cargar_numero_muertes

 ld iy , # numero_muertes
 ld hl , #0xC15F
  call my_drawStringM0

  call finalizamos

  ret


finalizamos::
game_over3:

 ld  hl, #Key_R
  call cpct_isKeyPressed_asm

  jr z, game_over3

 ld a, #0
 ld ( IndiceNiveles),a
  ;ld ( contador_vidas),a

  call  _main

  ret
```

Si tocamos a la princesa y ganamos el juego, se llamará a "ganar" que nos mostrará la pantalla de victoria junto con todas las veces que has muerto y esperará a que presionemos el botón R para resetear el juego.

```
cargar_numero_muertes::
 ld hl , # numero_muertes
 ld bc , # contador_vidas

 ld a , (hl)
 ld e , a
 ld a , (bc)
 ld d , a
 ld a , e
 add d

 ld (hl) , a ;; millar

 inc hl
 inc bc
```

```
ld a , (hl)
ld e , a
ld a , (bc)
ld d , a
ld a , e
add d

ld (hl) , a ;; centenas

inc hl
inc bc

ld a , (hl)
ld e , a
ld a , (bc)
ld d , a
ld a , e
add d

ld (hl) , a ;; decenas

inc hl
inc bc

ld a , (hl)
ld e , a
ld a , (bc)
ld d , a
ld a , e
add d

ld (hl) , a ;; unidades

 ret
```

Por último, cargamos el número de veces que hemos muerto para mostrarlo por pantalla una vez ganemos el juego.

- Collisions:

```
comprobacion primera 0 entidad::

  ld     a, 0(ix) ;; cargo el primer tipo  1000
  and    b                ;              0000
  sbc    a,b              ; 0000 - 0100 = 00 - 04 = FB -> 1111 1011

   ret

comprobacion primera 18 entidad::

  ld     a, 18(ix) ;; cargo el primer tipo  1000
  and    b                ;              0000
  sbc    a,b              ; 0000 - 0100 = 00 - 04 = FB -> 1111 1011

   ret

;;Input B ->tipo
comprobacion segunda 0 entidad::
```

```
  ld      a, 0(iy) ;;             0000
  and     b                     ; 0000
  sbc     a,b                 ; 0000 - 0001 = 00 - 04 = FF -> 1111 1011

   ret

;;Input B ->tipo
comprobacion segunda 18 entidad::

  ld      a, 18(iy) ;;            0000
  and     b                     ; 0000
  sbc     a,b                 ; 0000 - 0001 = 00 - 04 = FF -> 1111 1011

   ret
```

En cuanto a las colisiones, primero comprobamos qué tipo de entidades son las que están colisionando; si son del primer tipo o del segundo y dentro de cada uno comprobamos las propiedades 0 y 18 de esas entidades.

```
colision check::

  ld      a, 1(ix)
  add     3(ix)
  sub     #3
  sub     1(iy)
   jr       c, no colision

  ld      a, 1(iy)
  add     3(iy)
  sub     #3
  sub     1(ix)
   jr       c, no colision

  ld      a, 2(ix)
  add     4(ix)
  sub     2(iy)
   jr       c, no colision

  ld      a, 2(iy)
  add     4(iy)
  sub     2(ix)
   jr       c, no colision

   ;; IF DE PLAYER
  ld      b , # e type input
   call     comprobacion primera 0 entidad
   jr       nz, no player

   ;; IF DE ENEMY
  ld      b, # e type ia
   call     comprobacion segunda 0 entidad
   jr       z , matar_player
```

```
   ;;IF BULLET ENEMY
  ld      b , # e type bullet enemy
   call    comprobacion segunda 18 entidad
   jr      z, matar player

   ;; IF ARROW
  ld      b , # e type arrow
   call    comprobacion segunda 18 entidad
   jr      nz, no enemy

matar player:
   call    entity set4destroy

   call    mismo nivel

no enemy:
  ld      b, # e type princess
   call    comprobacion segunda 18 entidad
   jr      nz, no princess

   call    ganar

no princess:
  ld      b, # e type portal
   call    comprobacion segunda 18 entidad
   jr      nz, no player

   call    next level
no player:
   ;; IF ENEMY - BULLET
  ld      b, # e type ia ; 0100
   call    comprobacion primera 0 entidad
   jr      nz, no colision

  ld      b, # e type bullet ; 0100
   call    comprobacion segunda 0 entidad
   jr      nz, no colision

   ;;IF BULLET ENEMY
  ld      b, # e type bullet enemy ; 0100
   call    comprobacion segunda 18 entidad
   jr      z, no colision

  ld      a,  # e type dead ;;cargo en a el tipo invalido
  ld      0(ix), a
  ld      0(iy), a

   call    reset disparo

no colision:
    ret
```

En la etiqueta colision_check comprobamos si existe colisiones entre las diferentes entidades como enemy-player, bullet-enemy, etc.

```
mismo nivel::
  ld      a, ( IndiceNiveles)
  dec     a
  dec     a
```

```
  dec     a
  dec     a
  ld      ( IndiceNiveles), a

   ;ld       a , ( contador vidas)
   ;inc      a
   ;ld       ( contador vidas), a
  pop af
  pop af
  pop af
  pop af
  pop af
  pop af
  pop af
  pop af
  pop af
  pop af
  pop af
   call     cargador nivel

   ret

next level::

  pop af
  pop af
  pop af
  pop af
  pop af
  pop af
  pop af
  pop af
  pop af
  pop af
   call     cargador nivel

   ret
```

Las etiquetas globales "mismo_nivel" y "next_level" sirven para extraer de la pila todo aquello que esté usándose en ese momento y que vaya a dejar de usarse cuando reiniciemos el mismo nivel por muerte o pasemos al siguiente nivel.

```
colision update::

  ld      d, # e type collision
  ld      hl, #colision update one entity
   call     man entity forall

  ld      d, # e type bullet
  ld      hl, #colision update bullet entity
   call     man entity forall

   call     entity dead
   jr       z, salida
  ld      a, #0x01
salida:
   ret
```

Posteriormente llamaremos a "colision_update" para comprobar si estamos haciendo colisión con una entidad enemiga o con una entidad bala.

```
colision_update_one_entity::

  push    ix
  pop     iy

   ld       bc,#entity_size ;; carga la cantidad de atributos que tiene
entity  en bc
  add     iy, bc

 desloop2:
   ;; IF ENEMIGO PEQUEÑO
  ld      a, 18(ix)
  ld      b, # e type enemy small
  and     a, b
  sbc     a,b
   jr       nz, col normal

   call     colisiones enemigo pequeño
   jp       seguir
col normal:

   call     colisiones

seguir:
  ld      a, 0(iy) ;; cargo tipo de entidad  0000 0011
  or      a ;; resto los tipos               0000 0011
   ret      z ;;                              0000 0011

   call     colision_check

   ld       bc,#entity_size ;; carga la cantidad de atributos que tiene
entity  en bc
  add     iy, bc
   jr         desloop2
```

En la etiqueta "colision_update_one_entity" comprueba si las colisiones de las entidades son o no son bala.

```
colisiones_enemigo_pequeño::

  ld      b, #6
  ld      c, #3
   call colision abajo map
  or      a
   jr nz, colision izquierda peq

  ld      b, #6
  ld      c, #2
   call colision_abajo_map
```

```
  or      a
   jr nz, colision_izquierda_peq

  ld      b, #6
  ld      c, #1
   call colision_abajo_map
```

En "colisiones_enemigo_pequeño" comprobamos las colisiones izquierda y abajo del enemigo *shuriken*.

```
colisiones::


  ld b, #15
  ld c, #3
   call colision_abajo_map
  or a
   jr nz, colision_izquierda

  ld b, #15
  ld c, #5
   call colision_abajo_map
  or a
   jr nz, colision_izquierda

  ld b, #15
  ld c, #1
   call colision_abajo_map
```

En "colisiones" comprobamos las colisiones izquierda y abajo del player

```
colision_update_bullet_entity::

  ld      b, #6
  ld      c, #3
   call     colision_abajo_map
  or      a
   jr       nz, matar_bullet

  ld      b, #6
  ld      c, #0
   call colision_abajo_map
  or      a
   jr       nz, matar_bullet

  ld      b, #6
  ld      c, #6
   call colision_abajo_map
  or      a
   jr       nz, matar_bullet

col_izquierda:
  ld      b, #0 ;Y   -> hacia abajo
  ld      c, #4 ;X   -> hacia la izquierda
```

```
   call    colision_izquierda_map
  or     a
   jr      nz, matar_bullet

  ld     b, #6 ;Y
  ld     c, #4 ;X
   call    colision_izquierda_map
  or     a
   jr      nz, matar_bullet

col_derecha:
  ld     b, #0
  ld     c, #6
   call    colision_derecha_map
  or     a
   jr      nz, matar_bullet

  ld     b, #6
  ld     c, #6
   call    colision_derecha_map
  or     a
   jr      nz, matar_bullet

   jp      salir_col_bullet

matar_bullet:
  ld     0(ix), # e_type_dead

   ;;IF BULLET ENEMY
  ld     b , # e_type_bullet_enemy
   call    comprobacion_primera_18_entidad
   jr      nz, no_bullet_enemy

   call    reset_disparo_enemy

   jp      salir_col_bullet
no_bullet_enemy:
   call    reset_disparo

salir_col_bullet:
   ret
```

En "colision_update_bullet_entity" comprobamos si se está colisionando con una bala y si se produce esa colisión, comprobamos si está siendo por la zona derecha, izquierda y debajo.

```
reset_disparo_enemy::

  ld     h , 15(ix)
  ld     l , 14(ix)

  push   ix  ;; guardo ix de bullet

  push   hl  ;; guardo ix de enemy
  pop    ix  ;; saco ix de enemy

  ld     13(ix), #0 ;; enemy

  pop    ix ;; sacar el ix de bullet
```

```
   ret
```

Aquí reseteamos el disparo del enemigo shooter, es decir, cuando dispare y colisione con el jugador o con un bloque del mapa, se eliminará esa bala y se podrá volver a disparar otra.

```
reset_disparo::

  push    ix
   call     entity_man_getEntityArray IX
  ld      13(ix), #0
  pop     ix

   ret
```

Aquí hacemos lo mismo que con el disparo del shooter pero esta vez con el de player.

```
colision_mata_jugador::
  ld      a ,(hl) ;; INDICE DEL TILE DEL LADO IZQUIERDO

  cp      #2
   jr       z, matar_personaje

  cp      #3
   jr       z, matar_personaje

  cp      #4
   jr       z, matar_personaje

  cp      #5
   jr       z, matar_personaje

  cp      #7
   jr       nz, fin_matar


matar_personaje:
   call mismo_nivel

fin_matar:
   ret
```

En "colision_mata_jugador" comprobamos si alguna colisión con algún enemigo, bala enemiga y obstáculo está en contacto con el jugador, si es así, el jugador muere.

```
reposicion::
  cp      #1
   jr       z, reposicion_salir;;COLISIONA CON TILE SUELO

  cp      #6
   jr       z, reposicion_salir;;COLISIONA CON TILE SUELO

  cp      #8
   jr       z, reposicion_salir;;COLISIONA CON TILE SUELO
```

```
  cp      #9
   jr       z, reposicion_salir;;COLISIONA CON TILE SUELO

  cp      #10
   jr       z, reposicion_salir;;COLISIONA CON TILE SUELO

  cp      #11
   jr       z, reposicion_salir;;COLISIONA CON TILE SUELO

  cp      #12
   jr       z, reposicion_salir;;COLISIONA CON TILE SUELO

  cp      #13
   jr       z, reposicion_salir;;COLISIONA CON TILE SUELO

  cp      #14
   jr       z, reposicion_salir;;COLISIONA CON TILE SUELO

reposicion_salir:
   ret
```

En "reposición", reposicionamos al jugador cuando entra en contacto con algún tile del tilemap que no es obstáculo, es decir, algún bloque que no pueda traspasar.

```
colision_arriba_map::

   ;;TX = x/4
   ;;TY = Y/8
   ;;TW = tilemap-width (20)
   ;; P = tilemap + TY * TW + TX

   call    colision_up_down_map
   ret     z ; NO COLISIONA

   call    reposicion
   jr      z, reposicion_up

  ld     b, # e type input ; 0100
   call    comprobacion_primera_0_entidad
   jr      nz,  fin_up

   call    colision_mata_jugador
   jr      nz, fin_up

reposicion_up:

  ld     b, # e type bullet
   call    comprobacion_primera_0_entidad
   jr      nz , seguir_reposicion_arriba

   call    matar_bullet

seguir_reposicion_arriba:

;;  REPOSICION EN Y

  ld     a, 2(ix)
  ld     b, #5
```

```
  add     a, b

  ld      2(ix), a
  ld      6(ix), #1
  ld     11(ix), #-1

fin_up:
   ret
```

En "colision_arriba_map" comprobamos todas las colisiones posibles por la zona superior de la entidad que se está tratando en ese momento.

```
colision_abajo_map::

   ;;TX = x/4
   ;;TY = Y/8
   ;;TW = tilemap-width (20)
   ;; P = tilemap + TY * TW + TX
   call    colision_up_down_map
   ret     z ; NO COLISIONA

   call    reposicion
   jr      z, reposicion_down

  ld     b, # e_type_input ; 0100
   call    comprobacion_primera_0_entidad
   jr      nz,  fin_down

   call    colision_mata_jugador
   jr      nz, fin_down

reposicion_down:

  ld     b, # e_type_bullet
   call    comprobacion_primera_0_entidad
   jr      nz , seguir_reposicion

   call    matar_bullet

seguir_reposicion:
;;  REPOSICION EN Y
  ld     a , c
  add    a , a ; ty *2
  add    a , a ; ty * 4
  add    a , a ; ty * 8
  push   af

  ld     b, # e_type_enemy_small
   call    comprobacion_primera_18_entidad
   jr nz, col_down_normal
  pop    af
  add    #-6
   jp      seguir_down

col_down_normal:
  pop    af
  add    #-15

seguir_down:
  ld     2(ix) , a
```

```
  ld      6(ix), #0
  ld      11(ix), #0


fin_down:
   ret
```

En "colision_abajo_map" comprobamos todas las colisiones posibles por la zona de debajo de la entidad que se está tratando en ese momento.

```
colision_izquierda_map::

   ;;TX = x/4
   ;;TY = Y/8
   ;;TW = tilemap-width (20)
   ;; P = tilemap + TY * TW + TX

   call    colision izq der map
   ret     z ; NO COLISIONA

   call    reposicion
   jr      z, reposicion left

  ld     b, # e type input ; 0100
   call    comprobacion primera 0 entidad
   jr      nz,  fin left

   call    colision mata jugador
   jr      nz, fin left

reposicion left:


;;  REPOSICION EN X
  ld a, 1(ix)
  ld b, #-1
  sbc a, b

  ld      1(ix) , a


  ld      5(ix), #1

  ld     b, # e type bullet
   call    comprobacion primera 0 entidad
   jr      nz , fin left

   call    matar bullet



fin left:
   ret
```

En "colision_izquierda_map" comprobamos todas las colisiones posibles por el lado izquierdo de la entidad que se está tratando en ese momento.

```
colision_derecha_map::
```

```
   ;;TX = x/4
   ;;TY = Y/8
   ;;TW = tilemap-width (20)
   ;; P = tilemap + TY * TW + TX
   call    colision_izq_der_map
   ret     z ; NO COLISIONA

   call    reposicion
   jr      z, reposicion_right

  ld     b, # e type input ; 0100
   call    comprobacion primera 0 entidad
   jr      nz,  fin right

   call    colision mata jugador
   jr      nz, fin right

reposicion right:
  ld     b, # e type bullet
   call    comprobacion primera 0 entidad
   jr      nz , continuar

   call    matar bullet

continuar:
;;  REPOSICION EN X
  ld     a, 1(ix)
  ld     b, #0
  sbc    a, b
  ld     1(ix) , a

  ld     5(ix), #0xFF


fin right:
   ret
```

En "colision_derecha_map" comprobamos todas las colisiones posibles por el lado derecho de la entidad que se está tratando en ese momento.

```
colision_izq_der_map::

  ld     a, 2(ix)
  add    b
  srl    a
  srl    a
  srl    a  ;; Y/8

  ld     h , #0
  ld     l , a
  push   hl

  ld     h, #0
  ld     l,a
  add    hl,hl ; 2
  add    hl,hl ;4
  ld     d,h
  ld     e,l
  add    hl,hl ;8
  add    hl,hl ;16
```

```
  add     hl,de  ;; Y * 20

  ld      a, 1(ix)
  add     c
  srl     a
  srl     a ;; X/4

  ld      b,#0
  ld      c,a
  add     hl , bc
  ld      de, ( _tile_map_colision)
  add     hl, de
  dec     hl

  pop     bc
  ld      a ,(hl) ;; INDICE DEL TILE DEL LADO IZQUIERDO
  cp      #0

   ret
```

En "colision_izq_der_map", comprobamos si los valores por parámetro (usamos los registros b y c como si fuesen parámetros) corresponden a cualquier tile que encontremos en el mapa y si la entidad que estamos tratando en ese momento está colisionando con ese tile, la reposiciona haciendo que no atraviese los bloques de izquierda o derecha en caso de ser un bloque o mata al jugador en caso de ser un obstáculo.

```
colision up down map::

  ld      a, 2(ix)
  add     b
  srl     a
  srl     a
  srl     a  ;; Y/8

  ld      h , #0
  ld      l , a
  push    hl

  ld      h, #0
  ld      l,a
  add     hl,hl ; 2
  add     hl,hl ;4
  ld      d,h
  ld      e,l
  add     hl,hl ;8
  add     hl,hl ;16
  add     hl,de  ;; Y * 20

  ld      a, 1(ix)
  add     c
  srl     a
  srl     a ;; X/4

  ld      b,#0
  ld      c,a
  add     hl , bc
  ld      de, ( _tile_map_colision)
```

```
  add    hl, de

  pop    bc
  ld     a ,(hl) ;; INDICE DEL TILE COLISION
  cp     #0

   ret
```

En "colision_up_down_map", comprobamos si los valores por parámetro (usamos los registros b y c como si fuesen parámetros) corresponden a cualquier tile que encontremos en el mapa y si la entidad que estamos tratando en ese momento está colisionando con ese tile, la reposiciona haciendo que no atraviese los bloques de izquierda o derecha en caso de ser un bloque o mata al jugador en caso de ser un obstáculo.

- my_drawString:

```
.module cpct strings

.include "strings/strings.s"


my drawStringMO::

  jr     firstChar                    ;; [3] Jump to first char (Saves 1
jr back every iteration)

nextChar:
  ;; Draw next character
 push hl                             ;; [4] Save HL
  call  my drawChar Inner  ;; [5 + 824/832] Draws the next character
 pop  hl                             ;; [3] Recover HL

  ;; Increment Pointers
 ld   de, #4                         ;; [3] /
 add  hl, de                               ;; [3] | HL += 4 (point to next
position in video memory, 8 pixels to the right)
 inc  iy                                   ;; [3] IX += 1 (point to next
character in the string)

firstChar:
 ld     a, (iy)                      ;; [5] A = next character from the
string
 or     a                            ;; [1] Check if A = 0
  jr     nz, nextChar                      ;; [2/3] if A != 0, A is next
character, draw it, else end

endstring:
```

```
  ret
;; IX/IY Restore and Return provided by bindings
```

Para dibujar strings nos hemos servido de este código propio de CPCTelera con ligeras modificaciones.

- my_drawChar:

```
.module cpct_strings

;;
;; Include constants and general values
;;
.include "macros/cpct_undocumentedOpcodes.h.s"


my_drawChar_Inner::
   ;; Calculate the memory address where the 8 bytes defining the character appearance
   ;; ... start (IX = 0x3800 + 8*ASCII value). char0 ROM address = 0x3800
 sub #32
 rlca          ;; [1] | A = E = 8*ASCII. 3 RLCA leave A with this
  rlca               ;; [1] |     |hgfedcba| => 3*RLCA => |edcba|hgf| IXH         IXL
 rlca             ;; [1] | Then we need to move it to IX like this => |00111hgf| |edcba000|
 ld    e, a       ;; [1] \ That will be the final memory address where the definition starts
 and   #0x07      ;; [2] Isolate latest 3 bits of a |00000hgf|
  ;or     #0x38     ;; [2] Add the 3 ones in front, so that the address starts at 0x38xx => |00111hgf|
  ld   ixh a       ;; [2] Save it to IXH = |00111hgf|
 ld    a, e       ;; [1] Restore A status after 3*RLCA => |edcba|hgf|
 and   #0xF8      ;; [2] Isolate first 5 bits => |edcba|000|
  ld   ixl a       ;; [2] and save it to IXL = |edcba|000|
  ;; Now IX = |edcba|000||00111hgf| = 0x3800 + 8*ASCII

 ld  bc, #0x0040
 add ix , bc

 ld    bc, #dc_2pxtableM0     ;; [3] BC points to the 2 1-bit pixels to 2 4-bit pixels conversion table

  ;; Draw next line from the character to the screen
nextline:
 ex    de, hl       ;; [1] Put Destination pointer into DE (it is in HL)
 ld     a, (ix)     ;; [5] A = Next Character pixel line definition
                    ;; .... (8 bits defining 0 = background colour, 1 = foreground)
  ;; Copy the 4 bytes that compose the complete pixel line
  ;; repeating the code for each pair of pixels to maximize speed
.rept 4
   ;; Convert next 2-bits into 1 byte with 2 pixels in screen pixel format
  ;; and copy it to (DE) which is next screen location
 ld    hl, #0       ;; [3] HL = 0
 rlca               ;; [1] /     Put the 2 leftmost bits of A into the two
```

```
 rl    l               ;; [2] | ...rightmost bits of L. This is the
combination for the
 rlca                  ;; [1] | ...next 2 pixels (BG-BG, BG-FG, FG-BG,
FG-FG). We use it
 rl    l               ;; [2] \ ...as index for the dc 2pxtableM0 which
gets the actual pixel values.
 add   hl, bc        ;; [3] HL = BC + L (2pxtableM0 + Index = HL Points
to the converted pixel values)
 ldi                 ;; [5] Copy 2 pixels to the screen, incrementing DE
at the same time
 inc   bc            ;; [2] BC is decremented by LDI but we want it to
keep pointing to the table, so we add 1 again
.endm

endpixelline:
  ;; Move to next pixel-line definition of the character
    inc  ixl                 ;; [2] Next pixel Line (characters are
8-byte-aligned in memory,
                         ;; ... so we only need to increment IXL, as IXH
will not change)
   ld  a ixl             ;; [2] If next pixel line corresponds to a new
character
                          ;; .... (this is, we have finished drawing our
character), ....
 and   #0x07         ;; [2] ... then L % 8 == 0, as it is 8-byte-aligned.
  ret    z             ;; [2/4] If L % 8 == 0, we have finished drawing
the character, else, proceed to next line

  ;; Prepare to copy next line
  ;;  -- Move DE pointer to the next pixel line on the video memory
   ;; (We save new calculations on HL, because it will be exchanged
with DE at the start of nextline: loop)
 ld    hl, #0x800-4       ;; [3] | Next pixel line is 0x800 bytes away
in standard video modes
  add    hl, de                   ;; [3] | ..but DE has already being
incremented 4 times. So add 0x800-4 to
                            ;;       ..DE to make it point to the start
of the next pixel line in video memory
   ;; Check if new address has crossed character boundaries (every 8
pixel lines)
 ld     a, h                ;; [1] A = H (top 8 bits of video memory
address)
 and   #0x38                ;; [2] We check if we have crossed memory
boundary (every 8 pixel lines)
   jr     nz, nextline         ;; [2/3]   by checking the 4 bits that
identify present memory line.
                           ;; .... If 0, we have crossed boundaries
boundary crossed:
 ld    de, #0xC050        ;; [3] | HL = HL + 0xC050: Relocate DE pointer
to the start of the next pixel line in video memory
 add   hl, de               ;; [3] \ (Remember that HL and DE will be
exchanged at the start of nextline:)
  jr    nextline           ;; [3] Jump to continue with next pixel line

;; Conversion table from 2 1-bit pixels to mode 0 2 4-bit pixels.
Essentially, there are 4
;; possible combinations with 2 pixels and 2 colours: (00, 01, 10, 11
== BG-BG, BG-FG, FG-BG, FG-FG)
;;   We   reserve   here   4   bytes   that   will   be   filled   in   by
<cpct setDrawCharM0>
;;
```

```
dc_2pxtableM0:: .db 0x00, 0x40, 0x80, 0xC0    ;; Default colours BG=0, FG=1
```

Asimismo, también nos hemos ayudado de este código de CPCTelera para ayudarnos a dibujar Strings.

- pantallas:

```
.area _CODE

 string_next_level: .asciz "LEVEL COMPLETE"



pantalla_next_level::

 ld hl , #0x000F
  call cpct_setDrawCharM0_asm

 ld iy , # string_next_level
 ld hl , #0xC41F
  call my_drawStringM0

  ld      a , #1
  ld      (contar) ,a
esperar:
  ld      a , (cont_esperar)
  cp      #500
   jr       nz, esperar

  ld      a , #0
  ld      (contar) ,a
  ld      (cont_esperar), a

   ret
```

Por último, utilizamos este código para mostrar "LEVEL COMPLETE" cada vez que superemos un nivel.

Detalle de los pasos de elaboración

Materiales utilizados

- Distribución Linux Manjaro 20.1 XFCE
- Editor de código: Visual Studio Code
- CPCTelera
- Ensamblador
- Emulador Winape
- Emulador Retro Virtual Machine
- Gimp
- Arkos Tracker 1
- Tiled
- Discord
- WhatsApp
- Github

Problemas encontrados y soluciones adoptadas

1. Primer problema y solución:

   El primer problema que tuvimos fue el dibujado del sprite ya que no renderizaba por pantalla nuestras entidades.

2. Segundo problema y solución:

   El segundo problema que tuvimos fue el borrado de entidades una vez estaban eliminadas, se arregló dibujando un rectángulo negro encima de la entidad muerta.

3. Tercer problema y solución:

   Sólo podíamos crear una entidad al mismo tiempo y renderizarla, se solucionó revisando el código que actualizaba cada entidad ya que teníamos que hacer un push a la pila.

4. Cuarto problema y solución:

   Cuando se movía por pantalla una entidad dejaba un rastro de colores, se arregló mediante el repintado de tilemap.

5. Quinto problema y solución:

   El repintado de tilemap ralentizaba el programa y el doble buffer no servía. Se solucionó mediante el pintado con xor.

6. Sexto problema y solución:

   El personaje cuando saltaba dejaba un rastro, se solucionó implementando una tabla de saltos.

7. Séptimo problema y solución:

   Cuando moríamos muchas veces la pila incrementaba y colapsaba el juego ya que se sobreescribían posiciones de memoria. Se solucionó usando pops cada vez que morías.

8. Octavo problema y solución:

   Las colisiones del personaje con los obstáculos fallaban y colisionaban antes de tiempo. Se solucionó revisando el código de colisiones y adaptando un bounding box más pequeño al cuerpo del personaje para dar cierto grado de maniobrabilidad a la hora de esquivar enemigos y obstáculos.

9. Noveno problema y solución:

   Cuando se usó el doble buffer se implementaron animaciones pero como se tuvo que quitar, estas animaciones dejaban rastro por lo que se optó por no utilizarlas al usar el pintado xor.

10. Décimo problema y solución:

    Dibujaba las entidades a diferentes velocidades dependiendo de cuantas entidades mostramos por pantalla, se solucionó mediante la implementación de interrupciones.

11. Undécimo problema y solución:

    Al intentar hacer las interrupciones, cada vez que moríamos o cambiábamos de nivel, estas se veían alteradas. Se solucionó mediante dudas al profesor.

12. Duodécimo problema y solución:

    Que al dibujar la flecha usábamos un mismo tipo entidad para la flecha y para la bala, entonces si hacíamos un cambio en alguna de las dos, se alteraban las dos. Se solucionó un tipo de entidad diferente para la flecha y para la bala.

Reflexiones sobre las lecciones aprendidas

Gracias a esta práctica de crear un videojuego en Amstrad CPC 464, hemos aprendido a programar a bajo nivel, y todo lo que ello implica; es decir, hemos aprendido a gestionar de forma correcta la memoria y los recursos que nos ofrece Amstrad, con el fin de crear un videojuego que aproveche al máximo posible las características de este ordenador.

También hemos aprendido a ser autodidactas y a diseñar y programar las cosas nosotros mismos como ingenieros, cualidad que nos será muy útil en nuestra vida laboral.

Todos estos conocimientos a nivel teórico nos servirán llegado el momento de programar nuestro videojuego del ABP, ya que podremos aprovechar los aprendizajes que hemos adquirido con el fin de mejorar al máximo posible el producto final que se presentará en el ABP.