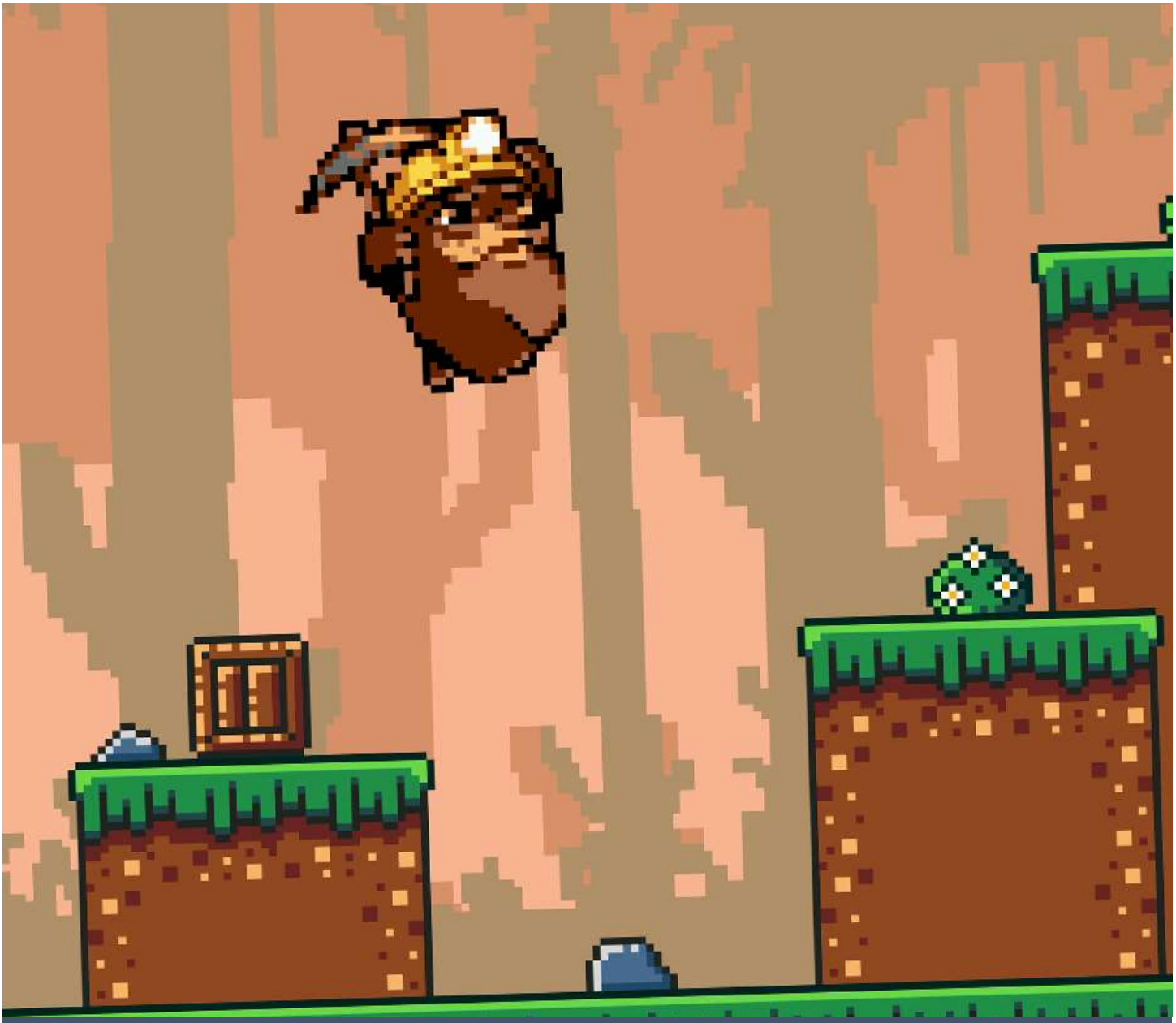


Desenvolupament de videojocs en 16-bits

Programari homebrew en ple 2021

Autor: Gerard Gascón Moliné

Tutora: Núria Betriu Caubet

Institut Joan Brudieu
2021-2022

RESUM

Avui en dia es desenvolupen aplicacions fent ús de les últimes tecnologies. En aquesta recerca s'ha plantejat fer justament el contrari, crear programari utilitzant la tecnologia del passat.

A partir de la base de coneixement obtinguda durant la investigació de l'arquitectura d'una consola del 1988, s'han escrit una sèrie de programes per descobrir el funcionament intern que té aquesta.

Seguint aquest concepte s'ha prosseguit amb el desenvolupament d'un producte original seguint les limitacions i estètiques de l'època. Posteriorment, s'ha redactat el procés i així mateix, s'han après encara més característiques internes que té aquesta.

Finalment, s'ha pogut veure com la construcció d'aquests programes donen peu a diferents mètodes de preservació de dades, juntament amb l'apreciació de les limitacions i com reduir el cost de rendiment que comporten certes estructures.

ABSTRACT

Nowadays apps are developed using the latest technologies. The goal of this project is to do the exact opposite, create a program that works in the technology of the past.

Starting with the base of knowledge obtained during the investigation of a 1988 game console, it has been written a series of programs for discovering how are its inner works.

In addition, it has been developed an original product with the limitations and aesthetics that were the day-to-day of the time. Every single step of this process has been written down, by doing this, we acquire a much deeper knowledge of the inside of the console.

Finally, it has been seen how these programs can lead to different methods for data preservation, along with the appreciation of the limitations and how to improve the performance by knowing the knockbacks of using certain inefficient structures.

Índex de continguts

Índex de figures

Agraïments

En l'elaboració d'aquest treball, vull agrair a la Núria Betriu, la meva tutora, per la confiança mostrada i el seu acompanyament al llarg de tot el projecte, a la meva tieta, la Laura Moliné, que m'ha deixat la seva consola de quan era petita la qual ha estat la base d'aquest treball, també a la meva padrina, la Teresa Grau, per la seva mania de no llançar res. Si no fos així, aquesta consola no hauria arribat a les meves mans. D'altra banda, al meu cosí Bru Arnau que ha imaginat el personatge principal del videojoc, i al grup de Telegram de desenvolupadors que han treballat anteriorment amb aquesta consola i m'han ajudat a entendre millor el seu funcionament. I finalment, a la meva germana, pares, padrí i iaia que m'han mostrat el seu suport en tot moment.

Introducció

Avui en dia, la immensa majoria de videojocs es desenvolupen mitjançant motors creats especialment per a ells. Sigui utilitzant Unity, Unreal Engine o qualsevol altre, els desenvolupadors en creen usant el màxim de facilitats que els hi aporten les noves tecnologies.

Tot i això, dins de tot aquell gran grup de desenvolupadors, n'hi ha un de petit que no busca la manera d'aprofitar el màxim potencial dels sistemes actuals. Si no que busca una manera de crear-ne un per a plataformes *retro* que ja han acabat obsoletes amb el pas del temps.

Aquesta pràctica ja pot semblar una manera d'aconseguir el somni de la infància de molts, en el que volien crear un videojoc per a la consola que ells havien tingut de petits. O bé pot ser un repte en el qual els programadors i dissenyadors intenten crear-ne un, adaptant-se a la gran quantitat de restriccions que comporta fer-ho per a alguna d'aquelles plataformes.

Justament és per aquest segon motiu pel qual he decidit fer aquest treball de recerca. Una manera de posar a prova les meves capacitats de programar. Limitant-me tant en àmbits artístics i musicals com computacionals.

Per a assolir l'objectiu final de crear un videojoc per a una consola dels anys 80-90, he buscat quines metodologies es poden seguir per a fer-ho. Així com llegir-me la poca documentació que existeix sobre la plataforma i el seu funcionament. Tot seguit, en haver obtingut una base de coneixement sobre la qual poder-hi treballar, he pogut començar a programar-lo utilitzant el llenguatge de programació C, juntament amb un IDE com ho és el Visual Studio.

El treball segueix una estructura progressiva, la qual s'inicia amb una breu explicació del terme que defineix el treball i posteriorment s'exposa el funcionament de la mateixa consola, les capacitats tècniques, les limitacions respecte a les actuals, etc.

Un cop es té una base sobre la qual entendre la lògica a seguir a l'hora de programar-hi, s'inicia el procés amb la programació d'un programa "Hola, món!". Finalment, s'acaba amb la creació del videojoc pas per pas, des de les primeres proves fins al resultat final.

1. Desenvolupament de programari Homebrew

Homebrew, quan s'aplica a videojocs, és un terme el qual utilitza la pirateria informàtica per a modificar un sistema, en aquest cas una videoconsola. I així capacitar-la per a poder exercir-hi altres funcions, entre les quals està inclosa la capacitat de crear-hi videojocs sense la necessitat de disposar d'un SDK oficial.

A partir de poder piratejar una videoconsola per tal d'executar-hi programari propi, es pot programar usant un *framework*, creat per apassionats. Aquest mètode facilita el desenvolupament mitjançant l'ús d'un llenguatge de programació més senzill d'emprar. Així com algunes funcions bàsiques per a estalviar temps de desenvolupament. Per altra banda, en el cas de les plataformes no tan modernes, es podria arribar a fer fent servir assemblador. Un llenguatge de baix nivell que bàsicament consisteix en un conjunt de mnemònics que representen instruccions bàsiques del processador, un llenguatge únic per a cada processador.

1.1. SGDK

SGDK és un exemple d'eina gratuïta per a crear aquest tipus de videojocs. Ha estat en constant desenvolupament des de la seva primera versió publicada el 2006 per Stephane Dallongeville. Aquesta et dona la possibilitat de crear videojocs per a la Sega Mega Drive utilitzant C en lloc d'assemblador, el qual era l'única manera de fer-ho fins aquell moment.

Figura 1.1: Logotip SGDK

1.2. Commodore 64

La Commodore 64 és una barreja entre consola i ordinador personal introduïda el 1982. La principal característica d'aquesta era ser programable per l'usuari. Això tot i semblar ser una consola pensada per al desenvolupament *homebrew*, trencava la principal característica d'aquest. Que era la necessitat d'haver-la de piratejar per a fer-ho. Per tant, els programadors no professionals que creaven videojocs per a la C64, simplement se'ls anomenava *hobbyists* o aficionats.


Figura 1.2: Commodore 64

2. Sega Mega Drive

2.1. Història

Figura 2.1: Sega Mega Drive model japonès.

La Sega Mega Drive, també coneguda com a Genesis a Nord Amèrica, és una consola de 16-bits que pertany a la quarta generació de videoconsoles. És la tercera consola desenvolupada per Sega i la successora de la Sega Master System. Es va posar a la venda el 1988 a Japó com a Mega Drive i el 1989 a Nord Amèrica com a Genesis. El 1990, va ser distribuïda com a Mega Drive per Europa, Oceania i Brasil. Aquesta consola compta amb una biblioteca de més de 900 títols distribuïts en cartutxos.

A Japó, la Mega Drive va ser un complet fracàs comercial a l'ombra dels seus principals dos competidors, la Super Famicom de Nintendo i el PC Engine de NEC, però va aconseguir una victòria considerable a Nord Amèrica, Brasil i Europa. Els principals contribuïdors a l'èxit de la consola van ser les diverses adaptacions de les recreatives, la popularitat de la sèrie de videojocs de Sonic The Hedgehog, diverses entregues d'esports populars i una campanya de màrqueting molt agressiva. En combinació a tot això, es va posicionar com a la consola "guai" per als adolescents.

Posteriorment el llançament de la SNES va provocar una batalla per la quota de mercat a Nord Amèrica i Europa coneguda com a *guerra de consoles*. Finalment la Sega Mega Drive va acabar venent més de 40 milions d'unitats.

2.2. Característiques

Aquest sistema es venia com a la primera consola de 16-bits. Això era una manera de fer publicitat contra les empreses competidores presumint de com el seu producte era millor. Realment aquest terme es referia a l'arquitectura del mateix processador i com aquest era capaç de treballar amb nombres més grans que les consoles anteriors. Per exemple, una consola de 8-bits, en unes condicions normals, només pot treballar en nombres del 0 al 255 o del -128 al 127, en canvi, una consola de 16-bits teòricament només pot treballar en nombres d'entre el 0 i 65.535 o del -32.768 al 32.767.

2.3. Processador

La consola disposa de dos processadors de propòsit general. El primer d'ells és el Motorola 68000, també conegut com a 68k, el qual funciona a aproximadament uns 7,6 MHz. Aquest processador ja estava present en bastants ordinadors d'aquella època, com l'Amiga, el Macintosh original, l'Atari ST, etc.

Figura 2.2: Placa base Mega Drive model europeu

El 68k és la principal CPU i s'utilitza per a fer funcionar la lògica del joc, gestionar l'entrada i sortida de bits (I/O) i fer els càlculs gràfics. Les seves característiques es resumeixen en què és un processador pensat per a treballar amb 16-bits tot i que té capacitats per a poder fer càlculs amb nombres de 32-bits amb l'inconvenient que per a fer-ho necessita 2 cicles, el doble de temps que tardaria a fer-ne un de 8 o de 16-bits.

L'altre processador dels quals disposa la consola és el Zilog Z80, a diferència del 68k, aquest funciona a 3,5 MHz. És el mateix processador que utilitzava la seva predecessora Sega Master System. S'usa principalment per al control de so i en el cas que el propietari introdueix-hi un videojoc de la consola de l'anterior generació, s'utilitza aquest com a processador principal i el 68k roman inactiu.

Les capacitats del Z80 es veuen força limitades a causa de la seva edat, i per això aquest només és capaç de fer operacions de 8-bits i tot i això, necessita 2 cicles per a fer-ho, a més només té 8 kB de memòria RAM a diferència dels 64 kB que té l'altre.

2.4. Gràfics

Les dades gràfiques són processades pel 68k i renderitzades per un xip patentat anomenat VDP el qual envia el fotograma resultant a la pantalla.

El VDP funciona a aproximadament uns 13 MHz i suporta múltiples modes de resolució depenent de la regió, fins a 320x224 píxels en NTSC (Japó i Amèrica) i fins a 320x240 píxels en PAL (Japó i Europa). En el cas que la consola sigui NTSC, aquesta emet un senyal de 60 fotogrames per segon i en el cas de ser PAL només de 50.

En aquest xip s'hi poden emmagatzemar els colors que s'estan visualitzant en aquell fotograma dins de quatre paletes de 16, tot i que el primer de cada una d'elles s'utilitza per a representar la transparència. En la pràctica només es poden veure fins a un màxim de 61 colors simultàniament, perquè el primer de la primera paleta s'usa per al fons. La imatge està dividida en quatre plans, el *Pla A* / *Window*, *Pla B*, *Pla Sprites* i el color de fons, a més de les seves respectives prioritats. Dins seu també compta amb 64kB de VRAM, que s'encarrega d'emmagatzemar totes les dades dels gràfics.

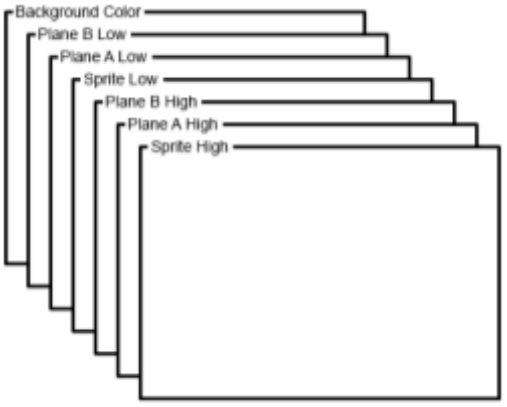


Figura 2.3: Esquema prioritats dels plans

Per a transportar les imatges des de la memòria del cartutx, disposem del DMA. Això ens proporciona una major velocitat a l'hora de transferir memòria sense la intervenció de la CPU. Aquest pot tenir una amplada de banda diferent segons el moment en el qual es faci servir. Addicionalment la CPU serà bloquejada mentre s'executi una transferència per DMA, això significa que hi ha d'haver certa precisió a l'hora de fer-ho, si envies més memòria de la que l'amplada de banda et permet pots veure certes pèrdues importants de rendiment.

2.5. Àudio

La Sega Mega Drive disposa de dos xips de so, el Yamaha YM2612 i el Texas Instruments SN76489. Cadascun d'aquests dos aporta unes funcionalitats molt diferents.

D'una banda, el de Yamaha és un sintetitzador FM que funciona a la mateixa velocitat del 68k i disposa de sis canals dels quals un d'ells pot ser utilitzat per a reproduir mostres PCM de resolució 8-bits i una freqüència de fins a 32 kHz.

Una de les tècniques que s'usa per al seu funcionament és la síntesi de modulació de freqüència o FM la qual n'és una de les tantes que s'utilitzen per a sintetitzar el so, aquesta va ser popularitzada durant la dècada dels 80. Aquest algoritme es basa en una ona ("transportadora") a la qual se li altera la freqüència mitjançant-ne una altra ("moduladora"), el resultat és una ona nova amb una freqüència i so nou.

L'altre mètode, la modulació per impulsos codificats o PCM n'és un el qual busca representar digitalment senyals analògics, tal com ho fa el format MP3. Això comporta que es poden enregistrar sons i llavors reproduir-los en el canal PCM, el

principal desavantatge que té aquesta tècnica és que requereix molta memòria per a emmagatzemar el so i en aquella època no era possible disposar-ne de molta.

D'altra banda hi ha el de Texas Instruments, un xip PSG el qual compta amb tres canals capaços de produir tres ones d'impuls diferents juntament amb un altre que produeix soroll blanc. Aquest xip ve integrat dins del VDP i funciona a la mateixa velocitat del Z80 perquè, igual que ell, també prové de la Sega Master System.

2.6. Cartutxos

Els cartutxos o PCBs de la Sega Mega Drive, originalment podien tenir una capacitat d'entre 512 i 8224 kB. Alguns d'ells podien portar SRAM, una petita memòria alimentada per una pila, aquest es feia servir per a guardar el progrés de la partida.

Per a bloquejar la importació de videojocs de l'estranger, tots els cartutxos tenien una forma diferent depenent de la regió. Tot i això, per a afegir-li un grau extra de seguretat, alguns jocs llegien el registre de versió de la consola per tal de comprovar que estiguessin funcionant en la versió adequada.

2.7. Construint un fotograma

En aquest apartat explico com el VDP dibuixa un fotograma, per a fer-ho utilitzo el videojoc *Sonic The Hedgehog* com a exemple.

2.7.1. *Tiles*

El VDP és un motor compost per *tiles*, cada un és una petita imatge de 8x8 píxels i cada fotograma es basa en el conjunt d'aquests. En cada un d'ells hi ha la informació per a saber quins colors i de quina paleta són els que li corresponen a cada píxel. Només es pot utilitzar una paleta per cada *tile*.

Cadascun d'ells pot estar capgirat horitzontalment o verticalment a més d'indicar-li la prioritat.

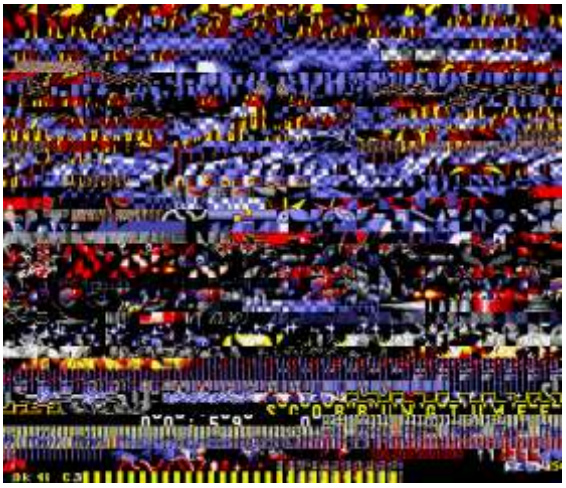


Figura 2.4: Captura de la VRAM del Sonic the Hedgehog

Els cartutxos en contenen a la seva memòria (ROM) però per a ser usats, han de ser copiats a la VRAM. Els *tiles* són emprats per a construir fins a un total de tres plans que se superposaran els uns als altres. Així que el VDP decidirà segons el tipus de pla i prioritat del *tile* quins seran visibles i quins no.

2.7.2. Pla B

El *Pla B* n'és un format per diversos *tiles* que es poden desplaçar. Aquest pot tenir sis dimensions diferents, totes múltiples de 8: 256x256, 256x512, 256x1024, 512x256, 512x512, 1024x256. Aquesta dimensió es pot escollir segons les necessitats que tinguis en el moment de desplaçar el pla.

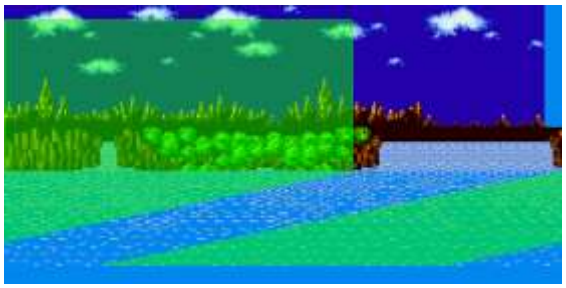


Figura 2.5: Captura del Pla B del Sonic the Hedgehog

Tal com es pot veure en l'exemple, la zona seleccionada per a mostrar-se en pantalla no és rectangular, i realment no necessita ser-ho. Això és perquè el VDP permet desplaçar certes línies dels plans horitzontalment, per la qual cosa només la zona verda es veurà en pantalla com si fos una sola imatge.

2.7.3. Pla A

El *Pla A* n'és l'altre, aquest té les mateixes propietats que el B, excepte que aquest pla té una prioritat més alta. Això comporta que els *tiles* que contingui aquest seran dibuixats per sobre dels del B.

Figura 2.6: Captura del Pla A del Sonic the Hedgehog

A més, aquest et permet dividir-se en un subplà, el *Pla Window*. La principal diferència és que aquest pla no es pot desplaçar.

2.7.4. *Sprites*

En aquest pla, els *tiles* són tractats com a *sprites*. És a dir que en lloc de moure's en distàncies de 8 píxels, aquests es mouen lliurement per una àrea de 512x512px i llavors la consola selecciona només aquells que estiguin dins de la resolució de la sortida de vídeo. A més, el VDP et facilita una funció de detecció de col·lisions força vella.

Figura 2.7: Captura del Pla Sprites del Sonic the Hedgehog

Els *sprites* es formen a partir de diversos *tiles*, des d'un sol, fins a un conjunt de 4x4, l'equivalent a 32x32px. Si tot i això en necessites un encara més gran, la solució és combinar-ne varis per tal d'aconseguir-ne un anomenat *metasprite*.

Hi pot haver fins a un màxim de 20 *sprites* alineats horitzontalment i un total de 80 en pantalla, si se n'afegeix algun més en un d'aquests casos, aquest no es dibuixa.

2.7.5. Resultat

Mentre el fotograma es dibuixa, el sistema estarà executant diverses interrupcions, aquestes poden ser de dos tipus: *H-Blank* ("cada línia horitzontal") i *V-Blank* ("cada fotograma").

Figura 2.8: Captura del Sonic the Hedgehog

L'*H-Blank* s'executa nombroses vegades, però està limitat a només poder executar unes rutines curtes que només poden accedir a la CRAM i a la VSRAM, la qual s'encarrega de fer el desplaçament vertical. Pel que durant aquestes, només es poden actualitzar les paletes o desplaçar verticalment els plans.

El *V-Blank* et permet executar unes rutines més llargues i més costoses, a més de poder accedir a tota la memòria de la consola. L'únic inconvenient és que només es pot executar unes 50 o 60 vegades per segon, depenent de si la consola és PAL o NTSC.

3. Creació d'un programa "Hola, món!"

Tot programa té els seus inicis, aquests sempre tenen un mateix nom, "Hola, món!". Això és definit pel seu propi contingut, tal com s'indica, simplement tracta d'escriure la frase "Hola, Món!" en la pantalla per a comprovar que el *framework* que s'està utilitzant funciona correctament.

En aquesta secció, en programo un de dues maneres diferents, en assemblador 68k i en C. Amb la finalitat d'experimentar amb els dos mètodes de programació. A més, busco un punt comú en tots dos que sigui relativament fàcil d'escriure i interpretar, per així poder-los comparar i saber les diferències que hi ha entre ells dos.

3.1. "Hola, món!" en assemblador 68k

Assemblador és un llenguatge creat en el 1949, però amb la principal diferència respecte als actuals que aquest canvia completament segons l'arquitectura del processador. Per tant un "Hola, món!" en la Sega Mega Drive no és el mateix que un en la Nintendo Entertainment System, així com tampoc ho és en la Neo Geo, la qual utilitza el mateix processador, però té unes adreces diferents.

3.1.1. Inicialitzar el programa

```
1. ;Variables RAM
2. Cursor_X equ $00FF0000      ;RAM posició X del cursor
3. Cursor_Y equ $00FF0000+1    ;RAM posició Y del cursor
4.
5. ;Ports vídeo
6. VDP_data    EQU    $C00000  ;Dades VDP, R/W només words o longwords
7. VDP_ctrl    EQU    $C00004  ;Control VDP, W només words o longwords
```

Primer defineixo dos símbols d'accés a adreces de la memòria RAM (Cursor *X* i *Y*), aquests els necessito més endavant per a poder escriure el text més fàcilment. També defineixo dos símbols més per a les dades i control de la VRAM, aquests els necessito per a llegir-la i escriure-la.

```
1. ; Adreces de memòria
2. DC.L    $FFFFFE00       ; Nombre del registre SP
3. DC.L    ProgramStart    ; Inici del programa
4. DS.L    7,IntReturn     ; Error bus, error adreça, instrucció il·legal,
   divisió entre 0, etc.
5. DC.L    IntReturn       ; TRACE
6. DC.L    IntReturn       ; Línia A (1010) emulador
7. DC.L    IntReturn       ; Línia F (1111) emulador
8. DS.L     4,IntReturn      ; Reservat / Coprocessador / Error format /
   Interrupció no init
9. DS.L    8,IntReturn     ; Reservat
10.DC.L    IntReturn       ; Interrupció spurious
11.DC.L    IntReturn       ; IRQ nivell 1
12.DC.L    IntReturn       ; IRQ nivell 2 EXT
13.DC.L    IntReturn       ; IRQ nivell 3
14.DC.L    IntReturn       ; IRQ nivell 4 Hsync
15.DC.L    IntReturn       ; IRQ nivell 5
16.DC.L    IntReturn       ; IRQ nivell 6 Vsync
17.DC.L    IntReturn       ; IRQ nivell 7
18.DS.L    16,IntReturn    ; Excepcions
19.DS.L    16,IntReturn    ; Varis (FP/MMU)
```

A l'inici del cartutx de la Mega Drive s'han de definir les excepcions. Aquestes són les actuacions que haurà de fer el processador en uns casos específics que són descrits en els diferents comentaris precedits per un punt i coma.

```
0. ; Capçalera
1. DC.B    "SEGA MEGA DRIVE"
2. DC.B    "(C)PROVA"
3. DC.B    "2021.JAN"
4. DC.B    "HOLA MON ASM"
5. DC.B    "HELLO WORLD ASM"
6. DC.B    "GM PROVA001-00"
7. DC.W    $0000
8. DC.B    "J"
9. DC.L    $00000000
10.DC.L    $003FFFFF
11.DC.L    $00FF0000,$00FFFFFF
12.DC.B    "            "
13.DC.B    "            "
14.DC.B    "UN HOLA MON EN ASM"
15.DC.B    "JUE"
```

La capçalera o *header* és una part fonamental de la ROM, aquest és el seu identificador. Conté absolutament tota la informació sobre el seu contingut, el qual es divideix de la següent manera i en el següent ordre:

1. Nom del sistema en el qual s'hi pot jugar.
2. Drets d'autor, o bé a qui li pertany.
3. La data en què s'ha creat aquella compilació.
4. Nom del cartutx.
5. Nom alternatiu o transoceànic de la ROM.
6. Número de sèrie, segueix l'estructura *TT NNNNNNNN-RR.* Les *T* simbolitzen el tipus de contingut, *GM* per als videojocs, les *N* indiquen el número del joc i les *R* la seva revisió.
7. El *Checksum* de la consola, és una mesura molt simple per a protegir les dades i verificar que no hagin estat corrompudes.
8. Dades dels controls, *J* per a indicar un control de 3 botons (*Joypad*), *K* per a teclats, *6* per al control de 6 botons, *C* per al CD-ROM.
9. L'adreça d'inici de la ROM.
10. Mida de la ROM.
11. Inici i final de la RAM.
12. Dades de la SRAM, per a guardar partides.
13. Indicar si aquesta ROM es pot connectar a internet.
14. Memòria, un text curt per a escriure una petita frase.
15. Regions compatibles, *J* per a Japó, *U* per a Amèrica, *E* per a Europa.

```
1. IntReturn:                ;Gestor d'interrupcions genèric
2.     rte
3. ProgramStart:             ;inicialitzar TMSS
4.     move.b ($A10001),D0   ;A10001 comprova la versió del sistema
5.     and.b #$0F,D0
6.     beq    NoTmss             ;executa la branca 'NoTmss' si no n'hi ha
7.     move.l #'SEGA',($A14000)  ;A14000 desactiva TMSS
8. NoTmss:
```

La capçalera necessita un gestor d'interrupcions genèric, aquest simplement és un *return*, s'encarrega de tallar l'execució del programa.

Ara que ja es pot iniciar el programa, el primer que faig és desactivar el TMSS en el cas que la consola en tingui.

3.1.2. Inicialitzar els gràfics

```
1.      lea VDPSettings,A5      ;Inicialitza els registres de pantalla
2.      move.l #VDPSettingsEnd-VDPSettings,D1 ;llargada de la configuració
3.
4.      move.w (VDP_ctrl),D0  ;C00004 llegeix estat de VDP
5.      move.l #$00008000,d5  ;Comandament registre VDP (%8rvv)
6.
7.  NextInitByte:
8.      move.b (A5)+,D5       ;Llegeix el següent byte de control de vídeo
9.      move.w D5,(VDP_ctrl)  ;C00004 escriu el comandament de registres al
    VDP
10.     add.w #$0100,D5       ;Apunta al següent registre del VDP
11.     dbra D1,NextInitByte  ;Repeteix per la resta del bloc
```

La pantalla la inicialitzo mitjançant *words* que les escric al *VDP_ctrl*. Aquests valors són llegits de la següent taula anomenada *VDPSettings*.

```
1.  VDPSettings:
2.      DC.B $04 ; 0 registre de mode 1
3.      DC.B $04 ; 1 registre de mode 2
4.      DC.B $30 ; 2 taula de noms desplaçament pla A (A=primers 3 bits)
5.      DC.B $3C ; 3 taula de noms pla Window (A=primes 4 bits / 5 en el
    Mode H40)
6.      DC.B $07 ; 4 taula de noms desplaçament pla B (A=primers 3 bits)
7.      DC.B $6C ; 5 taula d'atributs sprites (A=primers 7 bits / 6 en el
    H40)
8.      DC.B $00 ; 6 registre no utilitzat
9.      DC.B $00 ; 7 color de fons (P=paleta C=color)
10.     DC.B $00 ; 8 registre no utilitzat
11.     DC.B $00 ; 9 registre no utilitzat
12.     DC.B $FF ;10 registre interrupcions H (L=número de línies)
13.     DC.B $00 ;11 registre de mode 3
14.     DC.B $81 ;12 registre de mode 4 (C bits both1 = H40 Cell)
15.     DC.B $37 ;13 taula de desplaçament H (A=primers 6 bits)
16.     DC.B $00 ;14 registre no utilitzat
17.     DC.B  $02  ;15  increment  automàtic  (Després  de  cada  lectura  i
    escriptura)
18.     DC.B $01 ;16 tamany desplaçament (Tamany horitzontal i vertical del
    pla A i B)
19.     DC.B $00 ;17 posició horitzontal pla Window (D=Direcció C=Cel·les)
20.     DC.B $00 ;18 posició vertical pla Window (D=Direcció C=Cel·les)
21.     DC.B $FF ;19 Llargada DMA punt baix
22.     DC.B $FF ;20 Llargada DMA punt alt
23.     DC.B $00 ;21 Adreça DMA baixa
24.     DC.B $00 ;22 Adreça DMA mitjana
25.     DC.B $80 ;23 Adreça DMA alta (C=CMD)
26. VDPSettingsEnd:
27.     even
```

Aquí tinc un set d'eines per a inicialitzar una pantalla amb unes funcions bàsiques. Defineixo un byte per cada registre. Un cop fet això ja es poden començar a dibuixar elements en la pantalla.

```
1. ;Defineix paleta
2. move.l #$C0000000,d0    ;Color 0
3. move.l d0,VDP_Ctrl
4. ;         ----BBB-GGG-RRR-
5. move.w #%0000011000000000,VDP_data
6.
7. move.l #$C0020000,d0    ;Color 1
8. move.l d0,VDP_Ctrl
9. move.w #%0000000011101110,VDP_data
10.
11.move.l #$C0040000,d0    ;Color 2
12.move.l d0,VDP_Ctrl
13.move.w #%0000111011100000,VDP_data
14.
15.move.l #$C0060000,d0    ;Color 3
16.move.l d0,VDP_Ctrl
17.move.w #%0000000000001110,VDP_data
18.
19.move.l #$C01E0000,d0    ;Color 15 (Font)
20.move.l d0,VDP_Ctrl
21.move.w #%0000000011101110,VDP_data
```

Abans de tot, però, necessito una paleta de colors, per a fer això primer obtinc l'adreça en la qual es troba aquell color (*$C0NN0000*, on *N* és l'índex del color) i seguidament li escric el valor en binari (*----BBB-GGG-RRR-*, *B* és el blau, *G* és el verd i *R* és el vermell). Cada color primari ha d'estar definit per un valor entre 0 i 7, 000 i 111 en binari. Això són en total 9 bits de color, és a dir, fins a un total de 512 colors possibles.

```
1.  Font:                           ;Font d'1bpp - 8x8 96 caràcters
2.      incbin "\ResALL\Font96.FNT"
3.  Font_End:
4.
5.  ; Inicialitza la font
6.      lea Font,A1                 ;Adreça de la font en la ROM
7.      move.l #Font_End-Font,d6    ;La font conté 96 lletres
8.
9.      move.l #$40000000,(VDP_Ctrl) ;Comença a escriure a l'adreça $0000
10.                                 ;(Patrons en la VRAM)
11. NextFont:
12.     move.b (A1)+,d0     ;Guarda el byte de la font
13.     moveq.l #7,d5       ;Compte de bits (8 bits)
14.     clr.l d1            ;Reinicia el byte
15.
16. Font_NextBit:           ;1 color per nibble = 4 bytes
17.
18.     rol.l #3,d1
19.     roxl.b #1,d0
20.     roxl.l #1,d1
21.     dbra D5,Font_NextBit;Següent bit de la Font
22.
23.     move.l d1,d0        ; Canvia el color de la font de l'1 al 15
24.     rol.l #1,d1         ;Bit 1
25.     or.l d0,d1
26.     rol.l #1,d1         ;Bit 2
27.     or.l d0,d1
28.     rol.l #1,d1         ;Bit 3
29.     or.l d0,d1
30.
31.     move.l d1,(VDP_Data);Guarda el tile al VDP
32.     dbra d6,NextFont    ;Repeteix fins a completar-ho
```

Per a les lletres utilitzo una font formada per un mapa de bits, un format obsolet que utilitzava Windows en el passat. Cada caràcter d'aquesta font és de 2 colors (color i transparència) i té una mida de 8x8 píxels.

Primer s'han d'enviar totes les lletres a la VRAM, per a fer-ho, escric *$40000000* al *VDP_Ctrl*. Cada cop que es trameti una lletra, a aquesta se li ha de convertir la seva paleta de 2 colors a una de 16. Aquest procés s'ha de repetir per cadascuna de les 96 lletres.

```
1.      clr.b Cursor_X      ;Reiniciar la posició del cursor
2.      clr.b Cursor_Y
3.      addq.b #8,(Cursor_Y)    ;Canviar la posició a x=5 i y=8
4.      addq.b #5,(Cursor_X)
5.
6.      move.w#$8144,(VDP_Ctrl)   ;C00004 reg 1 = 0x44 engegar la pantalla
```

Un cop inicialitzats els gràfics, col·loco la posició del cursor a les coordenades (5, 8) i engego la pantalla.

3.1.3. Escriure text en pantalla

```
1.  PrintChar:
2.      moveM.l d0-d7/a0-a7,-(sp)
3.          and.l #$FF,d0           ;Guarda només 1 byte
4.          sub #32,d0              ;No hi ha caràcters per sota el 32
5.  PrintCharAlt:
6.          Move.L  #$40000003,d5   ;4=escriu, $3=Rang Cxxx
7.          clr.l d4                ;Tilemap en el $C000+
8.
9.          Move.B (Cursor_Y),D4
10.         rol.L #8,D4
11.         rol.L #8,D4
12.         rol.L #7,D4             ;2 bytes per tile * 64 tiles per línia
13.         add.L D4,D5
14.
15.         Move.B (Cursor_X),D4
16.         rol.L #8,D4
17.         rol.L #8,D4
18.         rol.L #1,D4             ;2 bytes per tile
19.         add.L D4,D5
20.
21.         MOVE.L     D5,(VDP_ctrl)    ; C00004 escriu el següent caràcter
    al VDP
22.         MOVE.W     D0,(VDP_data)    ; C00000 guarda la següent paraula
    del nom
23.
24.         addq.b #1,(Cursor_X)    ;Suma 1 a la X
25.         move.b (Cursor_X),d0
26.         cmp.b #39,d0
27.         bls nextpixel_Xok
28.         jsr NewLine              ;Si arribem al final de la línia crear
    una nova
29. nextpixel_Xok:
30.     moveM.l (sp)+,d0-d7/a0-a7
31.     rts
```

Aquí defineixo la rutina que dibuixa un caràcter en la pantalla. Haig de calcular l'adreça en la VRAM per al següent *tile* que s'ha de dibuixar, el mapa de *tiles* és de 64x64 i comença al *$C000*, cada *tile* pesa 2 bytes, per tant la fórmula per a obtenir l'adreça és la següent:

$$Adreça = \$C000 + (Ypos * 128) + (Xpos * 2)$$

Un cop calculada l'adreça, l'escric al *VDP_Ctrl* i l'índex del *tile* al *VDP_Data*. Posteriorment s'ha de comprovar que no estic en el final d'una línia, en el cas que hi estigui, en creo una de nova.

```
1.  PrintString:
2.          move.b (a3)+,d0         ;Llegeix la lletra d'A3
3.          cmp.b #255,d0
4.          beq PrintString_Done    ;Para en el 255
5.          jsr PrintChar           ;Dibuixa la lletra
6.          bra PrintString
7.  PrintString_Done:
8.      rts
9.
10. NewLine:
11.     addq.b #1,(Cursor_Y)        ;Suma 1 a la Y
12.     clr.b (Cursor_X)            ;Reinicia X
13.     rts
```

Finalment, per a facilitar l'escriptura de texts, la rutina de *PrintString* la qual em permet escriure frases de menys de 256 caràcters. A més, també li creo la rutina de *NewLine* per a crear una nova línia quan aquesta arribi al final de la pantalla.

```
1.      lea Message,a3
2.      jsr PrintString                 ;Escriu una frase en pantalla
3.      jsr NewLine                     ;Crea una nova línia
4.      jmp *                           ;Para l'execució
5.
6.  Message: dc.b 'Hola, Mon!',255
7.      even
```

Per acabar, el mateix missatge, aquest es dibuixa en la posició (5, 8) que he establert anteriorment i utilitzant totes les instruccions que he explicat en conjunt, obtinc aquest resultat:

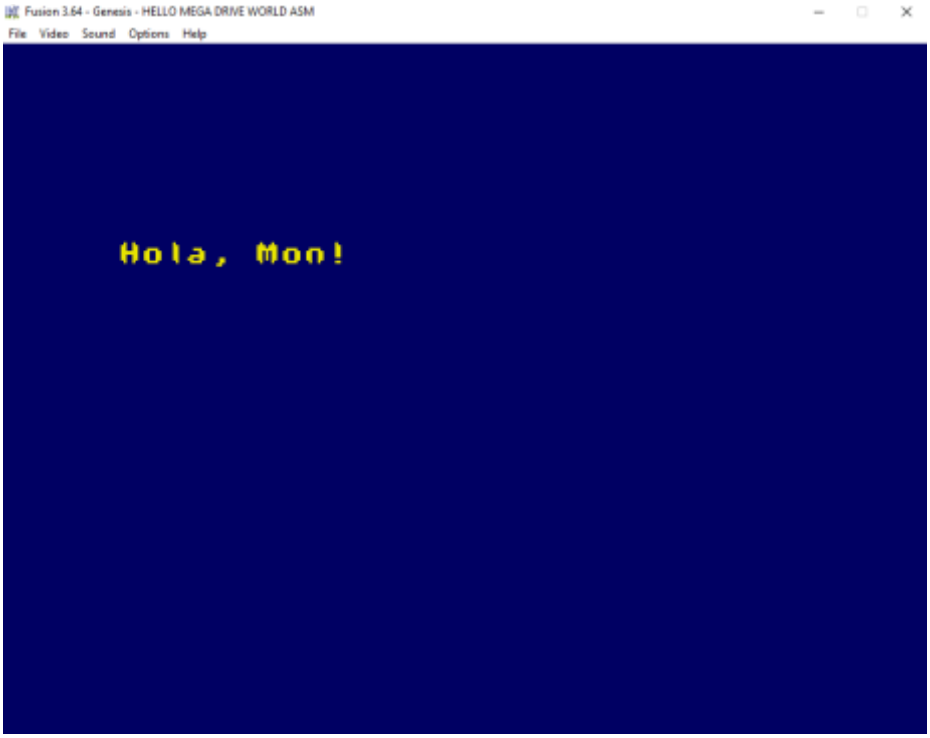


Figura 3.1: "Hola, món!" en assemblador

3.2. "Hola, món!" en SGDK

A diferència d'utilitzar un compilador juntament amb assemblador 68k, aquest utilitza C, un llenguatge creat el 1972 per a facilitar la lectura i el desenvolupament de programes entre plataformes, gràcies a un compilador que ho converteix a assemblador. Això juntament amb el *framework* de SGDK, el qual aporta un munt de funcions específiques per a la consola, puc crear un programa amb menys codi i més senzillesa a l'hora de llegir, a costa d'una petita pèrdua d'optimització.

3.2.1. Capçalera

La capçalera, en aquest cas es genera automàticament amb uns valors per defecte. Així fent que encara hàgim de programar menys tot i que ens dona la llibertat de modificar-lo quan vulguem.

3.2.2. Programa

```
1. #include <genesis.h>
2.
3. int main(){
4.     VDP_drawText("Hola, mon!", 5, 8);
5.
6.     while(TRUE){    // Bucle principal del joc
7.         SYS_doVBlankProcess();
8.     }
9.
10.    return 0;
11.}
```

En aquest cas, tal com es pot veure, el codi per a dibuixar un "Hola, món!" només ocupa 11 línies, en comparació a les prop de 200 de l'assemblador. En aquest cas primer importo la biblioteca de SGDK com a "*genesis.h*". Llavors s'inicia la funció principal, just abans de començar el bucle del programa, dibuixo en pantalla la frase en la mateixa posició que el cas anterior i ja finalment deixo el programa en un estat de bucle infinit.

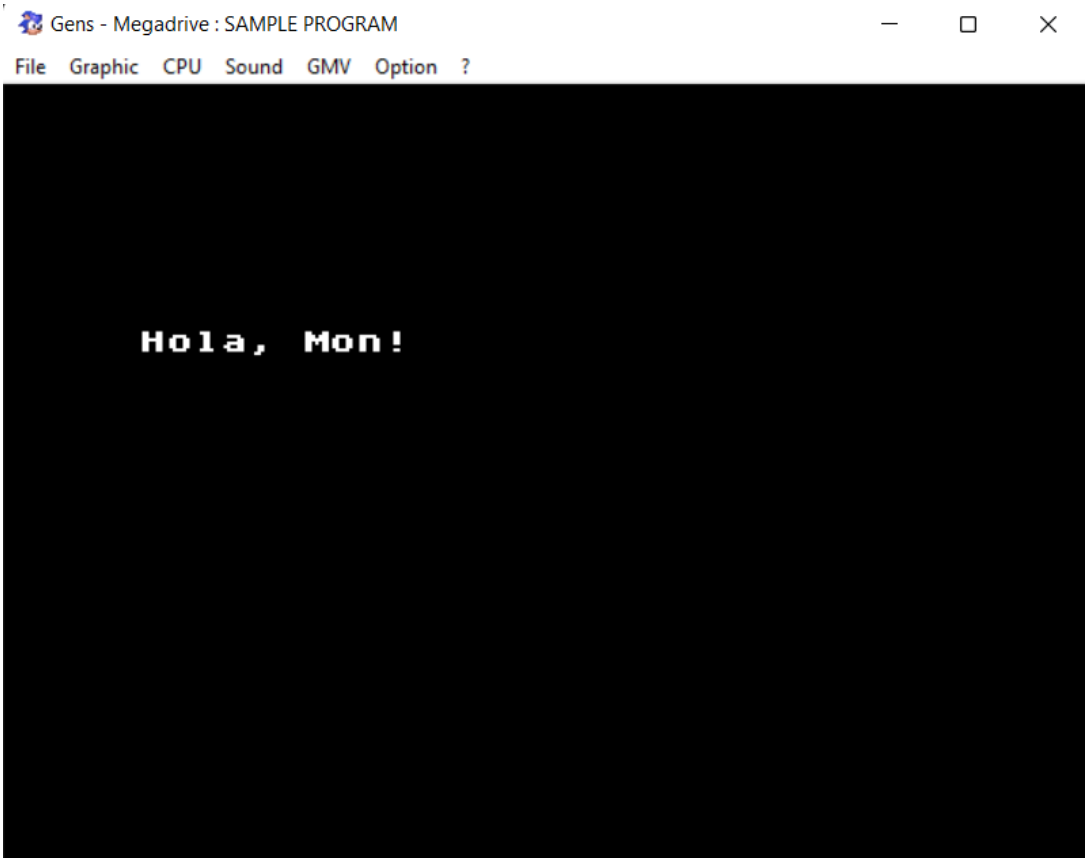


Figura 3.2: "Hola, món!" en SGDK

3.3. Diferències entre assemblador 68k i SGDK

Tots dos mètodes tenen tant avantatges com desavantatges. En el cas de l'assemblador, és que el seu codi està molt més optimitzat que la feina que et pot fer un compilador com el de SGDK. A diferència de l'època de la consola en l'actualitat és que avui en dia els compiladors de C estan molt més optimitzats del que ho estaven en el passat i el codi que compila és molt semblant al que ho podria ser si s'hagués escrit directament en assemblador, per tant la diferència de rendiment actualment és molt petita.

Per altra banda, SGDK et proporciona molta feina feta per defecte, això t'ajuda a centrar-te en el que realment és important i no tant en petits detalls que te'ls has de crear tu des de zero.

Per tant, en el cas d'aquest treball, prioritzo la facilitat d'ús abans que aquella petita millora de rendiment que probablement soc incapaç d'aprofitar. Així doncs obtinc una base ja feta la qual puc aprofitar per a aconseguir un videojoc més gran del que podria aconseguir seguint l'altre mètode durant el mateix període de temps.

4. Desenvolupament del videojoc

4.1. Primers contactes amb la Sega Mega Drive

Començar a crear un videojoc des de zero pot ser més o menys difícil depenent de la plataforma per a la qual l'orientis o el programa que utilitzis. En aquest cas, la Mega Drive no és una plataforma molt comuna per a la qual se li'n desenvolupen actualment. Per tant, les primeres setmanes tracten d'agrupar totes les poques guies de desenvolupament que hi ha publicades a internet i en conjunt amb l'assaig i error, posar-me a prova i crear alguns prototips molt bàsics amb els quals fer-me una idea de la dificultat que seria crear-ne un molt més gran que qualsevol d'aquests petits experiments.

4.1.1. *Megapong*

Megapong és una versió molt simplificada del *Pong* original per a un jugador. Cada cop que fas rebotar la pilota, reps punts i cada cert nombre de cops, aquesta accelera. La partida s'acaba quan la pilota surt per la zona inferior de la pantalla. Aquest primer prototip l'he pogut fer gràcies a una de les sèries de guies d'Ohsat, les quals ajuden a aprendre les bases sobre el sistema.

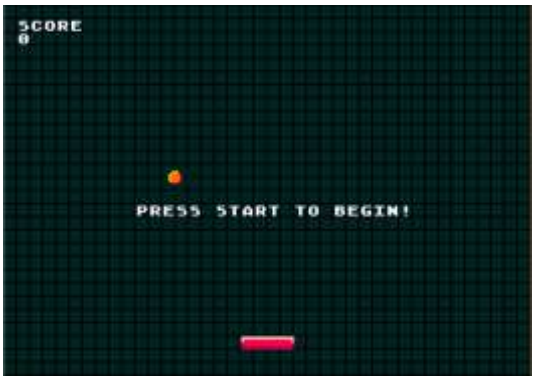


Figura 4.1: Captura de Megapong

4.1.2. *Megarunner*

Megarunner és un videojoc molt simple en el que has de saltar els obstacles per a guanyar punts. La partida s'acaba quan el jugador xoca contra un obstacle. Un exercici molt simple per a aprendre el funcionament del desplaçament de plans i l'animació de *sprites*. Igual que l'anterior, aquest també ha estat extret d'una guia d'Ohsat.

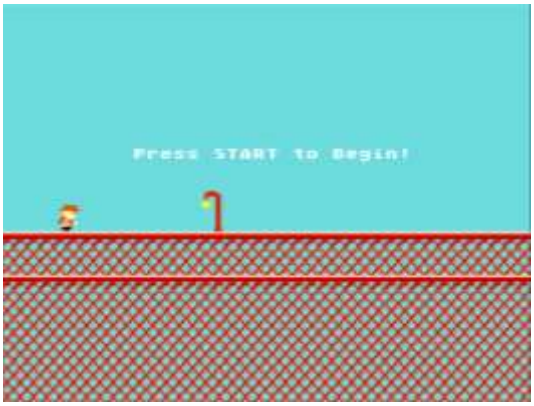


Figura 4.2: Captura de Megarunner

4.1.3. *Megalaga*

Megalaga és una versió molt simplificada del *Galaga* original. L'objectiu del videojoc és disparar als enemics que hi ha en la pantalla. Un cop els derrotes a tots, guanyes la partida. Un minijoc que, en l'àmbit de programació, dona a conèixer unes estructures més avançades que més endavant puc utilitzar per a la creació d'alguns elements del meu videojoc.

Figura 4.3: Captura de Megalaga

4.1.4. *Megatiler*

L'últim projecte de la sèrie d'Ohsat és el *Megatiler*, un minijoc de perspectiva aèria que tracta d'agafar totes les monedes i avançar al següent nivell. Aquest et dona una molt bona idea per a organitzar els nivells i les col·lisions de cadascun d'ells en un espai molt petit.

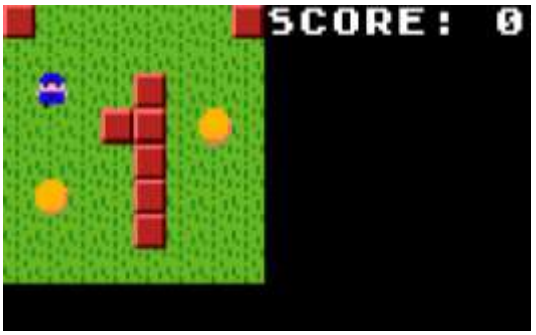


Figura 4.4: Captura de Megatiler

4.1.5. Modificació del *Sonic Example*

SGDK té en el seu interior una sèrie d'exemples per a poder descobrir algunes de les coses que s'hi poden fer. Un dels quals n'és el *Sonic Example*, un minijoc en el qual l'únic que pots fer és córrer i saltar amb el personatge de Sonic i uns quants enemics en pantalla. La meva modificació tracta d'afegir-li la generació de nivells del *Megatiler* i poder-hi jugar amb un fons generat gràcies a allò. Per a aconseguir-ho he utilitzat una altra sèrie de guies de Danibus, el qual m'ha aportat molt més coneixement sobre funcions específiques del sistema.

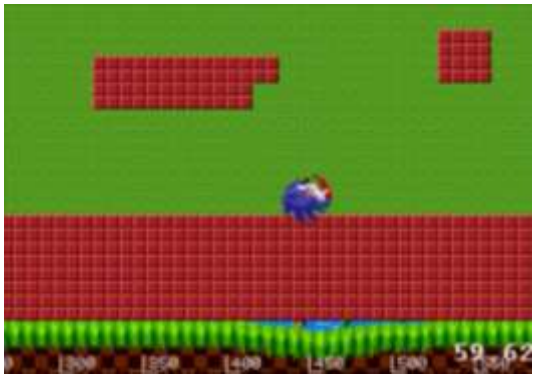


Figura 4.5: Captura de la modificació del Sonic Example

4.2. Creació dels fonaments

Si en lloc d'estar creant un videojoc per a la Sega Mega Drive, l'estigués fent per a ordinador, em podria saltar aquest pas perquè ja el tindria fet gràcies als motors. Aquesta primera iteració tracta de tenir una petita base sobre la qual poder treballar. És a dir, fer moure el personatge per la pantalla i generar un nivell.

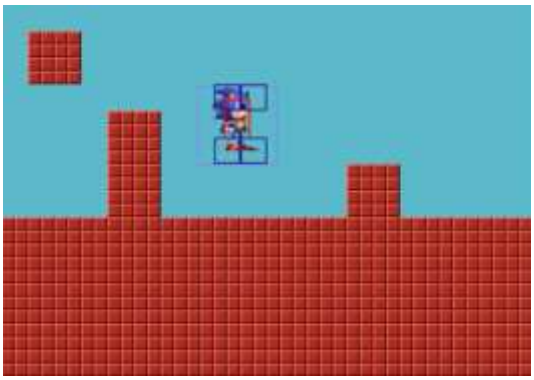


Figura 4.6: Captura de la primera iteració del videojoc

Per a fer això utilitzo els elements amb els quals ja he treballat anteriorment amb el *Sonic Example*. El resultat és un escenari molt senzill amb un personatge que es mou per la pantalla sense cap mena de col·lisió ni força de gravetat.

4.3. Motor de física

El videojoc es troba dins del gènere de plataformes, per tant necessita tenir un motor de física molt precís. Això s'aconsegueix generalment sent minimalista, per així evitar les diferents complicacions i poder-me centrar en el que realment és important, tenir un control suau i precís sobre les físiques del videojoc.

4.3.1. Mapa de col·lisions

El primer pas per a crear un motor de física és saber que és sòlid i que no, per a fer això utilitzo un *Array 2D*, també conegut com a matriu en l'àmbit matemàtic. Per a estalviar memòria i rendiment, faig servir menys caselles que píxels en pantalla, fent que cada casella equivalgui a 16x16 píxels, en total hi ha 20x14 caselles per cada sala. Aquest mètode no només em permet saber si una casella és sòlida o no, sinó que també em permet que en un futur pels hi pugui donar funcionalitat segons el tipus determinat a partir d'un altre número.

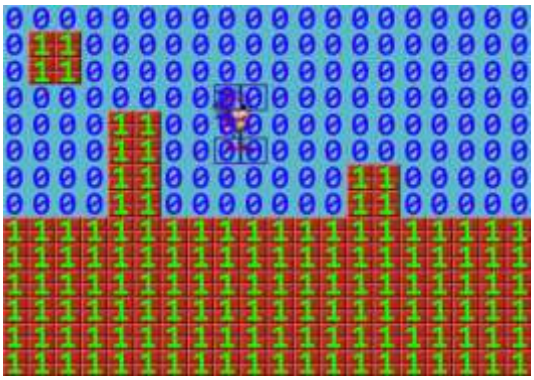


Figura 4.7: Representació visual del mapa de col·lisions

4.3.2. Resolució de col·lisions

Un cop tenint el mapa de col·lisions del nivell, s'ha de trobar la manera de convertir un moviment suau que pot tenir qualsevol magnitud de velocitat i ajustar-li la posició de cada element segons els altres que hi hagi al seu voltant.

Per a aconseguir això utilitzo una *Axis Aligned Bounding Box* ("AABB, una caixa que no es pot rotar"), aquesta aproximació s'adapta de manera ideal a les limitacions de la consola. Com que no puc rotar les imatges amb un rendiment acceptable em puc limitar a emprar solament AABBs. Tal com es veu a la figura, la caixa del personatge no necessita complir exactament les mesures del *sprite*.

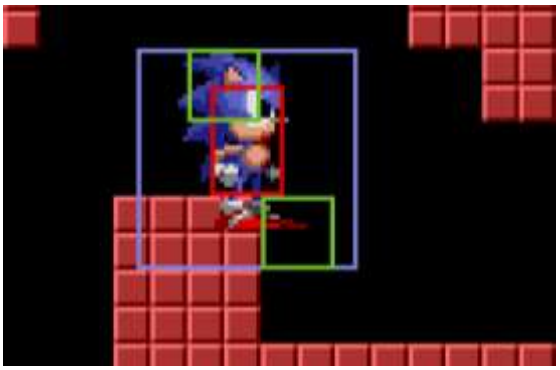


Figura 4.8: Representació detecció de les col·lisions

Assumint que no hi ha cap rampa, l'algoritme per a resoldre les col·lisions seguiria aquestes instruccions:

1. Descompondre el moviment en les components *x* i *y*. Per a resoldre cadascuna individual és seguirà l'ordre de primer la *x* i seguidament la *y*.
2. Llegir els valors del mapa de col·lisions en els que estarà en el següent fotograma (Els quadrats verds de la imatge representen el màxim i el mínim).
3. En el cas que es detecti si el jugador estarà col·lidint amb un bloc en el següent fotograma, col·locar-lo just en el costat corresponent de la paret o terra.
4. Actualitzar tots els valors de la posició i repetir-ho amb la component *y*.

4.3.3. Resolució de col·lisions entre AABBs

Amb el pas anterior ja es pot saber si el jugador està col·lidint amb qualsevol bloc que estigui representat en el mapa de col·lisions. Però en el cas que hi hagi una plataforma mòbil que no funcioni amb les posicions exactes del mapa s'haurà d'utilitzar un altre mètode.

Per a fer això faig servir una llista que conté a tots els elements mòbils amb els quals es pot interactuar, llavors itero per cadascuna de les AABBs i faig la mateixa comprovació que en l'anterior cas.

4.3.4. Velocitats i acceleracions

Un cop el personatge és capaç d'aturar-se quan està col·lidint contra un objecte sòlid, s'ha de fer que aquest es mogui a través de forces que es reflecteixin en acceleracions. Per a fer això es fa ús dels nombres racionals, però a diferència d'avui en dia, en aquella època els processadors no eren capaços de treballar amb decimals nativament. Per això s'ha d'utilitzar un altre mètode per a simular-los.

4.3.4.1. Operadors *bitwise*

Per a assolir l'objectiu final dels nombres amb decimals, s'ha de fer ús dels operadors *bitwise*, els quals serveixen per a operar els nombres en binari. Els principals operadors són aquests:

NOT

El NOT, és un operador unari que realitza la negació lògica per a cada bit, així fent que cada 1 passi a ser un 0 i a la inversa. El seu símbol en el llenguatge C és "~".

Entrada	0	1
Sortida	1	0

AND

L'AND, és un operador que, a diferència del NOT, aquest pren com a entrada dos nombres i en retorna un. El resultat de cada posició és 1 si en aquella mateixa posició també és 1 en les dues entrades. El seu símbol en el llenguatge C és "&".

Entrada	A	0	0	1	1
	B	0	1	0	1
Sortida	C	0	0	0	1

OR

L'OR, també conegut com a OR inclusiu, funciona prenent com a entrada dos valors i llavors el resultat en cada posició és 0 només si cap dels dos nombres té un 1 en la mateixa posició. El seu símbol en el llenguatge C és "|".

Entrada	A	0	0	1	1
	B	0	1	0	1
Sortida	C	0	1	1	1

XOR

El XOR o OR exclusiu, funciona semblant a l'OR amb l'única diferència que en el cas que les dues xifres d'entrada siguin 1, aquest retornarà 0. El seu símbol en el llenguatge C és "^".

Entrada	A	0	0	1	1
	B	0	1	0	1
Sortida	C	0	1	1	0

Aquests són els operadors *bitwise* bàsics, n'existeixen molts més, però em limito a explicar només aquests perquè la resta es poden crear a partir de les diferents combinacions possibles. Per exemple, l'operador NAND es crearia a mitjançant un operador NOT i un AND.

4.3.4.2. Operadors *bitshift*

Els operadors de desplaçament o *bitshift*, són un altre tipus d'operador que també funciona en l'àmbit binari. En aquest cas, tal com ho diu el mateix nom, aquest s'encarrega de desplaçar els bits cap a la dreta o cap a l'esquerra.

Shift cap a la dreta

L'operador de *shift* cap a la dreta, representat com a ">>", desplaça els bits el nombre de vegades que se li digui.

```
1. int a = 8; // 0000 1000
2. int b = 1; // 0000 0001
3. int c = a >> b; // 0000 0100 -> Equival a 4
```

En aquest exemple es veu com en fer-li un *bitshift* d'una posició al 8, aquest passa de ser un 1000 a un 0100. Com que el binari és de base 2, fer un *shift* cap a la dreta d'una posició equival a fer una divisió entre 2.

Shift cap a l'esquerra

El *shift* cap a l'esquerra, representat com a "<<", funciona exactament igual que l'anterior però en aquest cas cap a l'esquerra.

```
1. int a = 8; // 0000 1000
2. int b = 1; // 0000 0001
3. int c = a << b; // 0001 0000 -> Equival a 16
```

En aquest exemple es veu com passa el contrari que el de cap a la dreta, es desplaça un bit cap a l'esquerra i això suposa que en lloc de simular una divisió, simula una multiplicació per 2.

4.3.4.3. Aritmètica de coma fixa

A diferència dels processadors actuals, els quals funcionen amb aritmètica de coma flotant, la Mega Drive no té un processador amb suficient rendiment per a poder-nos permetre afegir-li una aritmètica que variï la posició de la coma segons el nombre de decimals i enters. En aquest cas s'utilitza el mètode de coma fixa, el qual segueix la següent representació.

1	0	1	1	0	0	1	1	1	1	0	0	1	1	0	1
Signe	Bits enters									Bits decimals					
-	207									13 → 13÷64=0,203					

Figura 4.9: Representació del nombre -207,203 en binari

Per a simular un nombre decimal creo una variable, en aquest cas de 16-bits i li assigno un valor utilitzant la següent *macro*:

```
#define FIX16(nombre)    ((fix16) ((nombre) * (1 << 6)))
```

Aquesta *macro* funciona a partir d'una constant, quan el programa es compila, cada nombre que s'assigni a partir d'aquesta es precalcula. En el cas que vulgui assignar un número a partir d'un altre que no sigui constant utilitzo la següent *macro*:

```
#define intToFix16(nombre)    ((nombre) << 6)
```

I si el que necessito és convertir una variable fix16 a un nombre enter (int):

```
#define fix16ToInt(nombre)    ((nombre) >> 6)
```

És clar, aquesta *macro* realment només retorna la part entera, és a dir, si per exemple el nombre és 0,9, aquesta retornarà 0 i si és -0,9, també retornarà 0.

Si el que necessito és que em retorni el nombre enter més proper, primer necessito aquesta *macro*:

```
#define fix16Frac(nombre)    ((nombre) & ((1 << 6) - 1))
```

Aquesta retorna un *fix16* que només conté la part decimal. Un cop tinc a aquesta, ja puc crear la següent que retorna el nombre arrodonit.

```
#define fix16ToRoundedInt(num)    \
    ((fix16Frac(num) > FIX16(0.5)) ? fix16ToInt(num) + 1 : fix16ToInt(num))
```

Aquí faig ús d'un operador ternari, el qual tracta de fer una comprovació afegint un signe d'interrogació després de la condició, en el cas que això es compleixi, retornarà el valor d'abans dels dos punts, si no retorna el que hi ha després.

Sumar i restar aquests nombres funciona exactament igual que els enters, però les multiplicacions i divisions funcionen d'una altra manera.

```
#define fix16Mul(num1, num2)    ((num1 * num2) >> 6)
```

Aquí es pot veure com primer multiplico i després, com que el programa es pensa que estem multiplicant dos enters, haig de tornar a fer el *bitshift* perquè torni a ser un racional. En el cas de les divisions:

```
#define fix16Div(num1, num2)    ((num1 << 6) / num2)
```

Aquí, a diferència de la multiplicació, primer li trec els decimals al primer número i així, al dividir entre el segon, aquest els hi torna a introduir.

En cap dels dos casos el resultat dona exactament el que hauria de ser, però com que el marge d'error no és molt gran, es pot conviure amb ells.

Hi ha bastants més *macros* per als decimals, però aquestes són les bàsiques amb les quals poder entendre el seu funcionament. A més, també es pot aplicar una lògica semblant per a fer-ho amb nombres de 32-bits.

4.3.4.4. Aplicar acceleracions

Finalment, un cop ja es poden utilitzar els nombres decimals, per a aplicar acceleració, utilitzo diverses variables: acceleració, velocitat i posició.

Per a accelerar s'ha d'augmentar la velocitat, per tant sumo la constant d'acceleració a la velocitat i llavors la velocitat sobre la posició. D'aquesta manera puc augmentar la velocitat d'una forma més gradual i així mateix fer-ho amb l'acceleració de la gravetat.

4.4. Motor de desplaçament

El desplaçament diferencial o de paral·laxi és una tècnica que s'utilitza per a crear una il·lusió de profunditat en una escena en 2 dimensions. Hi ha múltiples formes d'implementar-la, ja sigui en una direcció horitzontal, vertical o multidireccional. En aquest cas, la Sega Mega Drive és capaç de fer servir quatre mètodes diferents.

4.4.1. Mètode per capes

La Mega Drive té la capacitat de dibuixar dos plans, per a simular l'efecte de profunditat amb el mètode per capes, simplement s'han de desplaçar els dos plans a diferents velocitats.

4.4.2. Mètode per *sprites*

En alguns casos, si no es requereix una enorme quantitat de *sprites* per als personatges, es poden ajuntar per a crear un pseudo-pla i llavors tractar-les com a un sol objecte que es mou a certa velocitat, com en el cas del mètode per capes. Aquest mètode s'utilitza generalment juntament amb l'anterior.

4.4.3. Mètode per patrons repetitius

Aquest mètode és molt poc utilitzat i s'utilitza principalment per a fer animacions en bucle, però en alguns casos s'utilitzava per a crear l'efecte de paral·laxi mitjançant el canvi de colors dels *tiles*, generant així aquest efecte.

4.4.4. Mètode per ràster

Les imatges rasteritzades són aquelles que són refrescades de dalt a baix amb un petit retard entre cada línia. Això es pot utilitzar per a desplaçar cada línia individualment. Si es planeja bé, es poden crear efectes de distorsió o fins i tot, fer un efecte semblant al del mètode per capes amb diversos grups per un mateix pla. Aquest només es pot aplicar horitzontalment.

4.4.5. Casos particulars

Per una banda, hi ha el desplaçament per files, és un tipus de desplaçament semblant al mètode per ràster, però aquest només pot moure grups d'una alçada de 8 píxels cadascun i, igual que ell, també és exclusiu de l'horitzontal.

D'altra banda, l'altre mètode és semblant al de per files, però exclusiu del desplaçament vertical és el de per columnes, l'única diferència és que en aquest cas només es poden desplaçar grups de 16 píxels.

4.4.6. Creació del motor de desplaçament

4.4.6.1. Moviment de la càmera

Per a desplaçar el pla utilitzo una estructura que la nomeno càmera, aquesta s'encarrega de mantenir el personatge al centre de la pantalla i desplaçar el pla en direcció contrària per a fer la sensació de desplaçament.

```
1. if (camera.position.x != player.position.x){
2.     camera.position.x = player.position.x;
3.     MAP_scrollTo(bga, camera.position.x, camera.position.y);
4.     VDP_setHorizontalScroll(BG_B,          fix32ToInt(fix32Mul(FIX32(-
   camera.position.x), FIX32(.35))));
5. }
```

En aquest fragment de codi, primer comprovo si no està en la mateixa posició que el jugador. Si això es compleix, col·loca la càmera en la mateixa posició i desplaça el mapa cap a aquella, tot utilitzant la funció de la biblioteca MAP i posteriorment actualitzant el pla B per a fer la sensació de profunditat. En el cas del VDP, haig de donar-li la direcció en negatiu perquè el pla s'ha de moure en direcció contrària, en canvi per al MAP no, perquè aquest ja s'encarrega internament de negar-ho.

4.4.6.2. Zona morta de la càmera

La zona morta o *deadzone* de la càmera és aquella àrea en la qual ella no actua. En l'anterior cas aquesta ho és quan està en la mateixa posició que el jugador. Això en segons quins casos pot arribar a resultar marejant. Per a evitar-ho, amb una simple condició li amplio la zona morta.

Figura 4.10: Representació zona morta de la càmera

```
1. const int deadzone = 20;
2. if  (camera.position.x  +  deadzone  /  2  >  player.position.x  ||
   camera.position.x - deadzone / 2 < player.position.x){
3.     //Codi de la càmera
4. }
```

En la primera línia defineixo la constant de la zona morta com a 20 i posteriorment si el jugador està per sobre de la posició de la càmera més la meitat de la zona morta o bé està per sota de la posició menys la meitat, aquesta actuarà utilitzant una variació del codi anterior.

4.5. Disseny del personatge

Figura 4.11: Dibuix del protagonista

Tot videojoc necessita el seu protagonista, per a fer això puc utilitzar dos mètodes per a crear-lo. En puc agafar un d'un paquet d'animacions amb una llicència de *Creative Commons* i fer un videojoc basant-se en aquell personatge, o bé crear el meu propi des de zero. En aquest cas en tenir unes mecàniques de joc molt clares, aquest l'he dissenyat jo mateix amb l'ajuda de la imaginació del meu cosí.

4.5.1. Les habilitats del personatge

Figura 4.12: Fotograma victòria

El protagonista del videojoc és un talp miner, té les habilitats de córrer, saltar i atacar tant a l'aire com a terra. Un set de moviments força minimalista que em permet poder-me centrar molt més a polir-los i així tenir un resultat molt més agradable i divertit de jugar.

El *sprite* final del personatge té una mida de 60x50 píxels cada fotograma, en total n'hi ha més de 70 diferents. Aquest utilitza una paleta d'11 colors per a funcionar i en segons quina animació, utilitzo un *palette swap* que consisteix a canviar certs colors de la paleta perquè el personatge pugui tenir més variació sense requerir-ne de més quantitat.

4.5.2. Córrer

Figura 4.13: Fotograma córrer

El talp pot córrer, això ho faig aplicant-li certa acceleració, els valors que utilitzo són una velocitat màxima de 2 px, una acceleració 0,25 px i una desceleració de 0,2 px. Tots aquests valors s'apliquen cada fotograma, 50 per les PAL i 60 per les NTSC.

El fet d'usar aquestes xifres m'aporta un moviment molt més dinàmic i fluid. Si li apliqués una acceleració i desceleració més petites, el personatge semblaria que anés per sobre del gel i en el cas oposat, si les augmentés, el personatge tindria un moviment robòtic, és per això pel que aquests valors s'adapten bé al seu disseny.

4.5.3. Saltar

El salt és el component essencial de tot videojoc pertanyent al gènere plataformes, justament per això, aquí n'utilitzo un de gradual. Aquest funciona de tal manera que si el jugador prem el botó de saltar durant més temps, el talp saltarà més alt, en canvi, si el jugador prem el botó durant un curt període de temps, aquest ho farà considerablement més baix.

Figura 4.14: Fotograma saltar

Per a fer això, primer salta amb la velocitat màxima i en el punt en el qual deixi de prémer el botó reduir-la a la meitat, així la limito sense que el jugador noti un tall de la velocitat en sec.

4.5.4. Atacar

L'atac, tal com he dit abans, pot ser de dos tipus, aeri i terrestre. En cadascun dels dos casos la forma d'atacar funcionarà completament diferent l'un de l'altre.

4.5.4.1. Atac terrestre

L'atac terrestre s'acciona prement el botó B i s'executa cap a la direcció en la qual el jugador està mirant en aquell moment.

L'atac es pot començar a executar tant a l'aire com a terra. Si el personatge s'està desplaçant horitzontalment, aquest es frenarà un cop toqui el terra, mentre segueix-hi a l'aire, mantindrà la seva velocitat horitzontal.

Figura 4.15: Fotograma atac terrestre

4.5.4.2. Atac aeri

Aquest atac, a diferència de l'anterior, només es pot utilitzar mentre s'està a l'aire. Un cop el talp toqui a terra o accioni el terrestre, l'atac es cancel·larà.

En aquest cas l'atac aeri només afecta els enemics que estiguin per sota del personatge. Per tant, en cap moment de la partida es tindrà l'oportunitat d'atacar verticalment cap amunt.

Figura 4.16: Fotograma atac aeri

4.6. Obstacles

El videojoc té un personatge, un nivell, però no té cap mena de repte en el qual el jugador s'hagi de posar a prova. Per això necessita una sèrie d'obstacles que li dificultin el pas. Aquests ja poden ser des de simples punxes dispersades pel nivell, fins a enemics amb un comportament propi que puguin ferir al jugador.

4.6.1. Caixes

Un dels primers obstacles que li afegeixo són les caixes. El talp té com a habilitat principal el seu atac, sigui terrestre o aeri, per això un dels principals hauria de ser un que es pogués destruir. La caixa no té cap altre funcionament més que el d'estar estàtica en un lloc específic que li dificulti el pas del jugador. Quan aquesta és destruïda, apareix una moneda aleatòria amb una tendència a valors més alts en les caixes més grans i el contrari amb les petites.

Figura 4.17: Sprite caixa

4.6.2. Caminants

Un caminant és un tipus d'enemic que simplement es mou per la pantalla en una sola direcció. L'únic en el qual es fixa és en les parets del nivell. En el cas que un enemic detecti que està davant d'una paret, aquest canviarà de direcció per tal de no quedar-se estancat. En el meu cas, el represento com a un porc verd personificat que pot caminar sobre dues potes.

Figura 4.18: Sprite caminant

4.6.3. Caminants avançats

Aquest enemic, tal com ho indica el seu nom, és una versió avançada de l'anterior. Funciona de la mateixa manera que ho fa el caminant, però el que els diferencia és que en aquest cas no només canvia de direcció quan detecta una paret al seu davant, sinó que també ho fa quan aquest detecta que està a punt de caure per un precipici. Per tant, els porcs si veuen un precipici cauran, però aquests, que són representats per uns bolets, evitaran la caiguda.

Figura 4.19: Sprite caminant avançat

4.6.4. Voladors

Un volador és un enemic que es comporta d'una manera semblant als caminants, però en aquest cas ho fa a través de l'aire. Es transporta de banda a banda de la pantalla volant horitzontalment seguint una simple funció sinusoidal. Com no podria ser d'altra manera, aquest el represento com a un ocell de color blau.

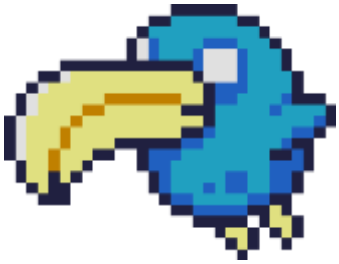


Figura 4.20: Sprite volador

4.6.5. Precipitats

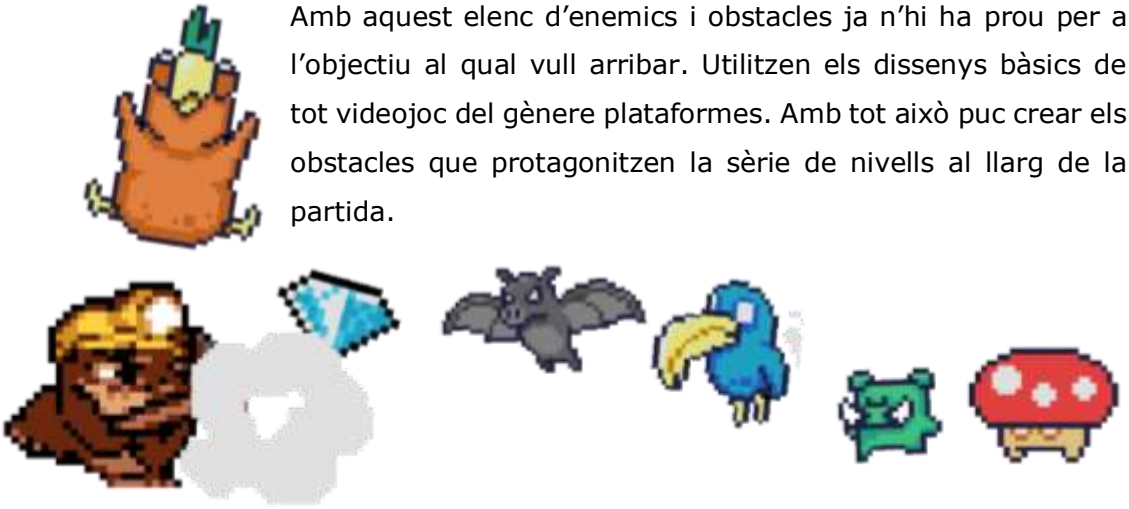
L'enemic precipitat és un que està estàtic seguint una simple animació en la part superior de la pantalla fins que aquest detecta que el jugador està dins de la seva zona d'atac. Quan ho fa, es deixa caure amb totes les seves forces per tal de ferir-lo. Un cop finalitzat l'atac, se'n torna a la seva zona inicial a l'espera de què el jugador hi passi per sota un altre cop. Aquest també és representat com a un ocell, però en aquest cas és més gras i d'un altre color.

Figura 4.21: Sprite precipitat

4.6.6. Perseguidors

Un enemic molt comú en els videojocs de perspectiva aèria, però no tant en la lateral, és el perseguidor. Normalment aquest es troba sigui estàtic o bé patrullant pel mapa. Un cop el jugador s'hi acosta suficientment a prop, comença a perseguir-lo. En el meu cas, al ser un ratpenat, aquest es troba en la part superior de la pantalla descansant i quan detecta que té el jugador a la vora, desperta i el comença a perseguir.

Figura 4.22: Sprite perseguidor



Amb aquest elenc d'enemics i obstacles ja n'hi ha prou per a l'objectiu al qual vull arribar. Utilitzen els dissenys bàsics de tot videojoc del gènere plataformes. Amb tot això puc crear els obstacles que protagonitzen la sèrie de nivells al llarg de la partida.

Figura 4.23: Elenc d'enemics

4.7. Sistema de guardat de partides

Per a guardar partides hi ha dues opcions, una és utilitzar contrasenyes que el jugador haurà d'introduir cada cop que reinicïi la consola o bé utilitzar SRAM, una petita memòria dins del cartutx alimentada per una pila. La segona opció sembla la més viable de les dues, però la PCB que faig servir jo no compta amb aquesta memòria. Per tant no em queda cap altra opció que la de crear un sistema de guardat per contrasenyes.

4.7.1. Anàlisi de sistemes existents

Metroid

Metroid és un videojoc que dona peu al seu propi gènere *Metroidvania*. Aquest tracta d'aconseguir una sèrie de millores i objectes per avançar en la partida. Per a guardar tot això utilitza contrasenyes formades per números i lletres, el resultat final és molt tediós d'introduir. A més, aquest sistema pot arribar a contenir unes contrasenyes que ja poden ser de caràcter ofensiu o una referència a personatges del món real com ho és la contrasenya "JUSTIN BAILEY".

Figura 4.24: Pantalla de contrasenyes Metroid

Castlevania IV

Seguint en el mateix gènere de videojocs, *Castelvania* també forma part del nom del gènere, i per això també requereix un sistema de guardat al nivell de *Metroid*. En aquest cas utilitza una graella de 16 caselles i cadascuna permet introduir-li tres valors diferents. Això permet escriure les contrasenyes sense haver de passar per un procés tan tediós com el de *Metroid*.

Figura 4.25: Pantalla de contrasenyes Castlevania

4.7.2. Creació d'un sistema de guardat

Per evitar el fet que se li faci tediós al jugador introduir contrasenyes, utilitzo un sistema semblant al del *Castlevania IV*.

4.7.2.1. Sistema numèric quaternari

Així com tenim el sistema decimal, hexadecimal o el binari, també existeix el quaternari. En aquest cas només es pot comptar amb quatre dígits. Així mateix utilitzant dues simples fórmules puc convertir un nombre de decimal a quaternari i a l'inrevés.

Decimal	0	1	2	3	4	5	6	7	8	9
Quaternari	0	1	2	3	10	11	12	13	20	21

4.7.2.2. De decimal a quaternari

Convertir un nombre de decimal a quaternari en un ordinador és molt senzill, els dispositius digitals funcionen amb binari. Per tant, tot aquell nombre que entri no-binari, com ho és un nombre decimal, automàticament es converteix en un nombre en binari. Com ja he mencionat anteriorment, el binari és un sistema numèric de base dos, per tant, si vull representar un altre sistema que utilitzi una base que estigui inclosa dins de la progressió geomètrica de 2^n (2, 4, 8, 16, etc.), cadascuna de les xifres es pot representar amb un nombre complet de bits. En ser el quaternari el segon de la successió, cada xifra d'aquest requereix 2 bits.

Binari	00	01	10	11
Quaternari	0	1	2	3

Per tant, per a convertir de decimal a quaternari utilitzo aquesta funció:

```
1. QuaternaryNumber decimalToQuaternary(u16 number) {
2.     QuaternaryNumber quat;
3.     quat.arr[0] = (number >> 12) & 0x3;
4.     quat.arr[1] = (number >> 10) & 0x3;
5.     quat.arr[2] = (number >> 8) & 0x3;
6.     quat.arr[3] = (number >> 6) & 0x3;
7.     quat.arr[4] = (number >> 4) & 0x3;
8.     quat.arr[5] = (number >> 2) & 0x3;
9.     quat.arr[6] = number & 0x3;
10.
11.    return quat;
12. }
```

El nombre màxim que necessito representar és el 9.999 que en quaternari és 2.130.033. Per tant no necessito més de set xifres per cada nombre. El funcionament d'aquesta funció tracta d'agafar els bits de dos en dos i anar-los guardant en una llista per a després poder-los ensenyar en forma de graella per a la contrasenya.

4.7.2.3. De quaternari a decimal

Per a convertir un nombre de quaternari a decimal la funció varia bastant:

```
1. u16 quaternaryToDecimal(u16* number, u16 length) {
2.     u16 num = 0;
3.     u16 power = 1;
4.     for (u16 i = 0; i < length; i++) {
5.         num += number[i] * power;
6.         power *= 4;
7.     }
8.     return num;
9. }
```

A simple vista pot semblar una funció molt complexa, però és molt més senzilla del que sembla, aquesta només està sumant les xifres seguint aquesta fórmula:

$$decimal(q) = q_0 \cdot 4^0 + q_1 \cdot 4^1 + q_2 \cdot 4^2 + q_3 \cdot 4^3 + q_4 \cdot 4^4 + q_5 \cdot 4^5 + q_6 \cdot 4^6$$

El decimal de cada nombre s'obté a partir de multiplicar cada xifra per la base elevada a la posició de la xifra començant per l'última. Tornant a la idea d'abans, la igualtat entre 2.130.033 i 9.999 s'obtindria així:

$$9.999 = 3 \cdot 4^0 + 3 \cdot 4^1 + 0 \cdot 4^2 + 0 \cdot 4^3 + 3 \cdot 4^4 + 1 \cdot 4^5 + 2 \cdot 4^6$$

4.7.2.4. Organització del sistema

El sistema està organitzat en una graella de 25 caselles, cadascuna d'elles significa un element important del videojoc que necessita ser guardat.

	Nombre de monedes
	Nombre de cors totals
	Nombre de cors actuals
	Nombre de vides
	Nivells superats
	Anti cheats ("antitrampes")

Figura 4.26: Representació components sistema de guardat

Tal com es pot veure, els punts no estan representats en aquesta graella, això és perquè requereix moltes caselles. Per tant, implementar-lo hauria ocupat molt espai, convertint-ho així en una tasca molt tediosa.

A més, s'hi pot veure una secció d'antitrampes, aquesta s'encarrega de validar el codi a més de fer certes comprovacions en què els nombres dels diversos elements siguin realistes. Per exemple, una de les comprovacions que fa és en la que unes caselles d'aquella secció indiquin quantes caselles tenen un dibuix. Si totes les dades són realistes, però aquella secció no correspon al nombre de caselles seleccionades, el codi s'invalida.

Per a dificultar el desxifrat del codi, aquest no està ordenat tal com mostro en la imatge, sinó que cada xifra està en una posició aleatòria sense seguir cap mena de patró.

4.8. Entrada al bucle de joc

Fins ara he estat parlant de com funcionen els diversos aspectes del videojoc, però en cap moment he parlat de com és el seu funcionament en si. El bucle de joc és un terme utilitzat en el disseny de videojocs per a explicar totes aquelles accions que estarà fent el jugador al llarg de la partida. Cada nivell que completes en el *Super Mario Bros*, enemic que mates en el *The Legend of Zelda* o cada *Pokémon* que captures són exemples de bucles de joc.

Quan el jugador comença la partida, accedeix a la pantalla de títol amb les opcions d'iniciar partida o continuar una que ja tens guardada en un codi. Des d'allà s'accedeix al selector de nivells i un cop escull el nivell, s'inicia el bucle.

El principal objectiu és simplement anar des d'un punt *A* a un punt *B* esquivant una sèrie d'obstacles. Aquest punt *A* és l'inici del nivell, vas avançant a través d'ell aconseguint monedes. Si durant la partida et maten o abandones, aquestes monedes i punts que has aconseguit els perds, però si arribes al punt *B*, el qual és la meta, aconsegueixes emportar-te-les totes.

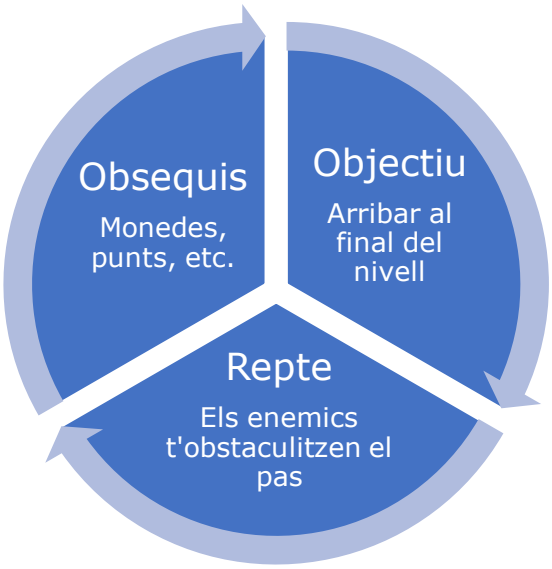


Figura 4.27: Bucle de joc

Amb tres simples passos tinc el disseny de pràcticament tot el videojoc, no es necessita res més que saber com es comença, com s'acaba i que passa quan s'acaba per a veure el que s'ha de crear.

4.9. Música

Compondre música per a consoles de fa trenta anys és complex. A diferència de sistemes posteriors, aquestes no tenien espai suficient per a emmagatzemar molta música com l'emmagatzemem avui en dia mitjançant formats tipus WAV o MP3.

4.9.1. Format VGM

VGM és un format que ha caigut en desús a causa de l'augment de capacitats en les memòries. El seu funcionament és molt senzill, l'arxiu VGM emmagatzema en el seu interior les ones de freqüència que utilitza com a so principal de la consola i llavors utilitzant les tècniques que ja he mencionat anteriorment en l'apartat d'àudio de les especificacions de la consola, puc modificar l'ona per tal de crear nous sons seguint les ordres de la partitura.

4.9.2. Creació de la música

Per a crear la música hi ha diverses opcions, una que es feia servir a la dècada dels noranta era utilitzar un programa creat per Sega que s'anomenava GEMS, aquest facilitava la creació de música per a la seva consola.

Per altra banda, han aparegut nous programes com ho és Deflemask que, a diferència de GEMS, aquests programes faciliten compondre música amb una millor interfície i compatibilitat gràcies al seu constant desenvolupament.

4.9.3. Funcionament de Deflemask

Deflemask funciona completament diferent de qualsevol programa de creació musical actual. Ja no tracta de compondre mitjançant l'ús de notes al complet, sinó que utilitza nombres per a representar aquelles notes i els efectes que se'ls hi ha d'aplicar a cadascuna d'elles.

Figura 4.28: Interfície Deflemask

Tal com es pot veure en aquesta imatge, es poden utilitzar diferents canals, aquests poden tenir el mateix so com en poden tenir un de diferent. A més, un d'aquests canals es pot fer servir com a reproductor PCM, per tant s'hi poden afegir mostres enregistrades.

4.9.3.1. Sistema hexadecimal

Com ja he mencionat abans el sistema binari i quaternari, en aquest cas s'utilitza l'hexadecimal. En tenir més xifres que el decimal, aquí se li afegeixen les sis primeres lletres de l'abecedari com a valors numèrics. La seva conversió és la següent:

Decimal	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Hexadecimal	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	10

4.9.3.2. Taula d'efectes globals

Codi	Nom de l'efecte	Detalls
0xy	Arpegi	xy = intervals
3xx	*Portamento*	xx = velocitat
4xy	*Vibrato*	xy = velocitat / profunditat
7xy	*Tremolo*	xy = velocitat / profunditat
8xx	Panoramització	01 = dreta, 10 = esquerra
9xx	Establir el valor de velocitat a 1	n/a
Axy	Desplaçar el volum	x0 = cap amunt, 0x = cap avall
Bxx	Salt de posició	Salta al patró xx
Dxx	Trencar el patró	Salta a la fila xx
E5xx	Afinar la nota d'origen	La nota d'origen és 80
ECxx	Tallar la nota	En tics (la meitat de la velocitat)
EDxx	Retardar la nota	En tics (la meitat de la velocitat)
Fxx	Establir el valor de velocitat a 2	n/a

Aquests són alguns dels efectes que es poden fer utilitzant *Deflemask*.

L'arpegi és un tipus de successió de notes en el que elles es reprodueixen en un to ascendent o descendent. El *portamento*, de l'italià "transport", és una transició que hi ha d'un so a una altura fins a una altra. El *vibrato* és una oscil·lació d'un so a una altura a certa freqüència. Per altra banda, el *tremolo*, a diferència del *vibrato*, és una oscil·lació entre dos sons.

La panoramització en música estèreo s'usa per a desviar un so cap a l'altaveu dret o l'esquerre, si no s'utilitza, el so es mantindrà en els dos altaveus. El desplaçament de volum, tal com indica el seu nom, augmenta o disminueix el volum a què es reprodueix el canal que se li indiqui.

Deflemask funciona per patrons, cada patró està format per les seves respectives files, en el cas que es vulgui saltar d'una fila d'un patró a un altre, es fa un salt de posició, però si es vol saltar de la fila d'un patró a una altra d'aquell mateix, es trenca el patró.

Finalment, tallar la nota serveix per indicar la duració que tindrà aquella i per altra banda, retardar-la serveix per a indicar quant de temps tardarà a sonar.

4.9.4. Implementació

Tot i ser temptador utilitzar aquesta tècnica, l'objectiu d'aquest projecte se centra en la programació. Pel que en el cas de l'àudio, optaré per les millores d'espai que han aparegut amb el pas dels anys, per així posar-li música que s'hagi enregistrat sense fer servir programes com Deflemask.

4.9.4.1. Format ADPCM

ADPCM està basat en DPCM que a la vegada està basat en PCM. És una tècnica utilitzada per a representar ones enregistrades utilitzant menys espai que amb PCM. La diferència és que si un ADPCM i un PCM tenen la mateixa freqüència, el PCM tindrà millor qualitat de so, però al pesar menys em puc permetre pujar-li la freqüència per tal de fer que s'escolti igual o millor, fent servir lleugerament menys espai que amb PCM.

4.10. Efectes de so

Un altre pas per a la creació del videojoc és afegir-li efectes de so, cada acció mínimament important tindrà el seu propi indicador auditiu. Aquests ja poden estar fets de dues maneres diferents, mitjançant FM o PCM.

4.10.1. Efectes de so mitjançant FM

FM o modulació de freqüència, és la tècnica principal per la qual es compon la música, això no comporta que no es pugui utilitzar per a la creació d'efectes de so.

La creació d'aquests sons té com a principal desavantatge la seva gran complexitat a l'hora de dissenyar-los a causa d'haver d'emprar notes per a simular-ho.

4.10.2. Efectes de so mitjançant PCM

Per altra banda tenim els efectes mitjançant PCM, són aquells que poden ser enregistrats perfectament amb un micròfon. Com ja he mencionat abans, aquesta tècnica requereix molt d'espai en la memòria, però en aquest cas al ser molt curt, aquella mida és gairebé imperceptible.

4.10.3. Implementació

Igual que amb la música, aquí també utilitzo ADPCM per a estalviar espai i a l'hora facilitar-me la creació d'efectes de so mitjançant curtes gravacions. El motiu principal d'aquesta decisió és el fet que SGDK no compta amb cap controlador de so compatible amb PCM i ADPCM simultàniament, per tant, aquesta és l'única opció que tinc.

4.11. Desmuntant un fotograma

En l'apartat de funcionament del VDP ja he parlat de com es dibuixa un fotograma. En aquest apartat explico com es desenvolupa cadascun d'ells des del codi fins a la part visual.

El primer que es fa és gestionar les col·lisions del jugador respecte el nivell i just després comprova les col·lisions respecte als objectes que funcionin amb un AABB. Els enemics fan exactament el mateix i en el mateix ordre, però ells no comproven les col·lisions respecte al jugador perquè seria el mateix que acabaria de fer ell.

Just després d'això se li aplica el moviment a la càmera. Per a evitar que hi hagi certa sensació de tremolor en la pantalla, aquesta s'ha d'actualitzar sempre després del jugador.

Si abans de tot això es detecta que el jugador està en el límit de la pantalla, es pausa l'execució del joc per a fer una transició des d'una sala del nivell a la següent. Just després d'executar tot el codi de cada fotograma, aquest passa a dibuixar-se.

La VRAM en el meu videojoc és dinàmica, va carregant-se a mesura que el joc necessita aquella part. A diferència de com ho fa *Sonic the Hedgehog*, aquí només hi ha carregat allò que o bé s'està dibuixant en aquell moment o algun objecte que apareix amb molta freqüència (comptadors, monedes, etc.). La majoria d'elements es carreguen a la memòria excepte el jugador, aquest al tenir una gran quantitat de fotogrames, només carregarà aquell fotograma que s'estigui dibuixant en aquell moment en concret.

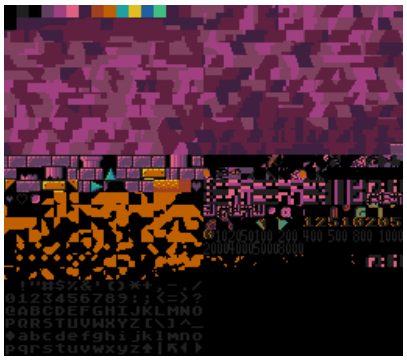


Figura 4.29: VRAM del videojoc

El pla *B* pot variar molt segons el nivell, en alguns casos aquest tindrà algun efecte especial o simplement formarà part de l'efecte paral·laxi. Tant un com l'altre s'actualitzen mentre la càmera es posiciona. Tal com es pot veure en la zona visible hi ha un efecte que distorsiona el pla, aquest l'explico en el següent apartat.

Figura 4.30: Pla B del videojoc

En el cas del pla *A*, aquest es dibuixa amb un mètode normal sense sofrir cap modificació. L'únic element estrany que hi ha són aquells *tiles* que no tenen res a veure amb el nivell. Aquests se'ls anomena brossa de la VRAM, és a dir, petites optimitzacions que resulten en dibuixos que no haurien d'existir en la zona no visible. En dibuixar-se en aquella zona, no m'he de preocupar en netejar tota la brossa generada.

Figura 4.31: Pla A del videojoc

Posteriorment es dibuixa el pla *Window*, aquí s'hi indiquen les dades bàsiques, com les monedes del jugador, les vides i els punts que té en aquell moment. Aquest es dibuixa en funció a unes variables emmagatzemades en la memòria.

Figura 4.32: Pla Window del videojoc

Finalment el pla *Sprites*, en aquest cas el pla només conté al jugador. En ser l'inici d'un nivell, per a no atabalar-lo, no inclou res més que no sigui ell. Si hi hagués altres elements, hi estarien inclosos tots aquells que utilitzessin un AABB, a més de les caixes.

Figura 4.33: Pla Sprites del videojoc

Figura 4.34: Captura d'un fotograma del videojoc

El resultat final és un nivell força viu amb moltes animacions en ell. Tal com es pot veure, el pla *B* apareix deformat a causa de l'efecte que se li aplica sobre ell.

4.12. Efectes especials

Les capacitats de la consola donaven peu a una gran quantitat d'efectes visuals per a reflectir aquell gran canvi que hi havia hagut. Al llarg d'aquest treball ja he anat fent petites mencions a alguns d'ells, en aquest apartat en parlo en més profunditat.

4.12.1. Desplaçament amb ràster

Per al desplaçament de plans ja he explicat que hi aplico un simple efecte de paral·laxi. Tot i que majoritàriament és així, en alguns nivells utilitzo uns efectes especials que encaixin amb l'estètica d'aquell. En el videojoc hi ha un nivell amb lava, aquesta a l'emetre calor genera una ona que distorsiona lleugerament la visió, per això li aplico un efecte de ràster al pla *B* per a fer aquella sensació.

Entre els efectes de ràster, hi ha el desplaçament per línies, aquest com ja he explicat anteriorment permet modificar les posicions de cadascuna de les línies que es dibuixen a la pantalla. Per a aplicar l'efecte de distorsió de la visió és tan senzill com usar una funció sinusoidal que s'actualitzi cada fotograma. Això ho faig emprant les següents funcions.

$$f(y,t) = amplitud * \sin(freqüència + (y + t) * velocitat)$$

$$g(y,t) = f(y,t) + posicióCàmera * k$$

La primera d'elles funciona a partir de dos paràmetres, la *y*, la qual és la posició de la línia i la *t*, el temps que ha estat funcionant. L'amplitud de l'ona, freqüència i velocitat són tres constants que defineixo anteriorment.

La segona funció depèn de la primera, aquesta serveix per a aplicar-li el desplaçament que li pertocaria segons la posició del jugador. La *k* és una constant que indica a quina velocitat es mou el pla, generalment menor que 1 per tal d'obtenir l'efecte de paral·laxi.

Tot el contingut de la funció *f(y, t)*, es precalcula amb un programa per tal d'estalviar recursos en el càlcul de desplaçament. Si no ho faig així, el resultat és d'aproximadament un 20% extra de consum de CPU.

4.12.2. Efectes llum i ombra

Els efectes de llum i ombra o com s'anomena en la documentació oficial *hilight and shadow*, probablement per una mala traducció. És un efecte que tal com indica el mateix nom, tracta d'enlluernar o enfosquir certes àrees de la pantalla. Aquest efecte es pot aplicar tant als *sprites* com als *tiles*, la diferència és que en el cas d'aplicar-lo a un *sprite*, aquest no pot afectar a altres *sprites*. Per tant si es volen assolir figures de bastanta precisió amb aquest efecte, s'haurien de combinar els dos plans.

Un dels nivells es desenvolupa en l'interior d'una cova, aquesta és fosca i per tant, l'única llum que hi ha és la del casc del talp. L'efecte funciona a partir de la combinació d'un pla i retalls d'un *sprite* molt gran.

Figura 4.35: Llum i ombra complet

Figura 4.36: Llum i ombra pla B

Figura 4.37: Llum i ombra sprites

Tal com es pot veure en les imatges, l'efecte està dividit en dos nivells, un d'ells és la figura primitiva que il·lumina els *sprites*. L'altre serveix simplement per a polir aquelles dents de serra i donar així l'efecte que realment és rodó.

4.12.3. Fos obrint/tancant

Les foses obrint i tancant, de l'anglès *fade-in* i *fade-out* són aquelles transicions generalment utilitzades en el món del cinema. Tracten, en el cas del *fade-out*, de canviar d'una imatge a un pla completament en negre i en el cas del *fade-in*, a l'inrevés.

Per a assolir aquest efecte, SGDK ja t'aporta una sèrie de funcions per a fer-ho. El seu funcionament és molt senzill, crea unes llistes de tots els colors pels quals ha de passar la paleta abans d'estar completament en negre o a color. Llavors itera per cadascun dels membres de la llista per tal de fer aquest efecte. L'únic inconvenient que hi ha és que al tenir només 512 possibles colors, si aquesta transició és molt lenta, es poden veure unes transicions força estranyes.

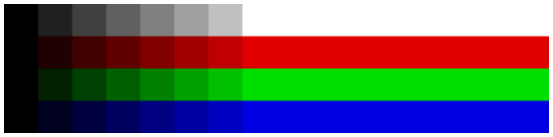


Figura 4.38: Passos que segueixen els diversos colors de la paleta

4.12.4. Animació de la lava

La lava és un fluid que hauria d'estar constantment en moviment per així fer veure com aquesta emet la calor que distorsiona la visibilitat del fons del nivell. Per això podria utilitzar un efecte de *palette swap* en el que la paleta simplement va variant de color seguint un cicle, però en canvi utilitzo una altra que tracta de fer un canvi en els *tiles*. Si canvio un *tile* per un altre en la VRAM, tots els que s'estiguin dibuixant a partir d'aquell índex, canviaran la imatge per la nova que li hagi introduït.

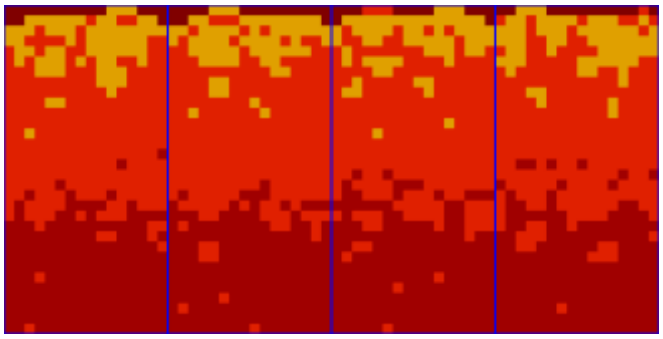


Figura 4.39: Diferents fotogrames animació lava

Si això ho faig repetidament, puc aconseguir una animació força fluida per a la lava i així donar-li encara més vida al nivell.

4.12.5. Animació pantalla de victòria

Per indicar que la seqüència del nivell s'ha acabat, li afegeixo una petita animació. Aquesta tracta de fer un fos tancant de totes les paletes de la pantalla menys a la del jugador i posteriorment dibuixar totes les monedes i punts que ha aconseguit en aquell nivell.

Figura 4.40: Pantalla final dels nivells

Per a evitar que el jugador sigui afectat pel fos, aquest utilitza la seva pròpia paleta. Cap altre objecte del nivell la fa servir. Per tant l'efecte quedarà en com el jugador segueix caminant mentre el nivell es va enfosquint.

Posteriorment, quan ja només queda el jugador, una de les paletes fa un fos obrint a una paleta per tal de dibuixar la puntuació i monedes que tenia el jugador abans d'entrar al nivell. Just després se li sumen les puntuacions que ha obtingut durant la seva partida en ell.

Finalment, quan ja s'ha sumat tot, el talp cava un forat a terra i hi entra, deixant així la pantalla completament buida per així poder fer una transició cap a la pantalla de selecció de nivells.

4.13. Empaquetat

El videojoc ja està acabat, només falta un empaquetament, fins ara només l'havia pogut fer funcionar a la consola mitjançant un Mega Everdrive X3, un cartutx amb una ranura per a poder-hi introduir els videojocs mitjançant una targeta microSD.

Això és una tècnica comuna actualment, però aquest tipus de targetes no van aparèixer fins a l'any 1999. Per tant, per a seguir amb el propòsit de fer un videojoc el més fidel possible a l'època, utilitzo una PCB amb EEPROM.

4.13.1. Programar una PCB

Ja he explicat que els cartutxos originals comptaven amb unes memòries ROM, en canvi la que utilitzo jo és de tipus EEPROM. És un tipus de memòria que pot ser modificada entre cent mil i un milió de vegades i llegida infinites vegades. D'altra banda, les ROM, només es poden escriure un sol cop.

Per a programar una PCB amb una memòria d'aquest tipus empro un programador fet especialment per a això. Tant la placa com el programador que uso són els dos de Krikzz.

Figura 4.41: Programador i PCB de Krikzz

Per a escriure-li les dades del videojoc, es fa mitjançant un programa molt simple al qual se li introdueix l'arxiu que el conté i aquest automàticament detecta si hi ha connectat un programador i inicia la neteja i escriptura de la PCB que estigui connectada.

Un cop fet això, ja es pot introduir la placa al lector de la consola. Però, abans de res, se li ha d'afinar les cantonades per assegurar-se de no danyar el lector. Ara sí que ja es pot connectar i immediatament en encendre la consola inicia el videojoc, sense passar per cap menú ni cap pantalla de càrrega, a diferència del Mega Everdrive.

4.13.2. Disseny de l'empaquetatge

L'empaquetatge és la coberta del videojoc, el que dona la informació sobre el seu contingut, consta de tres elements: la caixa, el manual i la carcassa del cartutx.

4.13.2.1. Disseny caixa

La caixa té dues cares, la caràtula, la qual dona informació sobre el nom del videojoc, juntament amb un dibuix que representi el seu contingut d'alguna manera, sigui amb el personatge o amb una il·lustració de l'entorn. L'altra cara és la contraportada, aquesta conté una breu descripció en vuit idiomes diferents juntament amb quatre captures extretes del videojoc.

Figura 4.42: Disseny caixa del videojoc del treball

Aquest és el disseny, tal com es pot veure conté el nom del videojoc, *Miner Mole*, juntament amb una versió a alta resolució del personatge. A la contraportada s'hi poden veure les captures i just al costat la informació bàsica escrita en català, castellà, anglès, francès, italià, alemany, neerlandès i suec. Totes aquestes traduccions les faig mitjançant el traductor de Google, per així simular la qualitat que hi havia en aquella època, on fàcilment li podies trobar errors.

4.13.2.2. Disseny del manual

El manual segueix la mateixa filosofia que amb la caixa, traduccions de baixa qualitat i amb un disseny el més semblant possible als originals.

Aquest compta amb un gruix considerable de pàgines explicant el contingut del videojoc, a més d'alguns secrets sobre els diferents apartats que es poden veure durant una partida en aquest.

Figura 4.43: Disseny caràtula del manual i primera pàgina del manual

4.13.2.3 Disseny cartutx

Aquest és l'element més senzill de dissenyar, normalment en els cartutxos s'hi col·loca una petita alteració del disseny de la caràtula amb el nom just a sobre. No hi ha cap mena de complicació a l'hora de dissenyar-se, és l'element que més s'ignora d'un videojoc.

Tal com es pot veure, en tots tres dissenys hi predomina una quadrícula sobre un fons negre, aquesta era la principal característica que diferenciava els videojocs europeus dels de la resta del món.

Figura 4.44: Etiqueta del cartutx

Conclusions

Finalment, el videojoc després de totes aquestes petites iteracions, ha acabat pesant poc més de 3 MB, la gran majoria d'aquests a causa del format de la música. Per a tenir un punt de comparació, en la dimensió d'aquest s'hi podria fer cabre una cançó d'uns 3 minuts de durada en format MP3.

El joc compta amb 4 nivells diferents, a més, gràcies a totes les eines que he anat creant durant el treball, puc afegir-li molt fàcilment tants nivells extres com vulgui. Hi ha 4 entorns diferents, un per cada nivell, s'hi troben el bosc, el castell, el temple i la cova.

Figura 4.45: Captures nivells

Gràcies a la gran feina de compressió de l'ADPCM, he pogut introduir 5 cançons diferents i una gran varietat d'efectes de so sense cap mena de complicació. En el cas de les cançons, aquestes sofreixen una lleugera pèrdua de matisos en els sons més aguts, però s'obté una gran quantitat de sons que serien molt complicats de reproduir utilitzant els canals FM.

Generalment no aconsegueixo esprémer al màxim la potència gràfica. Normalment faig servir les 4 paletes al complet i generalment no faig ús d'efectes especials amb elles. Tot i això, ha quedat un resultat que diferencia l'abans i després de les consoles de 16-bits.

Fent referència al codi, aquest petit experiment ha comportat més d'11.000 línies de codi en C i uns quants centenars més provinents de les petites aplicacions fetes en C# per agilitzar el procés de creació de nivells, efectes, etc.

Durant el procés de creació he estat emprant el Visual Studio 2022/Code per a la programació. Per a l'apartat artístic he fet servir aseprite com a eina d'edició d'imatges i una eina pròpia per a la modificació de paletes. Per a l'edició d'àudio he usat Audacity. A més, alguns dels elements que hi ha al videojoc com la música i part de l'art han estat extrets del portal de recursos *itch.io*. Finalment, alguns dels efectes de so han estat generats mitjançant diversos algoritmes implementats en el programa sfxr.

L'objectiu del treball era crear un videojoc amb l'aparença d'un que hagués estat desenvolupat durant l'època d'aquella consola. Això s'ha complert en gairebé el seu complet. L'únic problema amb el qual m'he trobat ha estat amb la composició musical.

Tal com s'ha pogut veure, el pas dels anys ha donat fruit a què nasquessin certes facilitats a l'hora de desenvolupar en aquests sistemes. Per altra banda, la manera d'accedir al coneixement es veu molt limitada per la mateixa causa. Durant aquest treball, la gran majoria d'elements difícilment els haguera sabut implementar sense l'ajut de gent que ha treballat amb la consola anteriorment, entre els quals estava el mateix creador del SGDK.

Justament per això mateix, una de les conclusions d'aquest treball és una breu reflexió sobre el nostre desconeixement de la informàtica del passat. Molts de nosaltres hem pogut veure com la tecnologia avança a passos agegantats, tot i que això no implica que hàgim de deixar de banda el baix nivell. L'actualitat es basa en la velocitat, però cada cop ens centrem menys en les optimitzacions d'un programa a causa de la nostra confiança en les millores actuals i aquest exercici ensenya com el coneixement del baix nivell t'ajuda a saber com adquirir aquesta velocitat extra. Això es podria obtenir documentant el funcionament de programes i sistemes teòricament obsolets a més de l'estudi d'aquests.

Per altra banda, s'ha vist com he creat el meu propi cartutx que fàcilment podria fer-se passar per un d'original. Això està molt relacionat amb la preservació de dades que tanta importància ha rebut durant els últims anys. Una de les maneres en les quals projectes com aquests hi contribueixen és gràcies a l'ús de programadors. A part d'utilitzar-los per a programar, també els podem utilitzar per a llegir i així crear còpies amb les quals en un futur, quan els circuits originals deixin de funcionar, poder tenir un sistema amb el qual replicar-los i per tant, allarga'ls-hi la vida útil així com la preservació d'aquests.

Seguint aquesta direcció, es podria plantejar una nova línia de recerca en la qual s'investigués el funcionament profund de sistemes obsolets i a través de la investigació, aportar-li elements extres pels quals no hi haguessin estat dissenyats. Les tecnologies del passat t'aporten moltes funcionalitats que s'han perdut en el temps, el coneixement i documentació d'aquestes pot ajudar a millorar la informàtica del futur.

Bibliografia

Andrej. (10 / 01 / 2021). *Tutorials*. Recollit de Ohsat: https://www.ohsat.com/tutorial/

Bitwise Operators in C/C++. (14 / 04 / 2021). Recollit de GeeksforGeeks: https://www.geeksforgeeks.org/bitwise-operators-in-c-cpp/

Bright, G. (20 / 06 / 2021). *Build a Bad Guy Workshop - Designing enemies for retro games*. Recollit de Game Developer: https://www.gamedeveloper.com/design/build-a-bad-guy-workshop---designing-enemies-for-retro-games

ChibiAkumas. (25 / 10 / 2021). *Hello World on the Genesis / Megadrive*. Recollit de ChibiAkumas: https://www.chibiakumas.com/68000/helloworld.php#LessonH5

Commodore 64. (02 / 11 / 2021). Recollit de Wikipedia: https://en.wikipedia.org/wiki/Commodore_64

Copetti, R. (10 / 8 / 2021). *Arquitectura Mega Drive*. Recollit de Copetti: https://www.copetti.org/writings/consoles/mega-drive-genesis/

Dallongeville, S. (16 / 12 / 2020). *SGDK*. Recollit de GitHub: https://github.com/Stephane-D/SGDK

Dallongeville, S. (15 / 03 / 2021). *SGDK Documentation*. Recollit de https://hardworld.webcindario.com/SGDK/index.html

Gryson. (10 / 09 / 2021). *History of the Sega Mega Drive / Genesis*. Recollit de Mega Drive Shock: https://mdshock.com/

Hacer juegos, mejor que jugarlos! (30 / 01 / 2021). Recollit de Danibus Games: https://danibus.wordpress.com/

Harrison, J. (20 / 07 / 2021). *A Guide to the Graphics of the Sega Mega Drive / Genesis*. Recollit de Raster Scroll Books: https://rasterscroll.com/mdgraphics/

Homebrew Development. (17 / 08 / 2021). Recollit de Technopedia: https://www.techopedia.com/definition/10649/homebrew

Keil, A. (20 / 9 / 2021). *Sega Genesis/Mega Drive VDP Graphics Guide*. Recollit de Mega Cat Studios: https://megacatstudios.com/blogs/retro-development/sega-genesis-mega-drive-vdp-graphics-guide-v1-2a-03-14-17

Keren, I. (02 / 05 / 2021). *Scroll Back: The Theory and Practice of Cameras in Side-Scrollers*. Recollit de Game Developer: https://www.gamedeveloper.com/design/scroll-back-the-theory-and-practice-of-cameras-in-side-scrollers

Khalifa, A. (19 / 11 / 2021). *Level Design Patterns in 2D Games*. Recollit de Game Developer: https://www.gamedeveloper.com/design/level-design-patterns-in-2d-games

Llenguatge d'assemblador. (17 / 08 / 2021). Recollit de Wikipedia: https://ca.wikipedia.org/wiki/Llenguatge_d%27assemblador

Ludos, D. (15 / 12 / 2020). *Making a SEGA Mega Drive / Genesis game in 2019*. Recollit de Game Developer: https://www.gamedeveloper.com/design/making-a-sega-mega-drive-genesis-game-in-2019

Mega Drive cartridges. (02 / 10 / 2021). Recollit de Sega Retro: https://segaretro.org/Mega_Drive_cartridges

Monteiro, R. (13 / 02 / 2021). *The guide to implementing 2D platformers*. Recollit de Higher-Order Fun: http://higherorderfun.com/blog/2012/05/20/the-guide-to-implementing-2d-platformers/

Plutiedev. (02 / 02 / 2021). Recollit de https://plutiedev.com/

Sega Enterprises. (03 / 04 / 2021). *Genesis Software Manual.* Recollit de Sega Retro: https://segaretro.org/images/a/a2/Genesis_Software_Manual.pdf

Sega Mega Drive. (30 / 08 / 2021). Recollit de Wikipedia: https://en.wikipedia.org/wiki/Sega_Genesis

Voltonov, F. (08 / 06 / 2021). *THE SEGA MEGA DRIVE/GENESIS VDP IN 16 PICS*. Recollit de https://hardworld.webcindario.com/VDPin16/

Llistat de sigles i acrònims

AABB	*Axis-Aligned Bounding Box* (capsa delimitada alineada amb l'eix)
ADPCM	*Adaptive Differential Pulse-Code Modulation*
CD-ROM	*Compact Disc Read-Only Memory*
CPU	*Central Processing Unit* (unitat central de processament)
CRAM	*Color Random-Access Memory*
DMA	*Direct Memory Access* (accés directe a memòria)
DPCM	*Differential Pulse-Code Modulation*
EEPROM	*Electrically Erasable Programmable Read-Only Memory*
FM	*Frequency Modulator* (modulació de freqüència)
IDE	*Integrated Development Environment*
NTSC	*National Television System Committee*
PAL	*Phase Alternating Line* (línia alternada en fase)
PCB	*Printed Circuit Board* (placa de circuit imprès)
PCM	*Pulse-Code Modulation*
PSG	*Programmable Sound Generator*
RAM	*Random-Access Memory* (memòria d'accés aleatori)
ROM	*Read-Only Memory* (memòria solament de lectura)
SDK	*Software Development Kit* (kit de desenvolupament de programari)
SGDK	*Sega Genesis Development Kit*
SNES	*Super Nintendo Entertainment System*
SRAM	*Static Random-Access Memory*
TMSS	*TradeMark Security System*
VDP	*Video Display Processor*
VGM	*Video Game Music*
VRAM	*Video Random-Access Memory*
VSRAM	*Vertical Scroll Random-Access Memory*

Glossari

.NET	Framework de codi obert, desenvolupat per Microsoft per a facilitar el desenvolupament entre plataformes.
Array	Estructura per a emmagatzemar dades d'un mateix tipus en una sola variable. També conegut com a matriu.
Creative Commons	Organització sense ànim de lucre dedicada a reduir les barreres legals per a compartir treballs creatius.
cartutx	Carcassa extraïble que conté arxius en una ROM destinats a ser connectats a una videoconsola.
framework	Entorn de treball que facilita la construcció d'aplicacions.
metasprite	Conjunt de sprites que actuen com a un de sol.
repositori	Sistema informàtic on s'emmagatzema la informació d'una organització amb la finalitat que els seus membres la puguin compartir.
sprite	Element gràfic que es pot desplaçar lliurement per la pantalla.
string	Seqüència de caràcters.
tic	Unitat de temps molt petita que s'utilitza per a representar el pas del temps en un sistema digital.
tile	Element gràfic que s'utilitza en conjunt a varis més per a formar una imatge.
word	Nombre màxim de bits que pot processar la CPU en un cicle.

A. Treballar amb SGDK

A.1. Instal·lació de SGDK

El primer pas que s'ha de fer per a instal·lar l'equip de desenvolupament per a la Sega Mega Drive és dirigir-nos al seu repositori de GitHub. A partir d'aquí podem dirigir-nos a la pestanya de llançaments o descarregar-nos l'última modificació directament des del botó verd de la part superior.

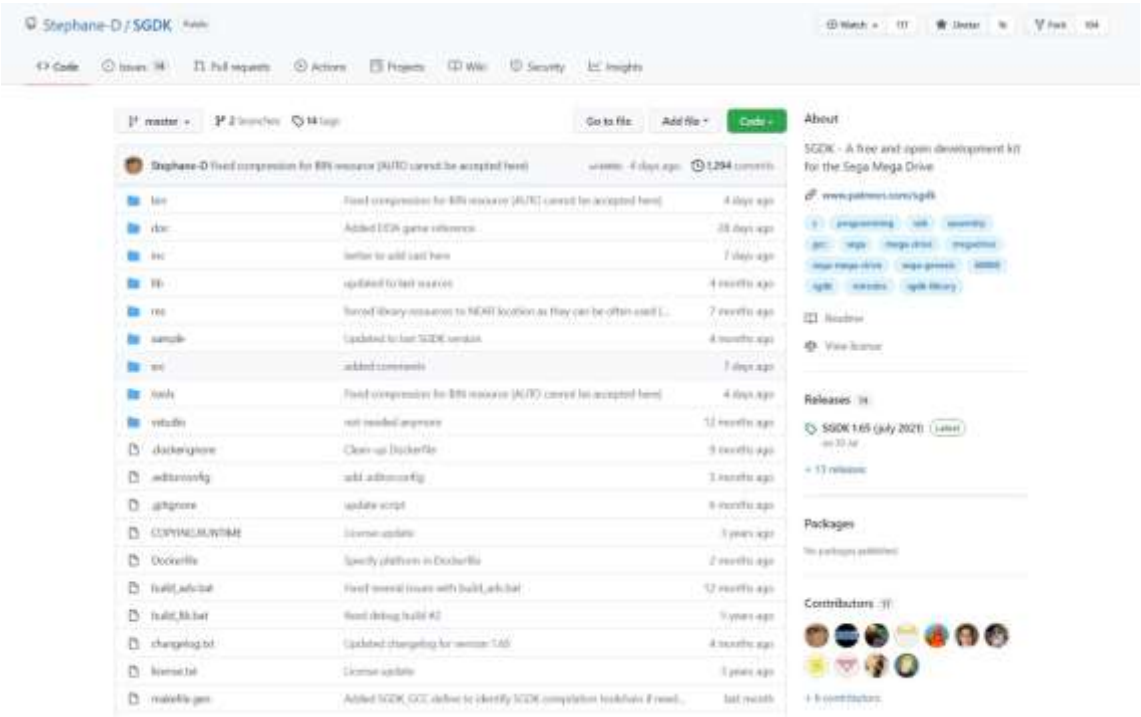


Figura A.1: Repositori de GitHub SGDK

En el moment de la redacció del treball, la versió més baixa que es pot descarregar és la 1.22 i la més alta la 1.65. Això és molt important perquè els processos d'instal·lació i ús del programa poden canviar d'una versió a l'altra, en aquest cas descarregaré l'última.

Seguidament, un cop descarregat el codi font, es crea una carpeta a l'origen del disc dur amb el mateix nom de l'equip, en la ruta "C:\sgdk". Dins seu s'hi aboquen tots els arxius del repositori.

Just després arriba l'hora de crear el projecte, però abans defineixo unes variables d'entorn per a accedir més fàcilment a la carpeta.

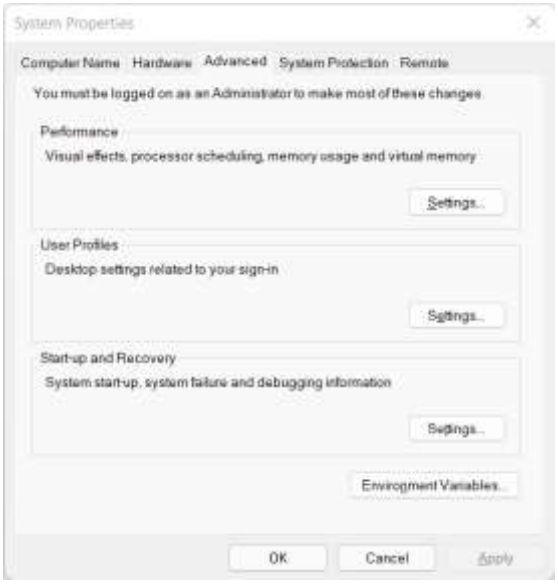


Figura A.2: Propietats del Sistema

Figura A.3: Variables d'Entorn

Per això, primer accedeixo a la configuració de propietats del sistema i posteriorment en el botó inferior de la pestanya d'avançat, accedeixo a la configuració de les variables. S'han de definir les següents:

Nom de la variable	Valor de la variable
GDK	C:/sgdk
GDK_WIN	C:\sgdk

Aquestes en versions anteriors a la 1.62 eren obligatòries, en el cas d'utilitzar una versió més recent, com ho és en aquest cas, em quedo només amb la "GDK_WIN" per a tenir-la com a accés directe a la carpeta de la instal·lació.

Un cop fet això, ja es pot crear un projecte en qualsevol editor de codi, fins i tot es podria fer amb el bloc de notes, però per a facilitar una mica el procés, faré servir el Visual Studio 2022.

A.2. Instal·lació de Visual Studio 2022

Per a instal·lar Visual Studio 2022, ens hem de dirigir a la pàgina de Microsoft, des d'allà, se segueixen les instruccions per a instal·lar la versió *Community,* aquesta és gratuïta i et prové unes eines més que suficients per a qualsevol classe de projecte.

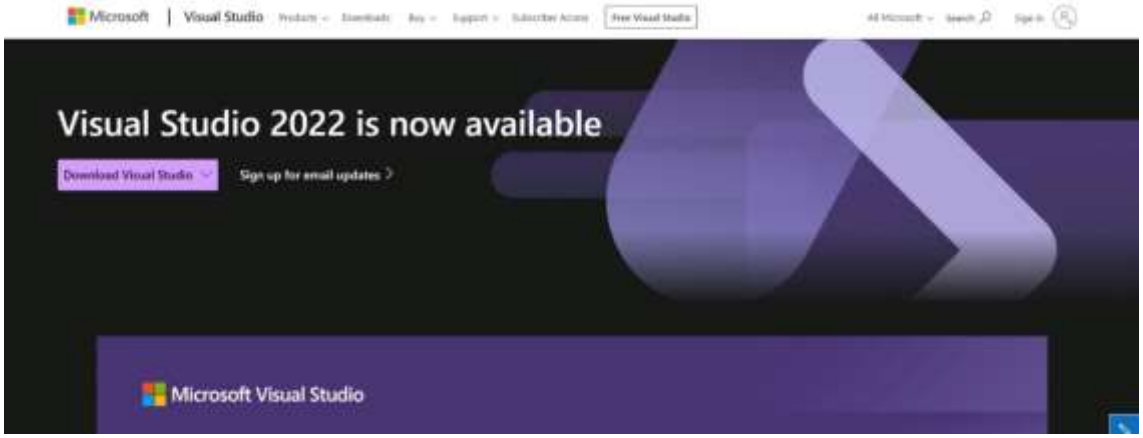


Figura A.4: Pàgina d'aterratge Visual Studio

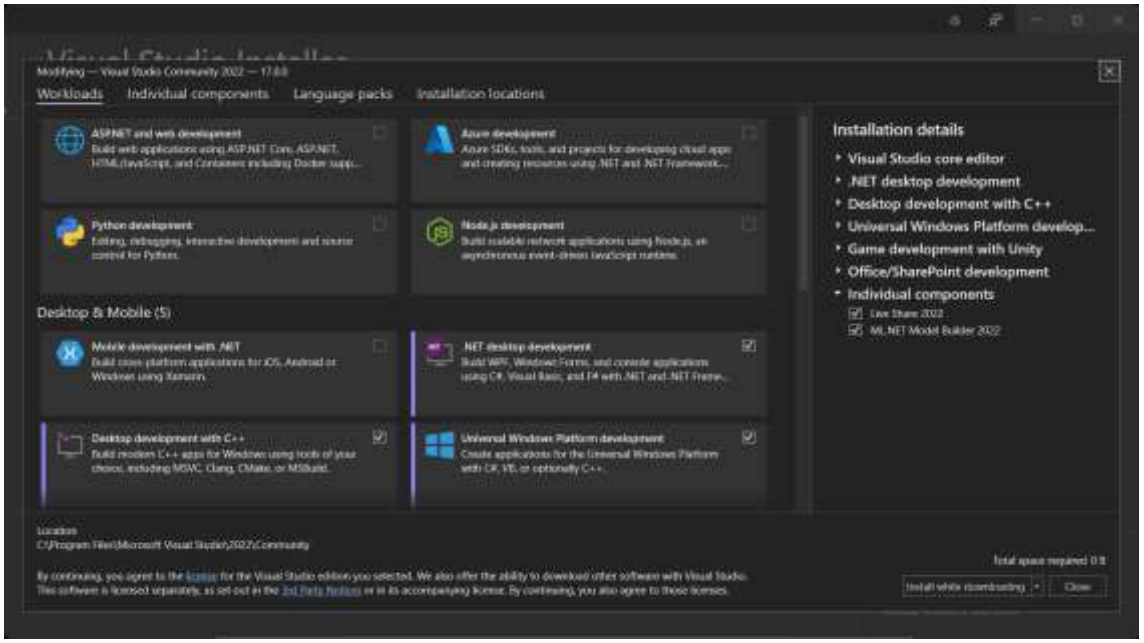


Figura A.5: Finestra instal·lació de components de Visual Studio

Un cop descarregat l'instal·lador, haurem de seleccionar la versió que vulguem i posteriorment a la pestanya de components, seleccionar "*Desktop development with C++".* Aquest ens permet utilitzar els llenguatges C/C++ i crear el tipus de projectes que necessitem per a aquests casos.

A.3. Creació d'un projecte de SGDK

Per a crear un projecte de SGDK amb Visual Studio, primer hem de seleccionar des de la pantalla inicial "Crear un nou projecte".

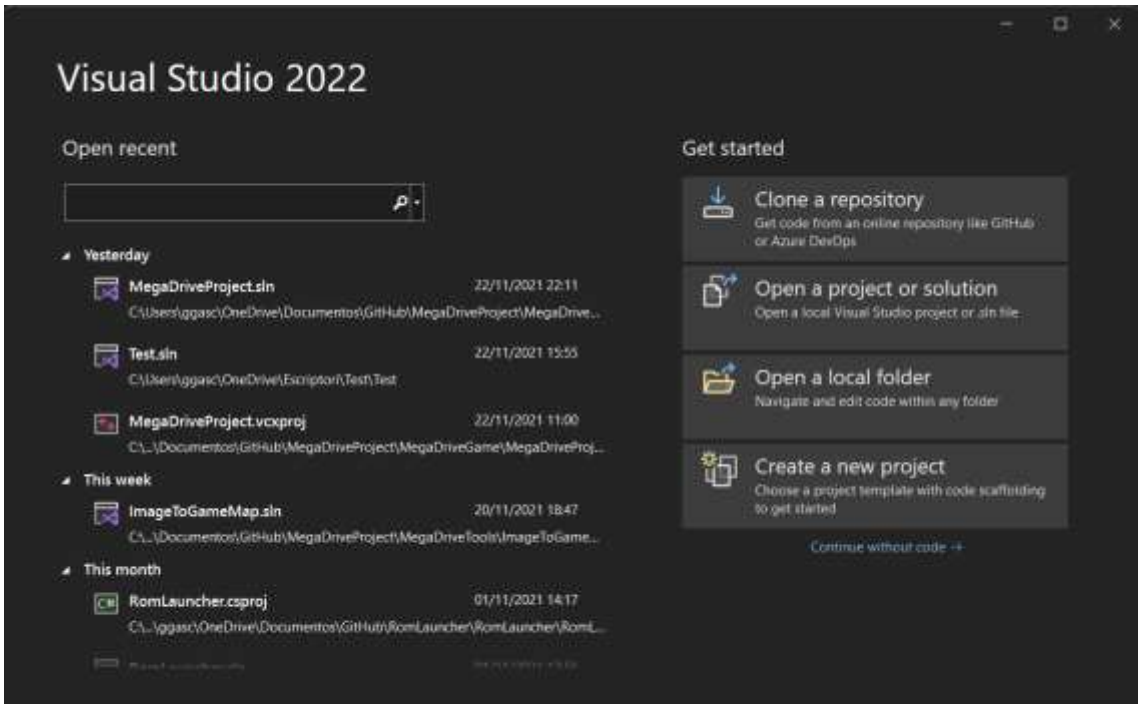


Figura A.6: Finestra d'inici del Visual Studio 2022

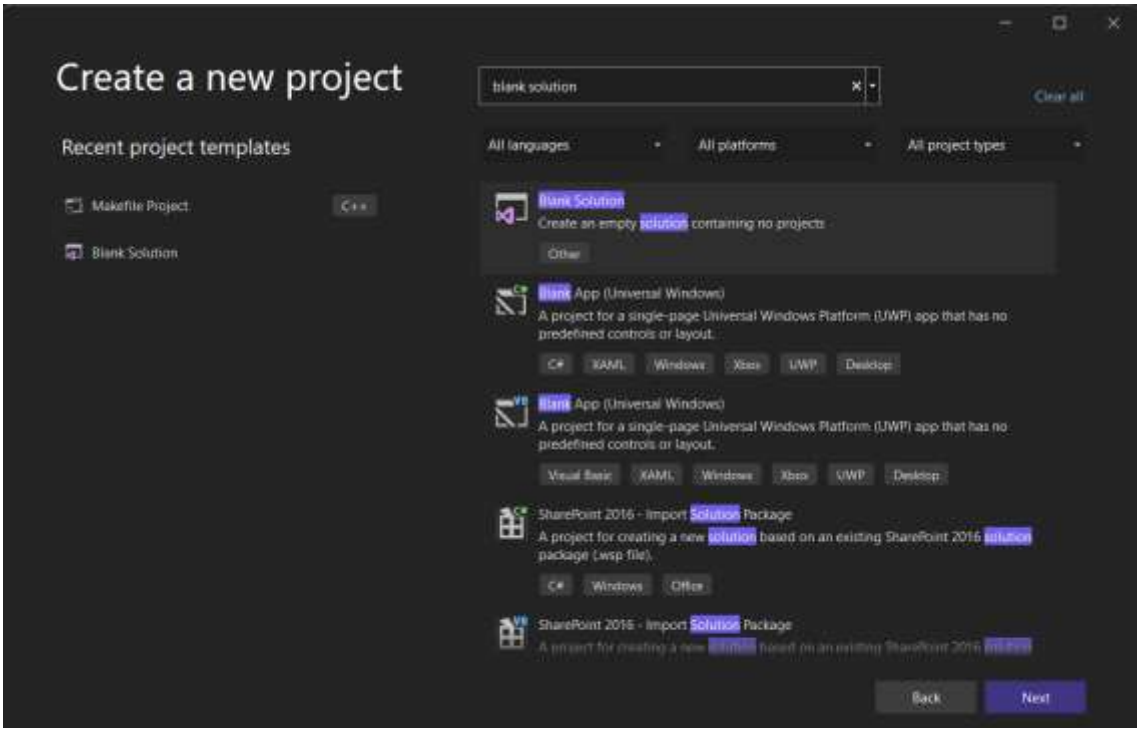


Figura A.7: Finestra creació d'un nou projecte al Visual Studio 2022

Seguidament, creem una "Solució en blanc", aquesta contindrà el nostre projecte en el seu interior.

Això ens enviarà en aquest menú, en el qual hem de seleccionar la ruta sota la qual se situarà la solució i el nom d'aquesta.

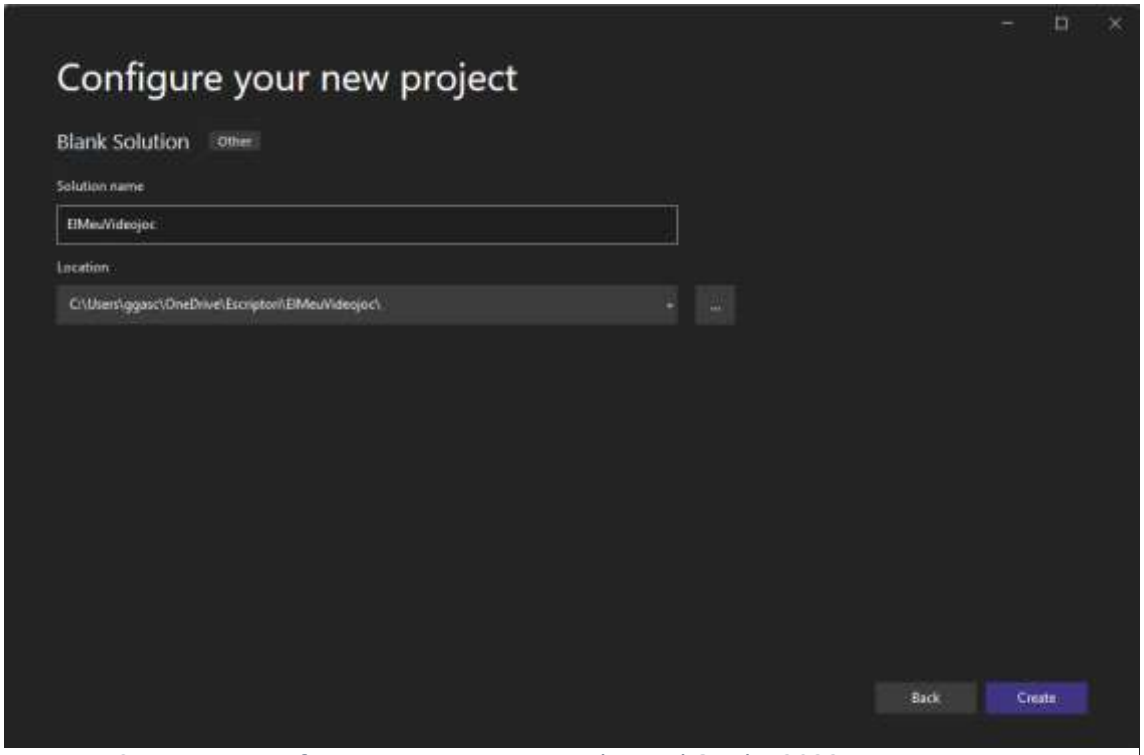


Figura A.8: Finestra configurar un nou projecte al Visual Studio 2022

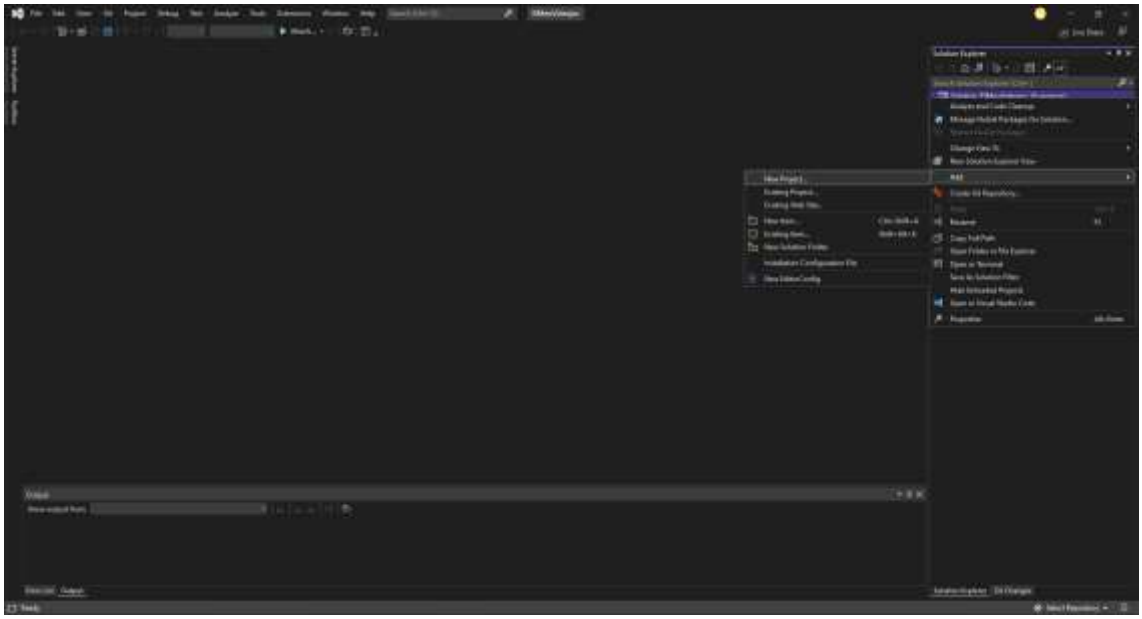


Figura A.9: Editor de codi Visual Studio 2022 buit

Un cop fet això, ja accedim a l'editor de codi, però encara no en podem escriure. Per això, hem de crear un projecte dins de la solució. Per a fer això, hem de fer clic dret a la solució de la pestanya anomenada "*Solution Explorer*" des d'allà, ens dirigim a afegir un nou projecte, el qual ens portarà al següent menú.

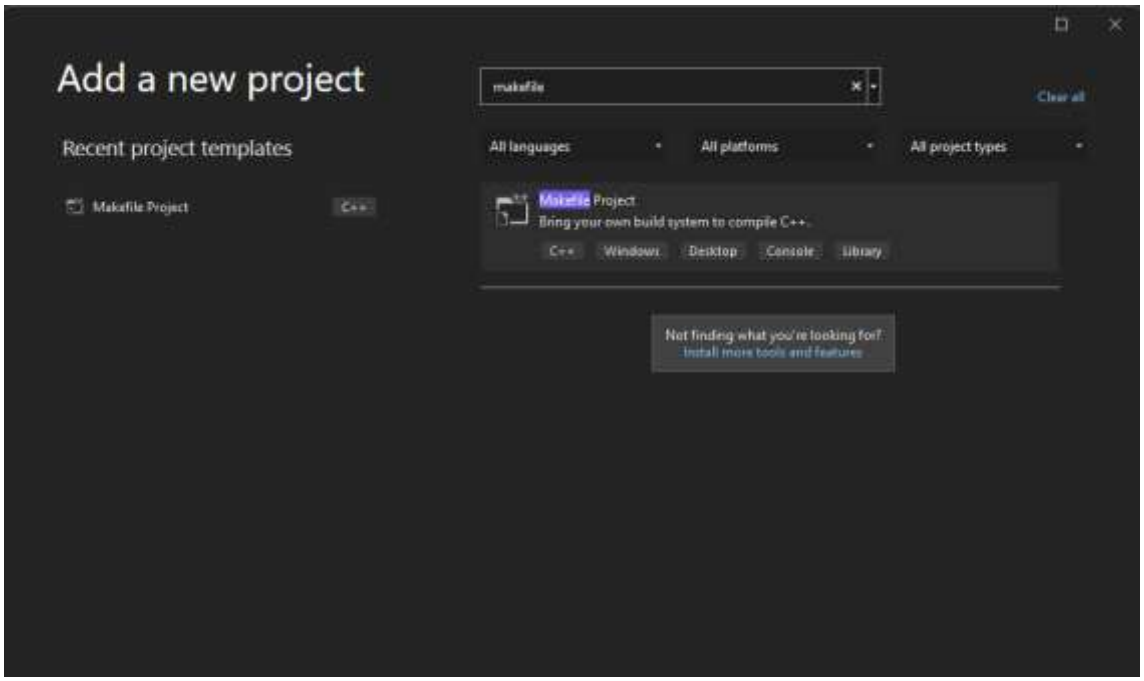


Figura A.10: Finestra crear un nou projecte makefile al Visual Studio 2022

Aquí hem de buscar el "*Makefile Project*", aquest és el tipus de projecte que necessitem per a compilar amb SGDK. Posteriorment ens sortirà el mateix menú que per a crear una solució, allà simplement li posem el nom que vulguem al nostre projecte i cliquem al botó de crear.

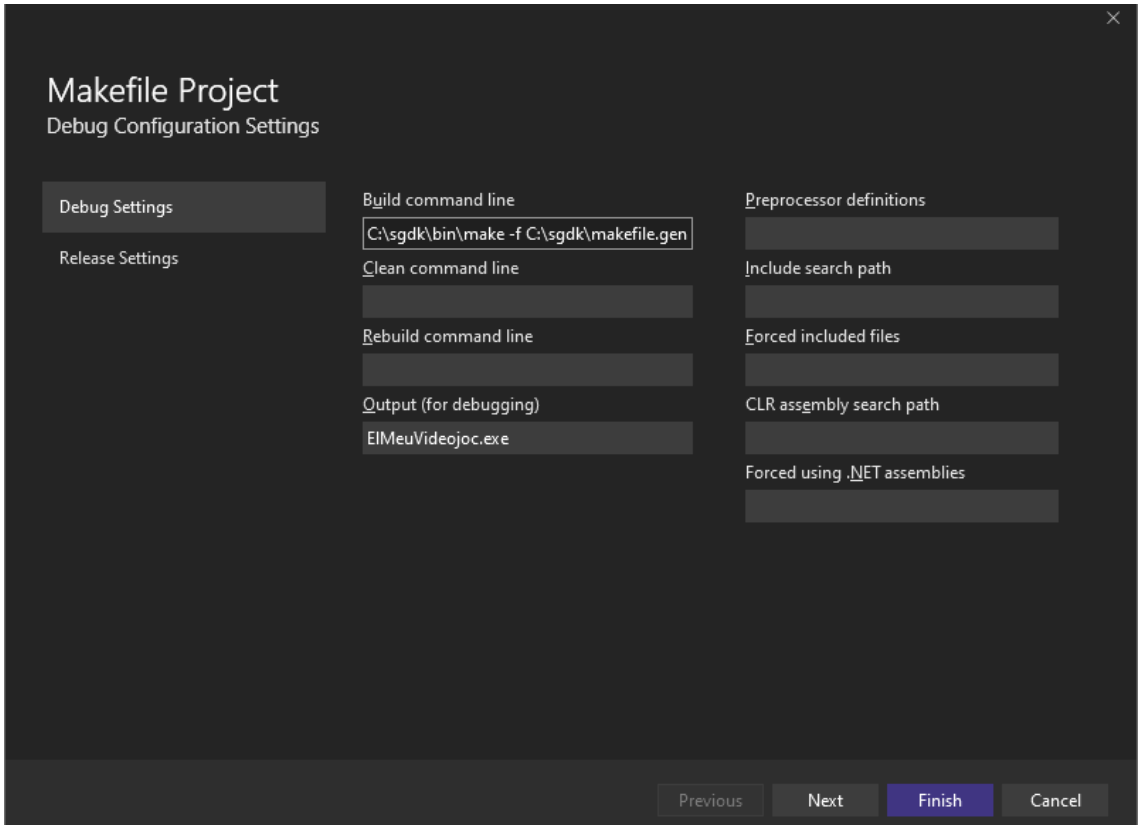


Figura A.11: Finestra configurar un projecte makefile al Visual Studio 2022

En aquesta finestra, li hem d'escriure el comandament per a compilar amb el *makefile* de SGDK, aquest és el següent:

```
C:\sgdk\bin\make -f C:\sgdk\makefile.gen
```

Aquesta línia serveix per a utilitzar el compilador, tot i que com he dit anteriorment, podem utilitzar la variable d'entorn. En aquest cas, el comandament seria el següent:

```
%GDK_WIN%\bin\make -f %GDK_WIN%\makefile.gen
```

En el segon cas, és possible que s'hagi de reiniciar l'ordinador per tal que la variable d'entorn sigui configurada correctament.

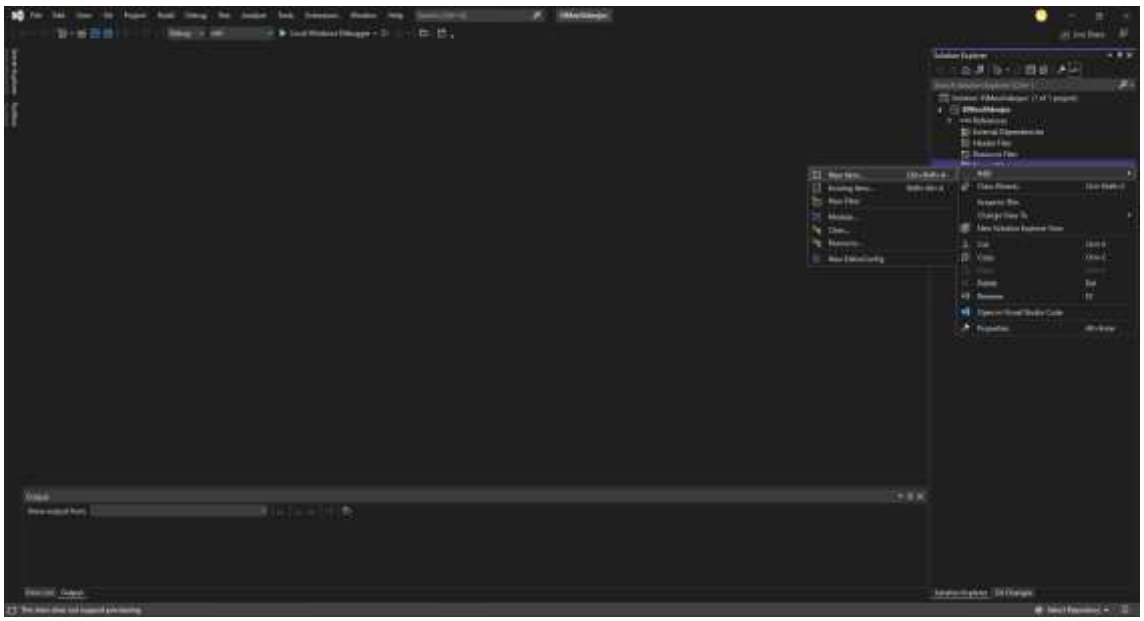


Figura A.12: Finestra de projecte buit Visual Studio 2022

Un cop creat, a la mateixa pestanya des d'on has creat el projecte, busques la carpeta de "*Source Files*" i amb clic dret li afegeixes un element.

En fer això, ens apareixerà aquest menú, hem de clicar a "*C++ File*", però SGDK no funciona amb C++, funciona amb C. Per això mateix quan nomenem l'arxiu, li canviarem l'extensió a .c en lloc de .cpp. Aquest script ha de portar el nom de main.c.

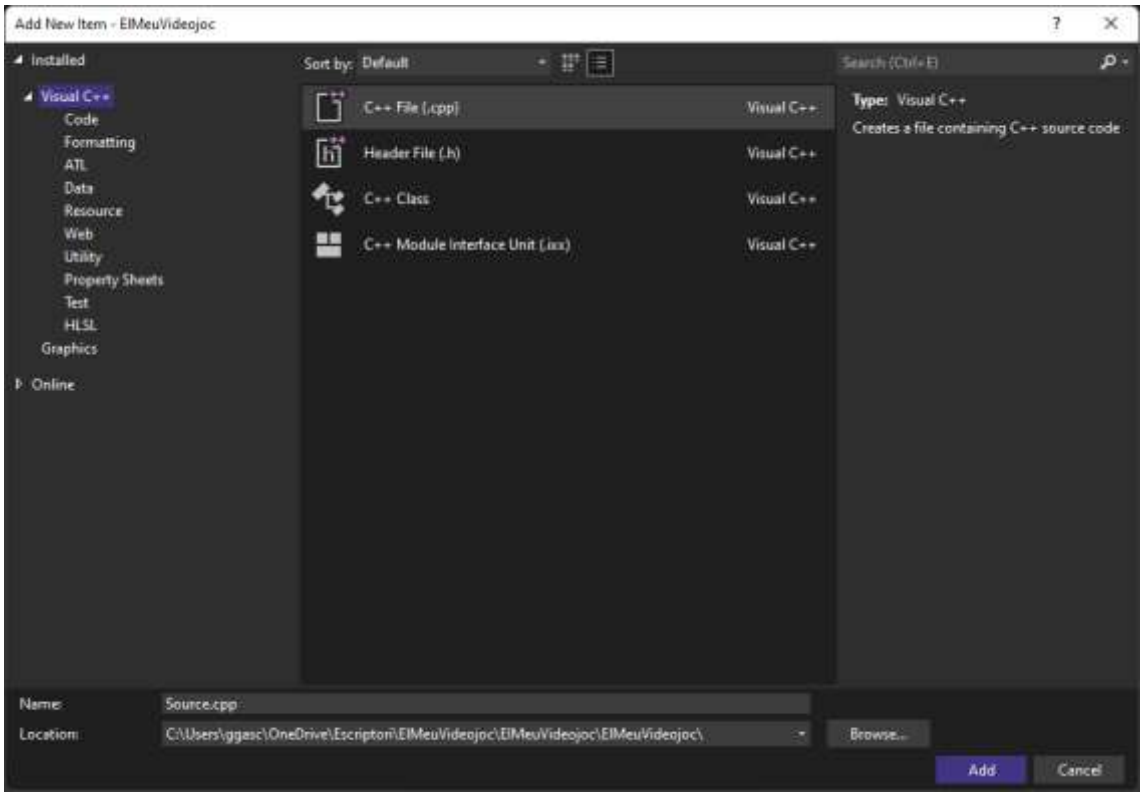


Figura A.13: Finestra afegir un nou element Visual Studio 2022

Ara ja tenim l'arxiu de codi creat i estem utilitzant el compilador de SGDK, però encara ens ho podem facilitar una mica més. Si en la pestanya on hem afegit el script sota de l'arxiu de la solució, on hi ha el projecte li fem clic dret i seleccionem propietats, accedirem a algunes de les opcions que ens interessen.

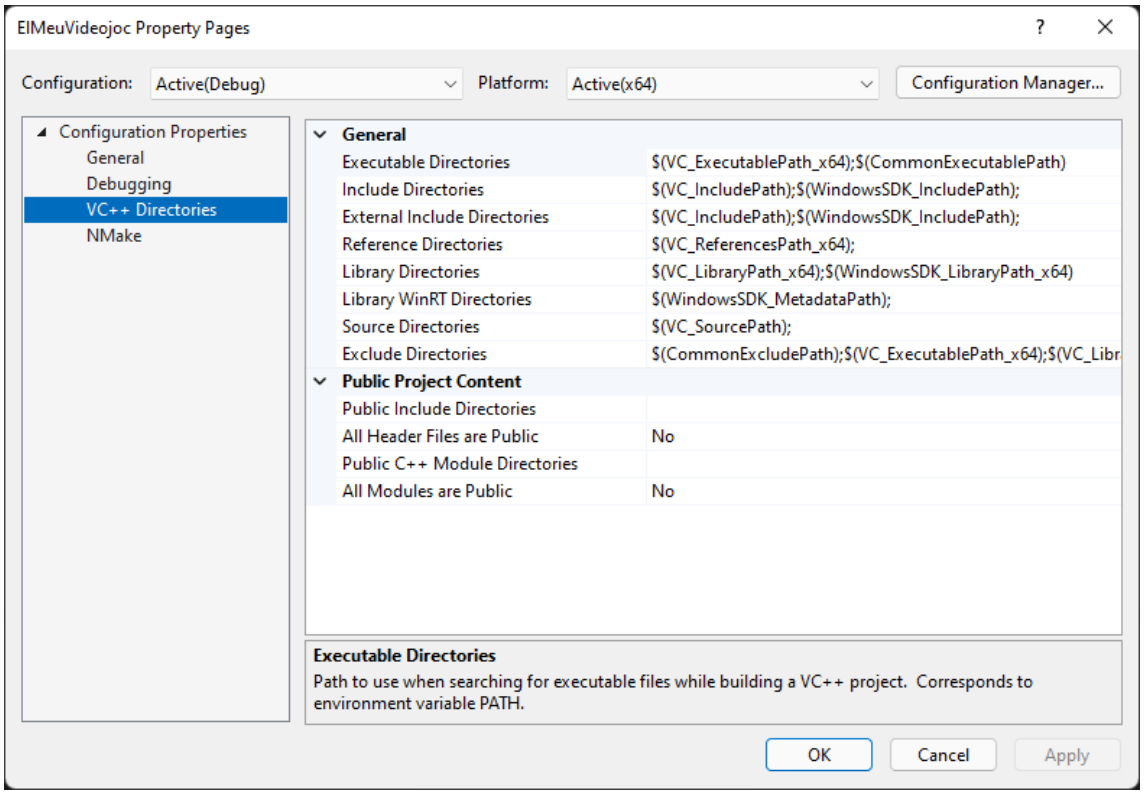


Figura A.14: Finestra de propietats d'un projecte Visual Studio 2022

Aquí seleccionem la pestanya de "*VC++ Directories*", en la secció que hi posa "*Include Directories*", si la seleccionem, ens apareixerà una fletxeta per a editar-la.

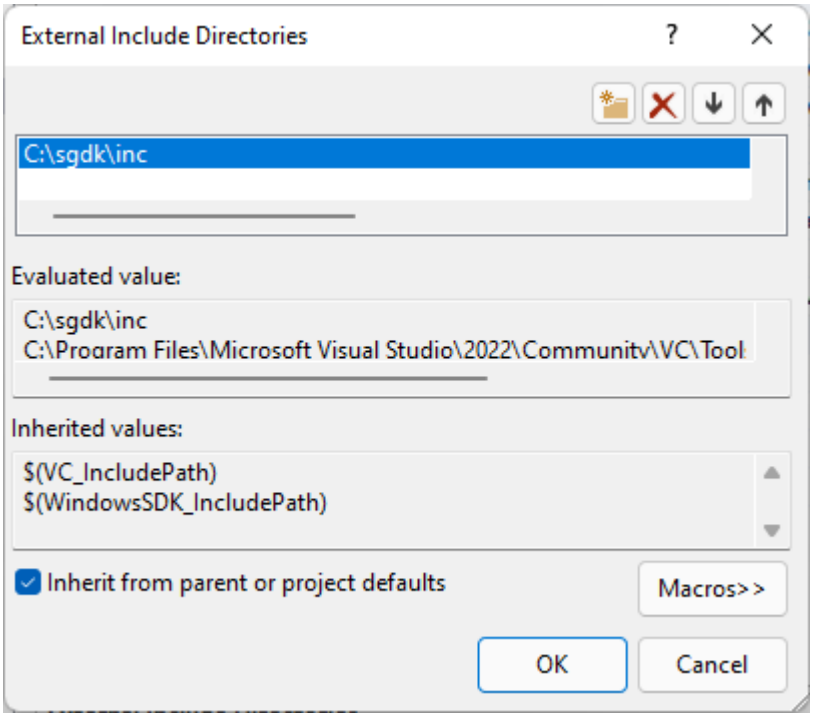


Figura A.15: Finestra d'edició de les direccions d'inclusions

Un cop seleccionada, ens apareix aquesta finestra, aquí li escrivim la direcció de la carpeta *inc* de SGDK, això ens proporciona una de les facilitats de Visual Studio com ho és l'autocompletat i descripció de les funcions.

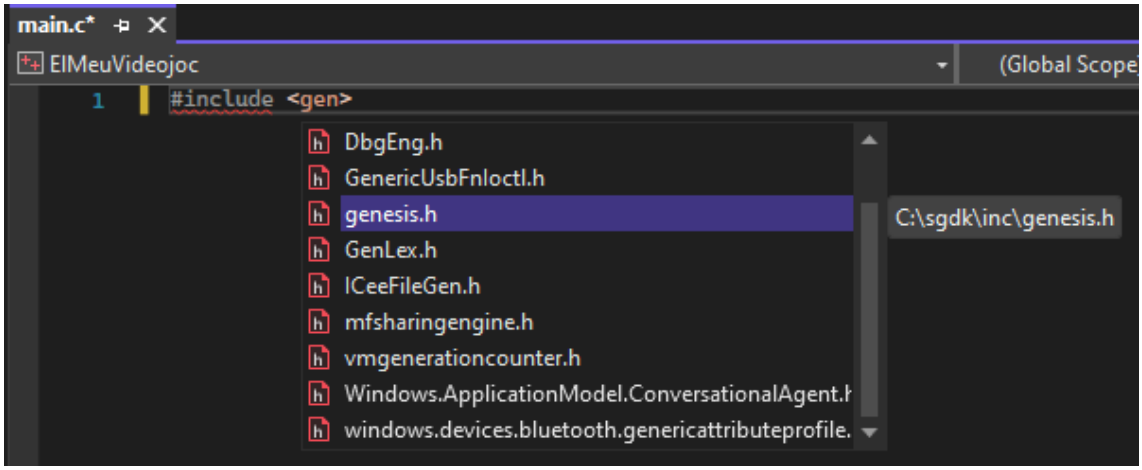


Figura A.16: Autocompletat del Visual Studio 2022

Si tot això ho hem fet correctament, a l'escriure aquesta línia ens hauria d'aparèixer *genesis.h* en el llistat d'autocompletat. Seguidament ja podem crear un programa seguint les instruccions explicades en l'apartat 3.2 del treball.

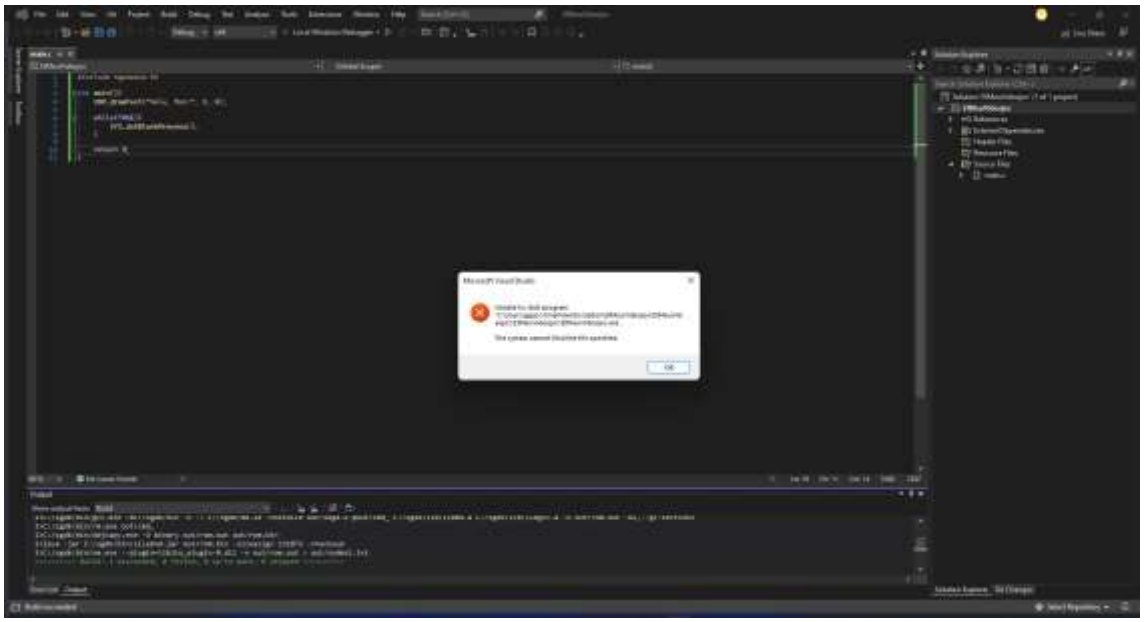


Figura A.17: Visual Studio quan acaba la compilació

Si cliquem a la barra superior el botó "*Local Windows Debugger*", ens apareix aquest error, però tal com es veu en la consola de la zona inferior, la compilació s'ha completat correctament. Aquest error és degut al fet que Visual Studio busca un arxiu en format *.exe* per a executar-lo, però la Mega Drive no utilitza aquest tipus de fitxers i per tant ens salta aquest error.

A.4. Instal·lació d'un emulador

Emular la Sega Mega Drive no és fàcil, això implica que no tots els emuladors funcionin igual, per això jo recomano tres d'ells, tots tres els he utilitzat en diverses ocasions al llarg del treball, el GensKMod, el BlastEm i el Regen. El primer d'ells, et proporciona un munt d'eines per a veure al detall les funcions internes que està fent la consola, a més d'alguna eina extra que funciona en conjunt amb SGDK. BlastEm té com a principal punt fort la seva fidelitat al funcionament de la consola, sent aquest el que millor emula els colors i els efectes que pot fer aquesta. Per últim, Regen no té cap apartat que destaqui per sobre de la resta, sinó que compta amb algunes funcions com el mode lent que t'ajuden a veure en detall el que està passant en el joc per així poder corregir errors en el codi.

Per a instal·lar qualsevol d'ells simplement hem d'anar a la seva respectiva pàgina web i descarregar-nos-el. En ser portables, no requereixen cap instal·lació, simplement hem d'obrir-los i arrossega'ls-hi l'arxiu anomenat *rom.bin* que ens hagi creat el compilador en el subdirectori *out* de la carpeta del projecte.

A.5. Depurar una compilació

Si durant les proves notem unes falles que no sabem d'on provenen, sempre es pot depurar la compilació, per això s'han de seguir certs passos per tal d'activar aquest mode.

A.5.1. Activar la compilació amb depurador

Primer hem d'activar el depurador, per això, el que fem és anar a la pestanya de *NMake* de les propietats del projecte i posteriorment canviar el "*Build Command Line*" que li havíem assignat anteriorment afegint-li *debug* al final de tot. De tal manera que quedaria així:

```
%GDK_WIN%\bin\make -f %GDK_WIN%\makefile.gen debug
```

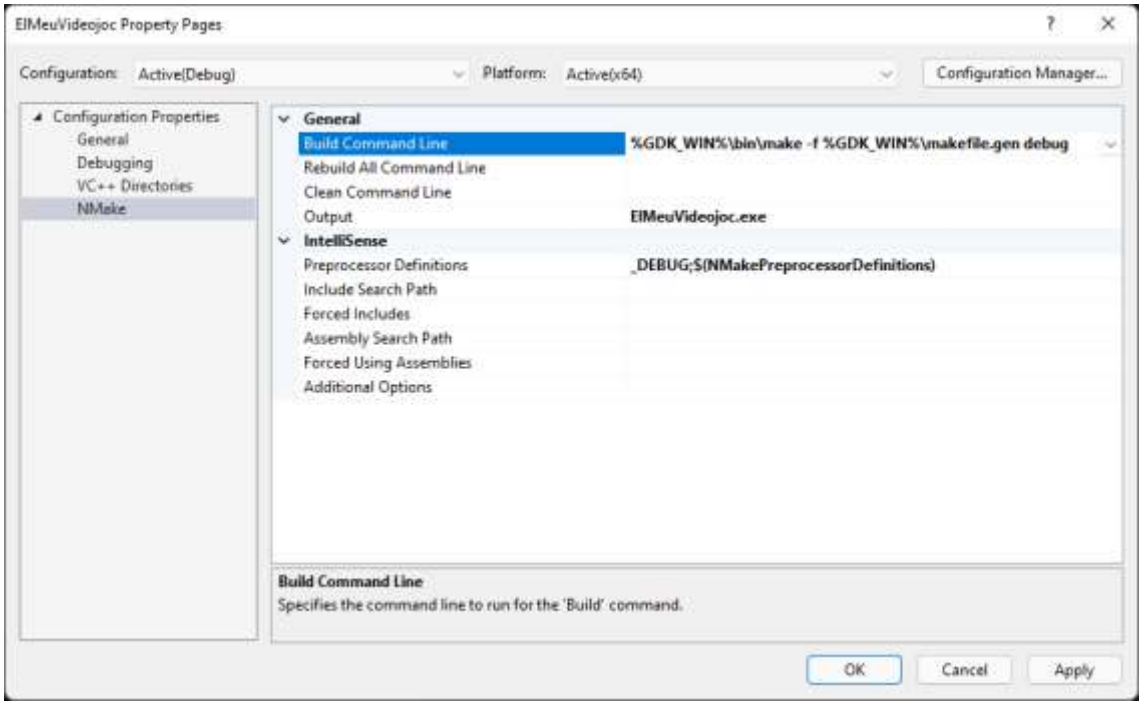


Figura A.18: Comandament de compilació per a depurar

A.5.2. Activar les funcions de desenvolupament de l'emulador

Tal com he dit anteriorment, GensKMod no és dels emuladors més fidels a la consola, però sí que és dels millors per a depurar el seu funcionament intern. Per activar aquest mode i així fer que aquest imprimeix-hi els missatges generats per SGDK, hem d'anar a la barra d'opcions i clicar a on posa "*Debug*". Se'ns obrirà una finestra i allà haurem d'activar la casella "*Active Development Features*".

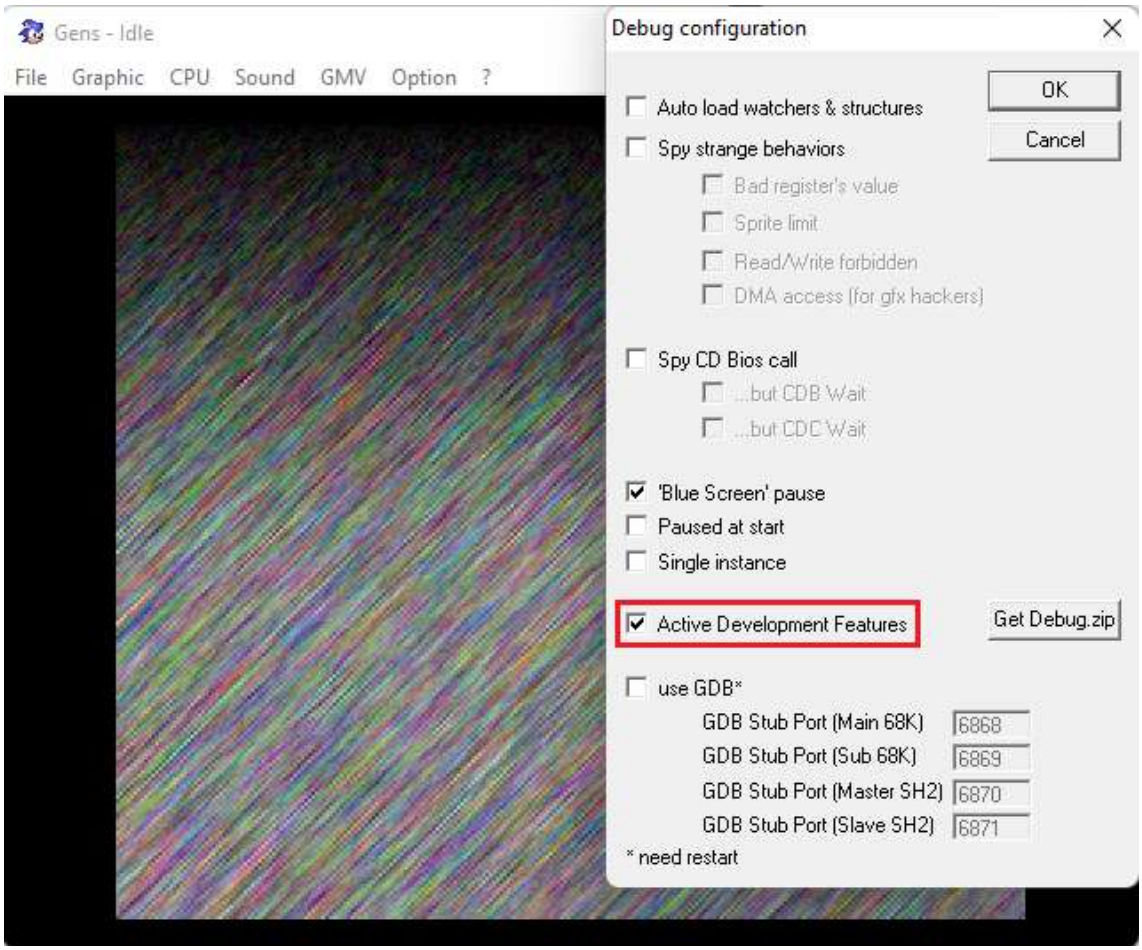


Figura A.19: Finestra activar depuració en l'emulador

A.5.3. Depuració simple

El videojoc el podem depurar de diverses maneres, la més simple és decidir quins elements imprimir en l'emulador. Per això, podem modificar l'exemple "Hola, Món!" per afegir-li aquesta funcionalitat.

```
#include <genesis.h>

int main(){
        VDP_drawText("Hola, Mon!", 8, 5);

        u16 i;
        while (TRUE) {
                KLog_U1("Hola, Mon! nº", i++);
                SYS_doVBlankProcess();
        }

        return 0;
}
```

Figura A.20: Exemple codi depuració

D'aquesta manera si seleccionem dins de l'emulador "CPU/Debug/Messages", hauríem de veure aquest resultat:

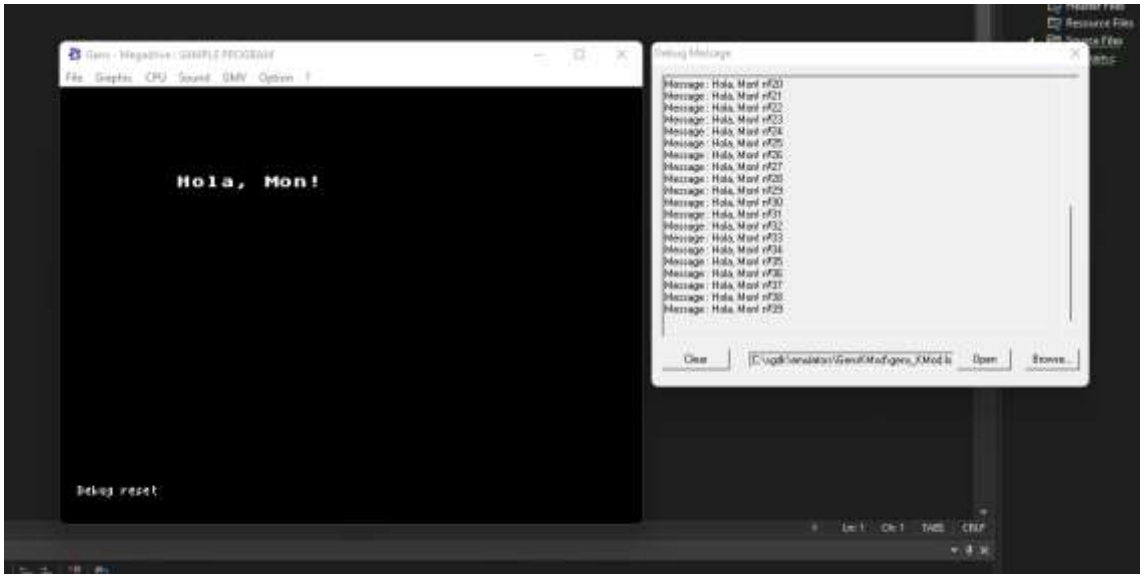


Figura A.21: Exemple depuració simple

El que fa el "KLog_U1" és imprimir un text juntament amb el nombre que se li escrigui com a segon paràmetre. Per a utilitzar això no fa falta compilar el videojoc com a *debug*.

A.5.4. Depuració avançada

Per altra banda, tenim la depuració avançada, en aquesta sí que s'ha de compilar amb el *debug* activat. En aquest cas, SGDK s'encarregarà d'imprimir absolutament tot el que falli durant la seva execució.

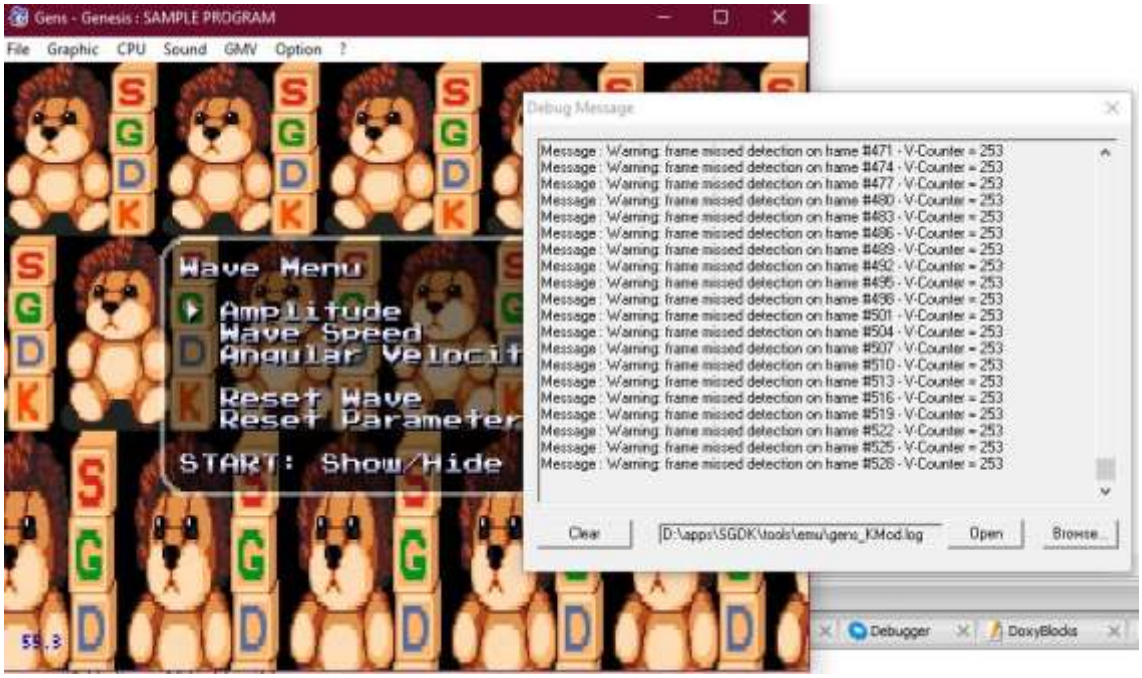


Figura A.22: Exemple depuració avançada

B. Programa per a agilitzar el procés de creació

Com ja s'ha pogut veure, és molt tediós haver d'obrir l'emulador, buscar l'arxiu del programa i executar-lo. Per això mateix, he escrit una molt senzilla aplicació que s'executi cada cop que es compili el programa per tal d'obrir l'emulador amb el videojoc sense haver de fer res.

B.1. Aplicacions en C#

El programa està escrit en C#, per tant, per a escriure'l s'ha d'instal·lar un nou component per tal de poder editar codi en aquest llenguatge. El component s'anomena "*.NET desktop development*", un cop instal·lat, s'ha de crear un nou projecte de "*Console App*", tal com es pot veure, és possible que hi hagi múltiples projectes amb aquest nom, ens hem d'assegurar que a sota hi posi que és de llenguatge C#.

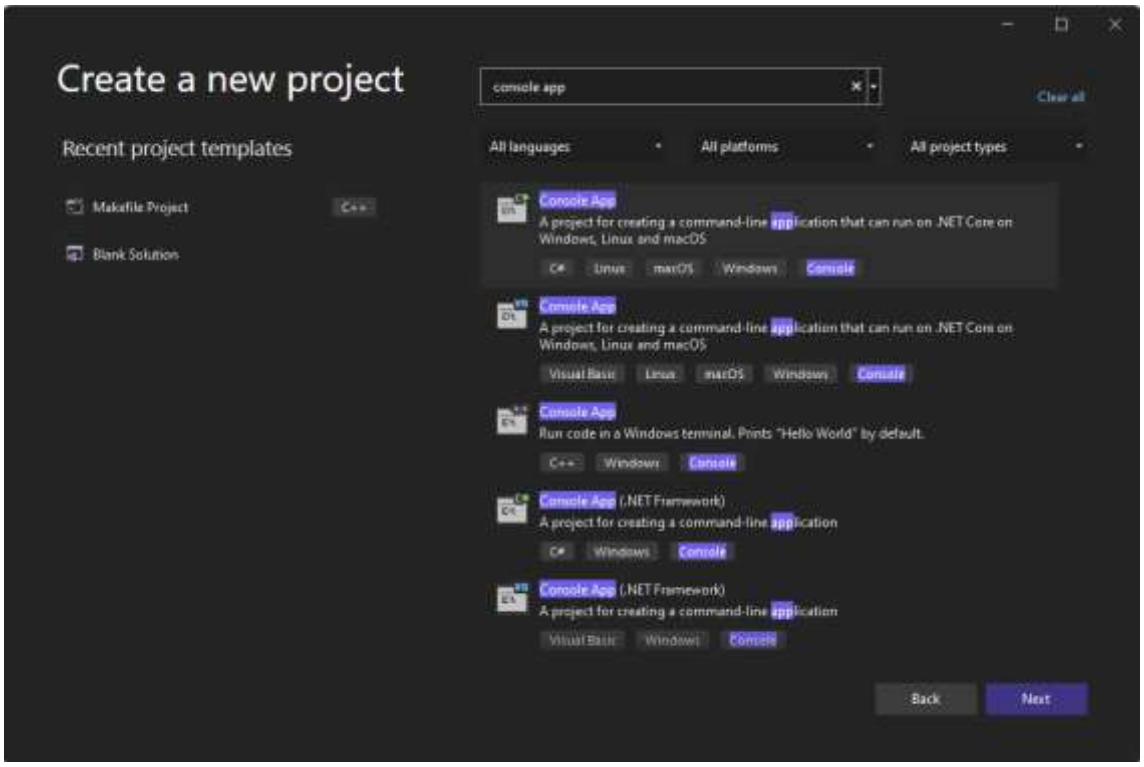


Figura B.1: Finestra de creació d'un projecte de consola Visual Studio 2022

En fer això ens apareixerà aquesta finestra, aquí hem d'escriure el nom del projecte i seleccionar a la casella per a crear la solució en la mateixa carpeta.

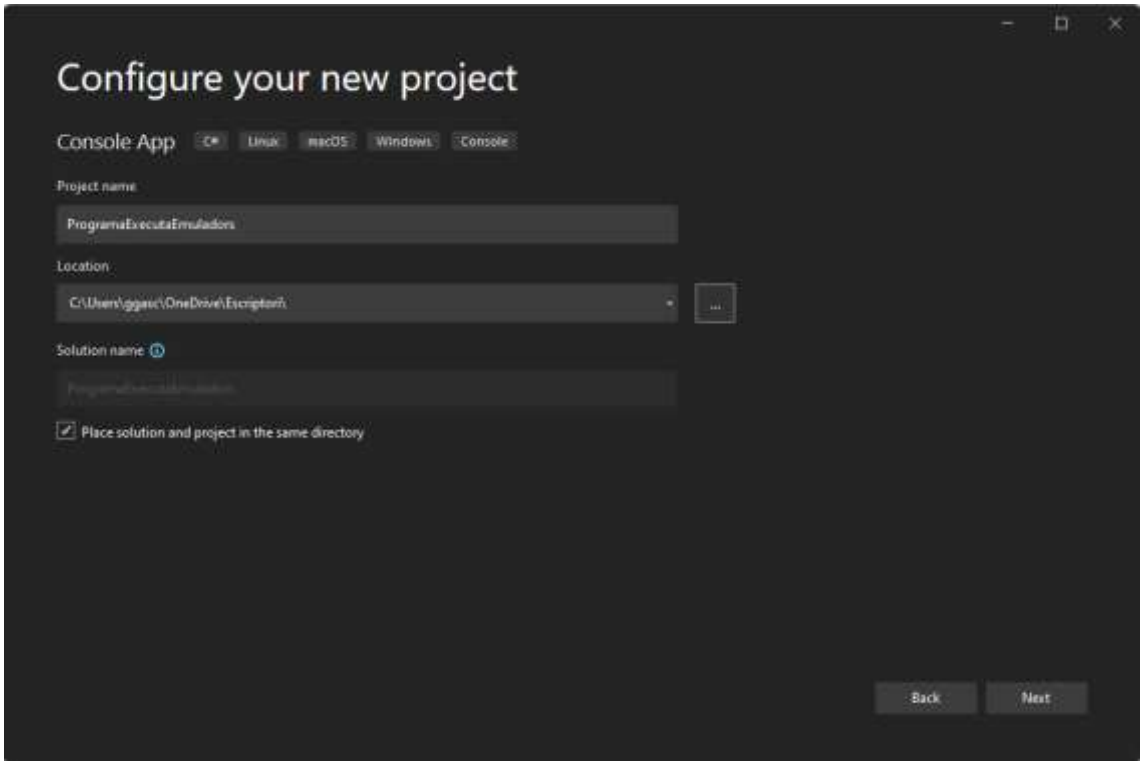


Figura B.2: Finestra configuració d'un projecte de consola Visual Studio 2022

Posteriorment ens enviarà en aquest menú, l'elecció que fem en aquest menú importa poc perquè més endavant la canviaré per una altra que no hi apareix.

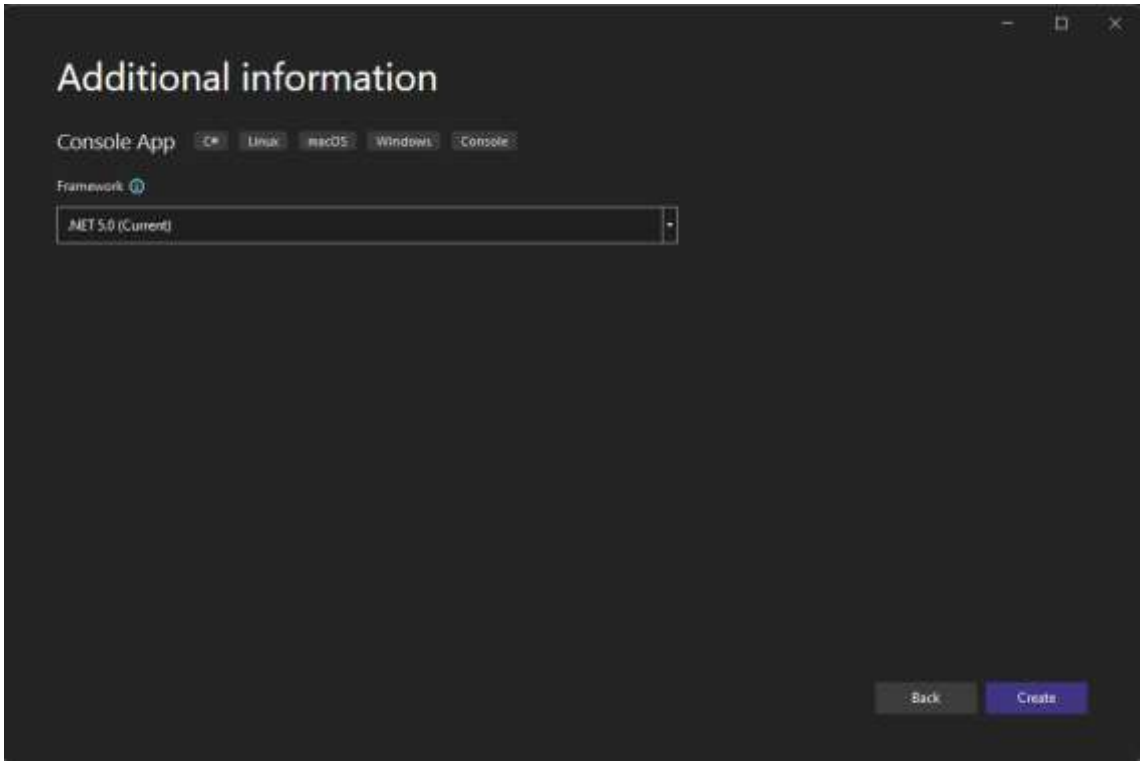


Figura B.3: Finestra selecció del framework del projecte Visual Studio 2022

Un cop creat ens apareixerà un fitxer de codi amb un programa "Hola, Món!", servint així com a exemple del que he mencionat en l'apartat 3 que tot programa s'inicia saludant al món.

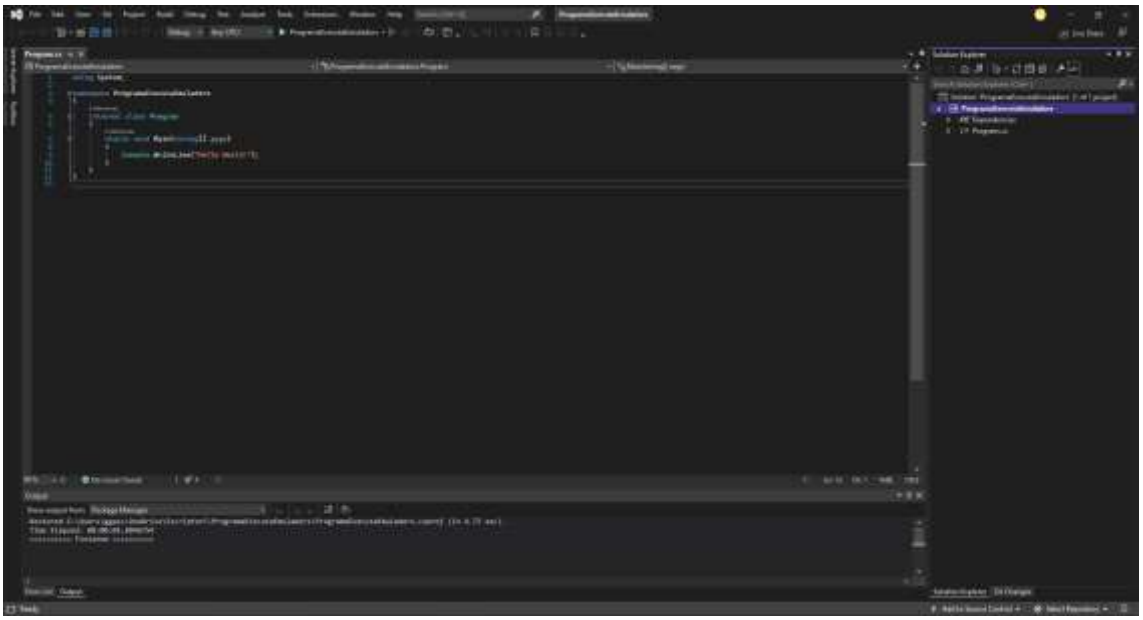


Figura B.4: Editor de codi d'un projecte de consola Visual Studio 2022

Si sense modificar-lo cliquem al botó superior que porta el mateix nom del projecte, se'ns obrirà una finestra amb el missatge "*Hello World!*".

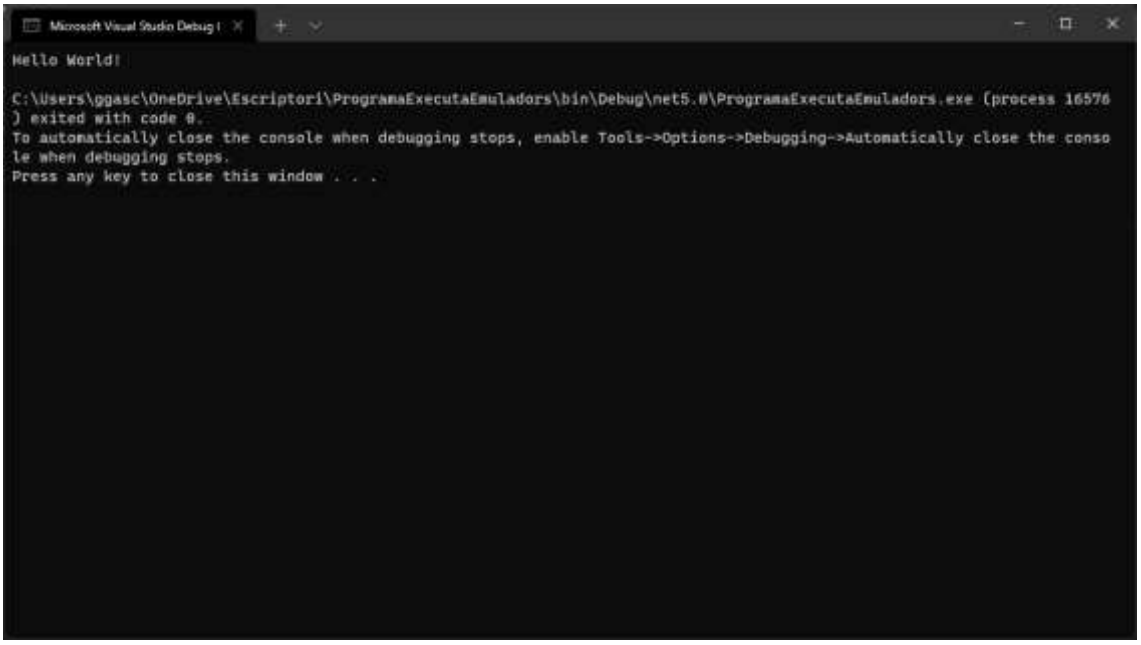


Figura B.5: Programa per defecte del projecte de consola

B.2. Aplicació que executa emuladors

Per a crear aquesta aplicació el primer que faré serà canviar la versió del .NET. Per això, primer ens dirigim a la pàgina de descarregar la seva versió 4.8. Des d'allà ens descarregarem el paquet de desenvolupadors i posteriorment l'instal·larem.

Figura B.6: Web d'instal·lació del framework .NET 4.8

Un cop fet això, ja podem accedir al Visual Studio de nou i obrir el panell de propietats del projecte, en aquest cas hauria de ser com el següent:

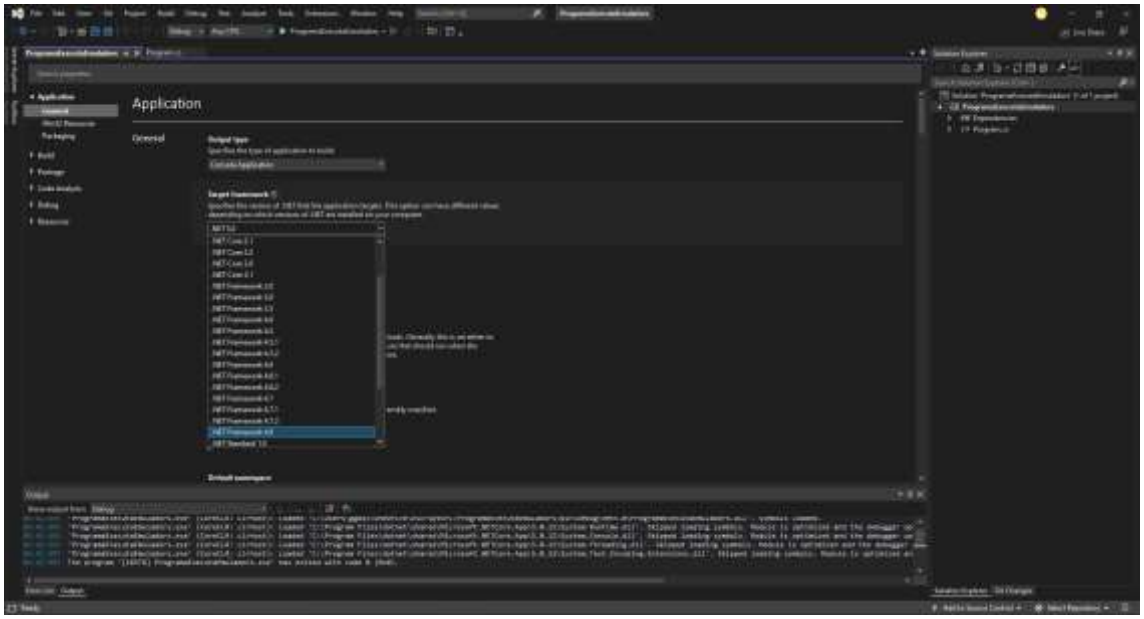


Figura B.7: Pestanya de configuració del framework de Visual Studio 2022

Un cop allà, seleccionem la versió 4.8 i guardem, això ens serveix per a comprimir la mida de l'executable, fent així que només necessiti un sol arxiu per a obrir-lo.

```
using System.Diagnostics;
using System;
using System.IO;

namespace ProgramaExecutaEmuladors {
    internal class Program {
        static void Main() {
            string appliactionPath =
                Path.GetDirectoryName(AppContext.BaseDirectory);
            string[] lines =
                File.ReadAllLines(appliactionPath +
                @"\emulatorPath.txt");

            DirectoryInfo di = new DirectoryInfo(appliactionPath);
            appliactionPath =
                di.Parent.FullName + @"\out\rom.bin";
            Process.Start(@lines[0], appliactionPath);
        }
    }
}
```

Figura B.8: Codi del programa executa emuladors

Aquest codi és tot el programa sencer, el seu funcionament és el següent:

- Primer crea una variable que contingui la posició en la qual es troba l'executable.
- Seguidament busca un fitxer del bloc de notes que contingui en la seva primera línia la direcció de l'emulador en la seva primera línia.
- Just després crea una variable que emmagatzemi la informació de la compilació del videojoc.
- Finalment, el programa executa l'emulador juntament amb el videojoc.

Però hi ha un minúscul problema, el qual és que el programa no es tanca automàticament a l'acabar la seva execució, per a arreglar això hem de seguir les instruccions que ens diu l'executable.

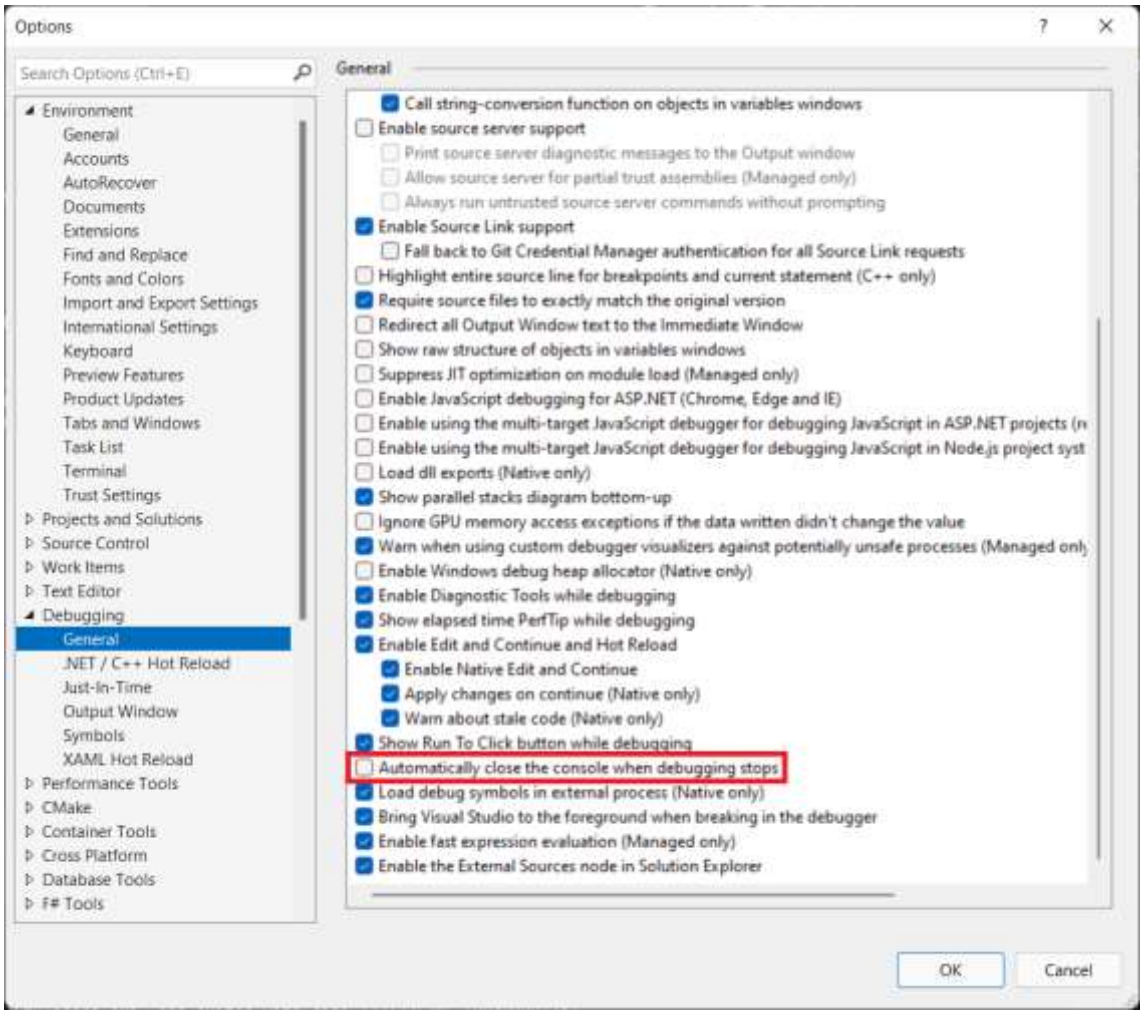


Figura B.9: Casella per a tancar automàticament les compilacions

En clicar aquesta casella, el missatge deixarà d'aparèixer i l'aplicació es tancarà automàticament. Finalment, la seva posició en la carpeta del projecte serà la següent, una subcarpeta a part a l'alçada de l'*out* que contingui en ella l'arxiu de text amb les direccions i l'executable.

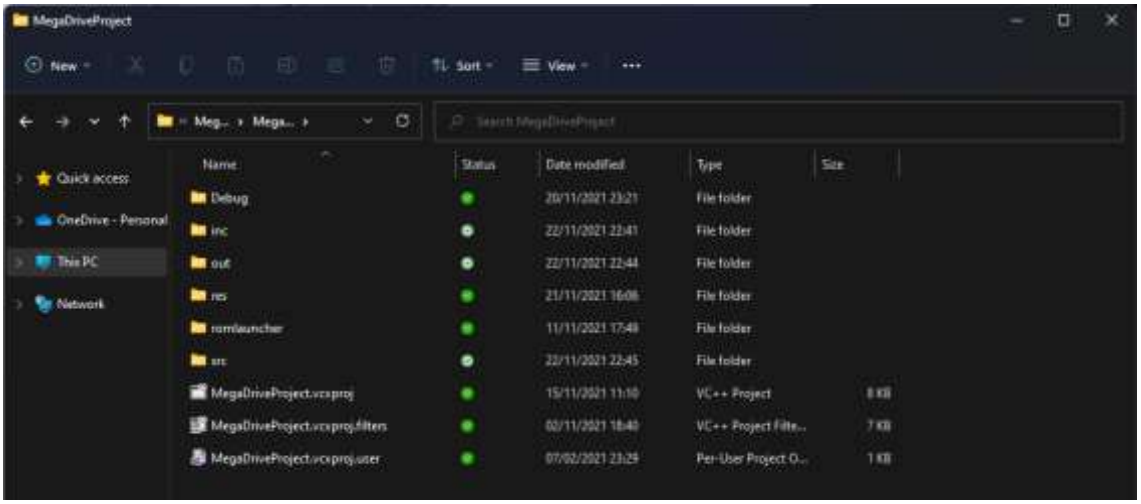


Figura B.10: Estructura carpetes d'un projecte amb SGDK

B.3. Implementació amb Visual Studio

Un cop tenim el programa col·locat en la seva respectiva carpeta, ja podem implementar-lo dins de l'editor de codi i així fer que aquest s'executi automàticament. Per a fer això, ens hem de dirigir al menú de propietats del projecte.

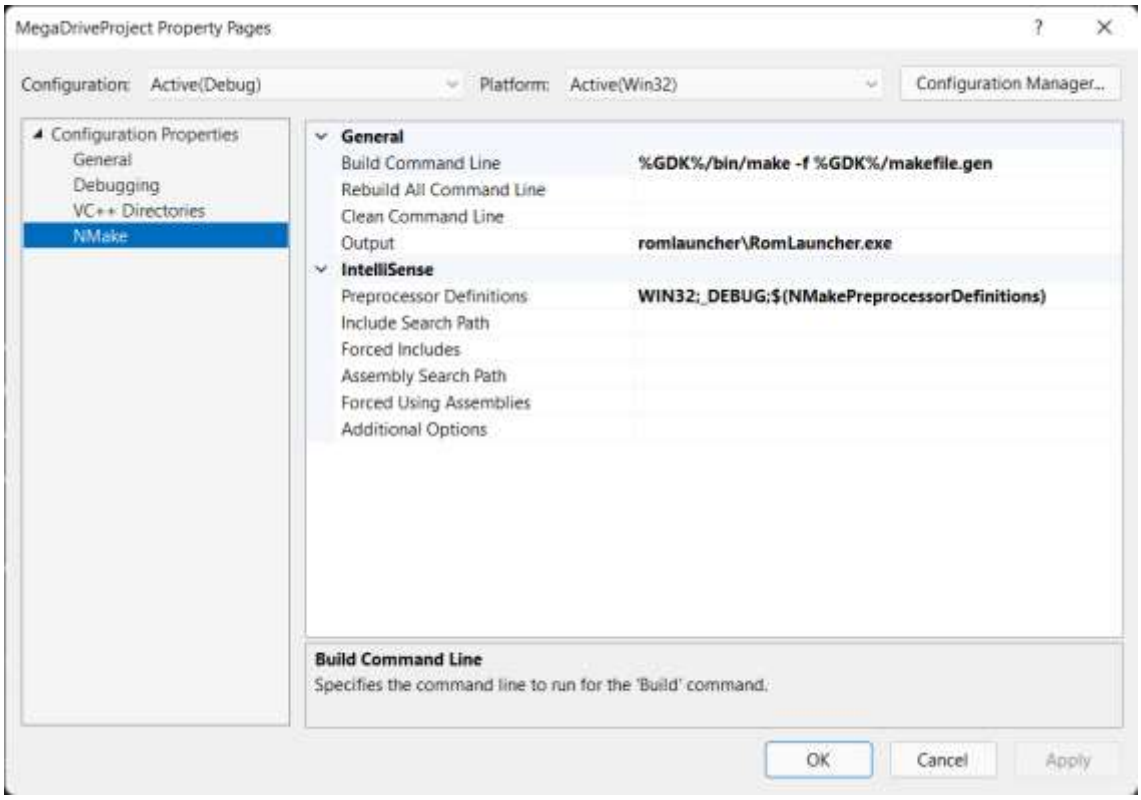


Figura B.11: Pestanya NMake del menú de propietats

Un cop s'obri la finestra de propietats, ens dirigim a la pestanya de *NMake*, en aquesta hi haurà l'opció *Output* amb el nom del projecte. El que farem serà canviar aquesta direcció per la direcció en la qual es troba l'executable, en aquest cas seria la següent:

```
romlauncher\RomLauncher.exe
```

C. Creació de gràfics per a la Mega Drive

C.1. Programes d'edició d'imatges compatibles

Generalment, qualsevol programa hauria de ser compatible, però en alguns casos es necessitaria un altre per a fer que els colors siguin detectables pel compilador. Per això mateix, posaré un parell d'exemples.

C.1.1. Aseprite

Aseprite és un programa centrat exclusivament en edició d'imatges píxel art, tot i així, és necessari seguir una sèrie de passos perquè quan s'exporti el resultat, aquest sigui compatible amb la consola.

Figura C.1: Logotip Aseprite

Primerament, en crear un nou projecte, hem de seleccionar una dimensió d'imatge que sigui múltiple de 8, això és a causa de la mida dels *tiles*. Seguidament, s'ha de seleccionar el mode de colors, en aquest cas hem de seleccionar *indexed*, això és perquè així nosaltres tenim el control total de l'índex dels colors i li donem la informació al compilador sobre quins estem utilitzant.

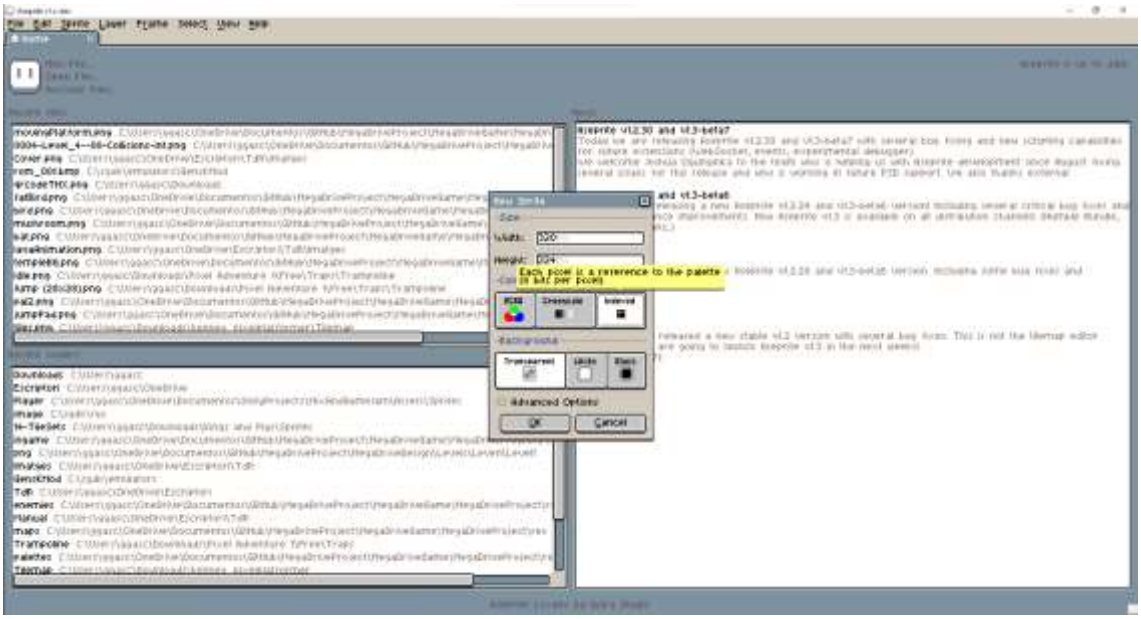


Figura C.2: Pestanya creació d'un sprite nou

Finalment, un cop creada la imatge, només caldrà assegurar-nos que la paleta de l'esquerra no supera els 16 colors, un cop hàgim completat la nostra obra d'art, ja la podrem exportar com a *.png* o *.bmp*, els dos formats compatibles, és indiferent quin es tria, tots dos pesaran el mateix un cop compilats.

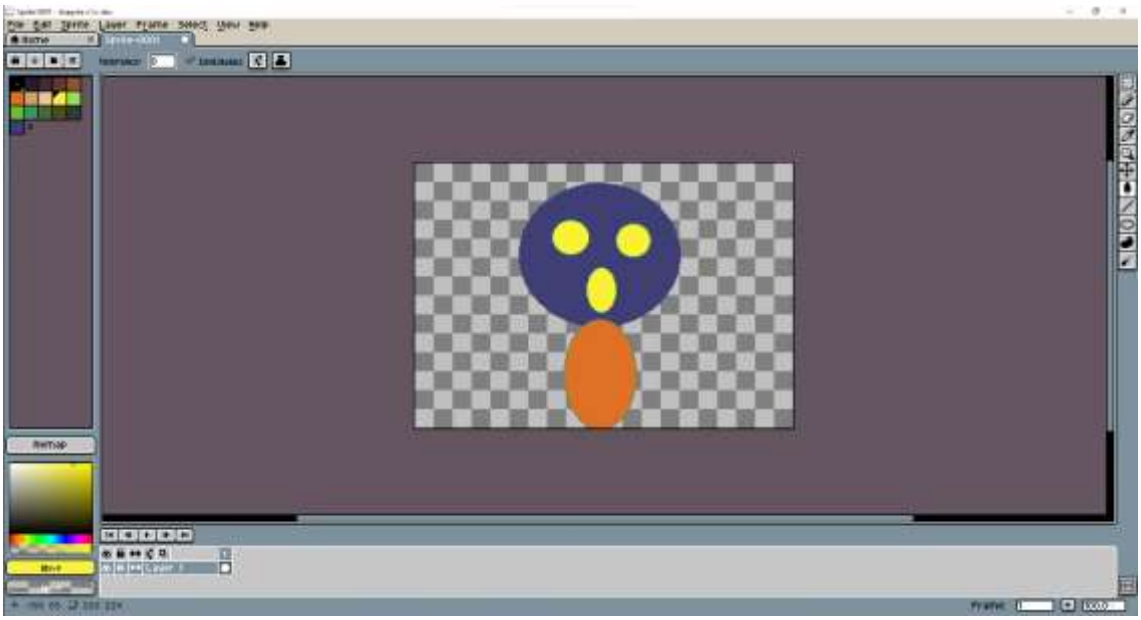


Figura C.3: Exemple il·lustració dins d'Aseprite

C.1.2. Photoshop

Photoshop és un programa àmpliament utilitzat en el món de l'edició d'imatges. Aquest també et permet crear imatges a molt baixa resolució, però no et dona el mateix control que Aseprite.

Per altra banda, ens proporciona molta més facilitat a l'hora de crear una làmina, ja que no hem de configurar-li el mode de color amb els problemes que ens pot ocasionar com en l'altre cas.

Figura C.4: Logotip Photoshop

A l'hora de crear una imatge, aquí simplement ens haurem de centrar a fer que aquesta sigui múltiple de 8 tant en amplada com alçada. En aquest cas, no ens interessa cap de les altres opcions.

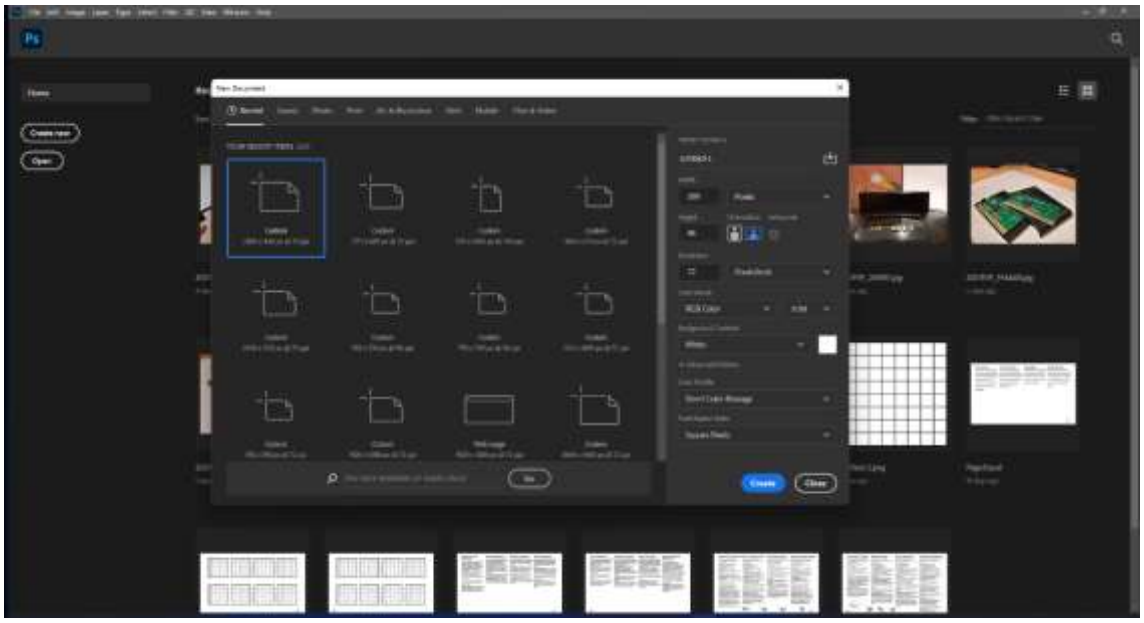


Figura C.5: Pestanya creació d'una imatge nova Photoshop

Posteriorment, podem iniciar la nostra creació. Això si, aquí no tenim cap control sobre el nombre de colors de la imatge com el d'Aseprite, per tant, serà més complicat controlar que no ens passem dels 16.

Un cop finalitzat, per a exportar haurem de configurar amb l'opció de mida més petita, aquesta ens redueix la paleta al nombre més petit possible. Per tant, si tenim 16 o menys, aquesta serà de la dimensió corresponent i així SGDK la podrà compilar.

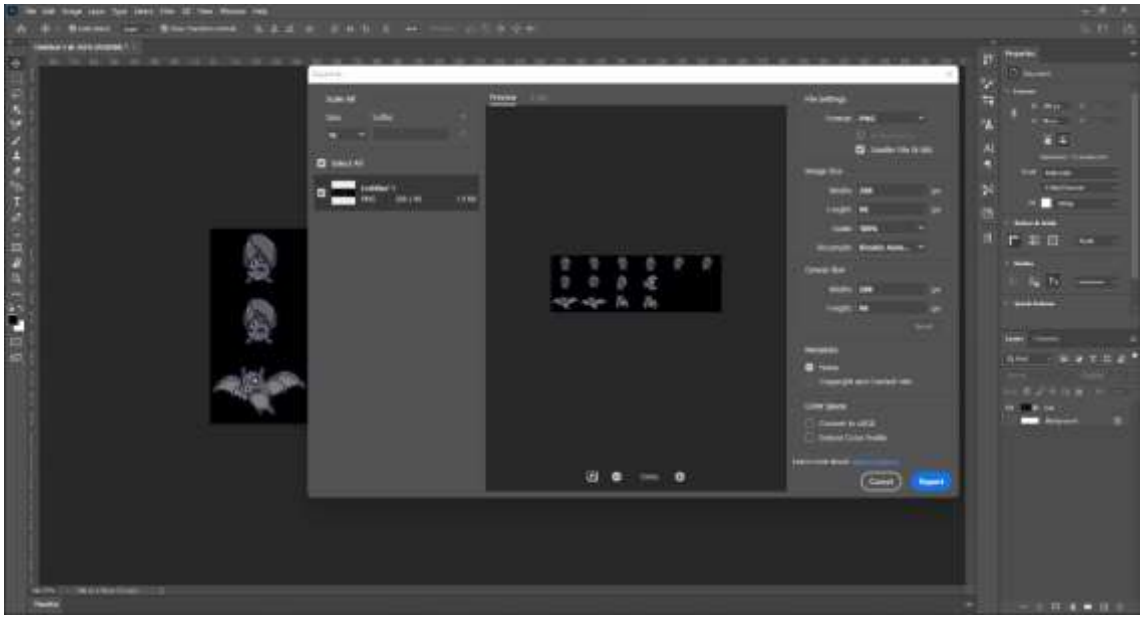


Figura C.6: Pestanya d'exportar una imatge Photoshop

C.2. Importar imatges a SGDK

Per a importar imatges, SGDK utilitza una eina pròpia que s'anomena rescomp. Aquesta llegeix un arxiu de text en format .res i a partir d'ell, crea la imatge.

Aquest fitxer es troba en la carpeta res que et crea SGDK amb la seva primera compilació. Aquest arxiu ha de seguir una estructura específica que s'indica en un fitxer del repositori de GitHub.

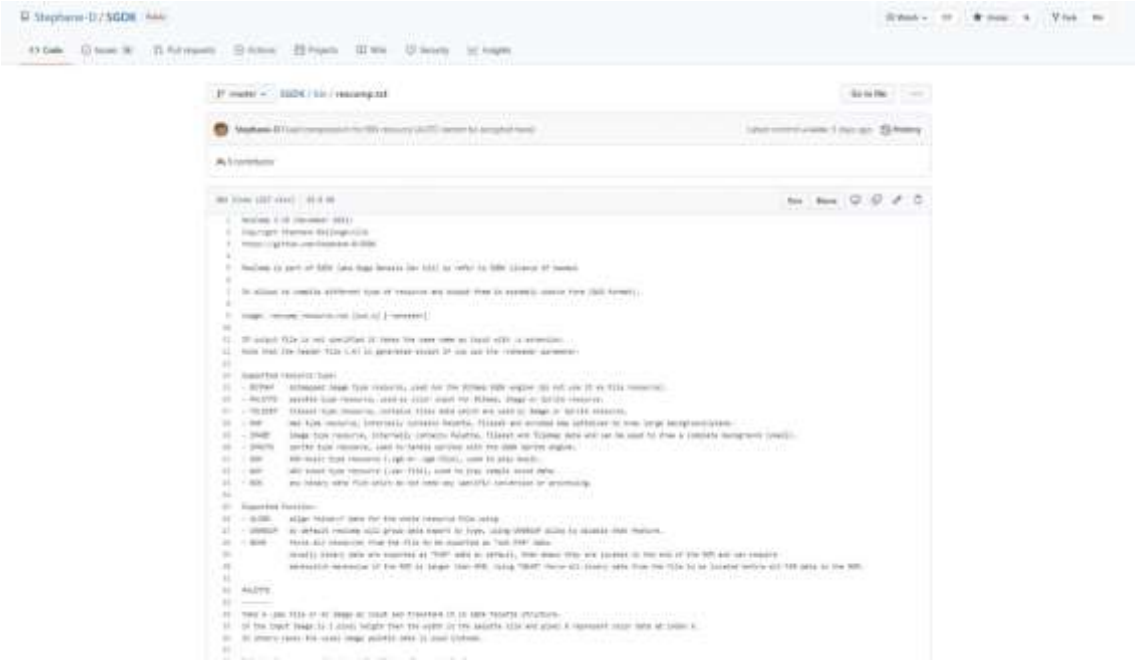


Figura C.7: Pàgina de GitHub del rescomp

Si seguim aquesta estructura, ens hauria de quedar alguna cosa semblant a aquesta en el nostre arxiu:

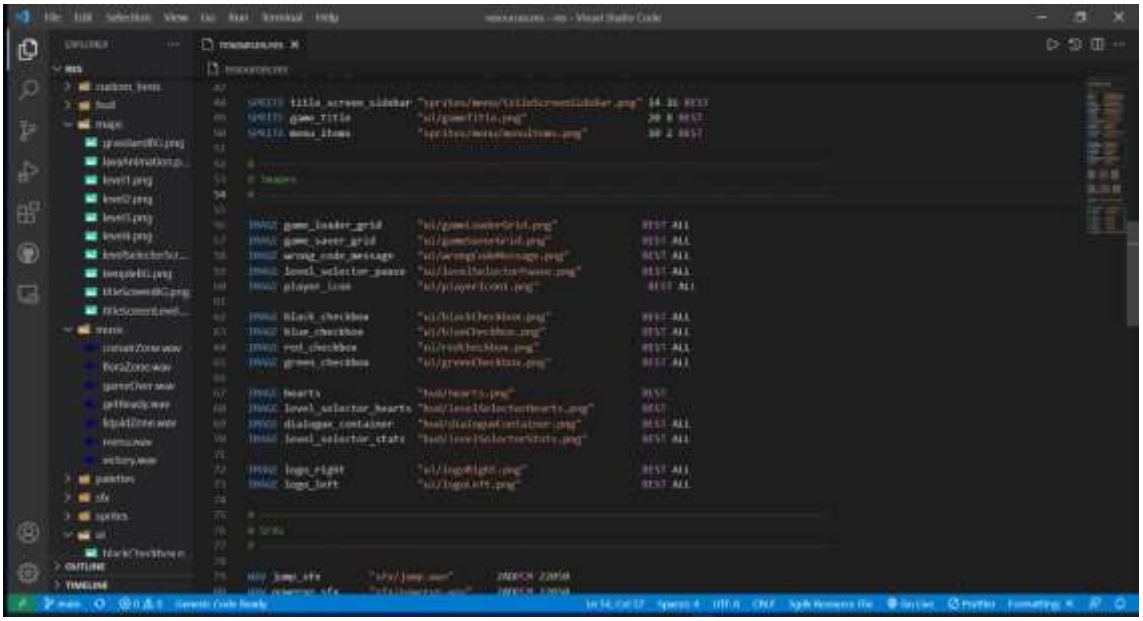


Figura C.8: Exemple arxiu de recursos

Un cop fet això, quan tornem al Visual Studio, en l'arxiu on vulguem importar aquestes imatges, li escriurem la següent línia a dalt de tot:

```
#include <resources.h>
```

Aquesta línia ens serveix per a fer-li saber al SGDK que volem compilar aquell fitxer, inicialment quan ho fem ens apareixerà un error assegurant que aquell arxiu no existeix, però un cop compilat ja podrem canviar aquesta línia per aquesta altra:

```
#include "../res/resources.h"
```

Això farà que Visual Studio ens detecti el fitxer i per tant, l'autocompletat inclourà les imatges noves.

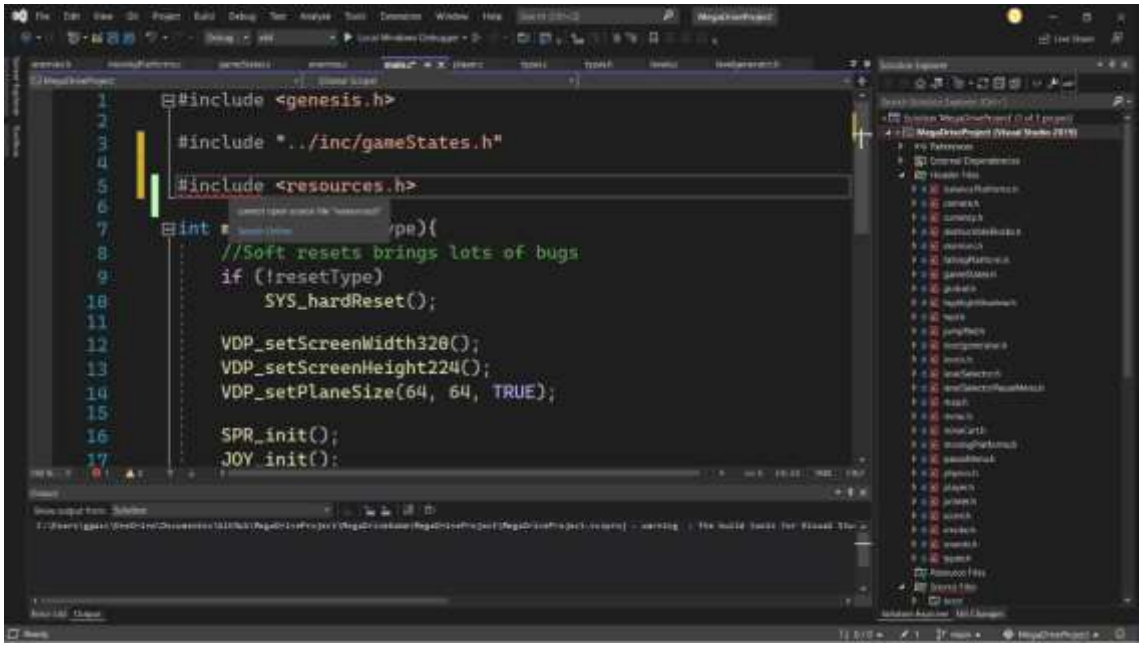


Figura C.9: Exemple incloure recursos

D. Canviar l'encapçalament de SGDK

En l'apartat 3 he parlat sobre com crear un programa "Hola, Món!" en assemblador 68k i SGDK. En assemblador haig de crear l'encapçalament manualment, però en SGDK, tal com he dit, aquest es genera automàticament. Per tant, el resultat per defecte és aquest:

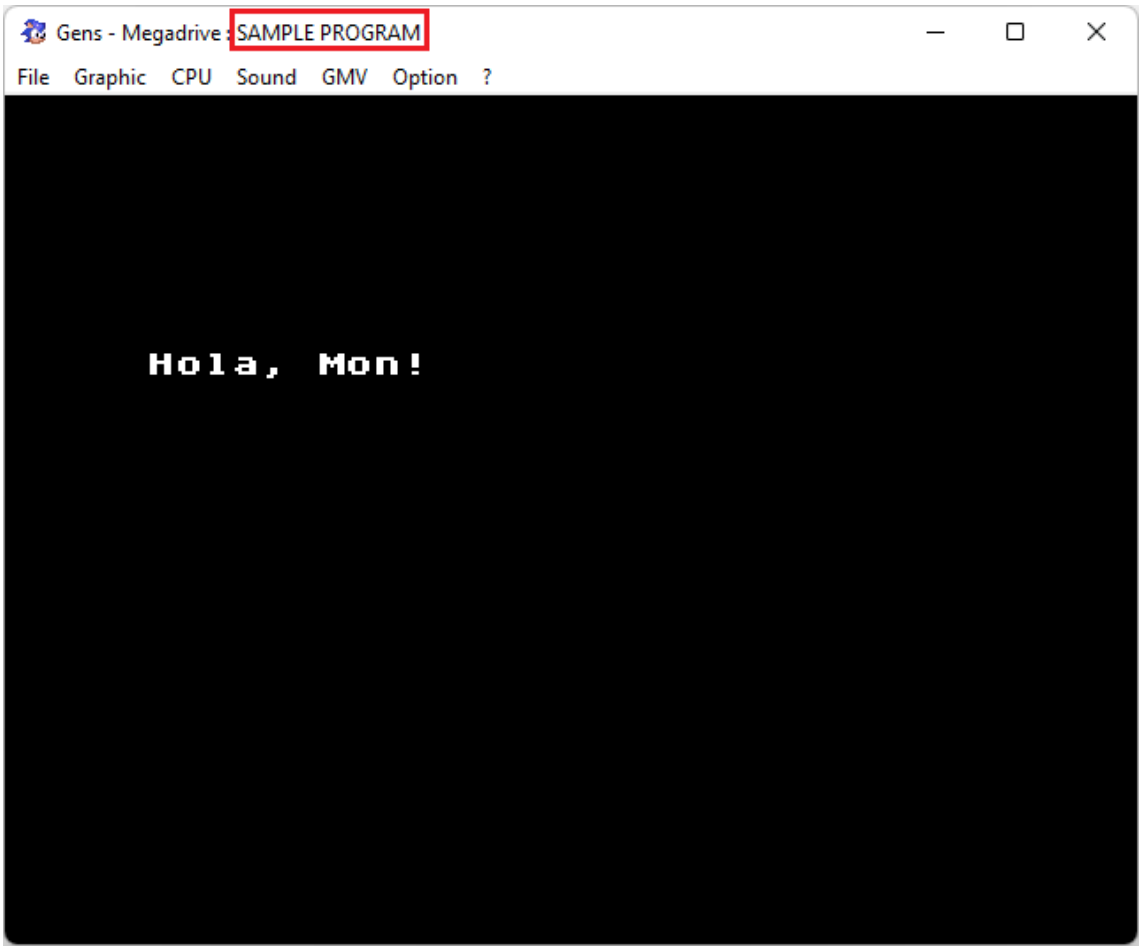


Figura D.1: Nom de la ROM per defecte

Per a canviar-lo, ens hem de dirigir a la carpeta *src* que ens haurà creat SGDK amb la seva primera compilació, en ella ens hi trobarem una altra anomenada *boot* i en l'interior hi haurà dos arxius, el que ens interessa és el que s'anomena *rom_head.c*. Si l'obrim ens trobarem amb el següent fragment de codi:

```
#include "genesis.h"

__attribute__((externally_visible))
const ROMHeader rom_header = {
#if (ENABLE_BANK_SWITCH != 0)
    "SEGA SSF        ",
#else
    "SEGA MEGA DRIVE ",
#endif
    "(C)SGDK 2021    ",
    "SAMPLE PROGRAM                                  ",
    "SAMPLE PROGRAM                                  ",
    "GM 00000000-00",
    0x000,
    "JD              ",
    0x00000000,
#if (ENABLE_BANK_SWITCH != 0)
    0x003FFFFF,
#else
    0x000FFFFF,
#endif
    0xE0FF0000,
    0xE0FFFFFF,
    "RA",
    0xF820,
    0x00200000,
    0x0020FFFF,
    "            ",
    "DEMONSTRATION PROGRAM                  ",
    "JUE             "
};
```

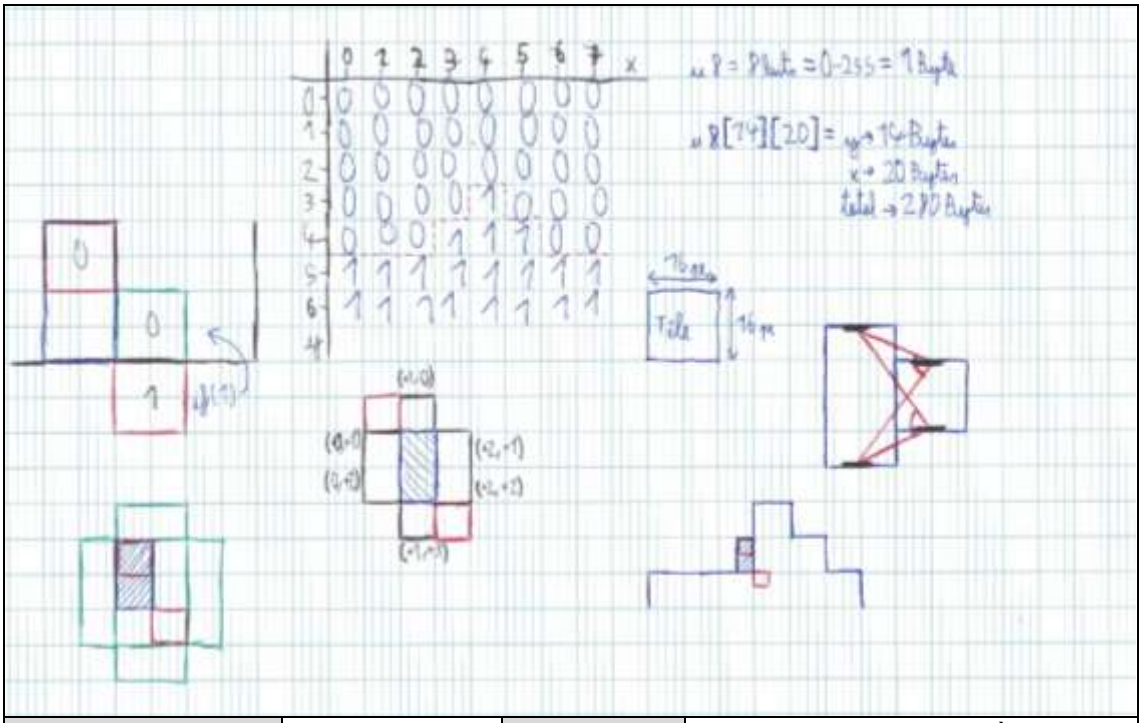
Figura D.2: Codi per defecte de l'encapçalament

Aquest ens proporciona les mateixes dades i en el mateix ordre que en l'assemblador. Les normes aquí segueixen sent exactament les mateixes. Un cop modificat tot això, si tornem a executar el compilador, podrem jugar amb el nou identificador de la ROM.

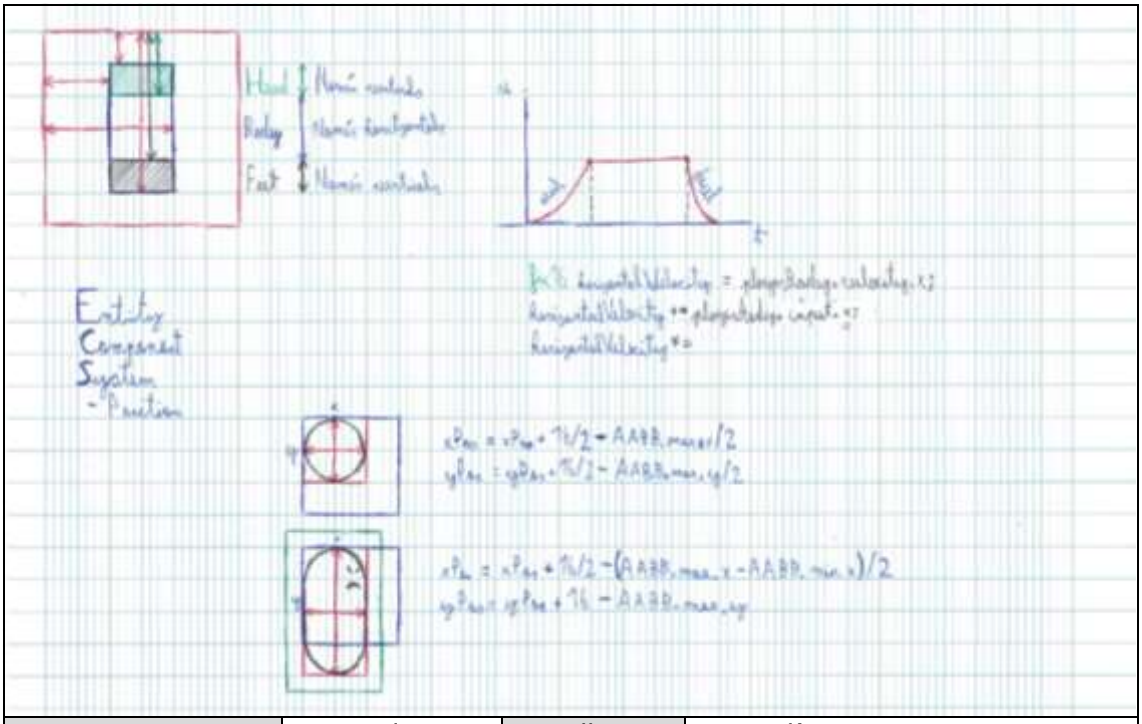
E. Esquemes de funcionament del videojoc

Durant el desenvolupament del videojoc, he anat dibuixant una sèrie d'esquemes que ensenyen visualment com encarava els problemes més complexos amb els que em trobava sobretot al principi.

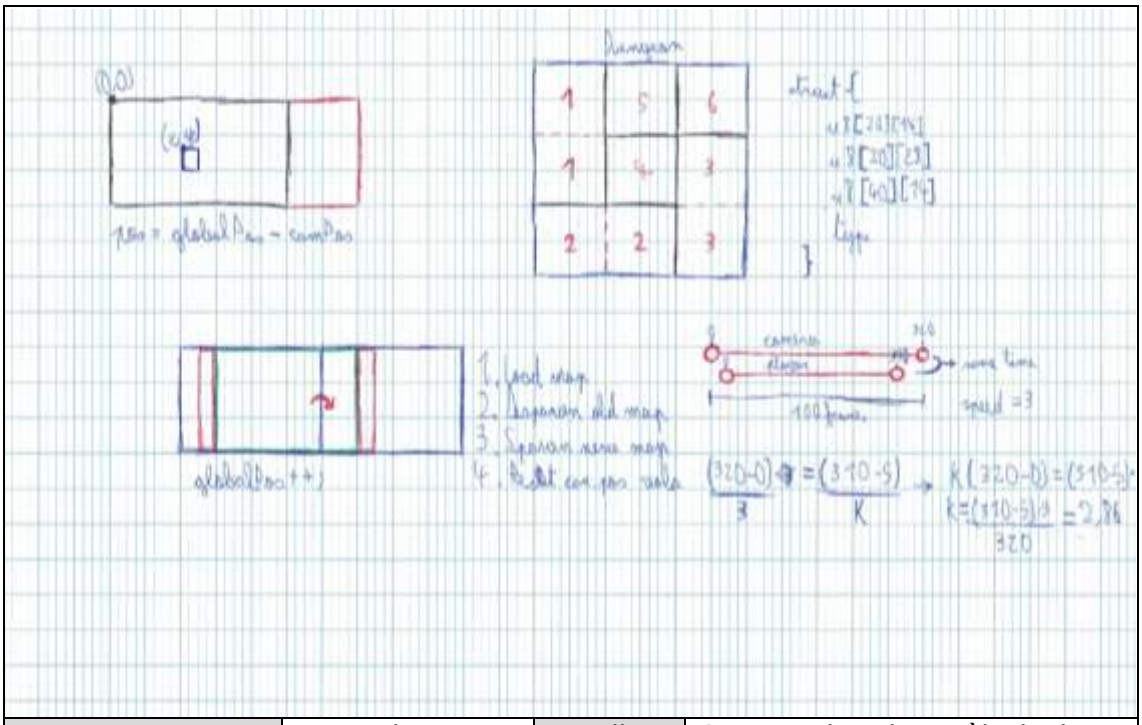
Crear un programa pot ser complicat, això comporta que tot i treballar digitalment, no hàgim de deixar de banda el paper i bolígraf.



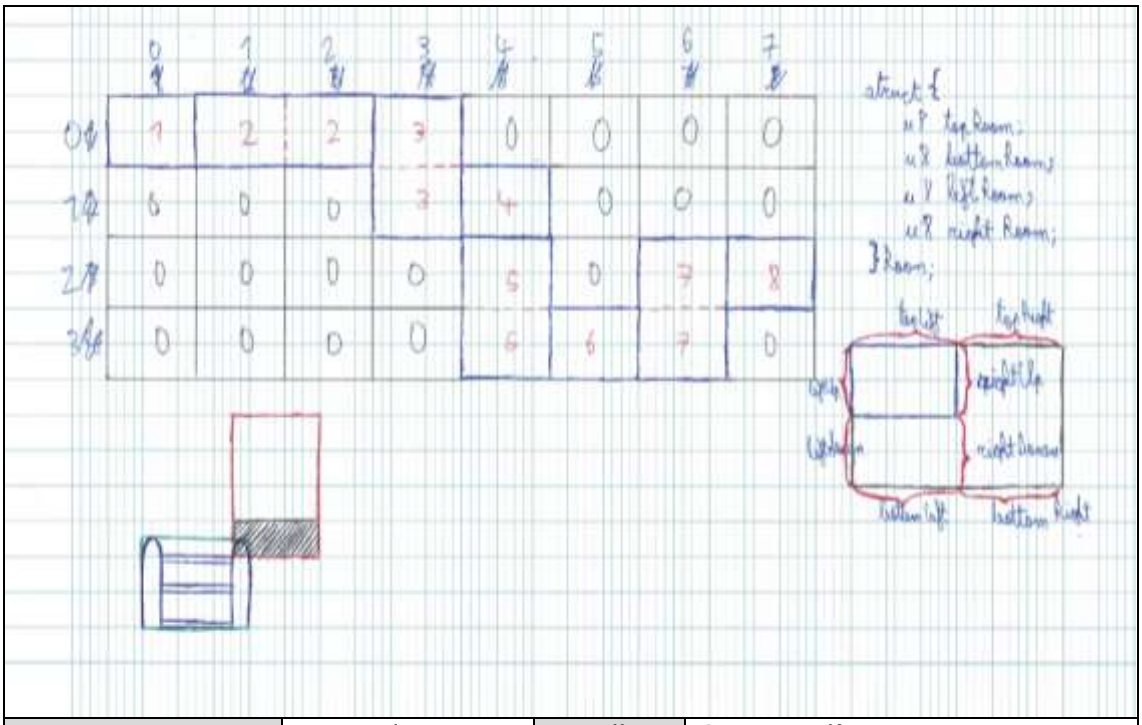
Nom	Motor de física I	Detalls	Sistema de graella i mètodes de detecció de col·lisions.
Descripció	En la part superior, es pot veure com era la meva idea inicial dels mapes de col·lisions, la qual no va canviar. Just a la dreta hi ha una sèrie de càlculs de quan ocupa cada sala. A la part inferior s'hi poden veure una sèrie d'intents dels diversos mètodes que vaig estar intentant utilitzar al principi per a resoldre les col·lisions.		



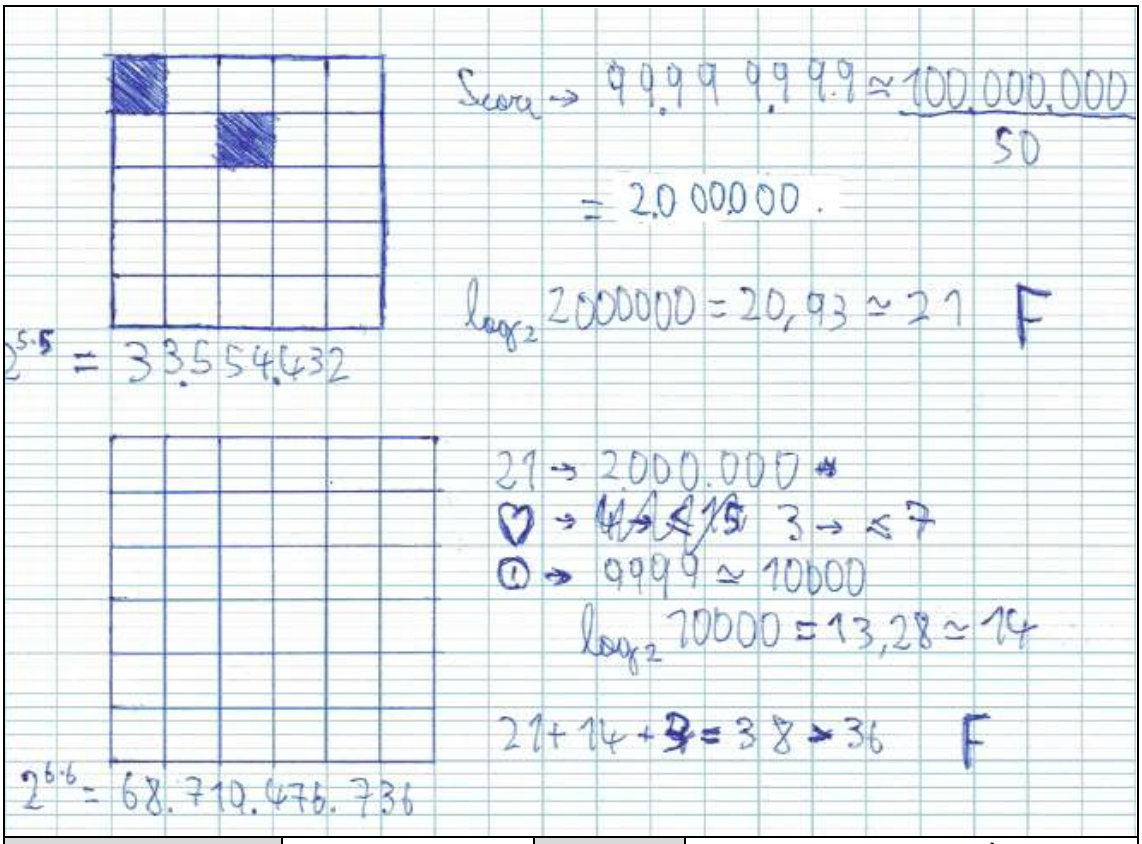
Nom	Motor de física II	Detalls	Parts d'un personatge, acceleracions i posicionament.
Descripció	En la cantonada superior esquerra s'hi pot veure l'esquema de les diferents parts que té un cos físic. Just a la dreta hi ha un esquema de com funcionen les acceleracions, amb la diferència entre el valor d'acceleració i desceleració. A la zona inferior hi ha un parell de càlculs sobre com simplificar els mètodes per a instanciar elements del nivell.		



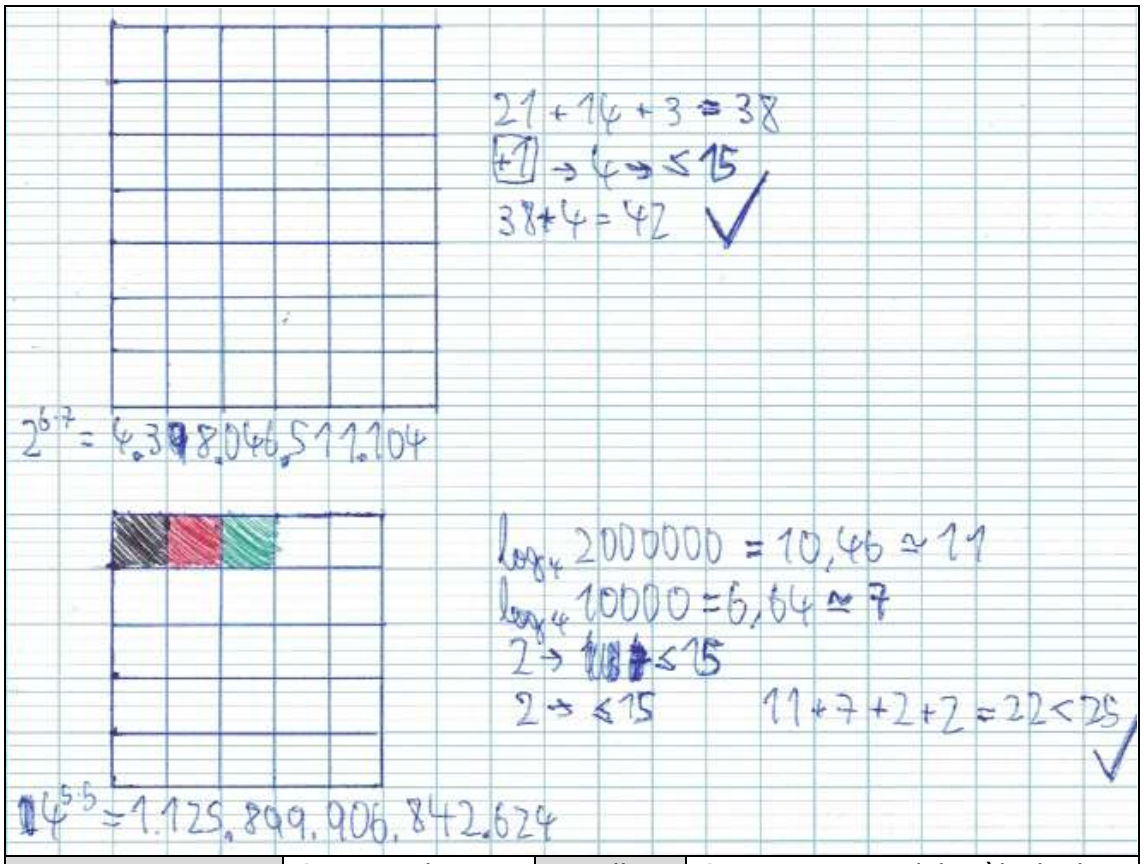
Nom	Motor de desplaçament I	Detalls	Sistema de sales i càlculs de desplaçament.
Descripció	En la cantonada superior esquerra hi ha representat el mètode que utilitzo per a posicionar el jugador respecte la càmera. A sota seu hi ha dibuixat un esquema del desplaçament entre sales al costat de l'ordre d'execució per a exercir el canvi, just a la seva dreta hi ha una regla de tres per a calcular les velocitats a les quals s'havia de moure cada pla perquè l'efecte s'exercís correctament. Finalment, en el quadrat superior hi ha representada una de les primeres idees que vaig tenir per a emmagatzemar el mapa del nivell, el qual més endavant m'assabentaria de la ineficiència que suposava.		



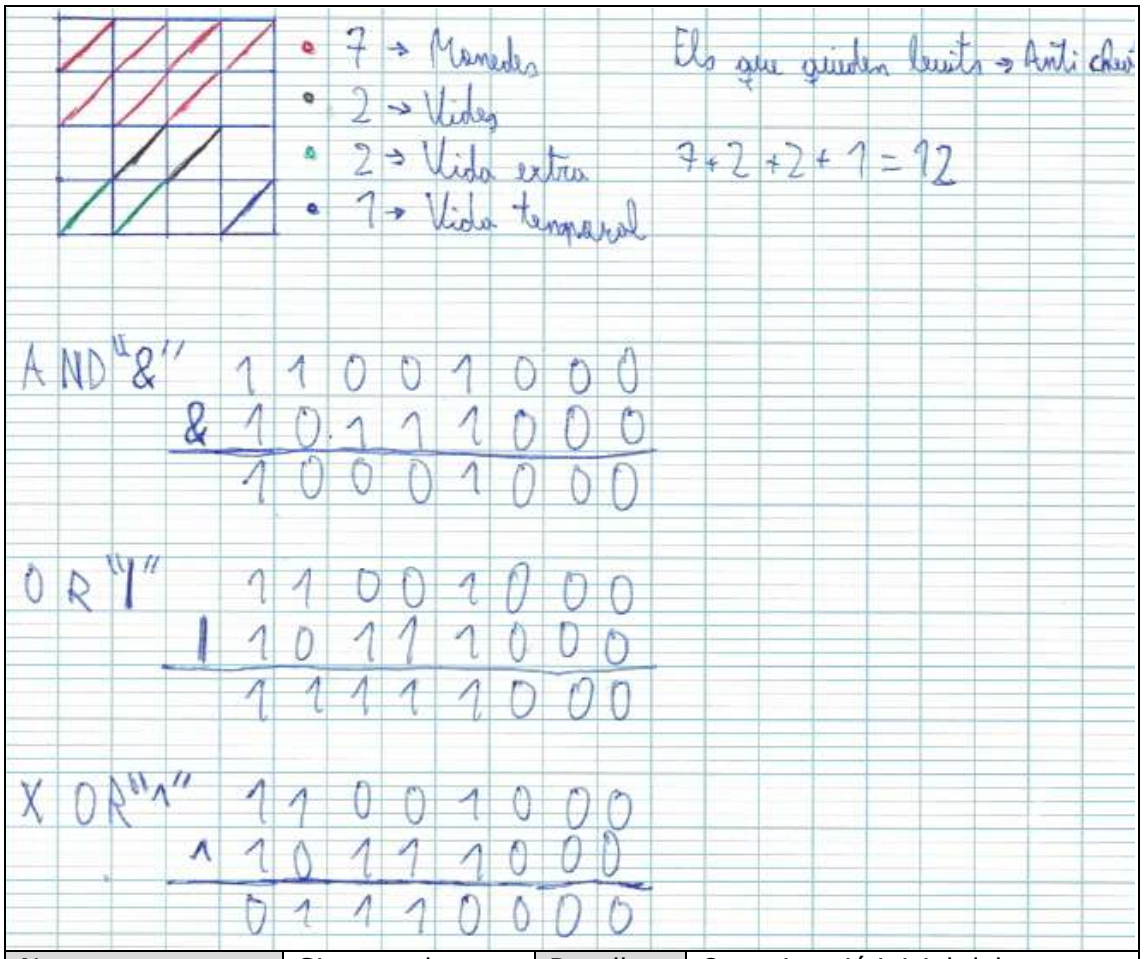
Nom	Motor de desplaçament II	Detalls	Sistema d'emmagatzematge de sales.
Descripció	Aquí es pot veure la representació visual de com emmagatzemo les sales, cadascuna ve representada per un nombre i una estructura com la de la seva dreta. Just a sota hi ha lògica que seguia la càmera per a canviar de sala segons el tipus, però això es va acabar transformant en un immens contratemps i vaig canviar els tipus de sales. A la cantonada inferior esquerra hi ha un altre afegit del motor de física, aquest és el tipus de col·lisions que li aplico a les escales.		



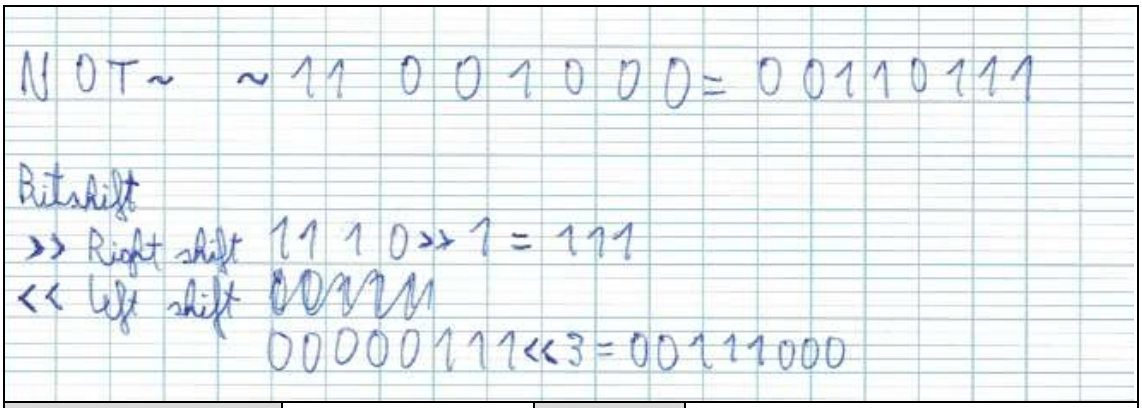
Nom	Sistema de guardat I	Detalls	Primera meitat dels càlculs de viabilitat.
Descripció	Aquí s'hi pot apreciar les primeres proves per a veure el tipus de sistema que intentava fer. Els càlculs consten d'intentar veure el nombre de caselles que necessito per a representar els nombres, en un inici anaven a estar en binari i per tant necessitava un gran nombre de caselles i les combinacions de 5x5 i 6x6, tal com es pot veure no em funcionaven.		



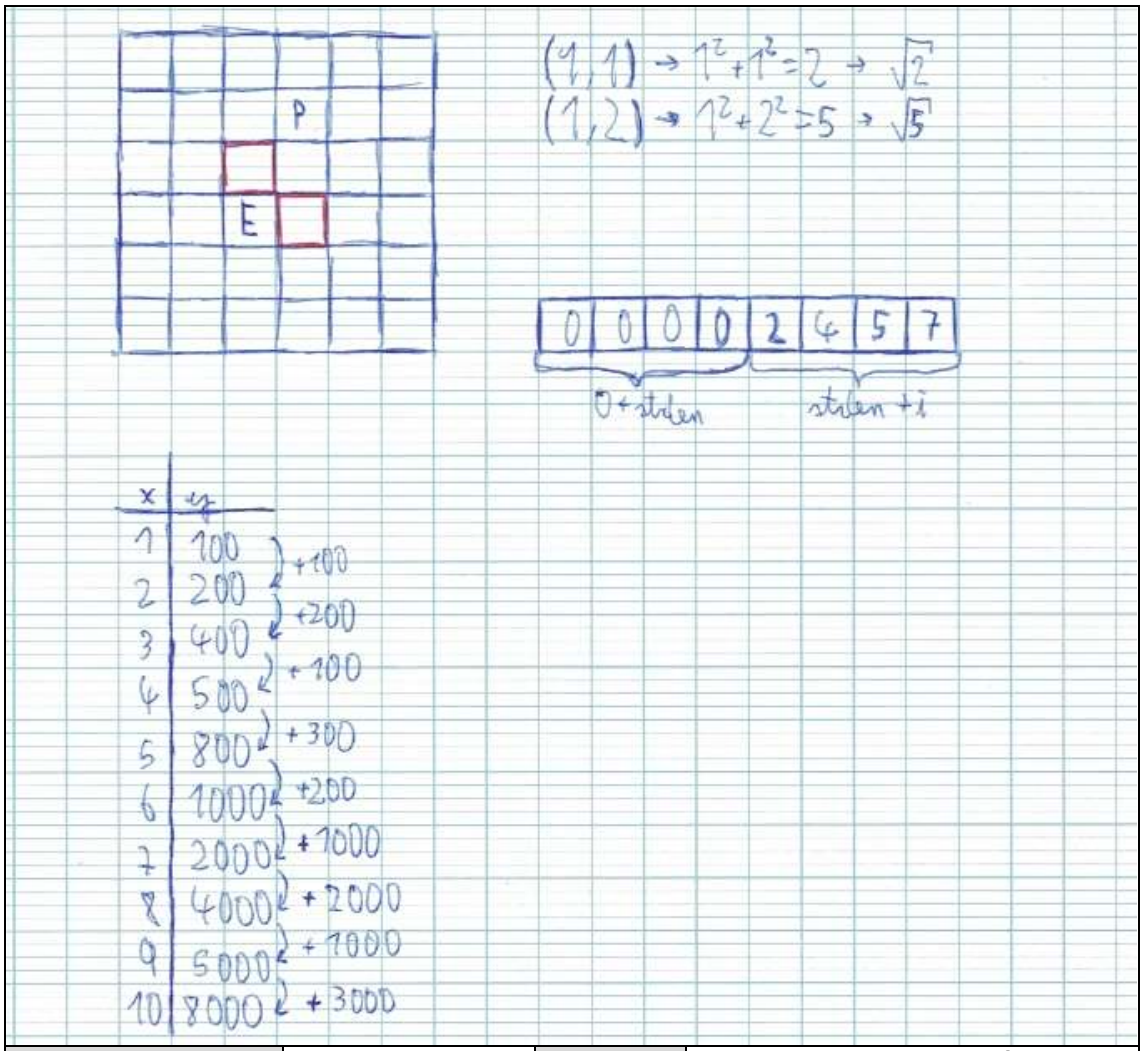
Nom	Sistema de guardat II	Detalls	Segona meitat dels càlculs de viabilitat.
Descripció	Seguidament, s'hi veuen els càlculs per a una graella amb una filera més. Tot i que tal com es pot veure, aquí ja em donava el nombre que volia, la graella era massa gran, així que vaig augmentar el sistema numèric de binari a quaternari i així vaig dissenyar el sistema que s'acabaria utilitzant.		



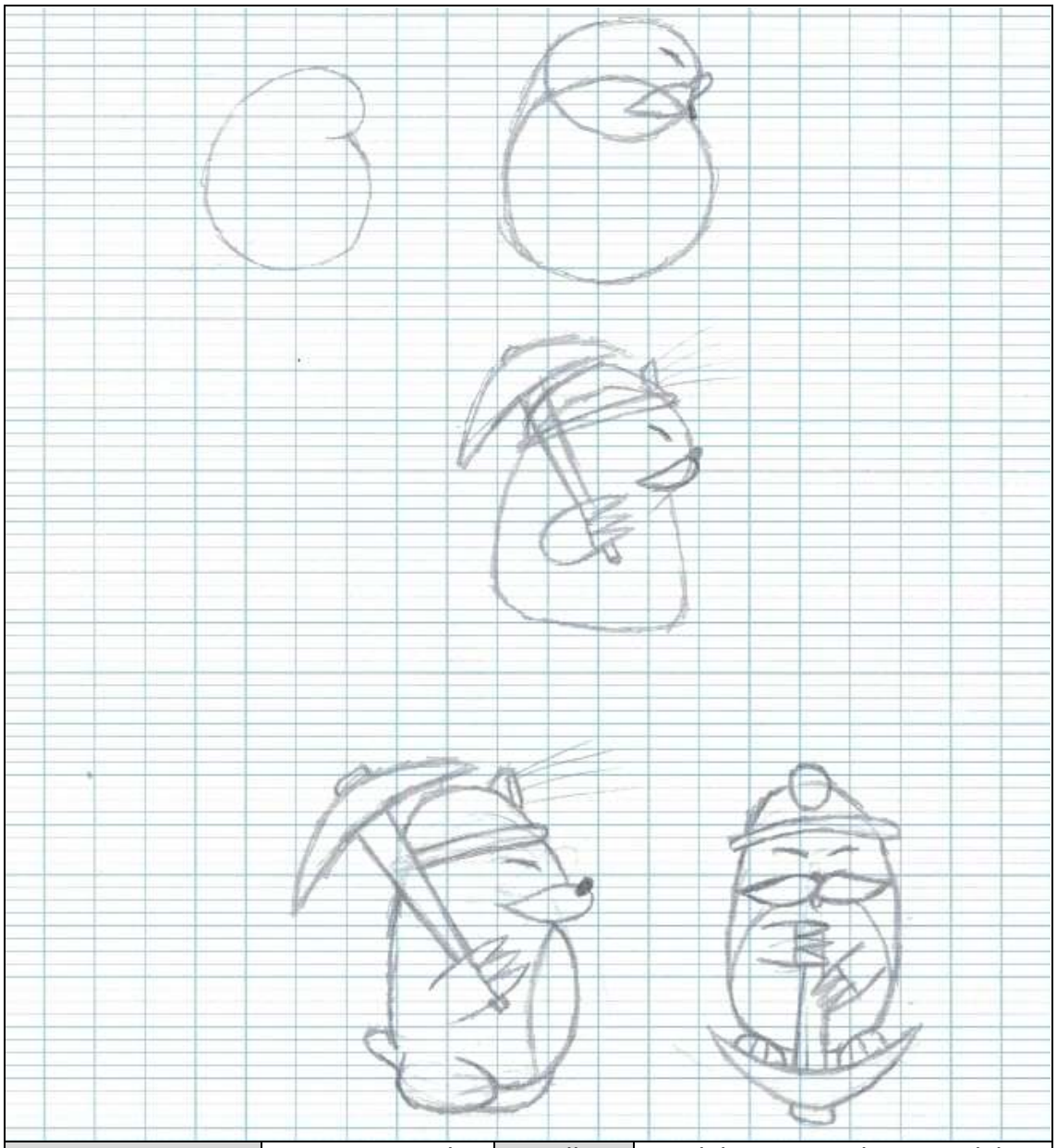
Nom	Sistema de guardat III	Detalls	Organització inicial del sistema.
Descripció	En la zona superior de la imatge hi ha dibuixat un esquema semblant al de l'apartat 4.7.2.4, en aquest cas era més petit perquè encara no havia pensat en l'organització dels nivells. A la part inferior hi ha uns breus apunts del funcionament dels operadors *bitwise* que vaig apuntar-me per a tenir-los sempre a mà.		



Nom	Sistema de guardat IV	Detalls	Segona meitat dels apunts dels operadors.
Descripció	Aquí s'hi troben la resta dels apunts, on només quedava l'operador NOT i els operadors *bitshift*. Tots estan exemplificats per tal de no haver de llegir a l'hora de revisar-los.		



Nom	Sistema de puntuació	Detalls	Esquema representació de la puntuació i enemics.
Descripció	En la zona superior hi ha una petita prova per al càlcul de distàncies, però just a sota hi ha la representació numèrica de la puntuació. Per a imitar les puntuacions clàssiques amb les vuit xifres, vaig fer-me aquest petit esquema sobre com es dibuixen aquests nombres. A la part inferior hi ha una seqüència de valors, els quals representen els punts que et donen els enemics, si els destrueixes amb l'atac aeri sense tocar el terra, guanyaràs més punts seguint aquesta successió.		



Nom	Art conceptual del protagonista	Detalls	Un dels primers dissenys del personatge final.
Descripció	Inicialment quan encara no tenia massa clar qui seria el personatge final del videojoc, havia de dibuixar diferents personatges fins que un d'ells acabés sent el definitiu.		