UNIVERSITAT DE BARCELONA

Treball final de grau

GRAU D'ENGINYERIA INFORMÀTICA

Facultat de Matemàtiques i Informàtica
Universitat de Barcelona

MAKING A VIDEO GAME FOR THE SUPER NINTENDO ENTERTAINMENT SYSTEM

Autor: Òscar Lek Boada

Director:	**Ricardo Jorge Rodrigues Sepúlveda Marques**
Realitzat a:	**Departament de Matemàtiques i Informàtica**
Barcelona,	**20 de juny de 2021**

Contents

Abstract

This project showcases the full development of a video game for the Super Nintendo Entertainment System, a console released in the year 1990. The game itself is made of multiple smaller games that intend to showcase as many functionalities and quirks of the system as possible. The games themselves have been selected to provide a variety of options the players can choose from.

These games include tic-tac-toe, a well-known, deterministic, simple game which became the target for the implementation of a simple AI; checkers, a more complex well-known, deterministic, strategy game; yacht dice, a strategy game with a prevalent element of randomness; tilecounter, a made-up game for this project, demanding reaction time and observation.

Aquest projecte mostra el desenvolupament d'un videojoc per la Super Nintendo Entertainment System, una videoconsola de l'any 1990. El joc està format per varis jocs més petits que pretenen mostrar tantes funcionalitats i peculiaritats del sistema com sigui possible. Els jocs en si han sigut escollits per tal de proveir una selecció variada d'opcions als jugadors.

Aquests jocs són el tres en ratlla, un joc conegut, deterministic i simple que va acabant rebent la implementació d'una IA; les dames, un joc d'estratègia conegut, més complex i deterministic; la generala, un joc d'estratègia amb elements d'atzar; el comptaquadres, un joc inventat per aquest projecte, requerint habilitats de reacció i observació.

Este proyecto muestra el desarrollo de un videojuego para la Super Nintendo Entertainment System, una videoconsola del año 1990. El juego está formado por varios juegos más pequeños que pretenden mostrar tantas funcionalidades y peculiaridades del sistema como sea posible. Los juegos en si han sido elegidos para proveer una selección variada de opciones a los jugadores.

Estos juegos son el tres en raya, un juego conocido, determinístico y simple que acabó recibiendo la implementación de una IA; las damas, un juego de estrategia conocido, más complejo y determinístico; la generala, un juego de estrategia y azar; el cuentacuadros, un juego inventado para este proyecto que requiere habilidades de reacción y observación.

Chapter 1

Introduction

The Super Nintendo Entertainment System (SNES) is a 16-bit video game console released by Nintendo in 1990. The console was regarded as a success overall, with many of its games being regarded as classics today.

We are used to modern programming, with powerful hardware and high level languages, so this project showcases the development of a video game with the constraints game developers had at that time.

The game is written in assembly language, as it is the only option available. This means that usual features of high level languages, such as variables or functions, have to be given a different approach when writing the code. For instance, displaying graphics requires making a tileset and arranging the tiles in a tilemap, then uploading them to a specific memory.

The main motivation to carry out this project has been the desire to conduct the full development of a game for the SNES, having some prior experience with making small modifications to already existing games for the console.

The rest of this document contains the planning of the project with the dates milestones were achieved, the objectives of the project, a glossary of terminology used throughout the document, a description of the environment used during the project, a development section detailing the process of initializing the console, making the menus and making each of the games within this project, and the conclusions derived from the project.

Chapter 2

Planning

Development of this project, as illustrated in table 2.1, began on 8th February, 2021, with the target being reaching a functional demo with one game by 19th March, 2021. The project managed to reach such a state in time, with the initialization, the menus and tic-tac-toe being finished.

After that, it was planned to add four more games, each showcasing a functionality of the console. The initial plan was to implement one game every two weeks so, as such, Checkers and Yacht dice were finished by 20th April, 2021.

By then, there was an idea to add an artificial intelligence to a game, so two weeks were dedicated to the implementation of a tic-tac-toe AI instead of another game, reducing the total to four.

By 18th May, 2021, the project had all four games finished and an AI for one of them, as well as the addition of some finishing touches.

The planning and writing of this document started thereafter, with its completion finishing on 20th June, 2021.

Date	**Activity**
08/02/2021	Start of the documentation phase, research about the first steps of the project.
28/02/2021	First graphics drawn on screen, minimalist gameloop.
12/03/2021	First iteration of the menu.
19/03/2021	The first game, tic-tac-toe, is finished.
06/04/2021	Checkers is finished.
20/04/2021	Yacht dice is completed.
04/05/2021	The AI for tic-tac-toe is completed.
18/05/2021	The last game, tilecounter, is finished.
20/06/2021	The writting of this document is completed.

Table 2.1: Milestones achieved throughout the project

Chapter 3

Objectives

The main objective of this project is to create a new SNES game from scratch. The intention is to create a collection of small games for two players in which both players are shown the same information.

To accomplish this, we want to have a menu where the player can select among those games, transition into the selected game and play it. We want the menus and the games to feel fluid and responsive.

Another objective of the project is to learn how to develop a full game without an existing base to build up from and to further improve my understanding about the mechanics and limitations of the SNES.

Chapter 4

Background

4.1 Glossary

This section intends to show several definitions of the terms used throughout this document, as they will make understanding the project more straightforward.

Memory Map The address space of the SNES is 3 bytes wide, resulting in all components of the SNES architecture needing to be located within addresses `$000000` through `$FFFFFF` so the CPU can access any of those components through their respective addresses.

Work RAM (WRAM) Multipurpose RAM used to load and store data for any use.

Video RAM (VRAM) Specialized RAM used to store tilesets and tilemaps. Indirectly accessed via registers `$2116` through `$2119`.

Color Graphics RAM (CGRAM) Specialized RAM used to store the entire palette. Indirectly accessed via registers `$2121` and `$2122`.

Object Attribute Memory A location in memory designated to contain the properties of objects. Indirectly accessed via registers `$2102` through `$2104`.

Save RAM RAM that can persist even after the console is turned off.

Mirroring A location that can be accessed in more than one way in the address space.

Interrupt A signal that pauses current program execution and runs its respective subroutine.

Cathode-Ray Tube (CRT) The type of monitor used during the SNES era. Consists of a vacuum tube containing one or more electron guns, the beams of which are manipulated to display images on a phosphorescent screen one row at a time.

Scanline A row of pixels generated by the electron beam of a CRT.

Blanking A period in which the electron beam of the CRT is turned off.

Horizontal Blanking (H-Blank) The blanking period between two consecutive scanlines.

Vertical Blanking (V-Blank) The blanking period between the last scanline of a frame and the first scanline of the next frame.

Forced Blanking (F-Blank) The blanking period in which the beam is turned off when it would otherwise be on, controlled by register `$2100`.

Direct Memory Acces (DMA) Allows for fast transferring of data from anywhere in the 24-bit address space and any of the registers between `$2100` and `$21FF`.

H-Blank DMA (HDMA) Allows to time a DMA transfer during specific H-blanks.

Subroutine A self-contained block of code or instructions the program runs.

4.2 Execution and development environment

4.2.1 The compiler

In order to compile the project, an assembler is needed, which takes the source files containing the 65c816 code and other data, as well as resources such as graphics. Asar [4] is the assembler used to compile this project. It assembles the source files according to the specifications found in its manual [5]. Among other assemblers, it was chosen because it has many useful features, is easy to use and is the one I was familiar with prior to this project.

4.2.2 The emulator

After compiling the ROM, we need to test it somehow, and testing it on a real SNES would be cumbersome, as it would require transferring the ROM into a cartridge every time it needed to be tested. An emulator for the SNES allows a computing device to run the console's software, making programming and testing in a computer a streamlined process.

The emulator used to test the project is lsnes [7], chosen because it allows showing the values contained in specific addresses at any given time, useful for debugging. For just checking out the program, other emulators such as ZMZ [9] or bsnes [8] may be used.

4.2.3 The programming language

65c816 [6] is the assembly language in which this project's code is written, the only one supported by the console's main processor. The code can be divided in instructions and data. Instructions are the opcodes plus up to 3 bytes of data. For example, we can have the instruction `LDA $08` to load the value in address `$08` into the accumulator.

4.2.4 The graphics editor

To create and edit graphics, a suitable editor capable of generating a file containing the graphics in a format recognizable by the SNES is needed. The graphics editor used is YY-CHR [10]. This tool is a multi-platform graphics editor, supporting a variety of formats beyond those supported by the SNES.

4.2.5 The directory

The directory structure of this project consists of the main folder containing the ASM source files and a resources folder containing the tilsets, the tilemaps and the palette.

4.2.6 Testing the project

A script is included in order to compile the project. For a successful compilation, one only needs to ensure to have Asar on the main folder of the project. Alternatively, one can run Asar within the main folder of the project, type "main.asm" when prompted for the patch name and type "TFG.smc" for the output file. To run the generated ROM, one can simply open it with the emulator.

Chapter 5

Development

5.1 Starting up

In order to have a program running, we first need to take care of the initial set-up. The header data needs to be set up, as well as the initialization code that will run upon starting the SNES and the gameloop that will run every frame.

5.1.1 Header setup

The header [12][1, pp. 26–40] is located at address `$00FFC0` of the ROM and contains its metadata and the addresses of the interrupt vectors. The metadata contains information about the game, as seen in table 5.1, often used by emulators to identify the type of cartridge of the ROM.

Data	**Size (bytes)**	**Value**	**Meaning**
Internal game title	21	"Treball de fi de grau"	
Mapping mode	1	`$20`	LoROM, SlowRom
Cartridge type	1	`$01`	ROM and SRAM
ROM size	1	`$05`	32 kB
SRAM size	1	`$01`	2 kB
Region	1	`$0E`	Common/International
Developer ID	1	`$00`	N/A
Version	1	`$00`	v1.0
Checksum	2	`$42DD`	Sum of all bytes in the ROM
Checksum complement	2	`$BD22`	

Table 5.1: Metadata contents of the header

- The mapping mode determines how the addresses are mapped within the address space and can be LoROM or HiROM, supporting ROM sizes of up to 4 MB, with ExLoROM and ExHiROM bumping that limit up to 8 MB. This field also determines whether FastROM is enabled, providing an increased access speed for some regions of the ROM.
- The cartridge type indicates whether the cartridge contains a SRAM chip or any coprocessors.
- The ROM size determines the amount of space needed to store the whole program, calculated by the formula $\lceil log_2(\text{size in kB})\rceil$. A mere 32 kB are sufficient to store the entirety of the program.
- The SRAM size determines the size of the cartridge SRAM, with 2 kB, the smallest non-zero value, being more than enough.
- The remaining fields are self-explanatory.

The interrupt vectors are located just after the ROM metadata, starting at address `$00FFE0`. These vectors are 2-byte pointers that point to the subroutine that will run upon its respective interrupt firing. Only the NMI, RESET and IRQ interrupts are used in this project.

- The NMI interrupt occurs once per frame, at 60 fps, and is used to mark the start of a frame, useful for setting up the gameloop.
- The RESET interrupt occurs whenever the console is powered on or reset, and marks the start of the program.
- The IRQ interrupt can be turned on or off, and it can be set to happen at a specific scanline position, one or multiple times per frame. This interrupt will only be used during the game tilecounter and its use in this project will be explained in more detail in that game's section.

5.1.2 SNES initialization

In order to initialize the SNES properly, many registers may need to be reset manually, as well as the RAM, in order to clear any remaining data that could have been leftover from an earlier execution due to a console reset, for example.

This initialization is based on the one found in this example [14]. This process is shown in Algorithm 1, and it does as follows:

- Interrupts are disabled to guarantee the code to be run unimpeded.
- F-blank is enabled to allow some registers to be written to.
- Emulation mode and decimal mode are disabled, as they are always enabled on power-on and reset.
- Direct page, the base address used when using direct addressing, is set to `$0000`.
- The stack pointer is set to `$1FFF`, used mainly for keeping track of the return address of a subroutine.
- Set up the DMA transfers by allocating one channel per transfer.
- Transfer the WRAM, CGRAM and VRAM data by enabling the DMA channels.
- Set the tilemap size and tilemap registers to their initial values.
- Clear any remaining data leftover from a previous execution from the remaining, required registers.
- Load the SRAM data into WRAM unless it does not exist, in which case load the default values.
- Re-enable interrupts.
- Disable F-blank.
- Jump to the gameloop.

5.1.3 Gameloop

In order to get a gameloop running, the NMI interrupts are an essential component, as they mark the start of a frame, allowing certain code to be run at the start of V-blank. Algorithm 2 shows what the code at figure 5.1 does, which does not need to be run during V-blank, and is an endless loop that awaits an NMI (ignores IRQ), increments the frame counter, advances the random number generator and calls the subroutine resulting from the pointers table.

Some registers can only be updated during a blanking period, therefore, mirrors in WRAM are used to store the values obtained during the execution of a frame to then be copied to their respective registers.

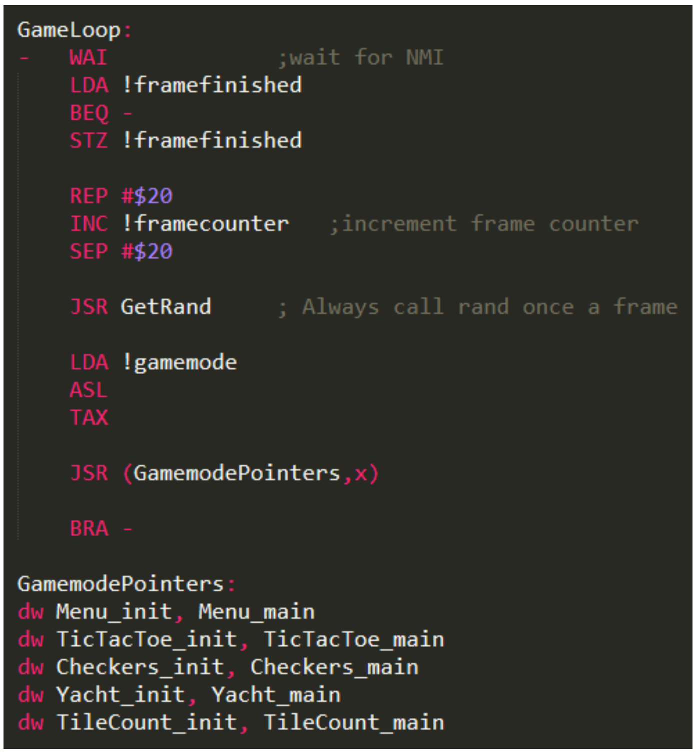
Algorithm 1: SNES Initialization

```
disable interrupts;
activate F-blank;
disable 6502 emulation mode and decimal mode;
set the direct page at address $0000 and the stack at address $1FFF;
set up upcoming DMA transfers;
transfer value $00 to WRAM addresses $0000 through $1FFF via DMA;
transfer palette data to CGRAM via DMA;
transfer tileset data to VRAM via DMA;
initialize tilemap address and size registers;
initialize all remaining registers to their default value, if required;
if SRAM data exists then
    load language, turn order and AI settings from SRAM;
else
    load default values for language, turn order and AI settings;
end
enable interrupts;
set F-blank to be disabled next V-blank;
jump to gameloop;
```

Algorithm 2: Gameloop

```
for forever do
    wait for interrupt;
    if frame has finished then
        increment frame counter;
        call the random number generator subroutine;
        call the subroutine corresponding to the current game state;
    end
end
```

In order to receive inputs from the player, the data sent by the SNES controller needs to be read and stored, as seen in lines 5 through 18 of algorithm 3. For that purpose, the console has a method for reading the controller data automatically: Auto-Joypad Read [13][1, pp. 79, 80]. It begins reading the data on the background at the start of V-blank and finishes shortly after, indicated by the least significant bit of register `$4212`. The data can then be read from registers `$4218` and `$4219` for player 1 and from registers `$421A` and `$421B` for player 2 [11].



```
GameLoop:
-   WAI             ;wait for NMI
    LDA !framefinished
    BEQ -
    STZ !framefinished

    REP #$20
    INC !framecounter   ;increment frame counter
    SEP #$20

    JSR GetRand     ; Always call rand once a frame

    LDA !gamemode
    ASL
    TAX

    JSR (GamemodePointers,x)

    BRA -

GamemodePointers:
dw Menu_init, Menu_main
dw TicTacToe_init, TicTacToe_main
dw Checkers_init, Checkers_main
dw Yacht_init, Yacht_main
dw TileCount_init, TileCount_main
```

Figure 5.1: Gameloop code and pointers

Every game state has its specific NMI subroutine as well, mainly in order to make specific adjustments to the tilemap.

5.1.4 Graphics

Evidently, the game needs to show something on the screen to the player. For that purpose, the SNES has a picture processing unit that takes care of all the graphics processing and rendering the screen, with its behaviour depending of the values of its registers [11].

Graphics can be rendered in a variety of ways depending on the current background mode, located at register $2105. Background mode 1 is used during the majority of the game, as it allows having two 4 bits per pixel (bpp) backgrounds, BG1 and BG2, plus an extra 2bpp background, BG3.

BG1 is used for the panel on the menu, as well as the various frames found in each game and their playfields. BG2 is used purely as a backdrop for the current game state. BG3 is used for text, as its reduced bit depth makes it a perfect candidate for heads-up displays and text in general.

Algorithm 3: NMI Gameloop

```
1  push the A, X and Y registers to the stack;
2  activate F-blank;
3  call the NMI subroutine corresponding to the current game state;
4  update all registers with their WRAM mirrors;
5  while auto-joypad read is not finished do
6      do nothing;
7  end
8  if controller 1 is connected then
9      controller 1 RAM ← controller 1 data;
10 else
11     controller 1 RAM ← controller 2 data;
12 end
13 if controller 2 is connected then
14     controller 2 RAM ← controller 2 data;
15 else
16     controller 2 RAM ← controller 1 data;
17 end
18 update buttons newly pressed this frame;
19 mark current frame as finished;
20 push Y, X and A registers from the stack;
21 return from interrupt;
```

There is an extra setting that provides with the option of displaying BG3 in front of other backgrounds. Since BG3 is used for text in this project, this functionality is activated.

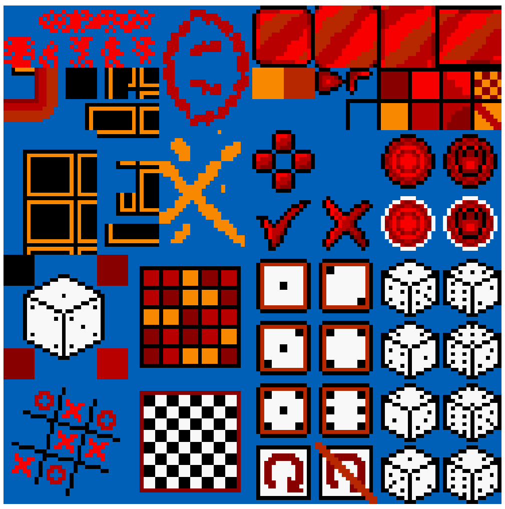
Besides backgrounds, objects are also rendered on the screen. Objects are individual, independent entities each having a 34-bit (4 bytes, 2 bits) entry located in OAM. 17 bits are used for position, 8 bits for the tile used and 9 bits for miscellaneous properties.

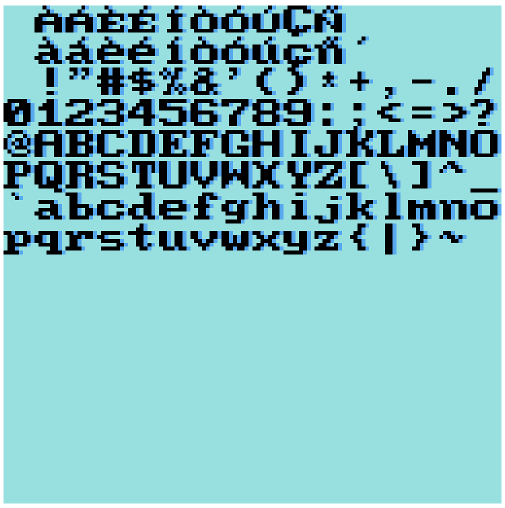
The graphics data consists of the tileset and the tilemaps. The tileset, as seen in figure 5.2, contains all the tiles used to form a tilemap. A tilemap is an arrangement of tiles used to fill an entire background. The tiles in 5.2a are the tiles used in BG1, BG2 and objects throughout the game in a bit depth of 4bpp, with the tiles in 5.2b being the font used for text in BG3 in 2bpp.

The original font [3] lacked some special characters, such as "ñ", so they had to be added. The characters were repositioned so that their tile indices matched their ASCII number, if possible, for convenience. Additionally, a shadow was added to all the characters to make the text stand out more.

Lastly, the palette consists of 256 colours, divided in 16 palettes of 16 colours each. In background mode 1, the first 8 are for backgrounds, while the other 8 are for objects. Each tile of a background or an object can display the colours of one palette. Note that the 8 palettes for BG3 have 4 colours each and are split among the first 32 colours of the entire palette. The first colour of a palette is always treated as transparency.

A tile size of 8×8 pixels and a tilemap size of 32×32 tiles (totaling 256×256 pixels) are used, as those are the minimum dimensions a background can have, enough to fit the default SNES screen resolution, which means that that is all you need for a static background. Even so, a background will wrap around the screen when scrolled, as seen in the main menu's backdrop.


(a) Main graphics in 4bpp



(b) Font in 2bpp

Figure 5.2: Contents of the tilesets

5.1.5 Cursor

A cursor provides a way to navigate through the menus and make selections during a game. Two addresses contain its x and y positions, which the player is able to manipulate when pressing up, down, left or right on the directional pad. Two addresses contain the maximum positions the cursor may have at a given time, wrapping around if the cursor would otherwise go out of range.

If the player holds down a direction, the cursor will move again after a certain amount of frames. This is known as delayed auto-shift and it is set to a value of

18 frames. If the direction is still held, the cursor advances again. This is known as auto-repeat rate and it is set to a value of 4 frames.

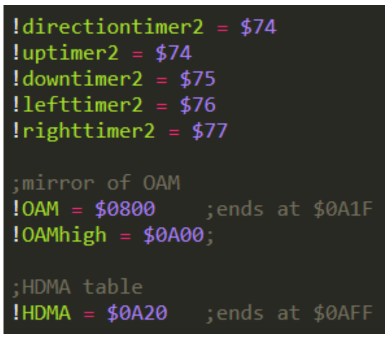
Two types of cursors were made: arrow and square. The arrow cursor is used during the menus, as it works best for navigating through a single column of elements. On the other hand, the square cursor is more suited towards selecting an element from a grid, therefore used during the games.

The graphics of the cursor are updated every time a game state is initialized or whenever a player's turn starts in order to accommodate for what kind of cursor is needed and whose player it belongs. Its animation is also updated whenever the cursor is active.

5.1.6 RAM map

In order to give a name to all the variables in the code, a dedicated file for address definitions is made. As illustrated in figure 5.3, variables representing a one byte value, a two byte value or a table of values, have been mapped to an address of one, two or three bytes depending on the addressing mode [6].

The commonly used variables, such as the frame counter and the playfield, are mapped to direct page addresses due to those only requiring one byte to address. On the other hand, the long tables of data that are not accessed too often, like the HDMA table, are mapped to an absolute address, requiring two bytes to address.



```
!directiontimer2 = $74
!uptimer2 = $74
!downtimer2 = $75
!lefttimer2 = $76
!righttimer2 = $77

;mirror of OAM
!OAM = $0800     ;ends at $0A1F
!OAMhigh = $0A00;

;HDMA table
!HDMA = $0A20    ;ends at $0AFF
```

Figure 5.3: Sample of the RAM map

5.2 Menu

The menu state is the state the game starts on power-on or reset. It consists of the title screen, the main menu and the options menu.

When switching between submenus within the menu state, a pixelation effect is applied to background 3 (text) in order to show a fluid transition. Such effect is achieved by writing to register `$2106` [11].

The BG2 tilemap, showing the backdrop, scrolls across the screen at a rate of 1 pixel per 2 frames horizontally and 1 pixel per 4 frames vertically.

5.2.1 Title screen

The title screen contains the game's title and a *Press Start* prompt. Naturally, the player may advance to the main menu by pressing the Start button.

5.2.2 Main menu

The main menu screen consists of four options, each selecting its corresponding game, as well as a fifth one for the options menu. Pressing the A button will advance to the corresponding selection, while pressing the B button will go back to the title screen.

5.2.3 Options menu

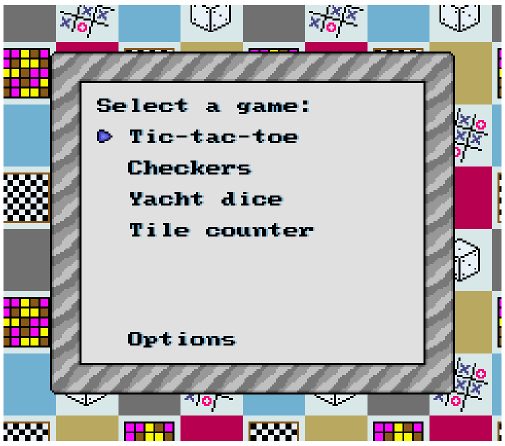
The options menu screen consists of three options, plus an option to go back to the main menu. Pressing the A button will cycle among the possible values each setting can have, while pressing the B button will go back to the main menu, just like selecting the *Back* option.

The Language setting may be set to *English*, *Español* or *Català*; the game is fully localized in those languages. The First move setting may be set to *Player 1*, *Player 2* or *Random*; it determines which player moves first for turn-based games. The Versus CPU setting may be set to *Off*, *Easy* or *Hard*; currently only available for tic-tac-toe, it makes player 2's moves determined by an in-game AI.

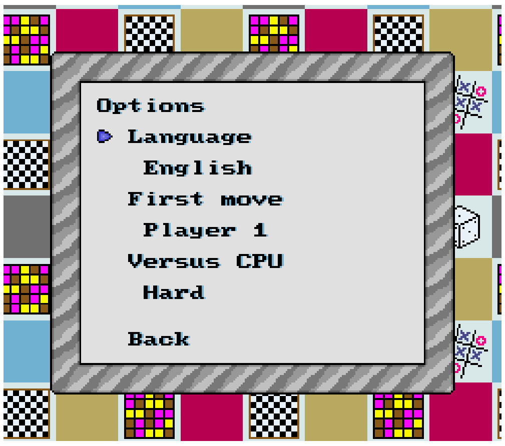
When exiting the options menu, the settings are saved into SRAM so they can be reloaded even after the console is reset or turned off and back on. This can be done by writing to addresses in banks `$70` and `$71`, mirrored at banks `$F0` and `$F1`.

5.2.4 Menu Results

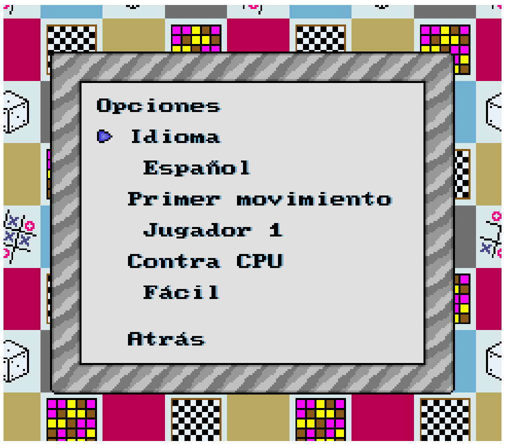
In figure 5.4 we see how the menu and its submenus ended up, with their selectable options through an arrow cursor and the menu to menu transitions.



(a) Main menu


Options
Language
English
First move
Player 1
Versus CPU
Hard
Back

(b) Options menu


Opciones
Idioma
Español
Primer movimiento
Jugador 1
Contra CPU
Fácil
Atrás

(c) Options menu after changing language

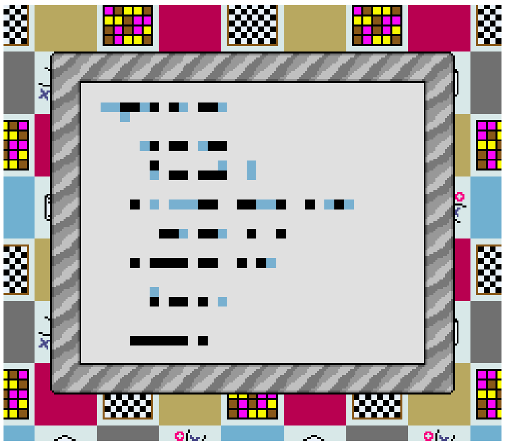

(d) Pixelation effect

Figure 5.4: Game screen of the menu

5.3 Tic-tac-toe

A widespread, well-known game in which two players take turns marking an initially empty grid with crosses or circles, with the objective of the game being connecting three of your own symbols in a straight line.

This was the first game to be developed, as its simplicity made it a perfect candidate for establishing a base for developing the upcoming games.

5.3.1 Tic-tac-toe initialization

At first, during NMI, the corresponding tilemaps are loaded into VRAM and the OAM is cleared. After NMI, many variables are set up in accordance to algorithm 4.

The frame after initialization is complete, the screen will start to fade in, getting brighter every frame until it achieves maximum brightness.

Algorithm 4: Tic-tac-toe initialization

```
brightness ← 0;
tilemap timer ← 0;
set background positions;
cursor position ← (0,0);
max cursor position ← (2,2);
set the cursor graphics position table to fit each cell of the playfield;
set default cursor palettes (blue for player 1, red for player 2);
if turn order = random then
    current player ← player 1 or player 2 at random;
else
    current player ← value from turn order setting;
end
initialize square cursor;
game mode ← tic-tac-toe main;
substate ← fade-in;
return;
```

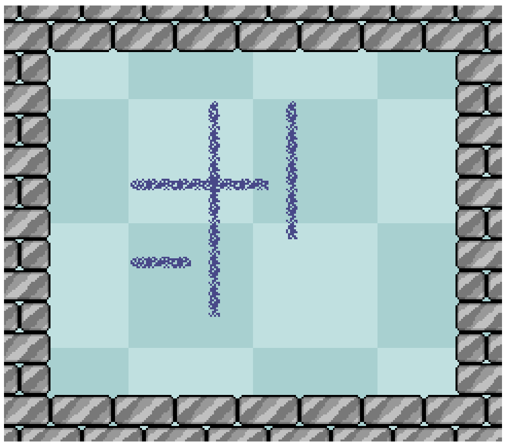


Figure 5.5: Tic-tac-toe grid being drawn

Finally, the tilemap is updated to display an opening transition that draws the 3×3 grid. This procedure happens during several frames until the grid is fully drawn, then play can start.

5.3.2 Tic-tac-toe main subroutine

On the first frame, text showing the player currently in control is displayed, just as an indication.

The current player is able to move the cursor through the grid to the desired location of their move, pressing the A button to confirm it, advance the substate and resolve whether the game reached a terminal state. The terminal state is reached when a player wins by lining three in a row or all nine spaces on the grid end up occupied.

5.3.3 Draw symbol

After a player makes a selection, their symbol is drawn akin to how the grid was drawn during setup. The tilemap is modified every frame until the full symbol is drawn, then play resumes for the opposing player unless a terminal state is reached, in which case the game advances to the end screen.

5.3.4 Artificial intelligence

A simple AI was made to demonstrate the game being able to play as one of the players. It uses the strategy described in algorithm 5. Note that on hard difficulty the AI will always play the optimal move [2], while on easy difficulty it will have a 50% chance to play a random move if there is no trivial (win or block) move.

The inclusion of an easy mode makes the AI need to react to any possible board state instead of just the follow-ups to the optimal moves, as well as to make it possible for the human player to win.

5.3.5 Tic-tac-toe results

This game laid the foundation for the development of the ensuing games, making the process of adding games more straightforward.

We have managed to implement the game successfully with an AI capable of responding to any given board state. In figure 5.6, we can see a possible outcome of a match against the AI.

Algorithm 5: Strategy used by the AI, assuming AI plays as circle

```
/* board is a grid with each square filled with 0, 1 or 2;
   representing empty, cross or circle, respectively.          */
Input: Board state
/* move is the position on the board where the AI will play.   */
Output: Move to make
move ← null;
for line in board.lines do
    if line has 2 of the same symbol  last space empty then
        move ← the empty space;
        if matching symbol is circle then
            /* Winning move found, returns immediately.        */
            return move;
        end
    end
end
if move $\neq$ null then
    /* A move that blocks a win from the opponent is found.    */
    return move;
else if difficulty = easy & random bit = 0 then
    return random element from the available squares;
else if a fork is available then
    return a square such that two or more ways to win are available next
      turn;
else if opponent has one or more forks available then
    if all forks can be blocked then
        return a square that blocks all forks;
    else if a win can be threatened then
        return a square that makes two in a line and threatens a win next
          turn;
    end
end
if center is empty then
    return center;
else if opposite corner is empty then
    return a corner opposite to an opponent move;
else if any corner is empty then
    return any available corner;
else
    return any available side;
end
```

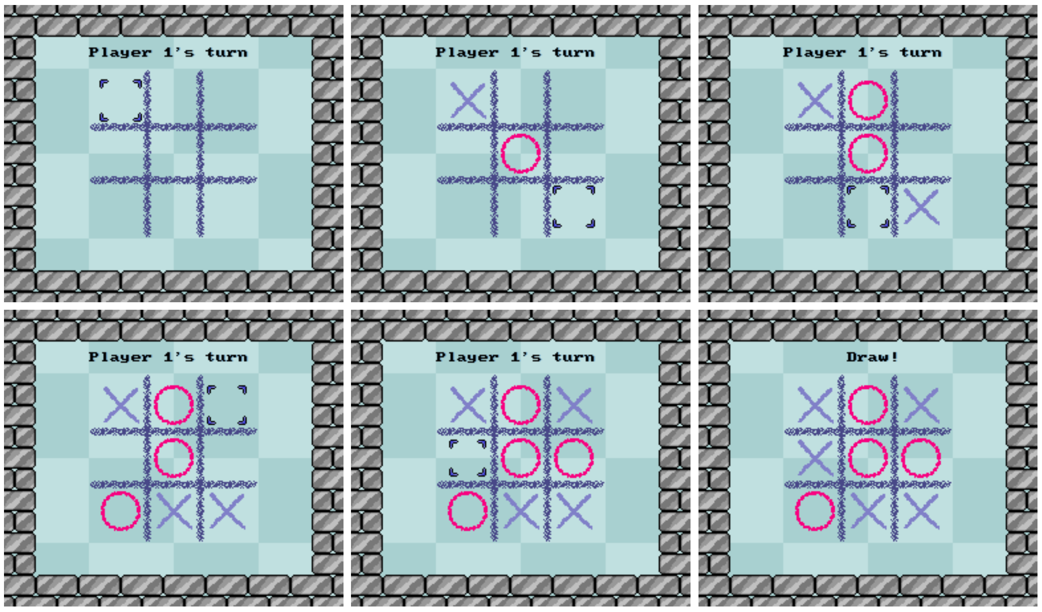


Figure 5.6: Possible match against the hard AI

5.4 Checkers

A well-known strategy game for two players which involves diagonal moves of pieces and mandatory captures by jumping over the opponent's pieces.

This game was chosen to illustrate the management of a large amount of objects on the screen, as well as the handling of a large board.

5.4.1 Checkers initialization

The general procedure is similar to what is done in Tic-tac-toe, with the maximum cursor positions having been accommodated for the 8×8 board, changing the cursor palette to red and white to match the checkers and initializing the checkers.

The checkers are put into the playfield, each one having a unique identifier, a colour and a crowned status. Following that, every checker is given object data to upload to OAM, but the OAM will not be updated until the board graphics are drawn.

For the opening transition, one square of each colour, starting at opposite ends of the board, are drawn each frame following a zigzag pattern contained in an offset table. Play can then begin.

5.4.2 Checkers main subroutine

Text indicating the current player is drawn below the board on the first frame.

The cursor is able to be moved freely across the board, being able to select any checker owned by the current player by pressing the A button. The checker will then be highlighted and will be able to be deselected by pressing the A button with the cursor over that checker or by pressing the B button. Pressing the A button on a different square makes the currently selected checker move there if such move is legal.

A checker may move diagonally forwards to an adjacent square if such square is unoccupied. If that square is occupied by an opponent's checker and the square immediately beyond is empty, the checker may be captured by jumping over it. If a capturing move is available, it is mandatory to make a capture. Furthermore, if a piece has another capture available after capturing, that piece is forced to make that capture within the same move. When a piece reaches the opposite row of the board, that piece becomes a king, allowing it to also move and capture backwards.

The terminal state of the game is achieved when the current player has no legal moves available, usually due to losing all of their pieces.

5.4.3 Move checker

This is the state used to move the object of the moved checker.

It moves the checker towards its destination one pixel every two frames for a regular move, and one pixel per frame for a capture. The captured piece, if any, is then placed off the board on top of any previous ones.

During a capture, the capturing checker needs to appear in front of the captured piece. Objects have a property that determines whether to display in front of or behind backgrounds, but the priority between two objects is determined by their order in OAM. Conveniently, register `$2103` can be used to give priority to an object other than the first one [11], so priority is given to the capturing piece.

After that, it recalculates the current player's legal moves to allow successive captures, otherwise it switches players and calculates their legal moves. If a terminal state is reached, the game advances to the end screen.

5.4.4 Checkers results

A faithful recreation of the well-established game was successfully made, managing to demonstrate the handling of several objects and proper restriction of legal moves. In figure 5.7 we see how the finalized version looks.

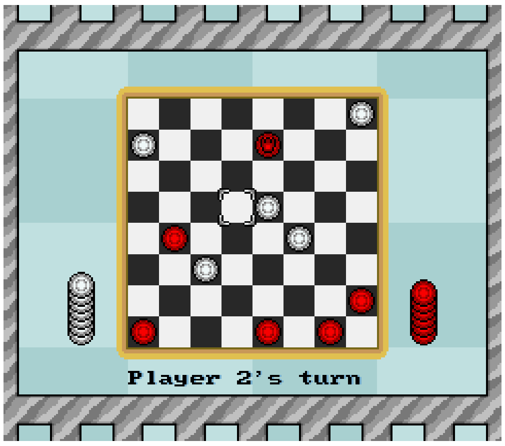


Figure 5.7: Game screen of Checkers

5.5 Yacht dice

A game in which two players take turns rolling five dice and fill multiple categories with their score.

This game intends to showcase the random number generator in the game, as there is no native alternative we can resort to.

5.5.1 Initialization

As seen beforehand, the initialization procedure is similar to the aforementioned games. This time, there are two cursor tables: one for categories and one for dice, switched by pressing left or right on the directional pad, without horizontal wraparound.

The registers `$2126` through `$2129` control the position of the windows [11], which allow the occlusion of backgrounds and/or objects in a specific region of the screen. These windows are used for the opening transition of this game as seen in figure 5.8, hiding the table of categories at the start and revealing it as time passes. This transition effect is achieved by changing the window offsets via HDMA in order to only affect the scanlines that contain the table of categories. The windows are shifted horizontally by changing the values written to their registers, while they are shifted vertically by modifying the amount of scanlines each HDMA entry is active for.

Another HDMA channel is set up in order to fit the BG3 text within the table of categories, moving the text up by four pixels for every line of text.

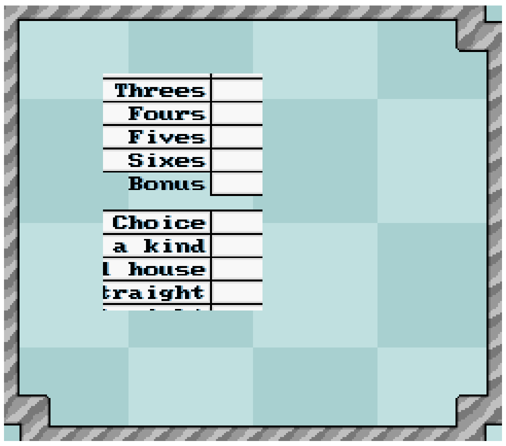


Figure 5.8: Yacht score table during the opening transition

5.5.2 Roll dice

Before a player takes control, the five dice are initialized by creating an entry on the OAM for each dice plus one entry for the re-roll button.

Next, the dice are rolled, animating until they reach their position and show their result. The animation frame for the dice is randomly determined out of eight possible frames, and is flipped both vertically and horizontally whenever it changes, making them feel like they roll.

The result of the dice roll is, naturally, determined at random with only one call to the random number generator, as there are $6^5 = 7776$ possible outcomes for rolling five 6-sided dice and $2^{16} = 65536$ possible outcomes for the random number generator, so 13 out of 16 random bits are sufficient, although extra calls may be needed if the 13-bit result falls in the range $[7777, 8191]$.

5.5.3 Yacht dice main subroutine

The current player can move around their column on the table of categories and the dice.

The player may select dice to set it aside and keep it instead of re-rolling it. The player may choose to re-roll the dice twice (for a total of three rolls).

At any point, the player in control may mark one of their unmarked categories on the table, regardless of whether they have re-rolls remaining or not, according to the rules described in table 5.2.

At all points the player is able to see the possible score the would obtain in a category in gray text and the categories they have already marked in black text.

5.5.4 Mark category

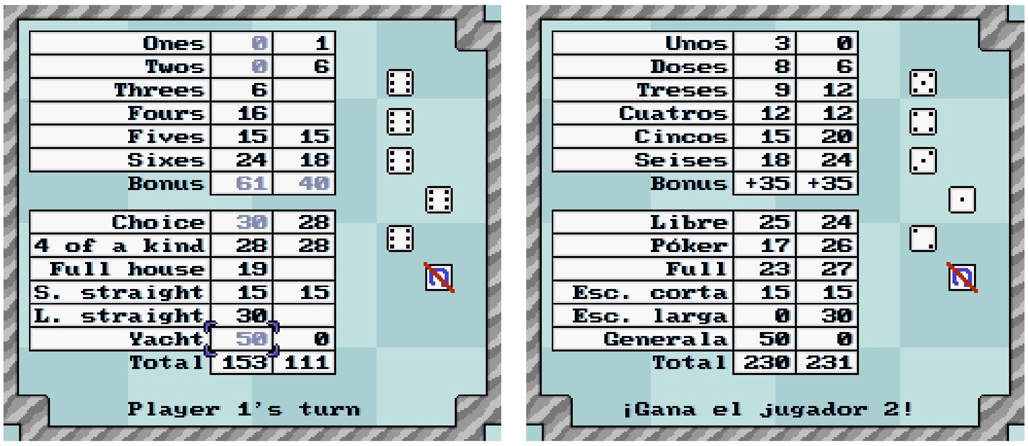
The score corresponding to the marked category is now displayed in black text, and the previous possible scores are erased. The bonus category is updated to show the current progress made there in gray text, and turned into a black "+35" or a "+0" when the bonus category criteria is either fulfilled or failed.

To make converting from a numeric value to the tiles needed for the shown text, the values are treated as binary-coded decimal by using the innate decimal mode of the SNES. This functionality can be activated by setting the decimal processor flag, and deactivated by clearing it. This simplifies the conversion by making each nibble represent a digit, so, for example, hexadecimal value of $25 would be treated as 25 in decimal.

Finally, the current player switches and the dice are reinitialized or, in the case all categories have been filled by both players, the game finishes.

Category	**Score**
Ones	Sum of dice with the number 1
Twos	Sum of dice with the number 2
Threes	Sum of dice with the number 3
Fours	Sum of dice with the number 4
Fives	Sum of dice with the number 5
Sixes	Sum of dice with the number 6
Bonus	35 if the sum of all number categories (ones through sixes) is at least 63, otherwise 0 (unselectable)
Choice	Sum of all dice
4 of a kind	Sum of all dice if at least four dice have the same number, otherwise 0
Full house	Sum of all dice if three of the same number plus the other two of the same number, otherwise 0
Small straight	15 if four numbers in sequence appear, otherwise 0
Large straight	30 if five numbers in sequence appear, otherwise 0
Yacht	50 if all five dice have the same number, otherwise 0
Total	Shows current total score (unselectable)

Table 5.2: Scoring rules for this variation of yacht dice



Ones	0	1
Twos	0	6
Threes	6	
Fours	16	
Fives	15	15
Sixes	24	18
Bonus	61	40
Choice	30	28
4 of a kind	28	28
Full house	19	
S. straight	15	15
L. straight	30	
Yacht	50	0
Total	153	111

Player 1's turn

Unos	3	0
Doses	8	6
Treses	9	12
Cuatros	12	12
Cincos	15	20
Seises	18	24
Bonus	+35	+35
Libre	25	24
Póker	17	26
Full	23	27
Esc. corta	15	15
Esc. larga	0	30
Generala	50	0
Total	230	231

¡Gana el jugador 2!

Figure 5.9: Game screens of Yacht dice

5.5.5 Yacht dice results

The end result of this game is a game that manages to utilize the random number generator and the decimal mode, as well as multiple simultaneous columns of variable height for the cursor. In figure 5.9 we can see a mid-game state and a terminal state.

5.6 Tilecounter

A made-up game about a 5×5 grid of coloured tiles in which the players need to determine the most plentiful tile over a total of 16 rounds. The amount of distinct tiles found on a given round varies, and so does the way the tiles are presented to the players.

5.6.1 Tilecounter initialization

The brightness, tilemap timer and background positions are all initialized just like in previous games, but the cursor is not used. Current player is set to `$0A` to draw the current round instead, as this is not a turn-based game. Background mode 2 is used to allow BG3 data to be interpreted as offset values[1, pp. 78, 208] for BG1 and BG2.

On NMI, the tilemaps and background offsets are uploaded to VRAM, draws a 0 on both score counters, sets up various HDMA channels, enables IRQ to happen at scanline `$A1` and points the background 3 tilemap address to the offset data.

5.6.2 Tilecounter main subroutine

The grid is generated by placing one of the correct tiles on it, then filling it with as many tiles as the round needs distributed equally, favouring the correct tile to ensure it is actually the one that appears the most. The grid is then shuffled.

There are always four possible answers the players are able to choose. There is always only one correct answer, with the other three tiles belonging to the incorrect answers being present in the grid unless there are less than four tiles in the grid, in which case it selects any remaining absent tile.

The players are then able to input their answer. Either the directional pad or the A, B, X or Y buttons can be used to select an answer based on its position. For example, either the Y button or left on the directional pad can be used to select the answer located on the left.

If a player selects the correct answer, it is awarded one point. If a player selects an incorrect answer or selects multiple answers, the opponent is awarded a point. In the event both players are either both correct or both incorrect, the round ends in a tie and no points are awarded. A symbol appears on the chosen answers, with a green checkmark signifying a correct answer and a red cross signifying an incorrect one.

Some time is given for the players to acknowledge the outcome of the round and a new grid is generated again, until the last round, where the game will end with the highest scoring player victorious or in a tie.

5.6.3 IRQ

IRQ is enabled on this game in order to make some graphics-related changes at a specific scanline, essentially splitting the screen in two portions. This separation is done to allow having both the tilemap offset change and a BG3 rendered on the same frame.

The top part is where the tile grid is displayed and the bottom part has the score counters for both players as well as the possible answers they can pick.

As seen in algorithm 6, the background mode is changed from 2 to 1 to disable offset change and enable BG3. BG1 is shifted 20 pixels up in order to match its intended display position, as the section showing the answers is ubicated lower down in the tilemap in order to not interfere with the vertical offset changes, as they could shift past their intended location.

If IRQ were to be disabled in a round that uses the tile offset function, the screen would look like in figure 5.10 due to no mid-frame register updating taking place.

Afterwards, the involved registers are set back to their original values in their mirrors during NMI.

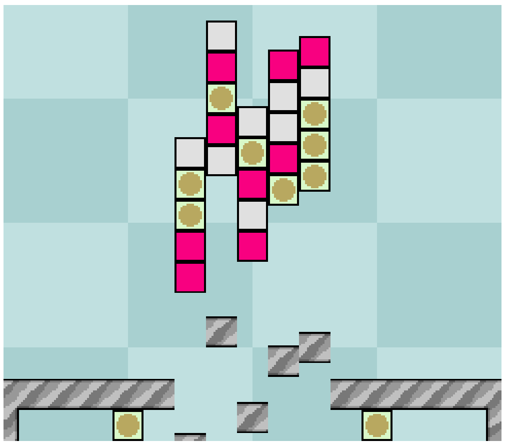


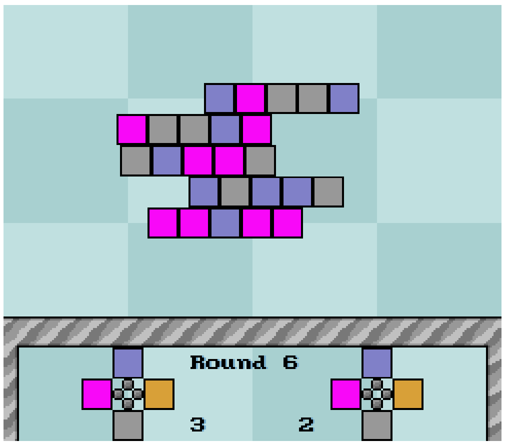
Figure 5.10: Hypothetical situation without IRQ

5.6.4 Tilecounter results

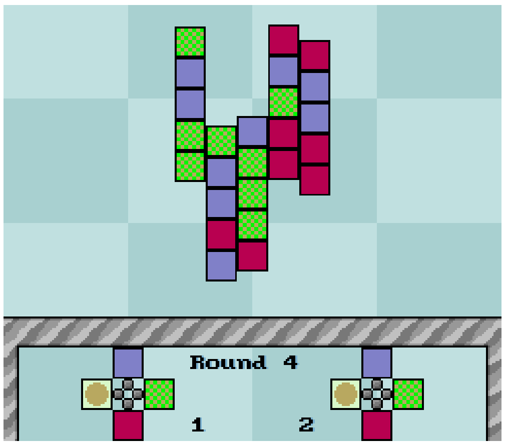
A reaction-oriented game is satisfactorily implemented, showing the capabilities of tile offsetting and the possible applications of IRQ. Figure 5.11 shows many graphic functionalities used along the rounds.

Algorithm 6: IRQ contents for tilecounter

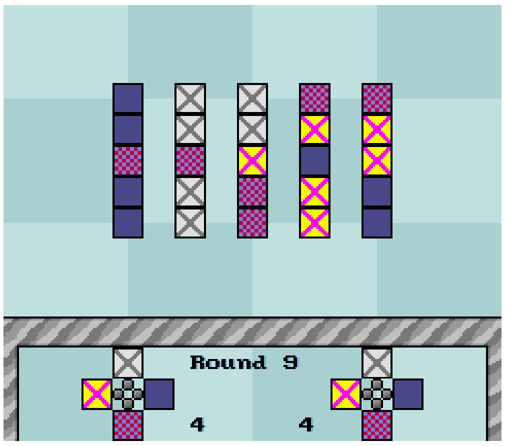
```
1 push the A register to the stack;
2 activate F-blank;
3 set background mode to 1, BG3 priority;
4 point the BG3 tilemap address to the tilemap;
5 adjust the BG3 position;
6 move BG1 up to match intended display;
7 disable the windows;
8 pull the A register from the stack;
9 return from interrupt;
```



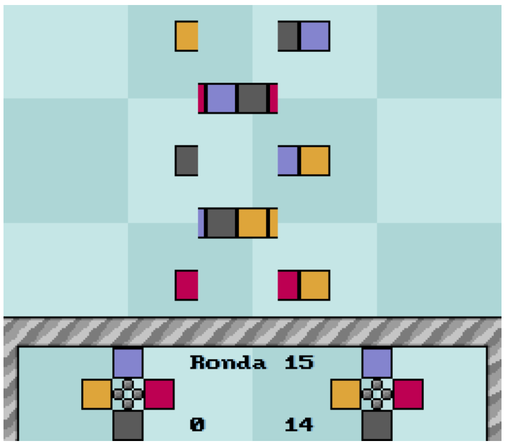
(a) Horizontal row shift through HDMA



(b) Vertical column shift through tile offsetting



(c) Separation between columns through tile offsetting



(d) Separation between rows through HDMA and window occlusion

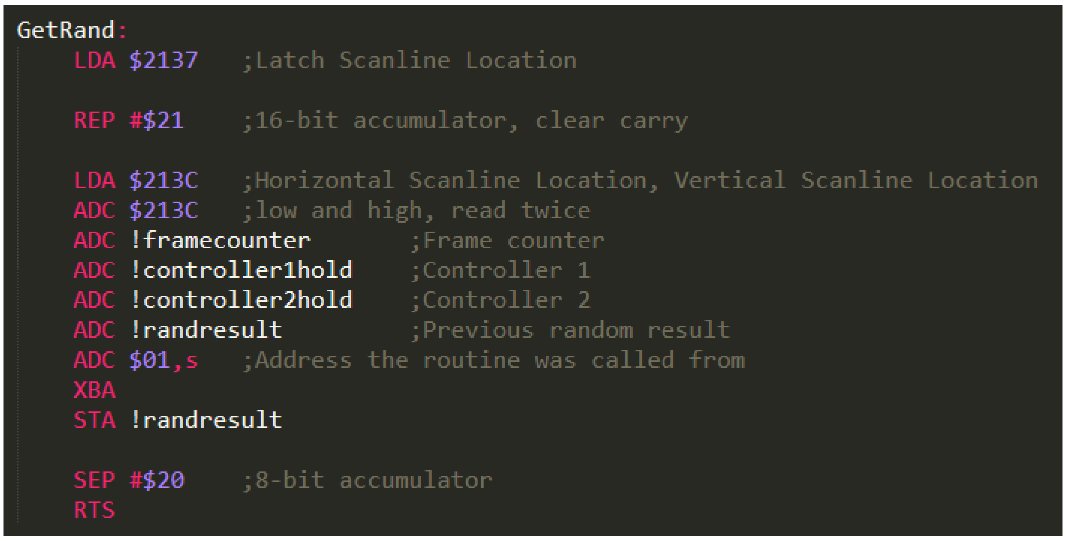
Figure 5.11: Multiple game screens of Tilecounter

5.7 Utility subroutines

Subroutines are snippets of code that can be called from another location of the code, just like a function would do in a higher level language.

5.7.1 Random number generator

The SNES does not have a native random number generator, so we have to develop one. As shown in figure 5.12, the 16-bit pseudo-random number is generated by taking the sum of the horizontal and vertical scanline locations, the current frame count, the inputs from controllers 1 and 2, the previous random result and the address the subroutine was called from; swaps the order of its bytes and stores it as the new random number.



```
GetRand:
    LDA $2137   ;Latch Scanline Location

    REP #$21    ;16-bit accumulator, clear carry

    LDA $213C   ;Horizontal Scanline Location, Vertical Scanline Location
    ADC $213C   ;low and high, read twice
    ADC !framecounter       ;Frame counter
    ADC !controller1hold    ;Controller 1
    ADC !controller2hold    ;Controller 2
    ADC !randresult         ;Previous random result
    ADC $01,s   ;Address the routine was called from
    XBA
    STA !randresult

    SEP #$20    ;8-bit accumulator
    RTS
```

Figure 5.12: Subroutine used to generate a pseudo-random number

5.7.2 Fading

This subroutine is used for both the fade-in and the fade-out effects. The fade-in effect is used as a substate after initialization but before play starts, and it makes the screen brightness increase from its minimum value to its maximum across several frames. The fade-out effect does the opposite by dimming the screen until it becomes black, used after a game's end screen.

5.7.3 Game end

A general subroutine for drawing on screen the outcome of the game. It loads a line of text on BG3 saying which player won, or that the game ended in a draw.

It awaits for any player to press either the A, B or Start buttons to then start a fade-out effect and return to the main menu.

5.7.4 Cursor initialization

For the arrow cursor, one entry in OAM with its position, tile number and palette is created. For the square cursor, one entry is made for every corner, totaling four, having their horizontal and vertical flip properties adjusted as needed and with their palette showing the current player in control.

5.7.5 Cursor update

Updates the position of the cursor graphics in relation to the position of the cursor itself and its current animation frame. It looks for the position each OAM entry is supposed to have for the given cursor position in a table and shifts it slightly according to the current animation frame.

5.7.6 Tilemap upload

These subroutines each upload a tilemap to VRAM in different ways. They are as follows:

- Load empty tilemap: it clears the selected amount of bytes from the selected tilemap by transferring zeros through DMA.
- Load uncompressed tilemap: it takes the raw tilemap data form the selected address and transfers it through DMA.
- Load compressed tilemap: it takes compressed tilemap data, decompresses it and uploads it directly to VRAM.

  A compressed tilemap allows for long sequences of repeating tiles to be represented as the tile in question and its amount. A string of tiles that only share properties is compressed by specifying the common properties followed by the tile numbers.
- Load partial tilemap: it takes multiple sequences of tiles and uploads them one by one in their specified positions, allowing for efficient uploading of sparse lines of tiles, like text.

5.7.7 OAM update

The OAM is updated by transferring its mirror through DMA. An OAM clear can be performed by first clearing the OAM mirror, also through DMA, and then performing the OAM update.

Chapter 6

Conclusions

Our initial objectives were to make a game for the SNES from the ground up, with fluid and responsive controls for the games and the menus.

Given those initial objectives, we can conclude that the objectives that were set were satisfactorily completed, as a game for the SNES was able to be made without a prior foundation, and the games themselves manage to have two players playing against each other with just the information presented on the screen.

The deviation of the project in order to forgo the implementation of an extra game in favour of an AI for an already implemented game paid off, as exemplifying the addition of an AI added an extra layer of complexity to the project.

From this project I managed to expand my knowledge about programming for the SNES by acknowledging the procedure needed to get a program running without a previous base to build upon, as well as learning more about the usage of some functionalities I was not familiarized with.

All in all, it has been a purposeful and interesting experience, as it allowed me to deepen my knowledge about a topic I was invested into.

6.1 Future Work

As for future work in this project, the obvious choice would be to develop more games, as the underlying infrastructure is there.

Implementing additional AIs for other games is also a potential extension for this project, as the concept is already exemplified in tic-tac-toe and the other games would benefit from the addition.

Finally, an addition to this project would be to add music and sound effects, as it was deemed too complex of a task to warrant development over the other areas of the project.

References

[1] Nintendo of America. *SNES Developer Manual*. 1993. URL: https://archive.org/details/SNESDevManual (visited on 18/06/2021).

[2] Kevin Crowley and Robert S. Siegler. "Flexible Strategy Use in Young Children's Tic-Tac-Toe". In: (1993), p. 536. URL: https://doi.org/10.1207/s15516709cog1704_3 (visited on 18/06/2021).

[3] GrandChaos9000 (username). *Modern DOS 8 Font*. 2014. URL: https://www.smwcentral.net/?p=section&a=details&id=9146 (visited on 18/06/2021).

[4] Alcaro (username). *Asar v1.81*. 2021. URL: https://github.com/RPGHacker/asar (visited on 18/06/2021).

[5] RPGHacker (username). *Asar User Manual*. 2021. URL: https://rpghacker.github.io/asar/manual/ (visited on 18/06/2021).

[6] Bruce Clark. *65C816 Opcodes*. 2015. URL: http://www.6502.org/tutorials/65c816opcodes.html (visited on 18/06/2021).

[7] Ilari (username). *lsnes*. URL: http://tasvideos.org/Lsnes.html (visited on 18/06/2021).

[8] Near (username) and byuu (username). *bsnes*. URL: https://github.com/bsnes-emu/bsnes (visited on 18/06/2021).

[9] Alcaro (username) and the ZSNES team. *ZMZ*. URL: https://www.smwcentral.net/?p=section&a=details&id=5681 (visited on 18/06/2021).

[10] YY (username). *YY-CHR*. 2020. URL: http://www.romhacking.net/utilities/958/ (visited on 18/06/2021).

[11] Anomie (username). *SNES hardware registers*. 2007. URL: http://www.romhacking.net/documents/196/ (visited on 18/06/2021).

[12] *SNES memory map*. URL: https://en.wikibooks.org/wiki/Super_NES_Programming/SNES_memory_map (visited on 18/06/2021).

[13] *Joypad input*. URL: https://en.wikibooks.org/wiki/Super_NES_Programming/Joypad_Input (visited on 18/06/2021).

[14] Aceman2000 (username). *Making a Small Game - Tic-Tac-Toe*. URL: https://wiki.superfamicom.org/making-a-small-game-tic-tac-toe (visited on 18/06/2021).