Aerts Sybran

Game Boy Game Development

Graduation work 2022-2023

Digital Arts and Entertainment

Howest.be

CONTENTS

ABSTRACT

This graduation project compares three approaches for developing software for the Game Boy: GB Studio, C with GBDK, and Assembly. GB Studio is a drag-and-drop game development tool that allows for easy creation of Game Boy games without requiring any programming knowledge. GBDK is a development kit including a C compiler for the Game Boy that allows for more advanced programming capabilities. Assembly is a low-level programming language that allows for maximum control and optimization but requires a strong understanding of the hardware and can be difficult to learn and use. The project evaluates the strengths and weaknesses of each approach and provides recommendations for which method is best suited for different types of Game Boy software development projects.

INTRODUCTION

This research paper aims to explore the various methods of developing software for the original Game Boy console (DMG-01). While other models of the Game Boy may be discussed, the focus of this research is exclusively on the DMG-01. As a pioneering handheld gaming console released by Nintendo in 1989[1], the Game Boy saw the release of over 1000 games during its lifetime[2]. The purpose of this research is to investigate the development of games for this console, plus which technology and tools are available today to achieve this and measure the results.

In order to compare these methods, the research will examine the differences between developing a game using GB Studio, C with GBDK, and assembly. Specifically, the research will compare the ease of use, performance, and flexibility of these three approaches in order to determine the most effective method for developing games for the Game Boy. GB Studio represents a more modern and easier approach to game development, while C with GBDK represents an intermediate level of difficulty. Assembly, on the other hand, represents the very low-level coding approach utilized by official developers in the 1990s.

Simply put: What are notable differences in a Game Boy game developed in GB Studio, C and ASM both during and after development?

CONTEXT

1. GAME BOY HARDWARE

1.1. SPECIFICATIONS

The original Game Boy is often referred to as the DMG-01. DMG stands for “Dot Matrix Game” because it utilizes a dot matrix screen. The Game Boy has a 1MHz, 8-bit processor. In reality, this processor is 4MHz, but each instruction takes 4 cycles, so it is often simplified to being 1MHz. Many people will draw similarities to the Zilog Z80 chip when talking about Game Boy, however the DMG chips themselves are much more similar to the Intel 8080 than the Zilog Z80. It also has 8KB of RAM and VRAM respectively. The screen has a resolution of 160 by 144 pixels and can display 4 colors. The sprites you can have on the screen are limited. You can only have 10 sprites on the same row and 40 in total, any excess sprites will simply not be drawn.[3] This is because the OAM (Object Attribute Memory) where the sprites are stored is 160 bytes, and each sprite is 4 bytes large.[4] Each sprite is 8 by 8 pixels; however, you can set them to be 8 by 16. In the past, there existed an official devkit developed by Nintendo for Game Boy game developers. This included a debugger and emulator, sprite editing software, demo tools, cartridge tools and more.[5] This however is impossible to be used in this project as they are extremely rare.

1.2. CARTRIDGE

To test the ROMs I will be making on actual hardware, I will use the EZ-FLASH Junior flashcard. This cartridge offers a micro-SD card of 4 to 32GB in FAT32 format to be inserted, which then holds the ROMs to be played on the hardware. The EZ-FLASH Junior supports Game Boy as well as Game Boy Color ROMs with a size of maximum 64Mbits and a 1Mbit maximum size for saves. For hardware, it supports “Gameboy, Gameboy Pocket, Gameboy Color, Gameboy Advance/SP, even with backlit mods and rechargeable batteries.” They are further known to be reliable and accurate flashcards.[6]

2. SPACE INVADERS

2.1. GAME

Space Invaders is a seminal arcade game developed by Tomohiro Nishikado and published by Taito in Japan and Midway in Europe and America in 1978. It is widely considered to be the first fixed shooter game and is credited with pioneering the shoot 'em up genre. The game was later released for a variety of consoles, including the Atari 2600 and Game Boy, and has been widely recognized for its cultural and historical significance.[7]

In the game, players control a ship that can move left and right and shoot at an array of alien invaders moving across the top of the screen. The invaders move in a predetermined pattern, changing direction and shifting downward each time they reach the edge of the screen. The player must also avoid the bullets fired by the invaders and can take cover behind barriers that are placed strategically on the field. Additionally, a UFO-like alien occasionally appears at the top of the screen, which the player can shoot for bonus points. The objective of the game is to score as many points as possible by destroying as many invaders as possible, while avoiding the bullets and other hazards. Variations of the game may include different gameplay mechanics, but the core objective remains the same.

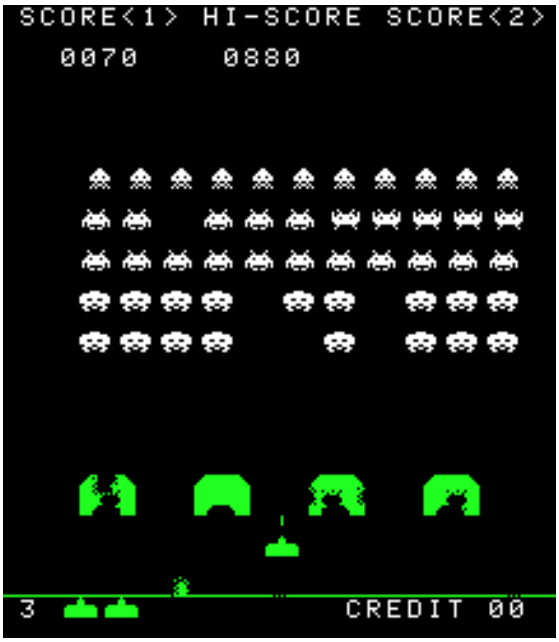


Figure 1: Arcade Space Invaders (from Wikipedia)

2.2. MY VERSION

In this version of the game, certain features from the original have been omitted in order to focus on incorporating the main elements. The player character possesses the ability to move horizontally and shoot a single bullet. The game comprises of five rows of eight invaders that move horizontally across the screen, descending and reversing direction upon reaching the edge of the screen. Furthermore, the invaders have the capability to shoot bullets at the player, with a maximum of three bullets active at any given time. The initial position of these bullets is determined randomly among the invaders. Additionally, the lives and scoreboard system has been simplified and placed at the bottom of the screen for ease of access.

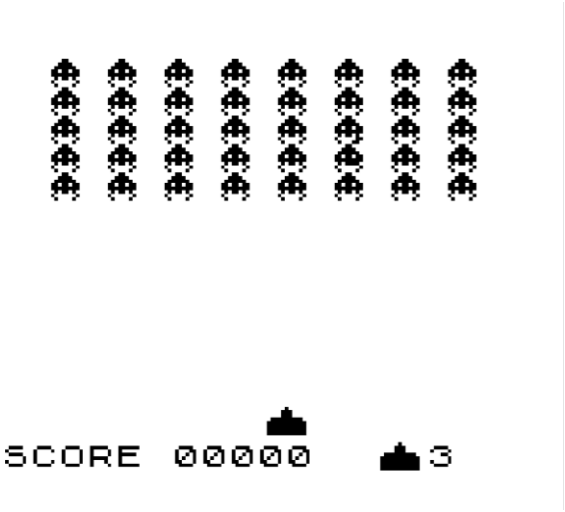


Figure 2: My Space Invaders version in C

3. DEVELOPMENT TOOLS

3.1. GB STUDIO

GB Studio is a game development engine created by Chris Maltby that enables users to create games for the Game Boy platform without requiring any prior programming knowledge. The engine utilizes a visual scripting language, which allows for the creation of interactive elements in games through the manipulation of pre-defined blocks of code. This approach simplifies the game development process and makes it accessible to a wider audience. In addition to the visual scripting language, GB Studio also includes a level editor and an in-engine music creator to aid in the development process, further streamlining the game creation process. These tools make GB Studio an accessible and user-friendly option for those interested in creating games for the Game Boy platform.[8]

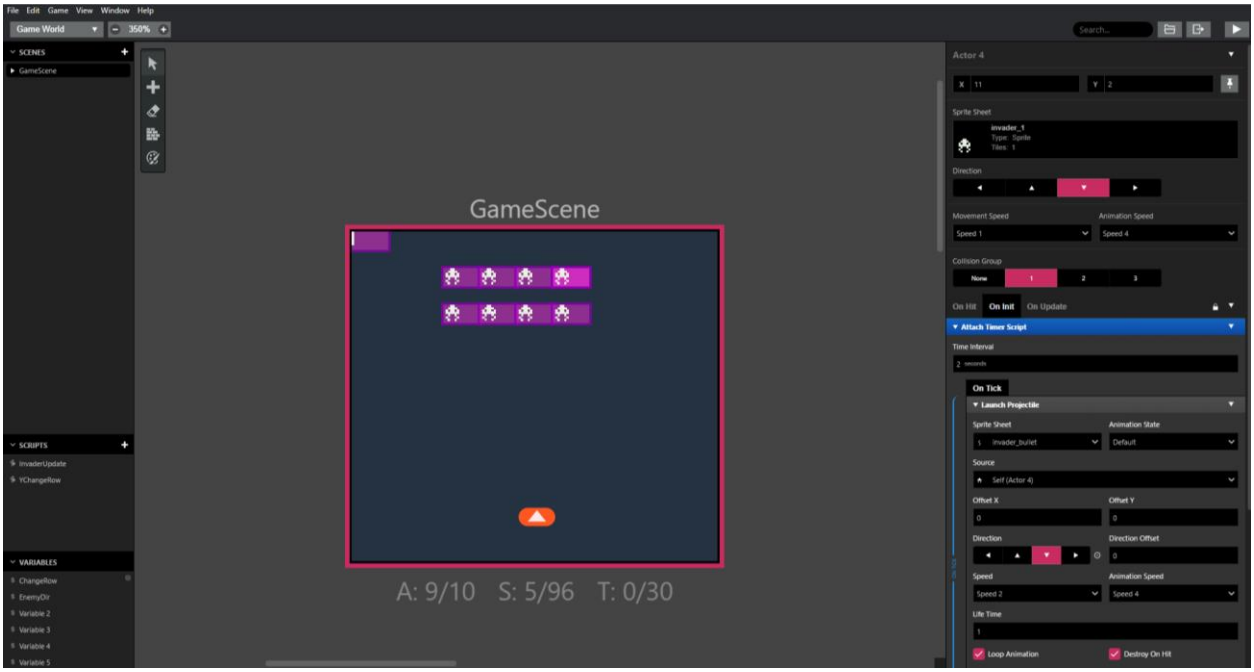


Figure 3: Scenes view in GB Studio

3.2. C + GBDK

GBDK-2020 (Game Boy Development Kit) is a cross-platform development kit that includes libraries, toolchain utilities and the SDCC compiler suite for C. It is made for any sm83 and z80 based gaming consoles, but with the Game Boy as main purpose. However, it also supports the Analogue pocket, Sega Master System, Sega Game Gear and some other consoles.[9] GBDK-2020 is extensible with ZGB or Retr0 GB, both being game engines written with GBDK as its core, as well as multiple other tools.[10]
The compiler suite it offers, SDCC, “is a retargetable, optimizing Standard C (ANSI C89, ISO C99, ISO C11) compiler suite that targets the Intel MCS51 based microprocessors (8031, 8032, 8051, 8052, etc.), Maxim (formerly Dallas) DS80C390 variants, Freescale (formerly Motorola) HC08 based (hc08, s08), Zilog Z80 based MCUs (Z80, Z180, SM83, Rabbit 2000, 2000A, 3000A, TLCS-90), Padauk (pdk14, pdk15) and STMicroelectronics STM8.” [11]
The kit, in conjunction with the SDCC compiler, allows for the creation of ROMs that can be run on both the hardware and within an emulator. To effectively develop software using this method, it is recommended to utilize an emulator for testing, execution, and debugging. Several options are available, including BGB, which is well-regarded for its accuracy and comprehensive debugging tools. Another highly recommended emulator is Emulicious, which offers similar advantages in terms of accuracy and debugging, but also includes an added benefit in the form of a VSCode(Visual Studio Code) plugin, enabling the ability to run the code directly from the editor and providing support for C source code debugging.[10] In light of these considerations, the latter emulator was chosen for use in this research project.

3.3. ASSEMBLY FOR GAME BOY

Writing assembly language for the Game Boy requires the use of a toolchain called RGBDS. RGBDS is a collection of programs that are used to assemble .asm files into ROMs, link them together, and fix the header. Additionally, RGBDS includes a program that can convert graphics. In order to properly write assembly language for the Game Boy, it is also necessary to use the "hardware.inc" file. This file is an include file that provides constants that allow for easy interaction with the Game Boy hardware registers.[12]
Besides this, VSCode with Emulicious will be used again for writing the code and running the ROM. Debugging is done in Emulicious’ built-in debugger.

CASE STUDY AND EXPERIMENTS

1. GB STUDIO

1.1. SETUP

When first starting GB Studio, it is advisable to examine one of the pre-existing sample projects to familiarize oneself with the layout and organization of the engine. Additionally, it is beneficial to engage in experimentation by making modifications to the example project, as this serves to gain proficiency in the development environment.

1.2. SPRITES

GB Studio does not have a built-in sprite editor like other game engines. Instead, it utilizes PNG images as sprites, which must be created using the correct color palette as specified in the engine's documentation. To facilitate this process, GB Studio offers a plugin for Photoshop and Aseprite, which provides the correct palette swatches for use in these image editing software. However, it is still possible to create sprites using other image editing software, as long as the correct colors are used. The engine does provide an editor for the properties of sprites, allowing for the manipulation of attributes such as flipping and animation speed.

When utilizing this engine, it is important to note that the default size for sprites is 16x16 pixels. This size constraint can potentially impede the ability to display a significant number of sprites on the screen. To circumvent this limitation, the solution is to use 8x8 sprites instead. Importing an 8x8 image as the sprite, the engine will not interpret the whitespace as part of the sprite, thus allowing for a higher number of sprites that can be displayed.

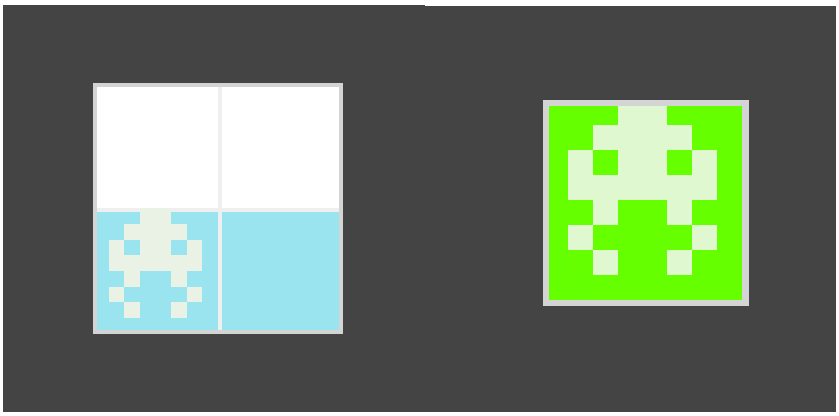


Figure 4: 8x8 sprite displayed in GBStudio

Figure 5: Invader sprite in the GBStudio color palette.

1.3. PLAYER MOVEMENT AND SHOOTING

When starting an empty project, developers can choose the “Shoot ‘em up” template from the pre-made templates. This template provides the correct movement for the player character without any more effort. The scene holds all the code for the player. Making the player shoot is quite straightforward as there is a built in “Launch projectile” event that can be attached to the button press. This event requires input of a sprite for the projectile, a designated direction, and additional parameters. To implement enemy characters, a new actor can be added to the scene and placed within the same collision group as the projectiles, enabling the enemy to be hit by the player's shots.

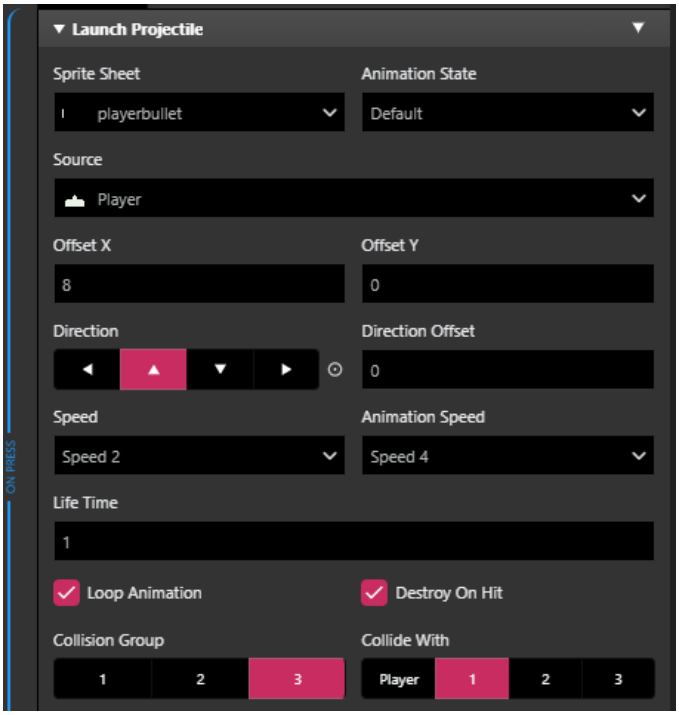


Figure 6: Launch projectile code block.

1.4. ENEMIES

Adding an enemy object and implementing its movement and shooting functionality is a relatively straightforward task. A move script can be attached to the enemy object, and its movement can be regulated by checking the position of the object in relation to the edges of the game screen.

The ability to shoot can be added by implementing a timer and a fire projectile script. However, when multiple enemy actors are introduced and given the same scripts through duplication, it becomes apparent that there are limitations to the engine. Specifically, only the last invader actor is able to shoot, and shooting any of the invaders causes the shooting to stop for all of them.

This issue can be attributed to there being a singular global timer that controls the shooting. As a result, the shooting script in the timer is only applied to the last invader actor spawned, making only one of them able to shoot. Additionally, the timer does not accept a variable for the time, making it impossible to randomize the shooting intervals.

One potential solution to this problem would be to create an array of the invader actors and have the timer choose a random invader to shoot from when it is time to shoot. However, the engine being used does not provide a way to create an array of objects. Furthermore, alternative approaches such as updating multiple actors at once resulted in noticeable delays in the program as well as not being properly functional either.

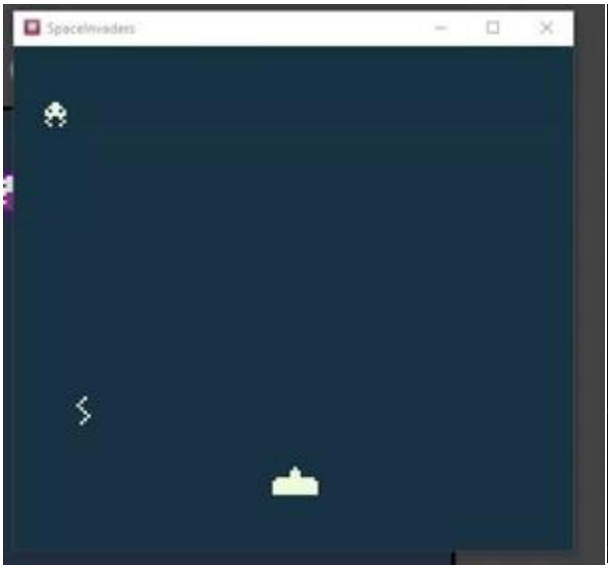


Figure 7: One invader with shooting and player character

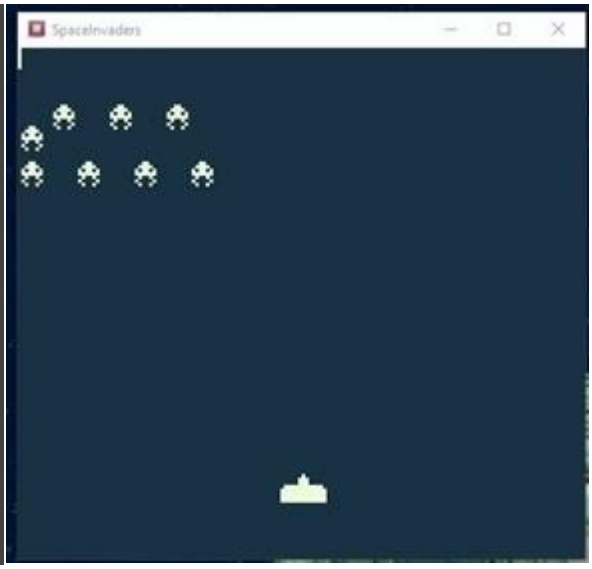


Figure 8: Multiple invaders updating when hitting edge of screen.

2. C + GBDK

2.1. SETUP

Prior to writing code, it is necessary to establish certain configurations within VSCode. In order to utilize the functionality of the Emulicious emulator, the installation of its corresponding plugin is required as an initial step. Additionally, it is highly recommended to create a launch configuration for debugging with Emulicious. Furthermore, the creation of a make.bat file, containing the commands necessary for compiling code into a ROM and its associated files, is also necessary. These steps can then be automated through the creation of a task within VSCode and subsequently linked as a pre-launch task within the previously mentioned launch configuration for the Emulicious emulator.

2.2. SPRITES

Sprites are represented as arrays of hexadecimal values. These values are either manually entered by the developer or generated through specialized software such as image editing software or image conversion software. Once the hexadecimal values have been determined, they must be initialized at the start of the program so that the game knows which image data to use for each sprite. Additionally, background tiles are created in a similar manner, however, they are initialized differently. Furthermore, both sprites and background tiles must be separately enabled using specific constants.

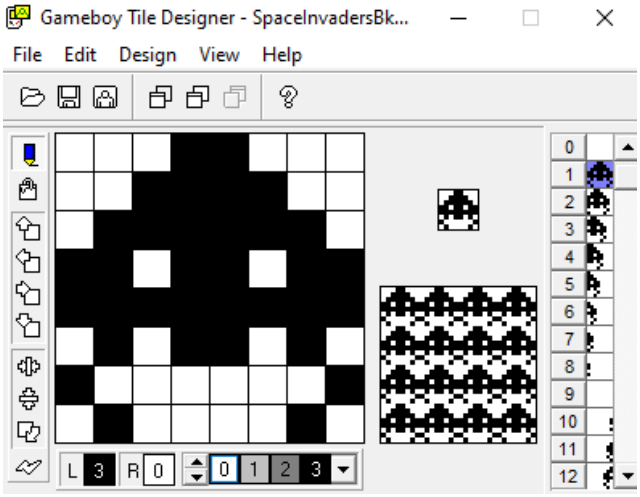


Figure 9: Game Boy Tile Designer

2.3. PLAYER MOVEMENT AND SHOOTING

The creation of a moving player character in this method is not as instantaneous as with GB Studio. In C with GBDK the function provided by GBDK to set a sprite's location is used. When your object exists out of multiple sprite tiles, this is called a metasprite. These metasprites do not present a greater difficulty in terms of movement. This is because the additional sprites can be easily manipulated in relation to the primary sprite.

One common approach for implementing bullet mechanics in otherwise object-oriented environments is to use dynamic memory allocation, however this is not an optimal solution for the Game Boy due to its limited memory capabilities. A more suitable approach would be to use a global variable and a Boolean flag to determine the active state of the bullet. This allows for efficient management of memory resources and the ability to easily deactivate or reactivate the bullet when necessary.

2.4. ENEMY DRAWING

Due to the Game Boy only supporting a maximum of 40 sprites on the screen and there being exactly 40 invaders poses a problem as traditional sprite-based rendering techniques would exceed the available resources. To overcome this limitation, the invaders are drawn as part of the background layer, which utilizes tiles of 8x8 pixels that can be set at specific positions on the screen. However, as the invaders move in increments of one pixel at a time, it is necessary to create background tiles for each possible position of the invaders. Furthermore, before drawing the invaders to the background, it is necessary to initialize all background tiles to be empty. Failure to do so will result in the display of garbage data on the screen. It is common practice to leave the tile at index 0 of the background data blank for this purpose.

Figure 10: Background tiles as displayed in Emulicious' VRAM viewer.

2.5. ENEMY MOVEMENT

In order to effectively move the invader on the background layer, a specific method must be employed to properly move on a pixel-by-pixel basis between tiles on the background. The full tile is initially drawn at the invader's current position and is subsequently replaced by a tile that has been offset by one pixel with each step. In this way, the tile adjacent to the current position will be the corresponding tile. Additionally, a count is kept of the number of steps that have been taken. Once enough steps have been taken, the invader is moved to the next tile position and the count is reset.[13] It is important to note that the offset from the current tile in memory to the corresponding tile is dependent on the direction of movement.

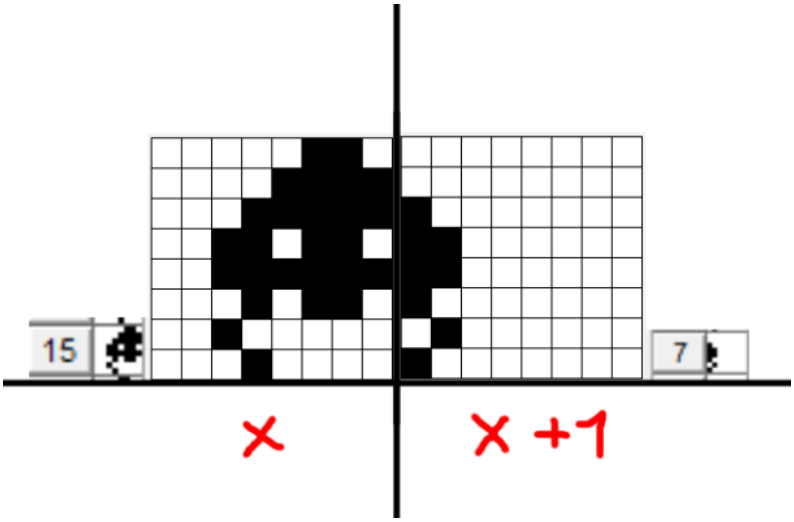


Figure 11: Invader moving from one background tile into the next

```
else if (invaders[i].slide > 0)
{
    set_bkg_tile_xy(invaders[i].x, invaders[i].y, invaders[i].spriteId + 16 - invaders[i].slide);
    set_bkg_tile_xy(invaders[i].x + 1, invaders[i].y, invaders[i].spriteId + 8 - invaders[i].slide);
}
else if (invaders[i].slide < 0)
{

    set_bkg_tile_xy(invaders[i].x, invaders[i].y, invaders[i].spriteId - invaders[i].slide);
    set_bkg_tile_xy(invaders[i].x - 1, invaders[i].y, invaders[i].spriteId + 8 - invaders[i].slide);
}
```

Figure 12: Invader drawing code in C+GBDK.

2.6. SHOOTING THE ENEMIES

To be able to accurately hit the invaders, their positions must first be converted from background tile coordinates to pixel coordinates. This is achieved by multiplying the tile coordinates by 8. However, the origin of the screen does not correspond to the same origin position for the sprites. Instead, the origin screen positions for those are outside of the boundaries of the screen (8 pixels for X, 16 pixels for Y), this is so the developer can place unused sprites off screen.[4] This offset must be taken into consideration when calculating the accurate position of the invaders. Additionally, it is necessary to consider the slide amount of the invaders, as they may not be aligned with a single tile. Therefore, the slide amount must be added or subtracted (depending on the move direction of the

invaders) from the previously calculated position. Once this is done, it is a matter of determining whether the bullet is within the boundaries of one of the invaders to make a successful hit.

2.7. ENEMIES SHOOTING THE PLAYER

The aforementioned positions for the invaders in pixels can now be used as the positions to spawn the bullets the invaders shoot at the player. To manage the number of bullets on the screen, a small pool is created with a maximum capacity that corresponds to the number of bullets that the invaders can simultaneously shoot. When a bullet is to be spawned, the system first checks if there are any inactive bullets within the pool that can be used, and if not, no new bullet will be spawned. A random invader is then selected, and the bullet is spawned at the position of the chosen invader.

2.8. HUD

The heads-up display (HUD) is a feature that is implemented as an additional layer on top of the primary screen. This layer is commonly referred to as the "window layer" and serves to display supplementary information to the user. To display text on the HUD, a font must be loaded into memory. In order for this to be achieved, the background tiles have to be repositioned in memory to a location after the font tiles. To display a score, the numerical value should be divided into its individual digits. These digits are then mapped to corresponding tiles on the window layer, resulting in the visual representation of the score. The window layer initially covers the entire screen, obscuring the background. However, it can be easily repositioned to display the score and remaining lives at the bottom of the screen.

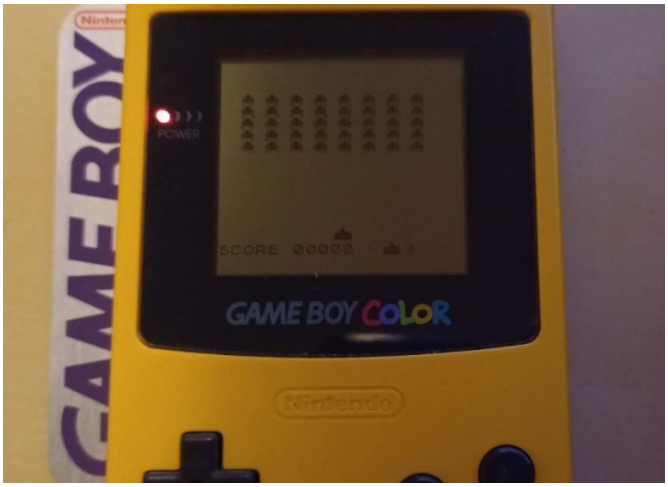


Figure 13: C+GBDK end project, working on hardware.

3. ASSEMBLY

3.1. SETUP

Before starting, it is important to install RGBDS and create a make.bat file that will facilitate the assembly of the code into a ROM format. Additionally, it is imperative to acquire and incorporate the hardware.inc file into the program. Furthermore, a similar configuration as used in the method of programming with C and GBDK in VSCode with Emulicious will be utilized.

When working with assembly language, it is important to have a thorough understanding of the hardware for which the code is being written. To gain this understanding, it is recommended to consult the unofficial documentation for the Game Boy hardware, commonly referred to as the "pan docs."[14] Additionally, for those new to assembly language, it is advisable to begin by following tutorials and seeking guidance from experienced developers within the community. A useful resource for this purpose is the GBDev community Discord server, where individuals can connect with experts who have contributed to popular resources such as RGBDS and hardware.inc, as well as the pan docs.

3.2. SPRITES

As with C, sprites are represented as an array of hexadecimal data. This data can be obtained using the same methods as with the C project. However, unlike in the C and GBDK project, the data must be manually copied to the appropriate memory addresses in VRAM for the sprites to be usable. Additionally, in order to use the graphics as objects within the game, they must be transferred to OAM, which comprises of four bytes per object. These bytes correspond to the X and Y coordinates, the tile ID, and special attributes of the object, respectively.

3.3. PLAYER MOVEMENT AND SHOOTING

Player movement can be achieved by altering the X attribute of the OAM address for the player's sprite. The corresponding tile for the player will also have to be moved accordingly. The same approach can be applied to moving a bullet object on the screen, which will be spawned on the location above the player and move upwards when set to alive and put outside of the screen's boundaries otherwise. However, there is an issue with accessing OAM; if OAM is accessed when it is inaccessible, it can lead to corruption of the data. To address this issue, this project employs the use of a "wait for VBlank" function, which is called each time before accessing OAM. This method, while functional, is relatively slow and can negatively impact the structure of the code by causing frequent pauses in execution. A more optimal solution would be to implement OAM Direct Memory Access (DMA) transfer, however, due to constraints on time and expertise, this approach was not implemented in the project.

Figure 14: OAM Corruption

3.4. ENEMY DRAWING

To draw the enemies on the screen, a method similar to the one used for copying data to memory, such as the one utilized in the previously mentioned C project, is employed. Instead of copying memory, this technique involves copying the necessary tile ID to the VRAM address of the desired background tile. The process previously implemented in the C project can be translated into assembly language in order to effectively create moving invaders in the game.
After implementing the above, it became apparent that certain tiles were not being rendered correctly when the edge of the screen was reached. Further investigation revealed that this was due to the behavior of the Game Boy's PPU (Pixel Processing Unit). Specifically, it was found that the PPU has 4 different “modes”. Mode 0 being HBlank, mode 1 VBlank, mode 2 Searching OAM and mode 3 transferring data to the LCD controller.[14] In mode 3, the VRAM is locked. As a result, when VRAM is accessed in this mode, no changes are made to the display. To address this issue, it was necessary to include a check that determined whether the PPU was in mode 0 or 1 before attempting to modify VRAM. Additionally, mode 2 was skipped in this check as it is a very short mode that comes right before mode 3 and checking for modes 0 and 1 is more efficient in terms of processing time.

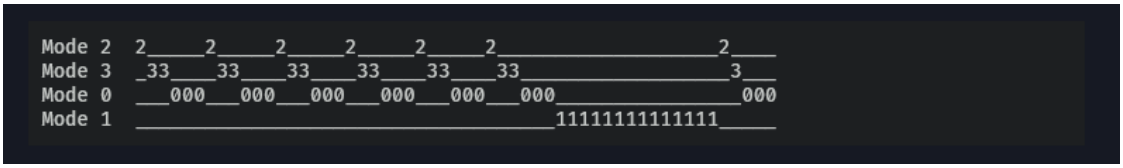


Figure 15: PPU Modes

3.5. SHOOTING THE ENEMIES

In order to shoot the enemies, it is necessary to determine their position. To accomplish this, an array can be implemented that stores the current VRAM address of each invader, as well as a binary value indicating their status (alive or dead). For this system to function properly, the loop responsible for rendering the invaders on the screen must be rewritten to utilize the addresses stored within the array, and the array must be updated in real-time as the invaders move.

In the update function of the bullet, the VRAM address inside of the array is checked each frame. To determine the corresponding X and Y positions, a method is employed to extract the tile X and Y coordinates from the VRAM address, as this address contains these coordinates encoded within it. Subsequently, the extracted coordinates are then used to calculate the corresponding position in pixels, which can then be compared with the position of the bullet. If a collision is detected, the "alive byte" is set to 0, which is then checked in the draw loop. This results in the replacement of the sprite for the invader with a blank, indicating that it has been hit.

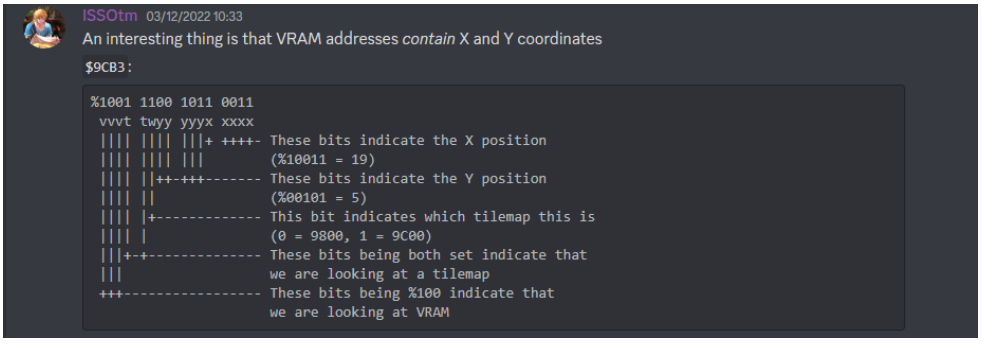


Figure 16: VRAM address containing X and Y position explained.

CONCLUSION & FUTURE WORK

CONCLUSION

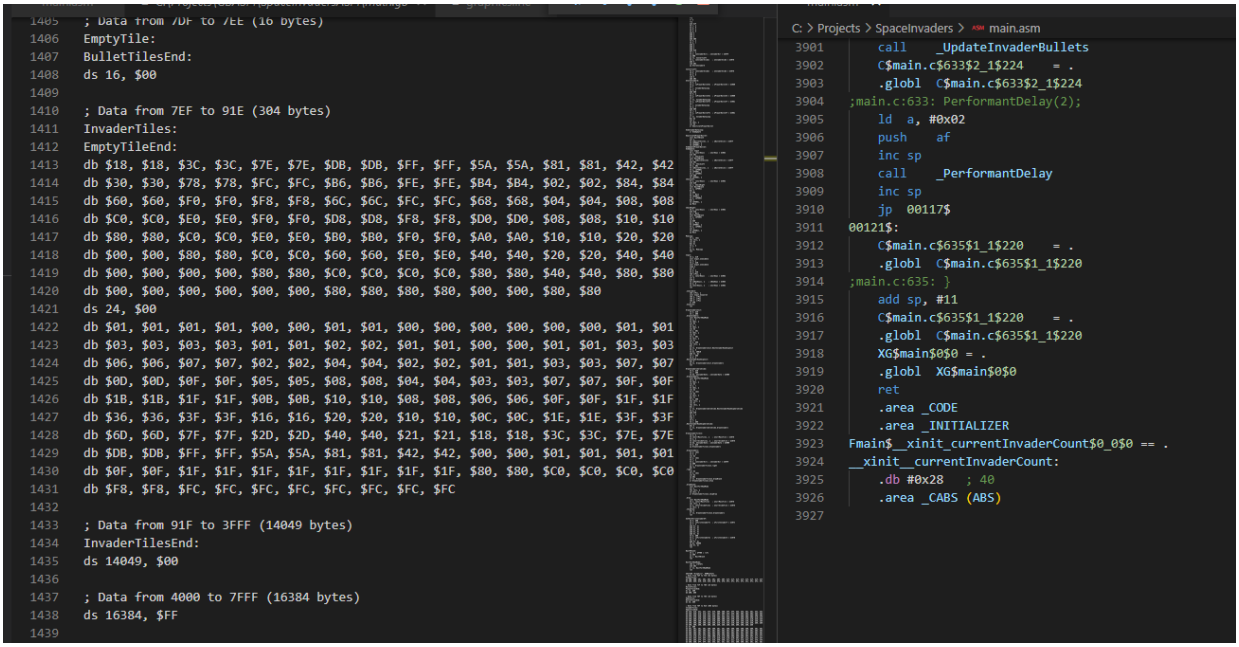
In this research project, three methods were compared for the creation of software for the Game Boy. The first method evaluated was GB Studio, a graphical tool with a user-friendly interface and a more visual coding language. While this method offered a simple and accessible way to create games, it was found to lack certain features necessary for the development of a more complex game, such as Space Invaders. GB Studio is deemed appropriate for beginners in game development or those looking to create small, novelty projects for personal enjoyment. Additionally, it is well-suited for creating games in the same format as the provided example projects.

The second method evaluated was the use of C with the GBDK library. This method offered users a high degree of freedom and control, while also providing the relative ease of use of the GBDK functions and a structured code with C. For individuals with experience in programming languages such as C++, C#, Python, or others, this method would be the most similar to what they are already familiar with.

The final method compared was the use of assembly. This method offers users complete control over the hardware, but at the cost of requiring a thorough understanding of the hardware to effectively utilize it. Assembly is a difficult language to learn, but those skilled in its use are able to write more performant code and accomplish tasks that are otherwise not possible with other methods.

Performance wise, GB Studio has been observed to exhibit significant delays when updating multiple actors simultaneously. This contrasts with using C with the GBDK library, which has been shown to perform well in terms of visual responsiveness. It is theoretically possible for assembly language to achieve even greater performance improvements over C and GBDK, but this requires a high degree of skill and expertise on the part of the developer. As a result, in cases where the developer lacks the necessary proficiency, the use of assembly language may result in poor performance, as evidenced by a noticeable delay in the example of a bullet being shot in the current project.

Upon comparison of the generated assembly code of the C+GBDK program and the assembly program, it can be observed that the assembly project exhibits a significantly lower number of instructions in comparison to the assembly generated by the C project. Specifically, the assembly program contains 1438 instructions, while the assembly generated by the C program comprises 3926 instructions. This difference in instruction count serves as an indication of the relative efficiency of the two approaches. It has to be taken into account however that the assembly example lacks certain features present within the C one.



```
; Data from 7DF to 7EE (16 bytes)
EmptyTile:
BulletTilesEnd:
ds 16, $00

; Data from 7EF to 91E (304 bytes)
InvaderTiles:
EmptyTileEnd:
db $18, $18, $3C, $3C, $7E, $7E, $DB, $DB, $FF, $FF, $5A, $5A, $81, $81, $42, $42
db $30, $30, $78, $78, $FC, $FC, $B6, $B6, $FE, $FE, $B4, $B4, $02, $02, $84, $84
db $60, $60, $F0, $F0, $F8, $F8, $6C, $6C, $FC, $FC, $68, $68, $04, $04, $08, $08
db $C0, $C0, $E0, $E0, $F0, $F0, $D8, $D8, $F8, $F8, $D0, $D0, $08, $08, $10, $10
db $80, $80, $C0, $C0, $E0, $E0, $B0, $B0, $F0, $F0, $A0, $A0, $10, $10, $20, $20
db $00, $00, $80, $80, $C0, $C0, $60, $60, $E0, $E0, $40, $40, $20, $20, $40, $40
db $00, $00, $00, $00, $80, $80, $C0, $C0, $C0, $C0, $80, $80, $40, $40, $80, $80
db $00, $00, $00, $00, $00, $00, $80, $80, $80, $80, $00, $00, $80, $80
ds 24, $00
db $01, $01, $01, $01, $00, $00, $01, $01, $00, $00, $00, $00, $00, $00, $01, $01
db $03, $03, $03, $03, $01, $01, $02, $02, $01, $01, $00, $00, $01, $01, $03, $03
db $06, $06, $07, $07, $02, $02, $04, $04, $02, $02, $01, $01, $03, $03, $07, $07
db $0D, $0D, $0F, $0F, $05, $05, $08, $08, $04, $04, $03, $03, $07, $07, $0F, $0F
db $1B, $1B, $1F, $1F, $0B, $0B, $10, $10, $08, $08, $06, $06, $0F, $0F, $1F, $1F
db $36, $36, $3F, $3F, $16, $16, $20, $20, $10, $10, $0C, $0C, $1E, $1E, $3F, $3F
db $6D, $6D, $7F, $7F, $2D, $2D, $40, $40, $21, $21, $18, $18, $3C, $3C, $7E, $7E
db $DB, $DB, $FF, $FF, $5A, $5A, $81, $81, $42, $42, $00, $00, $01, $01, $01, $01
db $0F, $0F, $1F, $1F, $1F, $1F, $1F, $1F, $1F, $1F, $80, $80, $C0, $C0, $C0, $C0
db $F8, $F8, $FC, $FC, $FC, $FC, $FC, $FC, $FC, $FC

; Data from 91F to 3FFF (14049 bytes)
InvaderTilesEnd:
ds 14049, $00

; Data from 4000 to 7FFF (16384 bytes)
ds 16384, $FF
```

C: > Projects > SpaceInvaders > main.asm

```
    call    _UpdateInvaderBullets
    C$main.c$633$2_1$224    = .
    .globl  C$main.c$633$2_1$224
;main.c:633: PerformantDelay(2);
    ld  a, #0x02
    push    af
    inc sp
    call    _PerformantDelay
    inc sp
    jp  00117$
00121$:
    C$main.c$635$1_1$220    = .
    .globl  C$main.c$635$1_1$220
;main.c:635: }
    add sp, #11
    C$main.c$635$1_1$220    = .
    .globl  C$main.c$635$1_1$220
    XG$main$0$0 = .
    .globl  XG$main$0$0
    ret
    .area _CODE
    .area _INITIALIZER
Fmain$__xinit_currentInvaderCount$0_0$0 == .
__xinit__currentInvaderCount:
    .db #0x28   ; 40
    .area _CABS (ABS)
```

Figure 17: Line count comparison between generated assembly of the C+GBDK and Assembly project

FUTURE WORK

The current assembly project for Space Invaders lacks certain features that are present in the C version of the project. Both projects are considered to be minimalistic and do not perform as accurately as the original Space Invaders game. However, with further development, either project has the potential to evolve into a full clone of the original game.

It should be noted that, as a first project, the code written in the assembly project is disorganized and lacking in performance. Through the application of skills acquired during the development of this project, or through additional tutorials and research on assembly programming for the Game Boy and its hardware, the code can be significantly improved and refined.

An alternative approach to this project could be to design and develop a game using GB Studio, and then use that as a model to replicate the game using the other methods, as opposed to choosing a pre-existing game beforehand.

Furthermore, it is deemed to be of interest to continue research on the Game Boy and its various models, as well as other old hardware and the proper development techniques for these platforms. This will aid in further understanding and optimization of software development for these older systems.

BIBLIOGRAPHY

[1] Wikipedia contributors. (2023 January 16), “Game Boy - Wikipedia.” https://en.wikipedia.org/w/index.php?title=Game_Boy&oldid=1134052561 (accessed Jan. 19, 2023).

[2] Wikipedia contributors. (2023 January 19), “List of Game Boy games - Wikipedia.” https://en.wikipedia.org/w/index.php?title=List_of_Game_Boy_games&oldid=1134540656 (accessed Jan. 19, 2023).

[3] Michael Steil, “The Ultimate Game Boy Talk,” *https://www.youtube.com/watch?v=HyzD8pNlpwI&t=2s*.

[4] ISSOtm, “GB ASM Tutorial.” https://eldred.fr/gb-asm-tutorial/part2/objects.html (accessed Jan. 16, 2023).

[5] “Gameboy (DMG & GBC) Development Kit Hardware · RetroReversing.” https://www.retroreversing.com/gameboy-development-kit-hardware/ (accessed Jan. 19, 2023).

[6] “EZ-FLASH.” https://www.ezflash.cn/product/ezflash-junior/

[7] Wikipedia contributors. (2022 December 17)., “Space Invaders - Wikipedia.” https://en.wikipedia.org/w/index.php?title=Space_Invaders&oldid=1127852595

[8] Chris Maltby, “GB Studio.” https://www.gbstudio.dev/

[9] “https://github.com/gbdk-2020/gbdk-2020.”

[10] gbdev contributors, “gbdev.io.” https://gbdev.io/resources.html.

[11] “SDCC - Small Device C Compiler.” https://sdcc.sourceforge.net/

[12] “RGBDS.” https://rgbds.gbdev.io/

[13] Larolds Jubilant Junkyard, “Space Invaders Tutorial for Gameboy.”

[14] “Pan Docs.” https://gbdev.io/pandocs/

Parts of this paper used the help of ChatGPT for the purpose of formatting and writing style.

APPENDICES

Space Invaders in C with GBDK: https://github.com/narbys/SpaceInvaders

Space Invaders in Assembly: https://github.com/narbys/SpaceInvadersASM