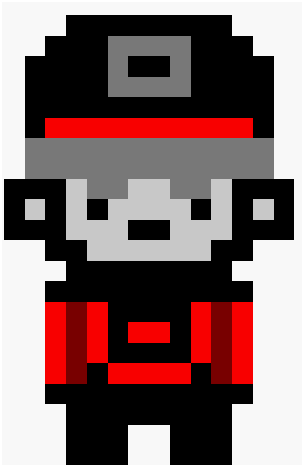


MIE438 Term Project Report:

The Creation of *Bit's Adventures* for the Game Boy Advance

19/04/2021

Connor Glossop (1005228576), Catherine Glossop (1004860741), Joel Kahn (1004496529), Julia Chae (100495554)

1.0 Bit's Adventures: A Retro Game Boy Advance RPG

The rise in popularity of retro game consoles and gaming fueled by nostalgia has resulted in emulated hardware and manuals for their development becoming much more accessible. Our team decided to create our own retro Game Boy game using these tools. Through this experience, we have been able to navigate both the hardware and software limitations of the Game Boy platform and the technical and creative process of developing a game from scratch.

Figure 1. Example gameplay

Bit's Adventure is an interactive RPG which draws inspiration from classic games like Pokemon Fire Red. The story follows Bit, a kid who ends up inside his Game Boy Advance and is tasked with stopping the villain Mal from overwriting important code and destroying the console's microprocessor. Bit meets a wise Frog who tells him about the fall of Game Boy Square and marches on to defeat the enemies in a pong-style minigame. *Bit's Adventures* can be played on the mGBA emulator and a README is provided in the appendix for download and play instructions. In Appendix B, there is also a link to a video which demonstrates the game play.

This report covers the hardware specifications of the Game Boy embedded system, a detailed description of the game implementation and tools used, challenges, and possible future improvements to the game. Some of the topics discussed include the hardware limitations of the Game Boy Advance platform and our choice of programming language, which together led to challenges with memory allocation, graphical limitations and providing smooth gameplay given a limited framerate.

2.0 Our Embedded System

2.1 Game Boy Advance Hardware Specifications

2.1.1 mGBA Emulator

This project was completed virtually and therefore we required a platform to be able to test throughout development. To simulate the Game Boy hardware, the open source platform mGBA [1] was used. mGBA provides many tools beyond gameplay for memory, tile and map visualization.

The emulator specifically emulates the Game Boy Advance which is able to run both games from the original Game Boy and the Game Boy Color, giving more freedom for the design of the game. The specifications of the emulator are as follows [2]:

- 31 general purpose 32-bit registers
- 9 status registers
- 16 kB BIOS RAM
- 16.8 MHz 32-bit ARM7TDMI chip
- 8.388 MHz Sharp LR35902 (for backward compatibility)
- 32 kB Internal RAM + 96 kB VRAM
- Input from joypad and the A and B button

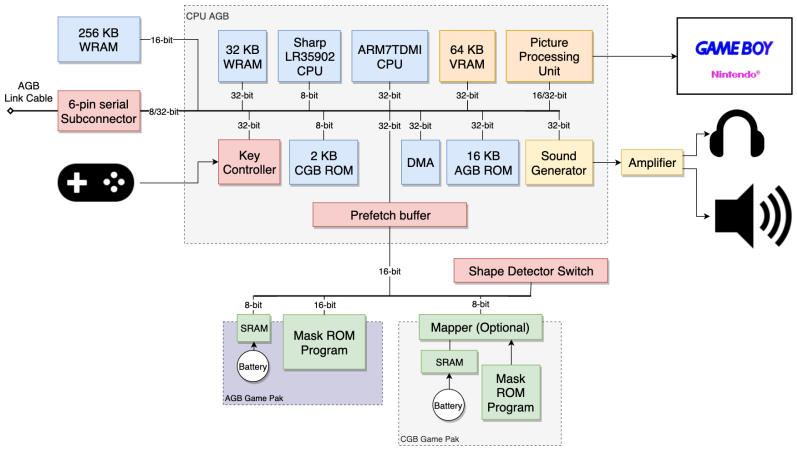


Figure 2. Architecture of the Game Boy Advance [2]

Figure 3. Input to Game Boy

2.1.2 Memory in Game Boy

The original Game Boy console and mGBA both have a separate memory model (memory locations 0x0000 to 0xFFFF) segmented into 10 distinct sections which cannot overflow from one to another [3]. The Game Boy only has 32kB from 0x0000 to 0x7FFF allocated for ROM, which is not a lot of memory especially for games with colour. In order to expand the limited space, the second half of the ROM space is switchable, meaning that different memory banks can be swapped in and out to take up that space. This is done through Memory Bank Controllers (MBC) which are in special game cartridges. The switchable banks allow up to 125 banks, significantly increasing the memory capacity of the embedded system. Additionally, 96 kB is allocated for VRAM that can be accessed freely during gameplay.

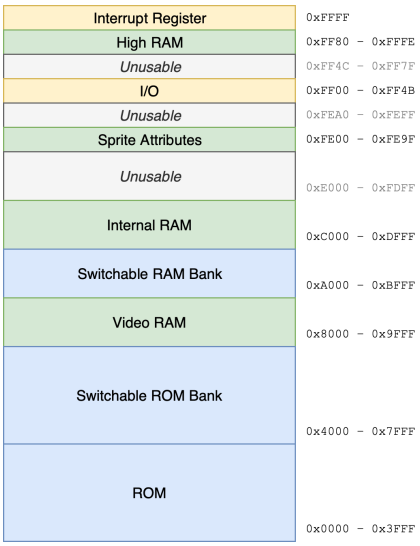


Figure 4. Memory Structure [3]

2.1.3 Moving to Physical Hardware

While this provides a realistic replacement for the actual Game Boy Advance hardware, there will be some steps required to go from the ROM file to a game cartridge for a physical Game Boy. An option for flashing the ROM is using a Flash Cart. To use a Flash cart, the ROM is loaded onto an SD card and is then inserted into the applicable Flash cart which acts as a converter between the SD card and cartridge storage mediums. Options include EZ Flash IV [4] and EZ Flash Omega which both have compatibility with the Game Boy Advance.

2.2 Software Tools

2.2.1 Game Boy Development Kit

To develop and compile the game, we used the Game Boy Development Kit (GBDK) [5], originally developed in the early 2000s and rebooted in a 2020 version. It provides a library of functions in C and a compiler to ease the development of a Game Boy game. Such functions as set_bkg_tiles, set_sprite_tiles etc. provide an easy way to access specific areas of memory and

initialize important data like the background and sprite information. More on its use is discussed in the following sections.

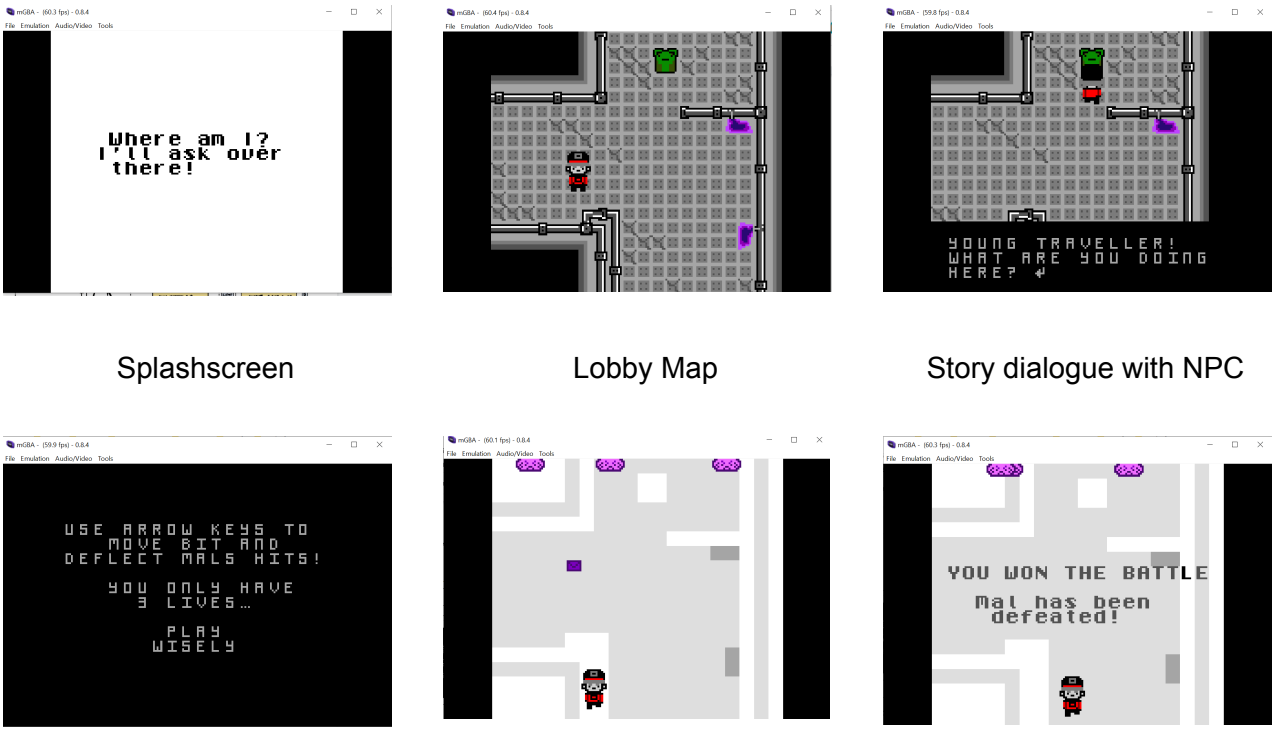
2.2.2 Building a Game Boy Game in C Youtube Series

The documentation for GBDK is sparse and a lot of base knowledge is required about the structure of the hardware and the external tools needed for development. To supplement the available documentation, we often referred to the Youtube Series “Learn how to develop your own Game Boy games” by GamingMonsters [6]. This provided a solid introduction to the basics of using GBDK and the tile and map designers discussed in section 4.1.1.

3.0 Our Game Overview

3.1 Gameflow

The game begins with a splash screen and moves into a main map where the player can move Bit to talk to the NPC on the screen. After their dialogue, the game transitions into the interactive combat level with a pong-style game. The player can repeat the game until Bit beats Mal, the villain AI. The game flows through the following stages:



Splashscreen | Lobby Map | Story dialogue with NPC

Game Description | Combat Game | Game End Screen

Figure 5. Gameflow overview

The gameflow can also be illustrated as a state machine:

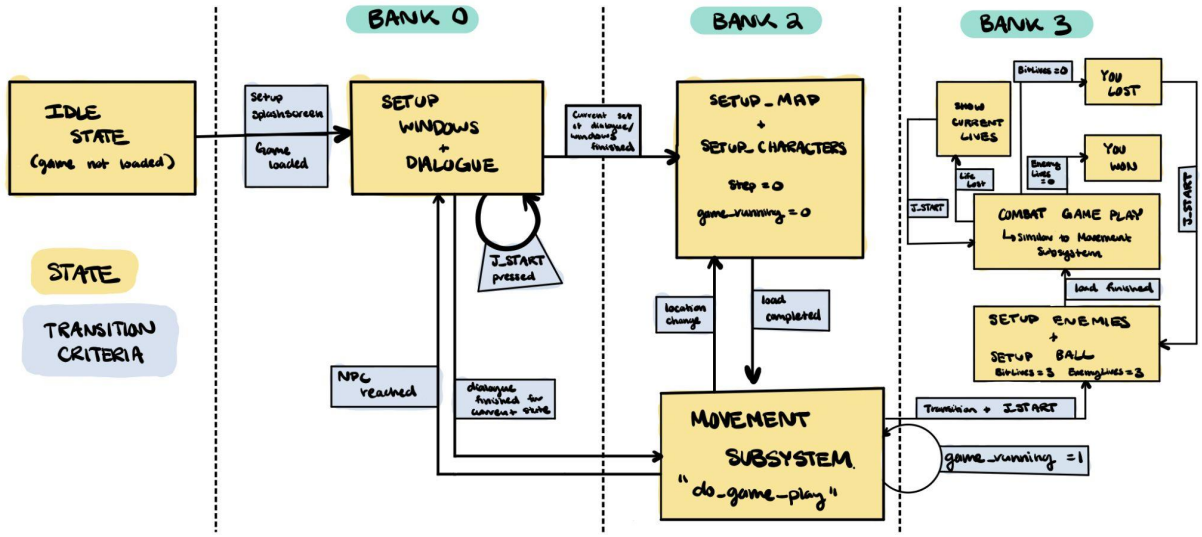


Figure 6. State machine of Bit's Adventures

The main sections in Figure 6 to observe are the transitions between user input states and loading states. The various code banks used are also highlighted in the diagram to better visualize the structure of the game within memory.

4.0 Detailed Bit's Adventures Implementation

4.1 Game Graphics

4.1.1 Designing the Graphics

To construct the graphics, two platforms were used that were developed in the late 90s. The Game Boy Tile Designer, created by Harry Mulder, allows you to create a set of desired tiles for both sprites and backgrounds. The final tileset can then be exported as a C file and included in the main script to be used for gameplay. Similarly, the Game Boy Map Builder can be used to organize tiles into maps [7, 8].

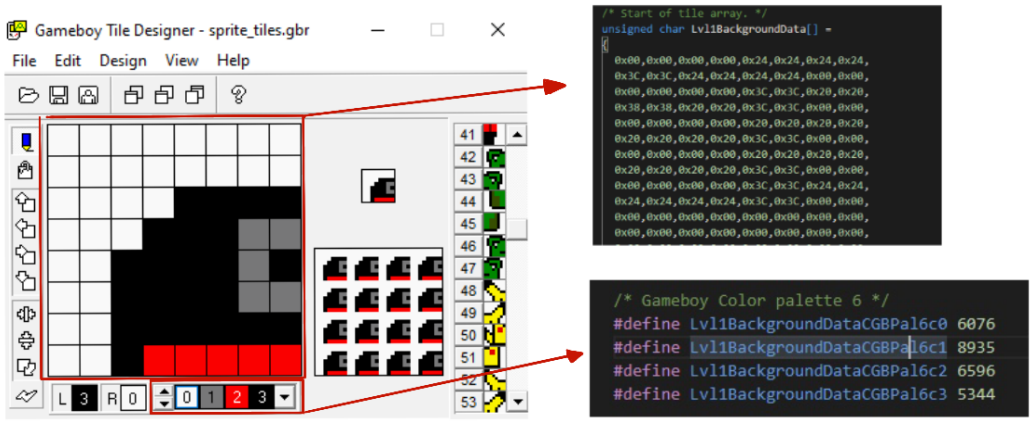


Figure 7. Graphical Interface to code definitions of sprite data

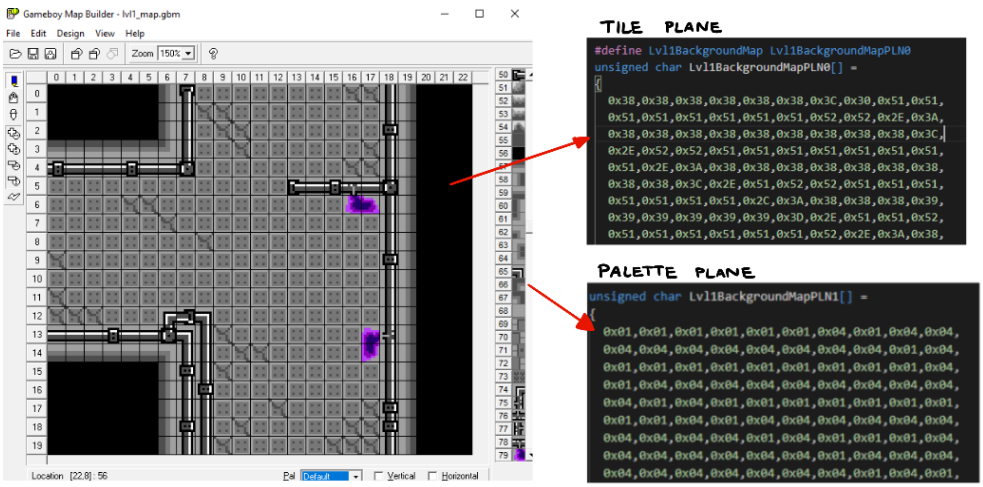


Figure 8. Planes used for graphics display

In C, the tiles are stored as an array of unsigned chars with hex values that describe the corresponding 8 x 8 tile. When using colour for Game Boy Color games, the colour palette information is also included as sets of 4 constants corresponding to the colour values used in the designer. The map is stored as two separate unsigned char arrays of the same length, one for the tile ids at each location on the map and one for the palette ids at each location on the map.

4.1.2 From Memory to LCD

The Game Boy Advance Emulator has a 96 kB VRAM and a 16 bit data bus that transfers data from memory to the peripherals. The VRAM is organized into the graphics core called the Picture Processing Unit or PPU [9] which does the actual rendering of the images from memory. Within this 96 kB structure, 64 kB stores the background data, 32 kB stores the sprite data, 1 kB 32-bit OAM stores the sprite attributes and 1 kB 16-bit PAL RAM stores the colour palette information for the sprites and the background.

The background map files are loaded into the microcontroller's memory as two separate planes and are stored in two separate video-memory banks. The palette plane is loaded into VBK_REG 1 and the tile information plane is loaded into VBK_REG 2. This allows more memory to be available for tiles which is essential for the amount of space the colour background takes up.

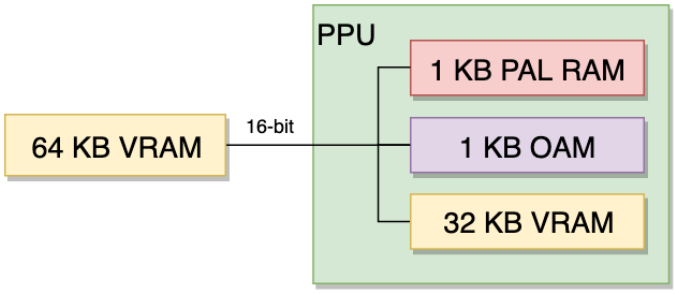


Figure 9. VRAM and PPU Structure [2]

Each set of frame information, composed of the background and sprite data, is loaded from the PPU through the 16-bit data bus. To allow for 32-bit instructions to travel on the data bus without a delay due to the 16-bit bus size, there is a "prefetch buffer"[10] that can store 8 x 16 bit values at a time, preventing any latency in the system.

The actual drawing of graphics from memory is achieved by calling a series of interrupts. The display consists of a series of scan-lines, each drawn by a call to the HDraw interrupt. When it is

complete, the HBlank interrupt is called. Once the entire image is drawn, it is 240 x 160 pixels and can display a 15 bit colour range. To create the smooth transitions between frames, the vertical blank (VBL) interrupt is triggered once all scan-lines are drawn to prevent the load of the next frame before the current frame has completed loading. This interrupt was implemented in our code to improve the flow of sprite movement by using the GBDK function wait_vbl_done(). This waits for the game to return from the VBL ISR before loading the next set of graphics, removing any choppy delays. The graphic drawing and switching process results in the game running at ~60 frames per second.

Type	Length	Cycles
Pixel	1	4
HDraw	240	960
HBlank	68	272
Scanline	HDraw + HBlank	1232
VDraw	160 * Scanline	197120
VBlank	68 * Scanline	83776
Refresh	VDraw + VBlank	280896

Figure 10. Timing of Graphical Display operations and interrupts [10]

The graphics are displayed in layers composed of the background, window and sprite layers. The sprite layer is completely separate from the background and window layers and is the most dynamic of the layers. The PPU cannot render more than 10 sprites per scan-line and 40 per frame, limiting the amount of movement that can be shown on screen at a given time. The background and window layers share a tile set and colour palette, and can be layered on top of each other.This can be useful for features such as dialogue and menus. The window layer was used for all dialogue that runs throughout the game, including interactions with the NPC and displaying instructions or results.

Figure 11. Designed map for first conversation with NPC

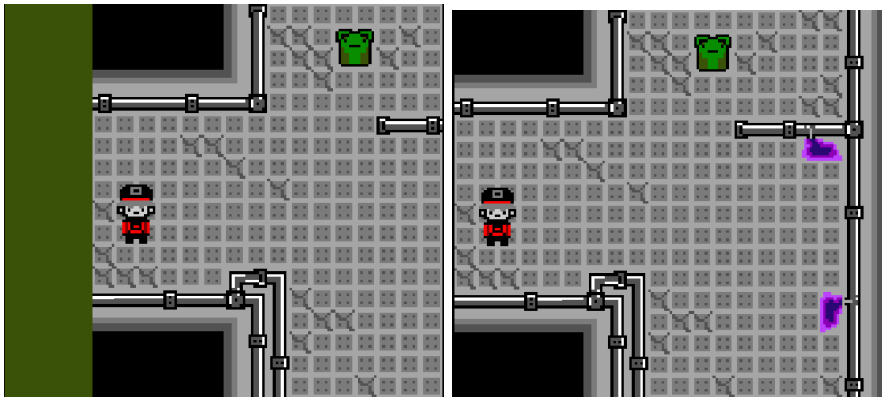
Figure 12. How dialogue appears on top of the background and sprite layers

4.2 Gameplay

4.2.1 Getting Around The Map

Movement within the game had to carefully consider a number of different factors, with a primary emphasis on the fluidity of the game. As explained above, the Game Boy screen is capable of displaying 240 x 160 pixels, or 30 x 20 tiles of 8 x 8 pixel size. The main map was 20 x 23, requiring screen scrolling to allow the player to explore the entire map. For this reason, two different types of movement were implemented - one where the Bit moves across the screen and another where Bit

stays stationary while the map scrolls underneath him (possible due to Bit being a sprite). In both cases, movement is specified to be one tile - that is, if the player presses “up” once, Bit will move one tile upwards. Careful tuning was necessary to ensure that the full background could be seen without causing graphical errors, as seen below. To ensure that the right movement option was being used, Bit’s location is stored as a global 2D array. When the player moves from one tile to the next, this location array is updated. When Bit is located at the center of the map, the screen scrolls underneath him, while if his location passes a certain threshold (either too high, low, left or right) his sprite scrolls on top of the map.



Figures 13, 14: Movement using exclusively map scrolling (left) and when both map and sprite scrolling are used (right). Note how the out-of-bounds textures (seen in green) are visible when proper scrolling has not been implemented

Collision was implemented in a similar way. As mentioned above, each background consists of two planes, with one containing the sprites at a given tile and the other containing the colour palette used. When the player attempts to move from one tile to the next, the tile id of Bit’s future location is collected. If the tile has been specified as one Bit can move to, then Bit’s location is updated and his sprite is moved. Otherwise, nothing is done.

When a movement key is pressed, the game exits its main loop to check if movement can be performed and if possible, moves the character. During this time and throughout the entire movement animation, the game does not check for key presses or perform any other calculations. This can cause some unintended effects, such as sprites updating much slower when the character is moving.

4.2.2 Defeating Mal

Bit’s fight with Mal is represented by a game similar to Pong. Bit is seen at the bottom of the screen and can only move on the left-right axis to fit with the pong-like nature of the game. Rather than have Bit move in 8 pixel chunks (one tile on the previous map), he moves 1 pixel every 8 frames (1/8th of a second) that the left or right joypad buttons are held. This provides Bit with fluid movement.

The ball is initially placed at the top-center of the screen, and then travels at a 45° angle towards the bottom left corner. When the ball hits one of the sides, it bounces off by flipping its x-velocity. Mal is represented by three pink sprites at the top of the screen. All three of the enemy sprites move from right-to-left at a constant speed, flipping direction once they hit one of the side walls. When the ball hits either Bit or one of the enemies, its y-velocity is flipped and it travels towards the opposite side of the screen.

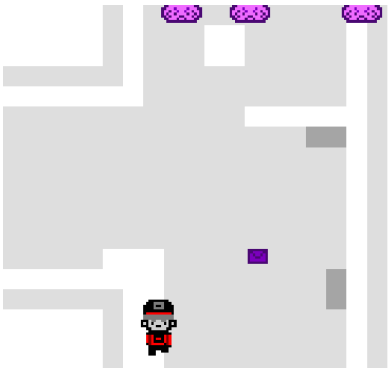


Figure 15 - Bit's combat level against Mal is highly similar to the classic game Pong

If the ball passes by either Bit or the enemies and reaches either the top or bottom of the screen, the game is reset and a life is removed from the side on which the ball passed by. The game then enters a simple loop, replaying the same combat level until either the player's or Mal's lives are reduced to zero. The ball always starts with the same location and direction while the enemies continue travelling from the same spot they stopped at the end of the previous loop cycle. If the player has no lives remaining, they are prompted to replay the game. If Mal runs out of lives, the player wins and the game finishes.

4.3 Code Integration, Organization and Compilation

4.3.1 Code Structure Overview and Motivation

When it came to integrating the maps, sprites, movement, collision and combat code discussed above, we realized that based on the requirements of our embedded system, special considerations had to be made. The final structure of our game repository with these considerations in mind is as follows:

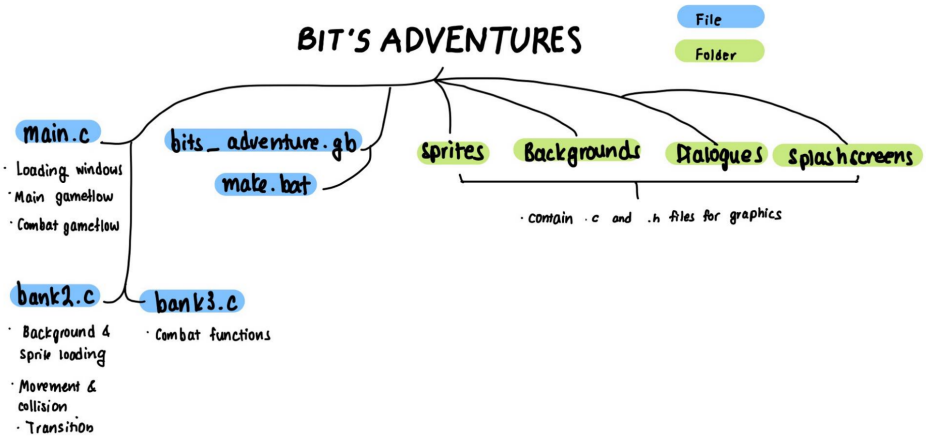


Figure 16. Repository structure of our game.

The main motivation behind our code organization was the lack of memory within the default memory model of our embedded system. As mentioned in section 2.1.2, the Game Boy console only has 32 kB of ROM memory, which is not sufficient for a large game, especially when using colour, as the colour tiles almost double the memory intake. To address this problem and to free up more room, we had to use a memory-switching method introduced in the same section as above known as banking, using Memory Bank Controller 1 (MBC1).

In the gbdk library, bank switching can be done simply by enabling it in main using the command ENABLE_RAM_MBC1, calling the function SWITCH_ROM_MBC1(X) prior to using the functions stored in bank X, and adding corresponding compilation flags for each bank files in the makefile of the game. The banks then switch in and out whenever the MBC1 command is called, expanding the amount of memory usable to us.

We split off our code into three main functions: main.c, bank2.c, bank3.c The main.c contained code which gets compiled into the permanent ROM while bank2.c and bank3.c contained code that gets stored in switchable ROM 2 and 3 banks respectively. When organizing the banks, we made sure that all groups of functions which directly relied on each other stayed in the same bank file to enhance the readability of our program.

4.3.2 Banking Challenges and Modifications Since the Proposal

Although we were able to implement banking and utilize it to fit the levels in our final program, we faced some major challenges during development. The majority of the challenges were due to GBDK's memory banking capabilities. Unfortunately, even though the gbdk library is easy to learn and very accessible, it is not the most transparent package when it comes to memory usage. In addition, based on our personal experiences along with comments on the gbdev forum and gbdk GibHub Issues pages, it became apparent that implementing banking with the gbdk library is often very difficult and can cause some issues that are hard to resolve unless you are coding at the lower level.

We ran into some of these issues during development, with a major issue being the "Memory Overflow" error between one bank into another. Although we were able to understand that we were loading in too much memory into a bank, debugging or resolving such a problem was extremely difficult, as it was hard to exactly map how the higher level C code we wrote populated the memory addresses in the Game Boy system. This was an issue we realized would be much more easily resolved if we had directly coded in a lower level language as we would have a better insight into memory usage during development and have more flexibility when switching banks. The memory errors more frequently persisted with each additional bank and for that reason, we had to scale down our original three-level-game proposal to a single level even though theoretically there was capacity in our hardware and the maps had been created.

4.3.3 Compilation: From C file to Game

The last and one of the most important parts of game development is converting the game from C code to the final playable .gb format. The code is compiled using GBDK and requires specific tags to be correctly compiled. A batch script is used to consolidate the necessary compilation actions. The compiler itself is an ANSI C Compiler based on lcc that generates relocatable code and uses a peephole optimizer, meaning the optimizer slides over the code with a window and makes direct substitutions. This provides some good cookie cutter optimizations but is not as powerful as optimizing manually [11].

The major manipulation of the code and interaction with hardware is through the tags used in each line of the compilation. Take this line for example:

```
C:\gbdk\bin\lcc -Wa-l -Wl-m -Wl-j -DUSE_SFR_FOR_REG -c -o main.o main.c
```

This line has the additional tag -DUSE_SFR_FOR_REG which allows a SFR to be used instead of a general purpose one. This SFR connects the microcontroller to the peripherals of the microprocessor.

```
C:\gbdk\bin\lcc -Wa-l -Wl-m -Wl-j -Wf-bo3 -c -o bank3.o bank3.c
```

The tag -Wf-bo3 must be used to tell the compiler which bank to use in compilation and how to allocate memory accordingly. This is important to prevent the bank overflow described above. The number banks must also be specified in the line of compilation from objective to ROM using the tag -Wl-yo4 for 4 banks.

5.0 Improvements and Extensions

5.1 Converting to Assembly

As mentioned in 4.2.2, a major improvement to this project would be moving development to a lower level language like Assembly. Assembly gives increased visibility on the allocation of memory and in general would enable much more direct interaction with the hardware of the Game Boy. This would allow us to allocate memory more efficiently without the restrictions of GBDK and therefore expand the game to more levels and include more graphics, using the full potential of the Game Boy Advance.

5.2 Updating the Toolchain

The tools used include the tile designer, map builder, and GBDK. Although GBDK was updated to a 2020 version, these platforms are meant for use primarily with the Game Boy and Game Boy Color with the graphics designers originating from the late 90s. This is perfect for designing a simple game such as the one we have now. To extend the graphics and memory capacity to that of a true Game Boy Advance game, given more time and resources for development, we could use a more fully furnished development toolchain. DevKitPro gives the ability to develop in C, C++, and ASM and provides direct interaction with memory-specific macros in C or C++ [12]. Due to the increased freedom of this platform, the learning curve is much steeper than GBDK and therefore would be useful to consider for longer term continuation of this project.

6.0 Conclusion

Building our own Game Boy game from scratch was a challenging but exciting experience for our group. During the creation of our project, we better understood the structure of our embedded system and learned how the libraries we were using directly affected it to give us the desired gameplay output. We also learned how important it is to be knowledgeable about the specifications of the hardware system you're coding for, and the consequences that arise otherwise. Overall, our group was able to have many valuable takeaways that can be applied to future embedded projects, or even potential future gameboy hobby projects

The code for Bit's Adventure is public on a GitHub repository (link) and the game can be played by downloading the .gb file from the repo and opening it with a Game Boy emulator. Detailed description for deployment is in Appendix B.

Appendix A: References

[1] mGBA. , *mGBA*. [Online]. Available: https://mgba.io/. [Accessed: 19-Apr-2021].

[2] R. Copetti, “Game Boy Advance Architecture: A Practical Analysis,” *Rodrigo's Stuff*, 30-Mar-2021. [Online]. Available: https://www.copetti.org/writings/consoles/game-boy-advance/. [Accessed: 09-Apr-2021].

[3] “Memory Bank Controllers,” *Memory Bank Controllers - GbdevWiki*. [Online]. Available: https://gbdev.gg8.se/wiki/articles/Memory_Bank_Controllers. [Accessed: 20-Apr-2021].

[4] “FLASH IV - EZ-FLASH,” *EZ-Flash*, 18-Feb-2021. [Online]. Available: https://www.ezflash.cn/product/ez-flash-iv/. [Accessed: 09-Apr-2021].

[5] Various, “gbdk-2020/gbdk-2020,” *GitHub*, 2020. [Online]. Available: https://github.com/gbdk-2020/gbdk-2020. [Accessed: 19-Apr-2021].

[6] *Learn how to develop your own GameBoy games*. YouTube, 2018.

[7] H. Mulder, *Harry Mulder's Gameboy Development: Gameboy Tile Designer*, 1997. [Online]. Available: http://www.devrs.com/gb/hmgd/gbtd.html. [Accessed: 19-Apr-2021].

[8] H. Mulder, *Harry Mulder's Gameboy Development: Gameboy Map Builder*, 1998. [Online]. Available: http://www.devrs.com/gb/hmgd/gbmb.html. [Accessed: 19-Apr-2021].

[9] R. Copetti, “Game boy Architecture: A Practical Analysis,” *Rodrigo's Stuff*, 12-Jan-2021. [Online]. Available: https://www.copetti.org/writings/consoles/game-boy/. [Accessed: 19-Apr-2021].

[10] F. Sanglard, “The polygons of Another World: Game Boy Advance,” Fabien Sanglard's Website , 26-Jan-2020. [Online]. Available: https://fabiensanglard.net/another_world_polygons_GBA/. [Accessed: 19-Apr-2021].

[11] “Game Boy Developer's Kit,” *GBDK*. [Online]. Available: http://gbdk.sourceforge.net/. [Accessed: 20-Apr-2021].

[12] *DevKitPro*. [Online]. Available: https://devkitpro.org/. [Accessed: 19-Apr-2021].

Appendix B: README

For a quick playthrough, watch this video

To play the game yourself,

1. Download mGBA here: https://mgba.io/
    a. Follow the installation instructions
2. Clone the GitHub repository: https://github.com/Maelstrum127/MIE438-GameBoy-Project.git
3. If you aren't already in the master branch, switch to that branch on your local device.
4. Open mGBA
    a. Click "Load ROM" in the top left corner of the window
    b. Locate the GB file "bits_adventures.gb" in the repository
5. The game should load automatically to the starting screen. Here are some basic instructions:
    a. To skip through screens and dialogue, click "Enter" (Note: this replaces the classic A and B buttons for simplicity despite on screen text used for authenticity)
    b. To move around maps and participate in combat, use the arrow keys:
        i. Main map: all arrow keys
        ii. Combat: left and right
6. The game is finished once the "Mal has been defeated!" text is displayed

Appendix C: Additional Graphics (that couldn't be included) for your Viewing Enjoyment