ME3241

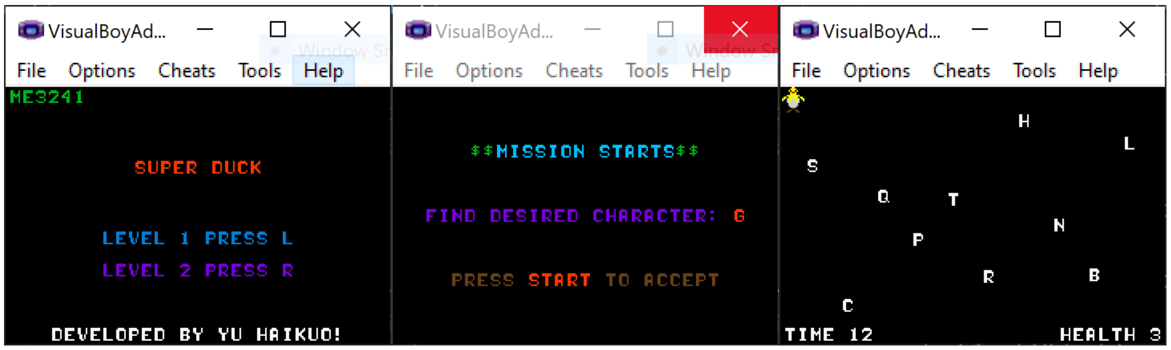
Super Duck Final Report

Student: Yu Haikuo

Matriculation Number: A0194553X

Brief Description

Super Duck is an adventure Gameboy Advance game in which the player is acting as a duck who needs to find her target character among the letters that move up and down within a limited time. In the beginning, the player uses button L or R to choose different difficulty levels. After the player confirms his/her choice, there will be a mission briefing that introduces the mission character. When the player presses START to enter the game, a lovely duck will appear at the top left of the window, and the other ten letters move from top to bottom. In level two letters are moving at a higher speed, and the given time is also reduced. The player uses arrow keys to control the movement of the duck to avoid hitting obstacles and approach the desired character. Once the duck hits the obstacles three times or the game times out, there will be a game lost page that allows the player to replay the game. After the player successfully approaches the target, the game will print a welcome message, and then directs the player to difficulty level two or the final thank you page. The player is also able to restart a new mission at any time by pressing L or R during game runtime or when the game ends.


VisualBoyAd...
File Options Cheats Tools Help
VisualBoyAd...
File Options Cheats Tools Help
VisualBoyAd...
File Options Cheats Tools Help
H
L
S
Q
T
N
P
R
B
C

ME3241

SUPER DUCK

LEVEL 1 PRESS L

LEVEL 2 PRESS R

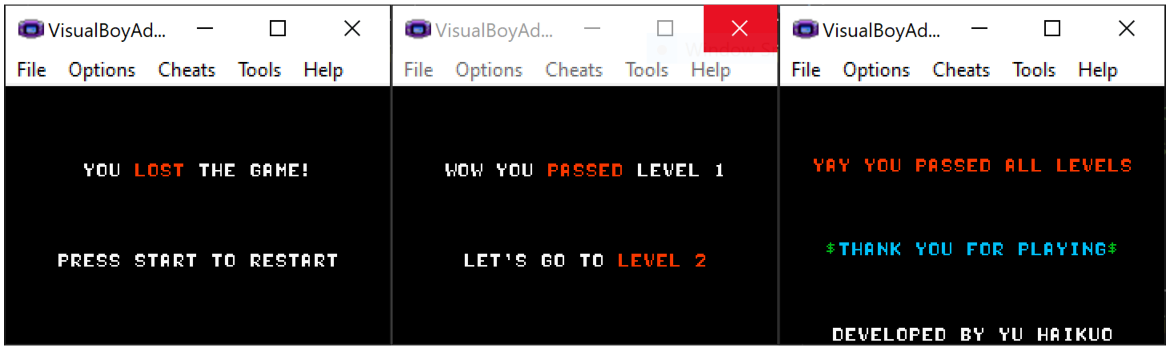
DEVELOPED BY YU HAIKUO!

MISSION STARTS

FIND DESIRED CHARACTER: G

PRESS START TO ACCEPT

TIME 12 HEALTH 3


VisualBoyAd...
File Options Cheats Tools Help
VisualBoyAd...
File Options Cheats Tools Help
VisualBoyAd...
File Options Cheats Tools Help

YOU LOST THE GAME!

PRESS START TO RESTART

WOW YOU PASSED LEVEL 1

LET'S GO TO LEVEL 2

YAY YOU PASSED ALL LEVELS

$THANK YOU FOR PLAYING$

DEVELOPED BY YU HAIKUO

Technical Aspects Behind

Main Code Structure and Logic:

The whole program consists of a main.c file and several header files. In the main file, there are several variables, structures, and *objectIndexArray[10]* defined outside the main function as global variables, global structures, and global array. There are also three functions – *interruptHandler()*, *checkButtonAtStart()* & *checkButtonDuringRuntime()* defined in main.c as they have to interact with global variables. In the main function, the first thing is to print home screen messages and set *INT_BUTTON*, *INT_TIMER0*, and *INT_TIMER1* interrupts. Then the whole game will be running in a *while (1)* loop. The main function watches and takes global variable *difficulty* into a switch statement, which branches into two cases – difficulty level 1 and difficulty level 2. Each case has the same structure. In each case, the program will first determine whether it is the first time to run this game, or the game is a restarted one and then do the necessary settings. After that, it will determine if it is the first time to start the game and if yes, in the same statement the program will determine and assign the randomized mission character. Then, when the game starts (the player presses *START* button), it will do the initialization and keep updating several variables, sprites, duck, and objects, and then print duck, objects, time left, and health status out on screen. Meanwhile, if the game times out, or the poor duck hits obstacles three times and her health becomes zero, the game will stop running and print a failure message. If the duck finishes the mission, the program will go to the next difficulty level, or go to the final thank you page according to the current level player is at. The player can also press L or R button at any time to restart a game with a new randomized character.

GBA Capabilities Used:

Interrupt – INT_BUTTON, INT_TIMER0, INT_TIMER1, and Interrupt Handler Function.

Sprites – fillPalette(), fillSprites(), and drawSprite().

DrawText – drawPixelInMode3(), drawSingleCharacter(), drawEntireString(), clearScreen().

Algorithm – swap(), abs(), shuffle().

Other C language capabilities such as struct and pointer.

My Contribution to the Project

I did not form a group with other students, so this project is 100% done by myself. I created this project with VHAM from scratch and then used Visual Studio Code as the text editor to finish the project.

Key Skills I Have Learned

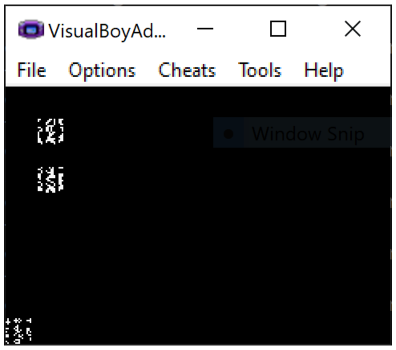
I have really learned a lot in this GBA project. I learned how to set and raise timer and button interrupts, how to write interrupt handler functions to deal with these interrupts, how to draw and use sprites, how to print strings with different colors, how to write and organize code in a while (1) loop, how to use pointers as function parameters, how to deal with object movement and collisions. In a word, how to write a simple but powerful GBA game.

Challenges I Encountered

I encountered a lot of interesting challenges during game development, and I managed to solve or detour away most of them. The first challenge I encountered is about addressing the interrupt handler function. I originally wanted to pass the button state to the handler function using pointers, so I can then pass the pointer into *checkButtonAtStart()* and *checkButtonDuringRuntime()* inside the handler function, which allows me to reduce the number of global variables and put these two functions into another a header file. However, when I write *REG_INT |= (int)&interruptHandler(&is_L_ButtonPushed, &is_R_ButtonPushed);*, I got this error: *invalid lvalue in unary '&'*. I did a lot of research on the Internet, but I did not get a solution. Hence finally, I have to give up doing this and put *checkButtonAtStart()* and *checkButtonDuringRuntime()* in main.c, and set *is_L_ButtonPushed* and *is_R_ButtonPushed* as global variables.

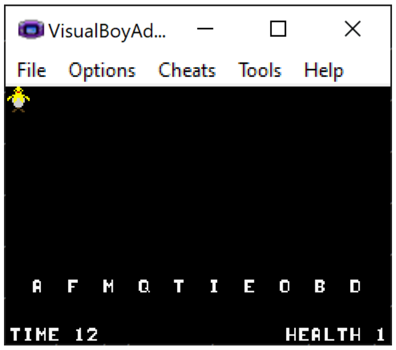
Another challenge I encountered is in file main.c at line 149 about *clearScreen(BLACK)* and setting parameters. I find out if I only run *clearScreen(BLACK)* before setting the parameters, the program will run too quickly to go to the next step (*clearScreen(BLACK)* takes time before it can be finished), and there will be some pixels left on the screen. I was thinking of using *sleep()* function from <unistd.h> to let the program hold on for a while, but I got this error: undefined reference to 'sleep'. It seems that it is because ARM-GCC does not support POSIX (Portable Operating System Interface) functions and other Unix-Specific functions [1]. Then I decided to run *clearScreen(BLACK)* again after setting all parameters, just for wasting some time, and in the end, finally I realized how stupid my idea was. I could have set all those parameters before executing *clearScreen(BLACK)* function!

In addition, I also encountered a problem with the sprites. Originally I was thinking of copying everything inside fonts[] from fonts.h into sprites2[] from sprites.h. But soon I found out after filling sprites2[], the given gift function *drawSprites()* could not work normally. It will draw something like this:



After reducing the number of elements to 31 (which includes numbers 0 to 9, letters A to T, and sprite duck), the function resumes working normally. I tried to make edits on the given gift function, but it seems that increasing the maximum boundary of *i* in *fillSprites()* does not have any effect on solving the problem. Therefore, in my option, I guess there may not be enough space specifically for those sprites in memory. The designed memory space could only contain 31 elements, not even one more. In sprites.h at line 2368, I was thinking of adding a heart sprite to better illustrate the duck's health status, but after compiling the whole game just crashed. I guess it might be because memory addresses assigned to global variables are limited, and the total sprites array exceeds the maximum size on its allocated part of memory, and thus it overwrites other bits that are out of the boundary and results in undesired behaviors. It could be reasonable as memory addressing in C is done by pointers, and a pointer can point and change anything, although it does not know whether it is pointing to the correct addresses itself.

A similar thing also happened when I was creating my last global variable *isGameEnded* in main.c at line 33. After creating *isGameEnded* the game became like this:



What a funny scene! All the letters were standing together in a line and started to drop. I was quite shocked and double confirmed that I did not make other changes but only added *isGameEnded* as a new global variable. By commenting and uncommenting *isGameEnded* the whole game will have a new ridiculous behavior! My inference is the same – probably those existing global variables have taken all the spaces Gameboy designs for them, and the new element has to overwrite some other important bits that are related to the positioning of objects.

In addition to these memory-related issues, I also encountered another two interrupt-related problems with the duck. The first problem is about *printFailureMessage()* in event.h at line 51. If I do not add *clearScreen(BLACK)* inside *printFailureMessage()*, the poor lovely duck could not go back to its original point (0, 0) when the game restarts if there are any arrow keys pressed when *printFailureMessage()* is called. Instead, she will continue to go a few steps based on the arrow buttons. It seems that there is a backlog in button interrupts when switching the video mode, and it requires two *clearScreen(BLACK)* to eat all these undesired interrupts. Later after encountering the next problem, I have a deeper understanding.

The second duck-related problem appears in objects.h at line 61. Originally, I was going to design a health system that gives this tiny poor duck two more opportunities when hitting the obstacles. However, the code *duck->life = duck -> - 1* will be executed for multiple times for only one hit, and the health will quickly reduce from three to zero. Finally, I figured out why – I forgot to set object->isObjectApproached == 0 as one of the essential conditions. Since there is a delay between the player pressing the button and sending out button interrupts, when the player stops pressing the button, the duck will continue to move a few minor steps farther due to backlogged interrupts. Therefore, multiple collision occurs, which quickly reduces the poor duck's health to zero.

Future Work to Do

In the future, I would like to make this game an infinite difficulty increasing one and adding a leaderboard. The number of objects and the velocity at which objects fall will increase with the difficulty level, and the player can record his/her highest difficulty level achieved together with his/her name. I will also add some bonus character, and some missions that have multiple mission characters with different priorities, too.

Bibliography

[1] "cant use sleep() in embedded c for stm32 - Stack Overflow." https://stackoverflow.com/questions/57591880/cant-use-sleep-in-embedded-c-for-stm32 (accessed Apr. 18, 2022).