Microprocessor Applications: GBA Game Assignment

Figure 1: Title/menu screen for game, with bouncing viruses and title and changing colour

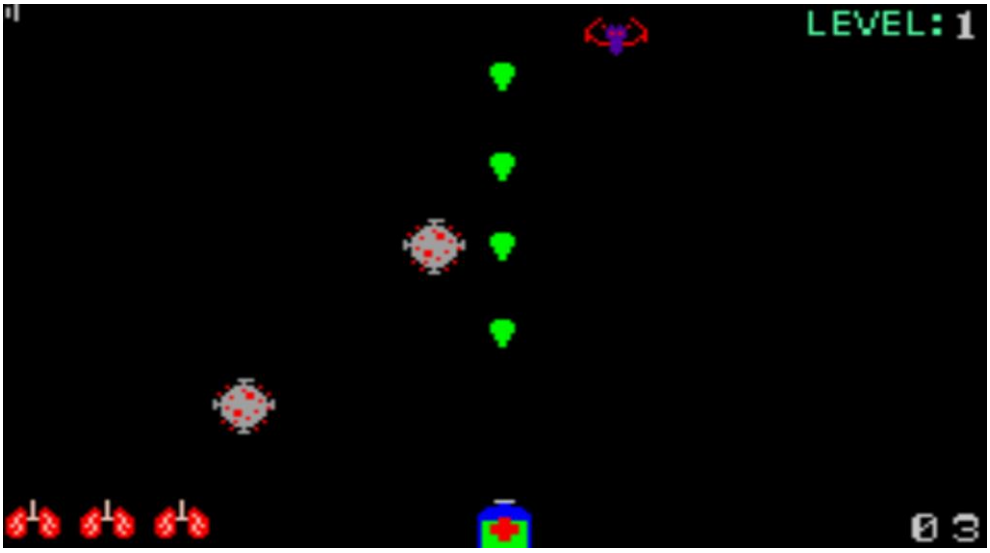


Figure 2: Typical gameplay screenshot of shooting down enemies

Task and Game Description:

Corona Killer is a Space Invaders style game where you must prevent the Corona Virus from reaching you by shooting it with hand-gel squirts. You have 3 lives denoted by lungs and must survive 3 levels of increasing difficulty. If you die or complete the game, you will get taken back to the menu screen where you can restart. If you beat all 3 levels your score will be displayed on the main menu afterwards. You get 10 shots of hand-gel before you must reload each time and your final score will be based on how few shots you used and how many remaining lives you have at the end of level 3.

Controls:

Button A (Z key):	Start game
Button B (X key):	Reset game to menu screen
Left/Right Arrows:	Move left or right
Up Arrow:	Shoot
Down Arrow:	Reload

Game Implementation:

We used C and ARM assembly code within the HAM editor.

Code Workspace:

File	Description
main.c	Main c code file with game loop and handler functions
assembler.s	File with assembly code functions
graphics.h	The colour palette and tile graphics
variables.h	All global variables and structure definitions
lazerFunctions.h	Functions related to shooting i.e. drawing, moving and collision with enemies
enemFunction.h	Functions related to the virus enemies
bossFunctions.h	Functions related to the bosses
menu.h	Functions to display the menu screen sprites
functions.h	All other functions miscellaneous functions such as drawing the health bar

Sprites:

Sprite Number in Memory	Assigned Sprite	Sprite Number in Memory	Assigned Sprite
0	Gun	71	Final Boss
1-3	Lungs (health bar)	80-81	Score Number
4-5	Laser counter	100-105	Level Characters
10-30	enemies	106-111	Score Characters
40-50	Laser shots	112-123	Game Title Characters
60-70	Bosses		

Creating Sprites:

For the Enemies, Bosses and Lazers there can be multiple sprites of the same type on the screen at the same time. Therefore, we have implemented structures for each of these object types, where each one will contain a list of attributes for that object such as its position, speed and health.

When we want to create one of these objects on the screen, we first check if there is an available slot in the objects array such as in the for-loop bellow:

```
int i = 0;
for (i; i < display_enemies; i++) {
   if (Enemies[i].inUse == 0) {
```

Once an empty space is found, then a new object is initialized there and Enemies[i].inUse is set to 1 so that the space is now occupied and the for loop is broken out of.

Drawing Sprites:

To draw the sprites, a function was created which loops through all the entities in the object array and if the object is in use then it will display the sprite. This speeds up the code as only in use objects will be drawn. As each sprite has to be allocated a unique space in memory, a sprite offset is applied in the code to make sure there is no overlap between different types of objects. Additionally, as all the sprite tile data is stored in one array, to extract the corresponding tile, an offset is applied to the tile destination. Bellow is an example for drawEnemy function where the enemy tiles start at tile 10 till tile 13:

```
void drawEnemy(int virus, int N, int x, int y)
{
   N = N + enemySpriteOffset;
   *(unsigned short*)(0x7000000 + 8 * N) = y | 0x2000;
   *(unsigned short*)(0x7000002 + 8 * N) = x | 0x4000;
   *(unsigned short*)(0x7000004 + 8 * N) = (virus + 10) * 2;
}
```

Deleting Sprites:

To delete a sprite, the InUse value is simply set to false so that it will no longer be drawn, and it will no longer interact with other sprites.

Moving Sprites:

To move the sprites, every timer 0 cycle, the movement function will be updated. For all the entities that are in use, their x and y_pos will be updated such as:

```
Enemies[i].x_pos = Enemies[i].x_pos + Enemies[i].x_speed;
```

Increasing the .x_speed attribute will directly increase the movement speed. Additionally, if this variable is negative, then the movement will be in the opposite direction.

Collision:

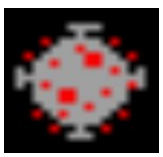
Collision between the lazer's and all the enemies works by looping through all the inUse lazers and then for each lazer, loop through each inUse enemy. If the lazer is within the hitbox for the enemy then 1 health is removed from the enemy and the lazer is removed. The hitbox is defined as a 16x16 square as this is the size of the enemy sprite. The following is some sample code:

```
for (i; i < display_lasers; i++)
{

   if (Lasers[i].inUse == 1)
   {

      int j = 0;
      for (j; j < display_enemies; j++) // if laser is within the collision box of an active enemy, then remove health from that enemy and remove the
      {
         if ((Lasers[i].y_pos >= (Enemies[j].y_pos / 10)) & (Lasers[i].y_pos <= (Enemies[j].y_pos / 10 + enemyHitbox)) & (Enemies[j].inUse))
         {
            if (Lasers[i].x_pos >= (Enemies[j].x_pos / 10) & (Lasers[i].x_pos <= (Enemies[j].x_pos / 10) + ((enemyHitbox) * 2) - 2))
            {
               Lasers[i].inUse = 0;
               Enemies[j].health--;
            }
         }
      }
```

For collision of the bosses with the edges of the screen, when the x or y position reaches the boundary then the x or y speed attribute is switched sign so that the sprite will move in the opposite direction.

Enemies:

These are the basic enemy (virus) which is a 16x16 sprite which falls downward. They have 1 health so are 1 hit 1 kill and will remove one player life if they reach the bottom of the screen.



We used a C language structure to give the enemy object the following attributes:

```
struct enemy { // s
      int inUse;
      int mode;
      int health;
      int x_pos;
      int y_pos;
      int x_speed;
      int y_speed;
};
```

We then created a size 20 array (called Enemies) with this structure, which allows us to have multiple objects at the same time on the screen.

```
struct enemy Enemies [20];
```

Bosses:

The bosses (bats) are 16x16 sprites which move side to side at the top of the screen and shoot virus down at the player. They have 3 health and will shoot a virus every 3 seconds. The key difference in attributes to the Enemies is that they also have an animation variable which will alternate between two values to animate the bat flapping its wings (will switch between two sets of 16x16 tiles). Additionally, a shot delay attribute is used to take note of when the next virus will be released from that bat.

```
struct boss { // structure for the project with three attributes
      int inUse;
      int health;
      int x_pos;
      int y_pos;
      int x_speed;
      int y_speed;
      int shot_delay;         // how long bewtween the attack of a boss
      int animation_timer;
      int animation;          // State of animaion
};
```

Level Design:

Level 0 (Menu):

Menu screen with viruses and title bouncing around for visual effect. The score from the previous game is displayed. Will proceed to level 1 when button A (Z) is pressed. Will return to menu if button B (X) is pressed at any time.

Level 1:

12 viruses fall towards the player at 2-3 second random intervals and at random positions at the top of the screen. A bat boss appears after the 4th virus is created. Once all enemies are dead, will proceed to level 2.

Level 2:

2 bat bosses are created simultaneously which fly around horizontally and vertically. Proceed to level 3 when both bosses are dead.

Level 3:

Big final boss which moves side to side while shooting out 3 viruses simultaneously which all move in different directs towards the player. Will return to menu screen with final score when Boss is defeated.

Game Loop:

The game starts in the main menu screen and will wait till A is pressed to start level 1. In the C code, a while loop is used to wait until level 1 is initiated. A variable called level is used to denote the state of the game with level=0 being the menu. If at any time the player loses all their health or the reset game button is pressed (B), then the level will be set to level=0 and the code will return to the start of the game loop using a goto statement. Here all variables are reset, the score is saved, all the active sprites are wiped and the game will wait until level 1 is initiated again. The health and game state are constantly checked with a for loop within the game while loop, so that if either are triggered, the game will reset using a goto start statement.

To create the timing intervals between enemies entering, an array is created at the start of each level which generates semi random intervals. When the global time equals the time in the current array position, then an enemy is created, and the code then waits for the next entry timing. Once the end of the array is reached no more enemies are created and the level will finish once all the enemies are dead.

A random function is also used to randomly select the start position of the enemies which means that every playthrough the timing and position of enemies will be different. For the random number generation the rand() function from the standard library is used. This is the only library that we used in our game.

Calculate Score (Assembly Code):

We used assembly code to write the definition for the calculateScore function:

```
.GLOBL calculateScore
calculateScore:
    swp     r4, r4, [sp]          @ pop last input argument from stack and put it in r4
    stmfd   sp!, {r4-r11, lr}     @ save content of r4-r11 and link register into the sp register

    @ function body:
   // r0: lasersNeeded (Score); r1: laserScore (lasersExceeded) , r2: int healthScore (livesLost), r3: varialbe for calculations
      rsb r2, r2, #3          // livesLost=3-healthScore
      subs r1, r1, r0            // lasersExeeded=laserScore-lasersNeeded

      mov r0, #100            // score=100
      bmi break               //if (lasersExeeded<0) {goto break}

      cmp r1, #15             // compare lasersExceeded to 15
      suble r0, r0, r1, LSL #1 // if (lasersExceeded<=15) {score=score-lasersExceeded*2}

      subgt r1, r1, #15    //if (lasersExceeded>15) {lasersExceeded-15)
      subgt r0, r0, r1     //if (lasersExceeded>15) {score=score-(lasersExceeded-15)}
      subgt r0, r0, #30    //if (lasersExceeded>15) {score=score-30}


      mov r3, r2, LSL #5         // x=32*livesLost
      sub r3, r3, r2, LSL #1  // x=x-2*livesLost
      subs r0, r0, r3         // score=score-x (x=30*livesLost)
      movmi r0, #0            // if (score<0) {score=0;}

      cmp r2, #3     //if(livesLost==3)
      moveq r0, #0   //score=0;
      break:
```

The function takes in 3 arguments: number of lazers needed, number of lasers used and health remaining. The number of lazers needed is the minimum number to be able to beat the game. The final score is therefore based on the ratio of number of lasers used and how much health is left over. The function then returns the final score in register r0.

Timers:

Timers 1 and 2 have not been used in this project.

Global Time Variable (Timer 3):

Increments every 0.1 seconds to allow precise control of the entry of sprites and timing of when they shoot. We used the 64x frequency timer with start point = 39316 to overclock every 0.1 second.

```
global_time += 1;
```

global_time is incremented by 1 each cycle, therefore 1 second will equal 10 units of time.

Movement, Collisions & Health (Timer 0):

This is a high frequency timer which each cycle will move all the sprites that are in use, check for collisions and check the health of sprites to see if they should still be in use. We also used the 64x frequency timer but with a start point of 64000. The functions within this timer was as follows:

```
if (level == 1 | level == 2 | level == 3)
{
   Collisions();

   moveLasers();
   moveEnemy();
   moveBoss();

   healthcheckEnemy();
   healthcheckBoss();

   bossShoot();
}

if (level == 0) {
   moveTitle();
   moveBackground();
}
```

Drawing Sprites (HBlank):

Every refresh cycle of the screen, all the sprites that are in use, will be drawn on the display. There is no reason to draw more often than this because the display would not have refreshed fast enough. The functions within HBlank is as follows:

```
// Draw the sprite
drawMenu();
drawTitle();
drawAllBackground(
// Draw the sprite
drawGun();
drawPlayerHealth()
drawShotCounter();
drawSprite(level,

drawAllLasers();
drawAllBosses();
drawAllEnemies();
```

Buttons:

When a button is pressed, the checkbutton() handler will run which finds the button that was pressed. Each button then has a specific function that runs if it was pressed. For example when KEY_DOWN is pressed, the function reload() runs which reloads the shot counter. Some functions within buttons you only want to work on certain conditions. Therefore, an if statement can first be added, for example when KEY_A is pressed, the level needs to be equal to 0 to start the game. If the level does not equal 0, then the button does not do anything.

Graphics and Colour Palette:

In total 62 8x8 graphic tiles were used and 13 different colours as follows:

```
*(unsigned short*)0x5000200 = 0; // Background Black
*(unsigned short*)0x5000202 = RGB(25, 25, 25); // White
*(unsigned short*)0x5000204 = RGB(31, 0, 0); // Red
*(unsigned short*)0x5000206 = RGB(20, 20, 20); // Grey
*(unsigned short*)0x5000208 = RGB(31, 31, 5); // Yellow
*(unsigned short*)0x500020A = RGB(31, 22, 5); // Skin
*(unsigned short*)0x500020C = RGB(31, 31, 25); // Cream
*(unsigned short*)0x500020E = RGB(10, 10, 31); // Navy
*(unsigned short*)0x5000210 = RGB(31, 25, 23); // Pink
*(unsigned short*)0x5000212 = RGB(0, 31, 0); // Green
*(unsigned short*)0x5000214 = RGB(10, 0, 20); // Purple
*(unsigned short*)0x5000216 = RGB(10, 31, 20); // Special Color (can change)
*(unsigned short*)0x5000218 = RGB(0, 0, 31); // Light Blue Bl
```

To draw the 16x16 sprites, a bitwise or operation of 0x4000 with x is added to draw 4x4 tiles rather than 1x1 tiles.

```
void drawEnemy(int virus, int N, int x, int y)
{
   N = N + enemySpriteOffset;
   *(unsigned short*)(0x7000000 + 8 * N) = y | 0x2000;
   *(unsigned short*)(0x7000002 + 8 * N) = x | 0x4000;
   *(unsigned short*)(0x7000004 + 8 * N) = (virus + 10) * 2;
}
```

In the menu screen, the colour of the text alternates between two colours. This is achieved by repeatedly reassigning the colour value in one of the corresponding memory locations.

Reflection:

As a group we worked well together. Despite social distancing preventing us meeting in person, we were able to work as a team by using a shared google drive to share the code. We also used a shared word document to record each version of the game with notes on who edited it, the changes made and any issues with the code.

This project taught me about how to work on a group coding project which I have not had to do before. It is different from other university projects because everybody's code interrelates with each other and so you cannot work independently then just put your work together at the end. This meant that communication was even more important than on a regular group project.

Some challenges we had was initially figuring out some of the basic functionality of the game such as drawing and moving the sprites. However, once these initial hurdles were overcome, the rest of the project was simply adding more content to the game.

In the code, we were sometimes inconsistent in our variable naming convention, using names that sometimes add confusion such as having 'enemies' object as one type of the games enemy. Therefore, next time as a group we would agree beforehand on the convention that we would use and make a brief write up of this plan.

Version Evolution:

Version Number	Functionality	Notes	Problems
1.0	Can move a sprite left and right with arrow keys and shoot out sprites		
1.1	Added a template for using structures for the shots. It is not functionality implemented yet but would be easy to do.	The functionality of the code has not changed	
1.2	Implemented Leo's structures and functions into the code, debugging and general cleaning of code	Code runs smooth, reload added, next goal: collisions	Wasn't able to implement handler function
1.3	Added template for enemy functions and used header files for the lazer functions and enemy functions. Added an experimental different way of shooting: you shoot when pressing the up arrow rather than holding it down and waiting for the time delay between next shots.	Functionally same other than the shooting change.	Wasn't able to implement delete sprite function so just used previous implementation
1.4	Added Collisions and bosses, right now one spawns when you press the Z/Y key	Now feels like a functioning game.	Headers need to be in certain order, maybe put all variables back in main
1.5	Added global_timer and random entry of enemies. Made 'bat' boss bounce around and produce viruses according to global time. Implemented Unmesh first virus graphics.	Adjusted some of the hit boxes because sometimes they were clipping through before. Added extra header file functions for misc functions.	
1.8	Trump functions/ structure implemented. Trump and Lung graphics implemented.		Tried to get big boss to turn when he faces other direction but didn't work.
1.9	Implemented game loop where enemies can enter at random timing at a random position at the top of the screen		
1.10	Implemented Health system and level system. Entered final graphics		
1.11	Added Menu screen which you will go back when you die or you reset the game.		

Work Distribution:

Part of Project	Person Contributed
Assembly Code	Leo, Franz, Unmesh
Graphic Design	Unmesh
Graphic Implementation (Drawing Sprites)	Leo
Timers	Franz
Button Handling	Unmesh
Menu Screen	Leo
Sprite/Object Movement	Franz
Game loop design and global timing	Leo
Collision	Unmesh
Sprite Structures	Franz