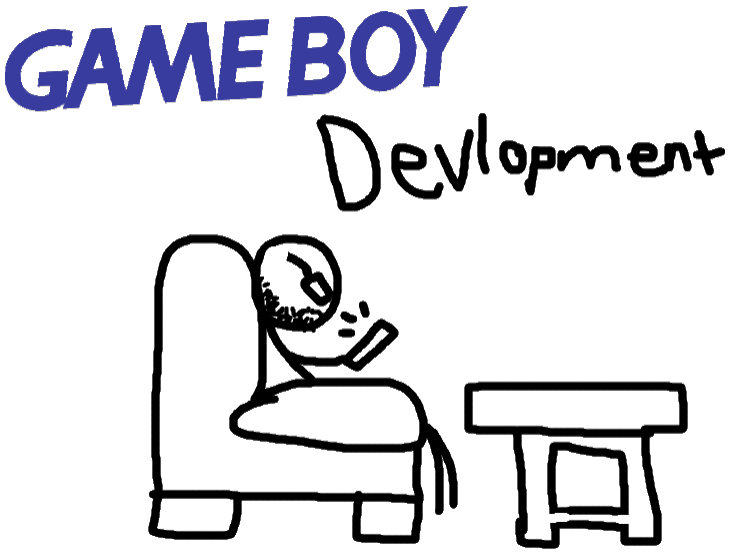
Game Boy Development

Alex Bianca

December 9, 2025



1 What am I doing?

My initial plan for this assignment was to develop a platformer for the ***Game Boy***, I wanted horizontal movement using the D-Pad, jumping using the A button, and sprinting using the B Button, and objects that the player would be able to interact with like enemies or a level ending object. However, after conducting more research into the developmental process and thinking further about the additional systems I would have had to create for such a game, I quickly realized how insanely out of scope this vision was for this assignment. In order to achieve what I was hoping to create, I would have had to create several complex systems from the ground up including a collision detection system that caused the player to walk on the ground, be stopped by walls, not be completely stopped when colliding with things above them, be able to interact with intractable objects, a physics system for the player so that jumping and sprinting could be achieved, a parallax scrolling graphics system, and in order to accomplish these tasks I would need to be incredibly efficient with the ***Game Boy***'s very limited memory so that the game be able to be built in a playable state.

So with all of these things in mind I pivoted to creating a more simple game, a clone of Atari's ***Breakout***, I choose this game because of it's simplicity in terms of it's game play, necessary systems, and potential room for expansion if I ended up having more time. The kinds of systems that a game like ***Breakout*** requires are far simpler than that of a platformer, with the lack of a need for parallax scrolling being a significant footnote, while systems such as a collision system, and a player input system were still requirements, and failure to implement them would still result in an unplayable game, they did not require the same amount of depth that would have been needed for a platformer. Also due to this games increased simplicity, the requirements for strict memory management were greatly decreased, giving me more much needed wiggle room in terms of console resource

management and making sure my game is as optimized as possible in order able to run in a stable state. Thanks to these to these cut backs and the extensive amount of helpful resources that would aid me in developing a game of this caliber I was able to massive cut down my projected development time making this project possible.

2 Why am I doing it?

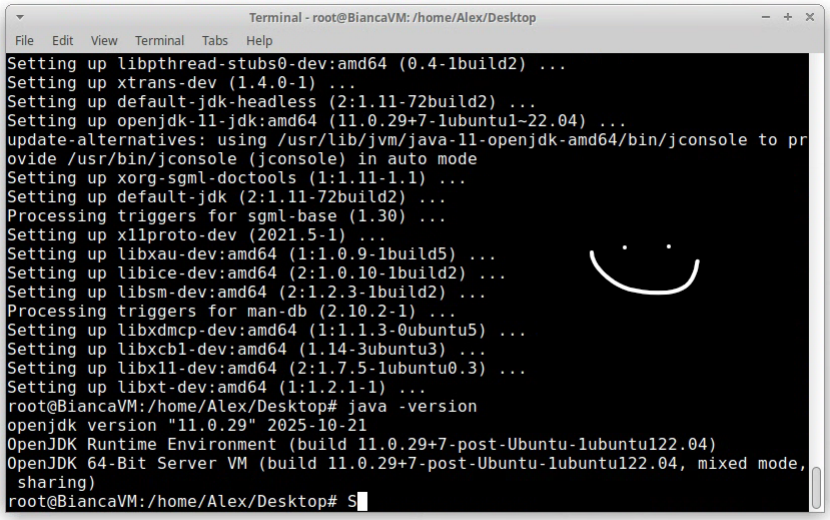
I am choosing to do ***Game Boy*** development because it has been something that I have been interested in since the beginning of the class, I knew from the beginning that I wanted to experiment with using assembly within the context of game development, but was unsure of how exactly I wanted to go about accomplishing it. But after watching the videos about the ***Game Boy***'s architecture that were provided in the class modules, doing some more external research, and seeing some examples of what previous students were able to accomplish for this assignment, I knew that I wanted to try my hand at ***Game Boy*** development for this assignment.

3 How am I doing it?

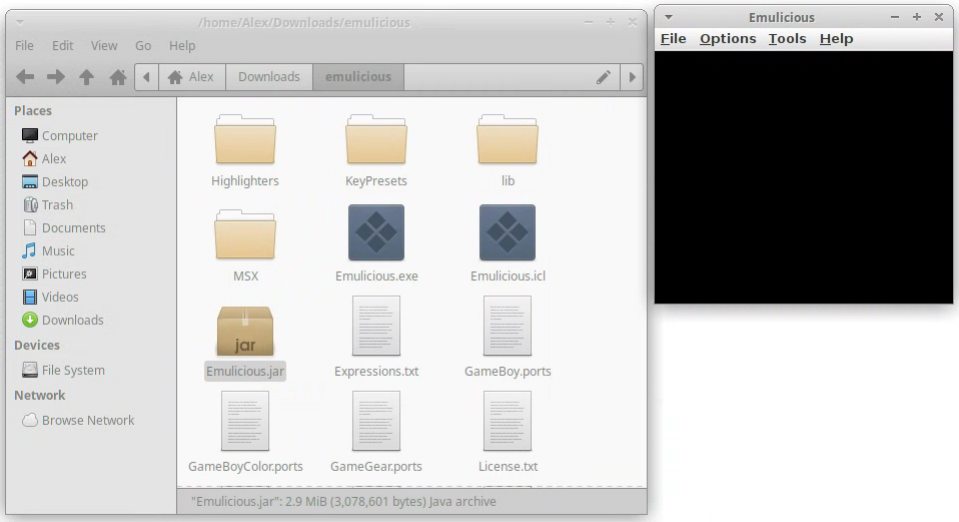
Initially, I planned on using the ***Game Boy*** Development Kit or GBDK, which is an open source tool chain that allows users to develop software or games for the ***Game Boy*** or ***Game Boy Color*** using a mix of C programming and assembly programming all within an IDE of the user's choosing. However, after further inspection of GBDK I realized that a vast majority of the coding was done in C rather than assembly, which didn't feel appropriate for this class or for this final so I went back to the drawing board in search of new ideas. Eventually, I stumbled upon a tutorial provided here which utilized ***RGBDS***, which is a free open source tool for ***Game Boy*** development that allows me to compile my written assembly code into an executable ***Game Boy*** ROM file that can then either be ran by an emulator or burned to a flash cart and ran on a real ***Game Boy***. For this project I choose to develop using an emulator to test my game simply just because it would be far more convenient that trying to use a real ***Game Boy***, the emulator I chose to use is ***Emulicious*** which is a Linux based ***Game Boy*** emulator. I made the decision to use ***Emulicious*** because it came equipped with a real time debugger that had built-in programs that covered the ***Game Boy***'s entire compilation pipeline with things like a memory editor, tile map viewer, and color pallette viewer, allowing for quick and easy debugging of my project and aiding greatly in my understanding of how exactly everything within the development process worked. I will also be using ***Sublime***, which is just a simple Linux based text editor making the process of writing code far simpler and quicker compared to just using a text editor like Notepad, and it's RGBDS syntax add on which is just an add on for ***Sublime*** that makes the code much more easily readable by giving the text some color in the editor environment.

4 What were the challenges?

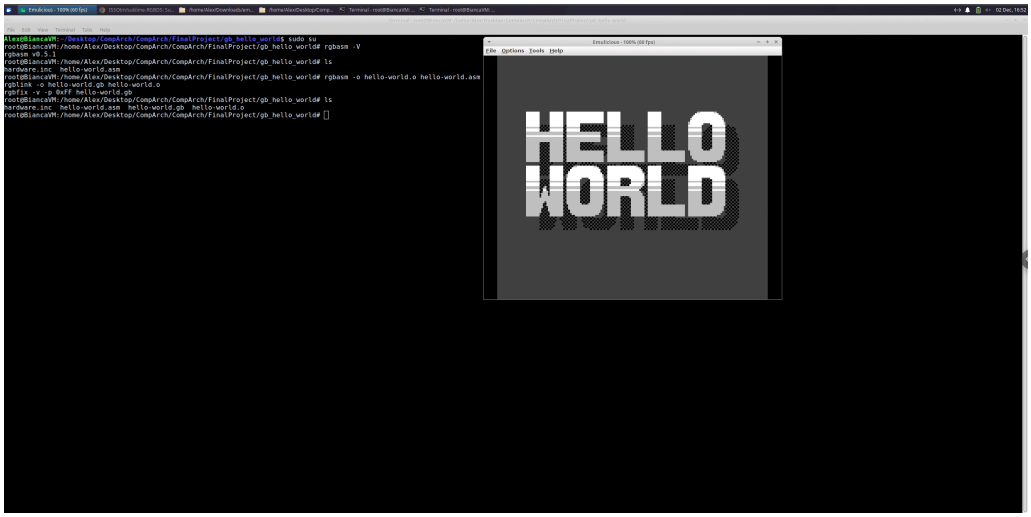
The first major challenge I encountered had nothing to do with the coding of the game, it was actually the process I had to go through in order to get the necessary development environment for this project setup and working, this project quickly became a lesson in general Linux usage, setup, and architecture in addition to everything else. The reason for it being because ***RGBDS*** and ***Emulicious*** required a Linux environment to run properly, and getting one set up was on my Windows PC was a lot more complicated and confusing than I thought.
First I tried using the Windows Sub System for Linux (or WSL) to get a Linux environment running within a Windows environment but this method was swiftly shelved due to a number of complications, for whatever reason I found the process incredibly confusing, even after reading documentation and tutorials I was still left confused on the steps I needed to take in order to create the environment I needed. Progress soon ground to a halt, any progress I was making towards getting everything to work was slow and painful. Fortunately, I had a backup plan, I had access to an Ubuntu virtual machine thanks to my Linux/Unix class providing me one for assignments, so I was able to utilize that environment which eliminated the need for me to have to set up my own Linux environment from scratch, saving me a lot of time on the set up process and headaches.



Using this environment didn't come without challenges either, in the beginning, I was still fairly unfamiliar with general Linux usage since I have been a Windows user for all my life, which led to me having to work through a lot of errors, especially while trying to install the needed tools I needed to begin development. ***RGBDS***, ***Emulicious***, and ***Sublime*** required specific Linux tools and firmwares to be downloaded into my environment, which meant I needed to undergo more research and learning into which specific versions of tools and firmwares I needed and more general Linux debugging when these downloads didn't work in the ways that were expected or at all.



But fortunately, after many hours of researching and problem solving my way through it all, I was able to get ***Emulicous***, and ***Sublime*** running, and installed the correct version of ***RGBDS***, so in order to verify that everything was working properly I loaded up a test file that was provided by the tutorial which resulted in the Hello World graphic below, showing that everything was working perfectly.



Now all that was left was to begin actually writing the code for my *Breakout* clone and getting it build and running on ***Emulicious***.

5 How does it all work?

RGBDS has a very simple series of commands for creating builds of ***Game Boy*** ROMs, the reason for this simplicity is because the chain of processes that the commands follow is a near one-to-one copy of the Translation Pipeline that we had been utilizing in class.
This series of tools begins with RGBASM, which is an assembler that takes in asm files which act as the source code for the ROM, and then assembles them into executable ***Game Boy*** code this is the part of the tool that does a majority of the heavy lifting in terms of actually creating an executable ROM that ***Emulicious*** can run.

This process is executed by inputting the follow command into the Linux terminal:
rgbasm -o main.o main.asm

```
root@BiancaVM:/home/Alex/Desktop/CompArch/CompArch/FinalProject/brick_bash# rgbasm -o main.o main.asm
```

This tells RGBASM to assemble the code found in main.asm into a new file called main.o

However, RGBASM often does not have enough information to produce a fully executable ROM file all by it's self, so it stores it's intermediary results in object files, this is where the RGBLINKER comes in. As the name implies, it is the linker that takes the object files generated by RGBASM and "links" them into the ROM filling in the gaps left behind by RGBASM.

This process is executed by inputting the follow command into the Linux terminal:
rgblink -o unbricked.gb main.o

```
root@BiancaVM:/home/Alex/Desktop/CompArch/CompArch/FinalProject/brick_bash# rgblink -o unbricked.gb main.o
```

This tells RGBLINKER to create a ***Game Boy*** ROM file called unbricked.gb based upon the objects main.o

The creation of the ROM is not complete just yet, as RBGLINK does not produce a usable ROM just yet, the ROM is still lacking one crucial component, it's header. A ROM's header is what contains it's metadata, this metadata includes things like the game's name, ***Game Boy*** Color compatibility, the game's licensee code, the Nintendo Logo, and many other important pieces of information the game ROM needs in order to be able to be run by *Emulicious*. The three most important pieces of data that must be accounted for in order for the ROM to be complete is the previously mentioned Nintendo logo, the ROM's data size, and two specific checksum, the Header checksum and the Global checksum.
These three pieces of metadata are crucial to be able to manage which is the job of RGBFIX, which populates these three specific fields of the ROM so that the ROM is able to be successfully ran, RGBFIX is necessary part of the process because of the boot ROM that is run at the beginning of every ***Game Boy*** game. This boot ROM first draws the Nintendo logo from the ROM and plays the

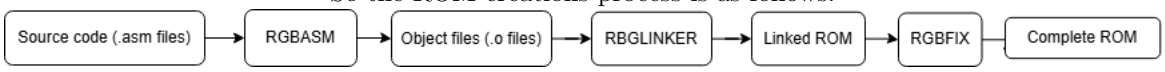
boot animation, once this animation is completed the console (or emulator in this case) checks if the logo on the ROM matches a copy of the logo that is stored internally on the console itself, and if the logos do not match, the console locks up and the ROM is not run. This is the result of an anti-piracy measure implemented by Nintendo to prevent unauthorized cartridges from running on the ***Game Boy***, since these unauthorized cartridges would not have access to the correct logo, these cartridges would be blocked from being able to be ran on an unmodified ***Game Boy*** console.

This process is executed by inputting the follow command into the Linux terminal:
rgbfix -v -p 0xFF unbricked.gb

```
root@BiancaVM:/home/Alex/Desktop/CompArch/CompArch/FinalProject/brick_bash# rgbfix -v -p 0xFF unbricked.gs
```

In a similar vein to this anti-piracy measure, the boot ROM also computes a checksum of the header to make sure that it is not corrupted in anyway, the header itself contains a copy of this checksum and if it does not match the checksum that boot ROM computed, the ROM is prevented from running, this header also contains the ROM's size. This is where RGBFIX works it's magic by using its -v option which injects the correct Nintendo logo needed to pass the anti-piracy check and also computes the correct check sums needed to prevent the ROM from being considered corrupted or unauthorized. Then the -p 0xFF options instructs the header to set the ROM's file size to a valid size and then that size is set to the corresponding value in the header field.

So the ROM creations process is as follows:



6 How does the *Game Boy* render graphics?

6.1 Tiles

The ***Game Boy*** utilizes tiles in order to render it's graphics which are essentially 8 x 8 x 2, or in some special cases 16 x 16 x 2, pixel squares that group pixels together assigning their data to one pixel rather than all of the individual pixels the tile occupies. So instead of storing data for every single pixel on the ***Game Boy***'s 144 X 160 screen with two bits being assigned to each pixel, tiles allow data to be assigned to groups of pixels rather than each individual pixel, allowing them to be reused easily with very little additional processioning cost, saving large amounts of memory. These tiles are using in order to create the graphics for everything seen on screen.

Tile creation code looks like this, a spinet of the code used to create the duck graphic:

```
423     ; Little duck logo
424     dw `22222222
425     dw `22222222
426     dw `22222222
427     dw `22222222
428     dw `22222222
429     dw `22222211
430     dw `22222211
431     dw `22222211
432
433     dw `22222222
434     dw `22222222
435     dw `22222222
436     dw `11111111
437     dw `11111111
438     dw `11221111
439     dw `11221111
440     dw `11000011
441
442     dw `22222222
443     dw `22222222
444     dw `22222222
445     dw `22222222
446     dw `22222222
447     dw `11222222
448     dw `11222222
449     dw `11222222
```

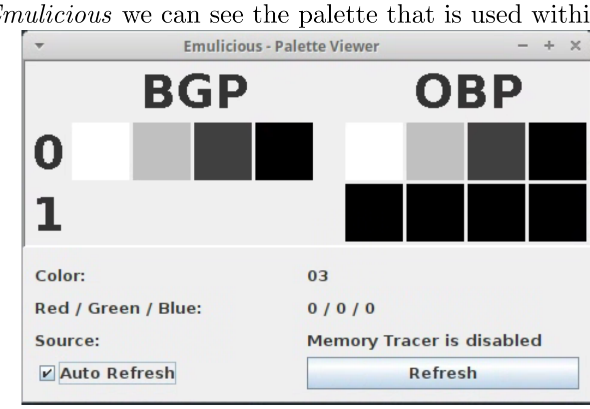
Here we can see values being assigned to specific pixels on the ***Game Boy***'s screen in waves of 8, for each pixel, each row of 8 pixels is stored in 2 bytes, but these bytes do not contain the info for 4 pixels, instead for each pixel the least significant bit of its index is stored in the first byte, and the

most significant bit is stored in the second byte. So then the leftmost pixel is stored in the leftmost bit of both bytes, and the pixel to its right in the second leftmost bit, and this continues for all of the bits. Meaning, the first pair of bytes stores the topmost row, the second byte the row below that, and so on, this process is called ”encoding” and it's how to ***Game Boy*** was able to keep its circuitry simple and run at a low power.

6.2 Palettes

As previously mentioned, each pixel on the ***Game Boy***'s screen has two bits assigned to them, the reason for this is because the color technique the ***Game Boy*** uses for it's pixels called ”bit depth”. Essentially, pixel colors are not stored in the tiles themselves but are instead colored by a palette given to the hardware. The ***Game Boy***'s color pallette has 4 colors ranging from 0 meaning white to 3 meaning completely black, which is why those numbers specifically were used in the code example above. The ***Game Boy*** has three separate palettes, one for the background called BGP or Back Ground Palette, and two for objects called OBP0 and OBP1 or Object Palette 0 and 1, with the back ground palette being applied to the tiles in the back ground and the object palettes being applied to object tiles.

Here in *Emulicious* we can see the palette that is used within the game:



6.3 Tilemaps

Tilemaps take the previously discussed tiles, and arrange them into larger grids of pixels to allow for cheap reuse, the tiles aren't stored in the tilemap directly, instead, tiles are referred to by an ID, of which there are 256 possible IDs, meaning that some IDs will refer to multiple different tiles simultaneously. These tiles are loaded into the ***Game Boy***'s VRAM and act like building blocks that the tilemap then uses to build the pixels of the entire screen, the tilemap's job is essentially to tell the console which tile to draw at each given location.

Here we can see the tilemap the is used to draw the entire screen

```
Tilemap:
    db $00, $01, $01, $01, $01, $01, $01, $01, $01, $01, $01, $01, $01, $02, $03, $03, $03, $03, $03, $03, 0,0,0,0,0,0,0,0,0,0,0,0
    db $04, $05, $06, $05, $06, $05, $06, $05, $06, $05, $06, $05, $06, $07, $03, $03, $03, $03, $03, $03, 0,0,0,0,0,0,0,0,0,0,0,0
    db $04, $08, $05, $06, $05, $06, $05, $06, $05, $06, $05, $06, $08, $07, $03, $03, $03, $03, $03, $03, 0,0,0,0,0,0,0,0,0,0,0,0
    db $04, $05, $06, $05, $06, $05, $06, $05, $06, $05, $06, $05, $06, $07, $03, $03, $03, $03, $03, $03, 0,0,0,0,0,0,0,0,0,0,0,0
    db $04, $08, $05, $06, $05, $06, $05, $06, $05, $06, $05, $06, $08, $07, $03, $03, $03, $03, $03, $03, 0,0,0,0,0,0,0,0,0,0,0,0
    db $04, $05, $06, $05, $06, $05, $06, $05, $06, $05, $06, $05, $06, $07, $03, $03, $03, $03, $03, $03, 0,0,0,0,0,0,0,0,0,0,0,0
    db $04, $08, $05, $06, $05, $06, $05, $06, $05, $06, $05, $06, $08, $07, $03, $03, $03, $03, $03, $03, 0,0,0,0,0,0,0,0,0,0,0,0
    db $04, $08, $08, $08, $08, $08, $08, $08, $08, $08, $08, $08, $08, $07, $03, $03, $03, $03, $03, $03, 0,0,0,0,0,0,0,0,0,0,0,0
    db $04, $08, $08, $08, $08, $08, $08, $08, $08, $08, $08, $08, $08, $07, $03, $03, $03, $03, $03, $03, 0,0,0,0,0,0,0,0,0,0,0,0
    db $04, $08, $08, $08, $08, $08, $08, $08, $08, $08, $08, $08, $08, $07, $03, $03, $03, $03, $03, $03, 0,0,0,0,0,0,0,0,0,0,0,0
    db $04, $08, $08, $08, $08, $08, $08, $08, $08, $08, $08, $08, $08, $07, $03, $03, $03, $03, $03, $03, 0,0,0,0,0,0,0,0,0,0,0,0
    db $04, $08, $08, $08, $08, $08, $08, $08, $08, $08, $08, $08, $08, $07, $03, $03, $03, $03, $03, $03, 0,0,0,0,0,0,0,0,0,0,0,0
    db $04, $08, $08, $08, $08, $08, $08, $08, $08, $08, $08, $08, $08, $07, $03, $03, $03, $03, $03, $03, 0,0,0,0,0,0,0,0,0,0,0,0
    db $04, $08, $08, $08, $08, $08, $08, $08, $08, $08, $08, $08, $08, $07, $03, $0A, $0B, $0C, $0D, $03, 0,0,0,0,0,0,0,0,0,0,0,0
    db $04, $08, $08, $08, $08, $08, $08, $08, $08, $08, $08, $08, $08, $07, $03, $0E, $0F, $10, $11, $03, 0,0,0,0,0,0,0,0,0,0,0,0
    db $04, $08, $08, $08, $08, $08, $08, $08, $08, $08, $08, $08, $08, $07, $03, $12, $13, $14, $15, $03, 0,0,0,0,0,0,0,0,0,0,0,0
    db $04, $08, $08, $08, $08, $08, $08, $08, $08, $08, $08, $08, $08, $07, $03, $16, $17, $18, $19, $03, 0,0,0,0,0,0,0,0,0,0,0,0
    db $04, $09, $09, $09, $09, $09, $09, $09, $09, $09, $09, $09, $09, $07, $03, $03, $03, $03, $03, $03, 0,0,0,0,0,0,0,0,0,0,0,0
TilemapEnd:
```

6.4 Objects

Previously mentioned in the palette section, the ***Game Boy*** has the capability to color things known as objects. Simply put, an object is a special kind of tile that is different from the previously discussed background tiles, these tiles are able to be edited in more ways than typical background

tiles are, objects are able to be drawn at any on screen position and have additional properties assigned to them referred to as "attributes". These objects are stored in a specific region of the ***Game Boy***'s VRAM called the OAM or Object Attribute Memory. Each object is stored within four bytes, one for the Y-coordinate, one for the X-coordinate, one for the tile ID, and one for the object's attributes. The OAM is 160 bytes long meaning a total of 40 objects are able to be accounted for by the ***Game Boy*** at any given time. One interesting and important piece of information regarding these objects data is that the X and Y coordinate data variables do not actually store the objects on screen position, the on-screen X position is the stored X position minus 8, and the on-screen Y position is the stored Y position minus 16. These offsets exist so that the ***Game Boy*** is capable of edge clipping of objects on the left and top edges of the screen, something previous Nintendo consoles, like the NES, were not capable of.

Utilizing these objects was the key to getting both the ball and the paddle in my game as objects allowing them to be able to be controlled by the player or moved dynamically.

Initialization of objects:

```
    ; Initalize paddle sprite in OAM
    ld hl, _OAMRAM
    ld a, 128 + 16
    ld [hli], a
    ld a, 16 + 8
    ld [hli], a
    ld a, 0
    ld [hli], a
    ld [hli], a
    ; Initalize the ball object
    ld a, 100 + 16
    ld [hli], a
    ld a, 32 + 8
    ld [hli], a
    ld a, 1
    ld [hli], a
    ld a, 0
    ld [hli], a
```

Paddle and Ball tiles:

```
Paddle:
    dw `13333331
    dw `30000003
    dw `13333331
    dw `00000000
    dw `00000000
    dw `00000000
    dw `000000009
    dw `00000000
PaddleEnd:

Ball:
    dw `00033000
    dw `00322300
    dw `03222230
    dw `03222230
    dw `00322300
    dw `00033000
    dw `00000000
    dw `00000000
BallEnd:
```

7 Game mechanics

7.1 Ball Momentum

The ball's momentum is actually quite simple, the ball has an X and a Y momentum variable, these variables are given an initial value and variables update the objects position variable which changes where on the screen the ball is drawn after it is updated in the OAMRAM.

Declaring the ball's momentum variables:

```
SECTION "Ball Data", WRAM0
wBallMomentumX: db
wBallMomentumY: db
```

Setting the ball's initial momentum:

```
    ; Make it so the ball starts going right
    ld a, 1
    ld [wBallMomentumX], a
    ld a, -1
    ld [wBallMomentumY], a
```

Updating the ball's position based off of its momentum:

```
; Add the ball's momentum to its position in OAM
ld a, [wBallMomentumX]
ld b, a
ld a, [_OAMRAM + 5]
add a, b
ld [_OAMRAM + 5], a

ld a, [wBallMomentumY]
ld b, a
ld a, [_OAMRAM + 4]
add a, b
ld [_OAMRAM + 4], a
```

7.2 Collisions

The way the ball's collision is handled is by the GetTileByPixel function:

```
; Convert a pixel position to a tilemap address
; hl = $9800 + X + Y * 32
; @param b: X
; @param c: Y
; @return hl: tile address
GetTileByPixel:
    ; Divide by 8 to convert a pixel position to a tile position
    ; Multiply the Y position by 32
    ; The two previous operations cancel eachother out so we only need to mask the Y value
    ld a, c
    and a, %11111000
    ld l, a
    ld h, 0
    ; This is the position * 8 in hl
    ; position * 16
    add hl, hl
    ; position * 32
    add hl, hl
    ; Offset the X position
    ld a, b
    srl a ; a / 2
    srl a ; a / 4
    srl a ; a / 8
    ; Add the offsets
    add a, l
    ld l, a
    adc a, h
    sub a, l
    ld h, a
    ; Add the offset to the tilemap's base address
    ld bc, $9800
    add hl, bc
    ret
```

Essentially what this function is doing is converting the position of the ball object to a location on the tilemap and getting which tile the ball is currently touching based off of said position, from this the tile the ball is currently touching is known. The tile that is returned from GetTileByPixel is then interpreted by the IsWallTile function which contains a list of the tiles that the ball is able to bounce off of.

```
; @param a: tile ID
; @return z: set if a is a wall
IsWallTile:
    cp a, $00
    ret z
    cp a, $01
    ret z
    cp a, $02
    ret z
    cp a, $04
    ret z
    cp a, $05
    ret z
    cp a, $06
    ret z
    cp a, $07
    ret
```

Then based off of the result of the IsWallTile, either a directional bounce function is called which updates the ball's momentum based upon which directional bounce function is called, shown below is two of the four directional functions.

```
TopBounce:
    ; OAM positions should be offset here
    ; (8, 16) in OAM coordinates is (0, 0) on the screen
    ld a, [_OAMRAM + 4]
    sub a, 16 + 1
    ld c, a
    ld a, [_OAMRAM + 5]
    sub a, 8
    ld b, a
    ; Returns tile address in hl
    call GetTileByPixel
    ld a, [hl]
    call IsWallTile
    jp nz, RightBounce
    ld a, 1
    ld [wBallMomentumY], a

RightBounce:
    ld a, [_OAMRAM + 4]
    sub a, 16
    ld c, a
    ld a, [_OAMRAM + 5]
    sub a, 8 - 1
    ld b, a
    call GetTileByPixel
    ld a, [hl]
    call IsWallTile
    jp nz, LeftBounce
    ld a, -1
    ld [wBallMomentumX], a
```

Finally, a check is done to test for a collision between the ball and the paddle, this collision is handled differently since there are no position conversions that have to be completed here, only position comparisons using the carry flag are used. First, there is a check to see if the ball is low enough to bounce off the paddle, if the ball isn't at the same Y position as the paddle, no bounce occurs. The checks regarding the X positions come in two distinct parts, first 16 is added to the ball's position and if the ball is more than 16 pixels to the right of the paddle, no bounce occurs. Secondly, another check occurs to see if the ball is more than 8 pixels to the left of the paddle and if it is, no bounce occurs.

```
BounceDone:
    ; Check if the ball is able to bounce off the paddle
    ld a, [_OAMRAM]
    ld b, a
    ld a, [_OAMRAM + 4]
    ; Fix for ball sinking into paddle when bouncing bug by adding an offset of 4 pixels
    add a, 4
    cp a, b
    ; If the balls Y position isn't equal to the paddles Y position then no bounce
    jp nz, PaddleBounceDone
    ; Ball's X position
    ld a, [_OAMRAM + 5]
    ld b, a
    ; Paddle's X position
    ld a, [_OAMRAM + 1]
    sub a, 8
    ; Compare ball and paddle's X positions, if they are not within the range no bounce
    cp a, b
    jp nc, PaddleBounceDone
    ; 8 to undo previous check 16 for the paddle's width
    add a, 8 + 16
    cp a, b
    jp c, PaddleBounceDone

    ld a, -1
    ld [wBallMomentumY], a
```

7.3 Player Input

Here we can see the code that checks for player input, updates the current directional input from the player, and moves the paddle according to the direction the player is inputting.

```
; Check left button
CheckLeft:
    ld a, [wCurKeys]
    and a, PADF_LEFT
    jp z, CheckRight
Left:
    ; Move the paddle one pixel to the left
    ld a, [_OAMRAM + 1]
    dec a
    ; If the the edge of the playfield is hit, paddle is not moved
    cp a, 15
    jp z, Main
    ld [_OAMRAM + 1], a
    jp Main

; Check right button
CheckRight:
    ld a, [wCurKeys]
    and a, PADF_RIGHT
    jp z, Main
Right:
    ; Move the paddle one pixel to the right
    ld a, [_OAMRAM + 1]
    inc a
    ; If the the edge of the playfield is hit, paddle is not moved
    cp a, 105
    jp z, Main
    ld [_OAMRAM + 1], a
    jp Main

UpdateKeys:
  ; I honestly have no idea how this input reading code works and neither does the tutorial but it's necessary inorder to be able to read inputs
  ; Poll half the controller
  ld a, P1F_GET_BTN
  call .onenibble
  ld b, a ; B7-4 = 1; B3-0 = unpressed buttons

  ; Poll the other half
  ld a, P1F_GET_DPAD
  call .onenibble
  swap a ; A7-4 = unpressed directions; A3-0 = 1
  xor a, b ; A = pressed buttons + directions
  ld b, a ; B = pressed buttons + directions

  ; And release the controller
  ld a, P1F_GET_NONE
  ldh [rP1], a

  ; Combine with previous wCurKeys to make wNewKeys
  ld a, [wCurKeys]
  xor a, b ; A = keys that changed state
  and a, b ; A = keys that changed to pressed
  ld [wNewKeys], a
  ld a, b
  ld [wCurKeys], a
  ret

.onenibble
  ldh [rP1], a ; switch the key matrix
  call .knownret ; burn 10 cycles calling a known ret
  ldh a, [rP1] ; ignore value while waiting for the key matrix to settle
  ldh a, [rP1]
  ldh a, [rP1] ; this read counts
  or a, $F0 ; A7-4 = 1; A3-0 = unpressed keys
.knownret
  ret
```

8 Conclusion

And that's it! This was unfortunately where I had to stop since I was quickly running out of time after finishing up the collision systems, if I had more time I would have loved to continue developing this game, adding things like blocks actually disappearing when they are hit and a scoring system that actively updates and shows the player their current score while they play. But beyond that, this project was incredibly eye opening to me in terms of just how much work went into the making of these older games. With such limited resources, it really forced developers to get creative with their games in terms of how they could achieve the effects they wanted in the most efficient way possible, these developers had to develop their games around the hardware they were working with, abusing every quirk and oddity they could. I was honestly a bit nervous about taking on this project initially, I knew going into this project that this was not going to be an easy project, I was aware of the struggles and frustrations that other people faced while also attempting to do something similar to me, I knew about the complexity of coding, compiling, and running an original ***Game Boy*** game but despite these fears I took on the challenge. In the end, I'm really glad that I did decide to take on this challenge for this project, I had a lot of fun working on this project even if at times it felt

very frustrating and tedious, just seeing it all slowly come together throughout the development process was very satisfying to me and being able to make a game on hardware this limited was a really uplifting accomplishment for me and kept me wanting to do more.

9 Work Cited:

References

Hardware explanation 1: https://www.youtube.com/watch?v=RZUDEaLa5Nw
Hardware explanation 1.5: https://www.youtube.com/watch?v=t0V-D2YMhrs
Hardware Explanation 2: https://www.youtube.com/watch?v=ecTQVa42sJc
GameBoy ASM Tutorial: https://gbdev.io/gb-asm-tutorial/part1/setup.html
RGBDS Website: https://rgbds.gbdev.io/install
Sublime Text Website: https://www.sublimetext.com/
RGBDS Sublime Text Syntax Package Website: https://packagecontrol.io/packages/RGBDS
Emulicious Website: https://emulicious.net/