# Assembly Tetris

By Finlay Holmes

For AQA A-level Computer Science

# Contents Page

Contents

[Assembly Tetris 1](#_Toc73783073)

[Contents Page 2](#_Toc73783074)

[Analysis 4](#_Toc73783075)

[The Problem 4](#_Toc73783076)

[Stakeholders 4](#_Toc73783077)

[Requirements: 4](#_Toc73783078)

[Assembly Code 5](#_Toc73783079)

[My research on Tetris 5](#_Toc73783080)

[History 5](#_Toc73783081)

[The game 5](#_Toc73783082)

[Interview 6](#_Toc73783083)

[My Questions: 6](#_Toc73783084)

[Have you played much Tetris before? 6](#_Toc73783085)

[In Your opinion what should a Tetris game Be able to do? 6](#_Toc73783086)

[Should the blocks be certain colours for each shape, or should it be chosen randomly? 7](#_Toc73783087)

[Which colours should be included? 7](#_Toc73783088)

[How large should the play area be? 7](#_Toc73783089)

[What controls do you believe may be helpful in the game? 7](#_Toc73783090)

[Success Criteria: 7](#_Toc73783091)

[Primaries: 7](#_Toc73783092)

[Secondaries: 7](#_Toc73783093)

[Design 9](#_Toc73783094)

[The Screen 13](#_Toc73783095)

[Stakeholder Input: 13](#_Toc73783096)

[Technical Solution 14](#_Toc73783097)

[The interrupts 14](#_Toc73783098)

[Testing 26](#_Toc73783099)

[Evaluation 28](#_Toc73783100)

[Condition 1 multiple shapes: 28](#_Toc73783101)

[1.1 each shape must have their own colour: 29](#_Toc73783102)

[1.2 the shapes must be randomly chosen: 29](#_Toc73783103)

[1.3 controllable left/right and rotation: 29](#_Toc73783104)

[1.4 the shapes must stop: 29](#_Toc73783105)

[Condition 2 lines must be removed upon completion: 30](#_Toc73783106)

[2.1 lines above must be moved down: 30](#_Toc73783107)

[Condition 3 the game must end when the shapes have stacked to the top of the play area: 30](#_Toc73783108)

[References 32](#_Toc73783109)

# Analysis

## The Problem

The intention of this project is to improve my understanding of assembly code by using the ARM lite emulator created by Peter Higginson. I intend to do this by recreating the classic game of Tetris in ARM assembly.

My initial idea was to create a Tetris game where the player plays against the computer but have since decided to stick with a simple traditional Tetris game. This program is intended for someone to play a game and pass the time. The system will require the user to be able to move shapes and spin them; also the system must be able to remove completed lines and drop the rest into the spaces which have been cleared. Furthermore, shapes cannot pass through other shapes and must stop on the top of the shape. Finally, a scoring system will be required as a method of competitiveness.

## Stakeholders

The typical client of this software would be someone with a computer and who would be interested in a casual game. Because of a lack of mobile phone controls on the emulator, the users would be limited to a browser on a computer.

Lots of browser games are only accessible to people on a computer so it is not an unusual situation and is considered satisfactory to most users.

To enable the programme to be suitable for the average user, I identified a stakeholder who was representative of the cliental likely to play such a game. The individual whom I interviewed was a middle aged man with no programming skills but who was used to playing games on his computer and would know the needs and requirements of a casual videogame.

### Requirements:

* Blocks to be chosen and shown what is next
* Blocks to collide with one another and the floor
* Blocks to fall downwards
* Completed lines to be removed
* The player to be able to control the blocks by moving them left and right and to rotate them.
* The game must finish if the blocks reach the top

With the completion of these a basic Tetris game will have been produced with a very high game speed as it will be made in assembly code.

I intend to complete all these tasks separately and then combine them as to hopefully simplify the creation of the software and to speed up the debugging of the software.

These requirements by themselves do not seem that complicated, however, I will be doing this in assembly code, a language that I have had no prior knowledge on or how to program in it before.

## Assembly Code

My project required me to learn assembly code from zero prior knowledge of how it worked or what to do with it. To learn assembly, I used a book from Richard Pawson’s computer science from the metal up series, titled: Assembly Language Programming. Many of the programming concepts in high level languages are present in assembly and Peter Higginson’s ARMLite (The emulator I used to write in assembly), however the way you made them was very different. With further focus of memory management than high-level languages, it was important to use as many tricks as possible to get a working program by the end of the project.

## My research on Tetris

### History

This classic video game was developed in 1984 in the former Soviet Union by Alexy Pajitnov. The original game was written on an Electronica 60 computer which lacked a graphical interface and so the shapes consisted of brackets and other ascii symbols. The Russian state became interested in his project and he received additional funding which enabled him to use an IBM PC with graphic capability. As a result he developed the tetris game which we are now familiar with and allowed the game to be sold the West.

### The game

In the game there are 7 shapes that can be used: a square, a line, a T, an S, a Z, an L and an r. These shapes can be spun round and all but 2 have 4 different turn shapes. The line and the square both have less than this as with the square, if you spin it, you only get the same orientation as it is symmetrical on both plains. The line, however, has 2 turn shapes as it is not symmetrical in one of the plains. The screen of the game is relatively simple. In most iterations of Tetris, there is a play area in the middle that is taller than it is wide, this is where the game will be played, and blocks will drop. On the right, there might be a description of what the next block will be. On the left, there may be a location where the player can hold their current block for later.

Blocks can be described as geometric shapes made of 4 pixels. In a classic game of Tetris, the shapes are described as: “T”, ”I”, ”O”, ”L”, ”r”, ”S” and ”Z”. The letters describe the shape of the blocks as the shapes look similar to how the letters are shaped. Interestingly the “O” refers to a square shape.

In the game, you are given control of a block that descends from the top of the screen. The user can move it left and right and also may spin it into different orientations to allow them to fit in a space. It will then land either at the bottom of the screen or it will land on top of a block that has already landed. Once this happens, the game gives you a new block at the top of the screen. Once you fill out a horizontal line with your blocks, that line is removed and the lines above are moved down one line. If the user is to fail at this, the blocks will fill up to the top of the screen and the game will end with a game over.

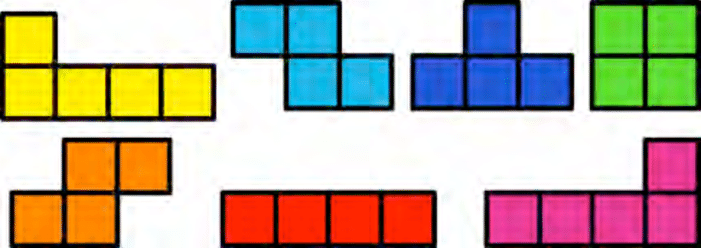


Figure 1: This shows the standard shapes from left to right, top to bottom, r,z,t,o,s,I,l

## Interview

For this interview, I need to know what a user would expect to see in a Tetris game. On top of that, it could be helpful to know what type of controls would be comfortable for Tetris whilst playing on a computer. The questions will outline the user’s opinions of Tetris and will help assist me with the design of the game.

### My Questions:

1. Have you played much Tetris before?
2. In your opinion, what should Tetris be able to do?
3. Should the blocks in Tetris be certain colours for each shape or just randomly chosen?
4. What colours should be included?
5. How large should the play area be?
6. What controls do you think would be helpful in the game?
7. Do you have anything else to say?

Question 1 was to learn the user’s history with Tetris and to get an idea on how biased they might be for the future questions.

Question 2 investigates a user’s expectations with the game and gives the user a chance to express any problems they may have had in the past with Tetris.

Questions 3-5 were designed to assist in decisions on how the game would look and play as I believe that the user would know more about what could be better looking in the user’s point of view.

Finally, the Questioning is concluded with a chance for the user to input anything else they see as necessary.

### Have you played much Tetris before?

“Yes, I have played Tetris before at an arcade and on my phone.”

### In Your opinion what should a Tetris game Be able to do?

“A Tetris game should be able to produce multiple shapes that can be swapped in and out of a bank at the side. These blocks should also fall at increasing speeds dictated by the number of lines completed with a small sound to play and notify you that you have completed a line. Furthermore, the next block should be displayed to the side of the play area, and the blocks should be chosen at random.”

Some of these requirements will be tough to implement but will still be viable to create these functions. Peter Higginson’s ARMlite does not, as of writing this, allow for the use of sound effects so I’m afraid that I will not be able to introduce a sound system into this.

### Should the blocks be certain colours for each shape, or should it be chosen randomly?

“Each shape should be given a defined colour so that players only need to recognise the colour when it shows up which will allow for greater reactions and speed while playing the game.”

### Which colours should be included?

“Preferably the shapes should consist of bright identifiable colours for example: Red, Yellow, Blue, Green, Purple, Orange, Pink.”

### How large should the play area be?

“The play area should allow for ease of creating lines and a lot of space to drop from. Because of this, I believe that the board should be comprised of 8:20 ratio.”

### What controls do you believe may be helpful in the game?

“the controls of the game should be easy to use and understand. And simple to make playing the game for long periods of time simple. Because of this, a maximum of 5 controls should be used: moving left and right, spinning the shape and storing it and maybe a button to send it straight down.”

With these important questions answered, it is now clearer on what a typical user would expect to find if they were looking for a Tetris game to play. With clearer goals, we now know what our success criteria can be.

## Success Criteria:

I believe that in order for this project to be considered a success, there must be:

### Primaries:

I believe these to be of top priority and the game will not be complete without these items

1. Multiple shapes
   1. Each with their own colour
   2. Randomly chosen
   3. Controllable
      * 1. Left and right
        2. Rotate
   4. The shapes must fall downwards and stop
      * 1. At the bottom
        2. On top of other shapes that are already there
2. Lines must be removed upon completion
   1. The lines above must be moved down after
3. The game must end when it reaches the top

### Secondaries:

These are the items that I believe are not totally necessary in order to call this a success but may be added if I have time

1. A scoreboard where one point is added whenever a line is removed. If this is done, I would recommend doing in the input/output box as drawing possibly 3-digit numbers in pixel form may be tough.
2. A box that displays the next shape to come up.
3. A place to hold the current shape and use later.
4. Ability to speed up the rate of decent as the game progresses

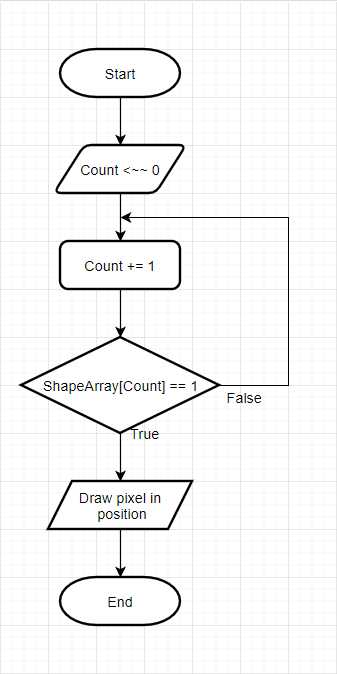
# Design

For the game to function appropriately, it will require several features such as a way of stopping blocks when they hit the floor or another block, we also want the blocks to have the ability to spin and be placed as such, on top of that, there will be a need for a system of storing one block to use later, furthermore a function to remove a completed line will be needed and finally the game must have a scoring system as a way of making the game more competitive and fun.

In Tetris there are several different types of blocks that can fall. I hope to implement these multiple blocks by storing the shapes as an array of 9 elements and running through it one by one placing pixels where the array dictates.

|  |  |  |
| --- | --- | --- |
| 0 | 1 | 1 |
| 0 | 0 | 1 |
| 0 | 0 | 1 |

Figure 2: In this example, to create the L shape, the array would store this as: 011001001



The function would then be run 9 times over each pixel.

It must be remembered, however, that after 3 pixels have been read, that you must add on the other 64 pixels to go down to the next line.

This does, however, present a problem of drawing the shapes that are not actually 3x3 in size for example, the “I” shape is a 4x1 shape which would not fit easily in a 3x3 area. To combat this, a check may be added each time to see if the shape is actually 3x3 or something else.

The colour of the shape can be stored in the same array and can therefore easily be retrieved before the shape is drawn with some indirect addressing.

Figure 3. flow chart describing the drawing algorithm

This leaves the problem of how to allow blocks to spin upon a user input. To tackle this, I had three ideas:

* To have 4 arrays for each shape
* To implement 4 ways of reading the array
* An array for each shape and for each rotation to state which pixel in the shape array to go to next.

The first idea would work where (for the shape above) it would have 4 arrays to represent itself which could be in a 2-dimensional array so that every time the user inputs a spin the function will use the array shown by the number in one of the registers.

The second option is to have 4 different functions, one for each direction. This would mean that less shapes would have to be stored in arrays and so less effort would have to be used by the creator. The four functions can be described as follows:

|  |  |  |
| --- | --- | --- |
| 1 | 2 | 3 |
| 4 | 5 | 6 |
| 7 | 8 | 9 |

This shows the shape box in its original form with the red box showing where the top is.

This function will work like the top with the function looping over each element in the array in order of appearance.

|  |  |  |
| --- | --- | --- |
| 7 | 4 | 1 |
| 8 | 5 | 2 |
| 9 | 6 | 3 |

This function would have to utilise an idea of using the element of -3(x-1) + {7,8,9} for each row, so it would loop over every element in the array but in a different order

|  |  |  |
| --- | --- | --- |
| 9 | 8 | 7 |
| 6 | 5 | 4 |
| 3 | 2 | 1 |

This one would be the same as the first one but in reverse, so instead of counting up the array, you’d count down it.

|  |  |  |
| --- | --- | --- |
| 3 | 6 | 9 |
| 2 | 5 | 8 |
| 1 | 4 | 7 |

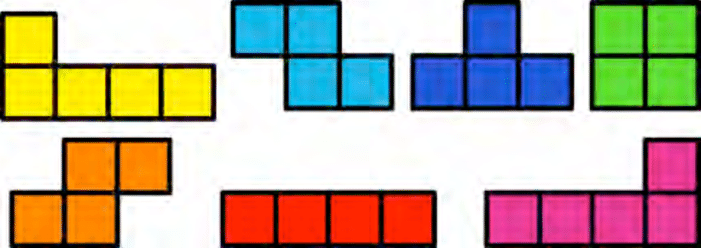
This one is not unlike the second one in idea as it will work the same as it but with a different function: 3(x-1) – {3,2,1}

Finally, these functions would have to be assigned values to be chosen by a counting loop.

The last idea was to have an array for the shape and 4 arrays to describe the direction it will face. This will mean that only one function is needed to draw the shapes and it limits the number of arrays needed to be written. However, the rotation indexes would need to be calculated and mistakes can make the shape look very weird.

All of these plans would require an index for the rotation it’s on so that the shapes are rotated the correct way round and not just sitting there doing nothing.

Personally, I went for the final method as it presented a simple algorithm and gave me options to edit it easily in the future if I so required.



The shapes used will all require their own array for their original start position but 2 of them (square and I) will require separate types as otherwise they may not fit in the 3x3 area and so will require separate functions for drawing them and their rotations. Furthermore, both the square and I shapes, if allowed to just use 4 bits will appear the same as both are four 1s. The way this can be implemented is with a simple comparison before drawing the shape and if the shape is either a line or a square then it will draw the corresponding shape. It will be hard coding each of the odd shapes in their own functions to avoid overcomplicating the program as I have already coded the shape’s drawing function to fit the general 3x3 shapes. If I were to redo the program, I would take account of this by adding another identifier at the end of each array to describe the dimensions of the shape, this would also fix the problem of shapes hitting the side of the arena without there actually being anything there because the shape is being affected by the 3rd empty row.

In Tetris, the blocks must stop moving down when the block comes in to contact with another block or when it reaches the bottom of the screen. I intend to do this by checking the pixel below to make sure it is clear or not too low. This seems simple enough, but it must also check the pixels under all the pixels as it must slot into spaces. On top of that it must check below different blocks when it spins. This means I need a function to check under the correct block possibly by checking if it has got a space below it in its array. If the check function is run while the shape isn’t drawn (after it has been erased), then the fact that it might get stuck on itself is mitigated. To make sure the spins land the right way up, the shape can only check the pixels where the shape will be. This can be done by drawing an “Invisible” version of the shape, done by using the same function as drawing it but instead of storing the pixel it can be loaded and the colour checked against the background colour.

With the removal of the lines, I fear that the emulator would not run fast enough to run through each pixel to make sure it is a complete line. Therefore, I believe it to be best that after a shape has been set (stopped moving down/being the shape controlled by the user) the three lines (or four if it is needed) can be checked to see if every pixel is not the same as the background colour. If a single pixel is found to be the same as the background, then it can be moved on to the line below and then if all three lines have a grey pixel, then nothing will change. Otherwise, the completed line(s) will be removed, and the blocks above will be moved down however many lines were removed.

To move the blocks above down. I can hold the number of removed lines as a variable and then work up each line, moving them down until I reach the top, at which point, I will re draw however many lines are needed at the top.

The Game must end once the blocks have been placed all the way to the top of the play area. But this is an easy problem to solve as all that is required is a simple check whenever a new shape is about to be drawn, to check if it will land on a shape that is already there. This is similar logic to the function earlier, where we checked for the shape to be hitting shapes on the bottom. This is different because instead of setting the shape, we must display a game over screen

Getting the shape to be drawn at regular intervals is easy enough as all that is required is to use a clock interrupt, whereby, whenever the clock hits a specific time, the interrupt pin will be signalled of a clock interrupt and the stored function will be run. It is during this time that I will be deleting the shape and moving it down one line. This is also where the shape’s next state is checked and the “Invisible” shape from before is drawn.

There will also need to be a keyboard interrupt to find out if the user has inputted one of the keys for a movement. The key pressed will have to be discovered during the interrupt and if the key pressed is not used as an actual movement key, then the interrupt must be ended as to not waste too much time.

## The Screen

ARM-lite gives the user a sample screen of 64x48 pixels. With this, I intend to use the centre 32 pixels to hold the shapes. The column to the right will include the next shape and to the left will be a score system.

Figure 4: Image of screen: 1:4 ratio

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |

This will mean that I must also not allow the blocks to travel too far left or right, confining the moving shapes to the central area. This can be achieved with a simple position check.

Score Play Area Next

Piece

I intend for the play area to have a black border on the left and right as it will make seeing the edge easier when a shape is there already and will define the area of play.

For ease of use, I believe that the game should be controlled by 3 buttons. ARMLite only allows for keyboard inputs from the letters so I am limited to only 26 buttons, which is still more than is needed.

The three buttons needed are for the controls of: left movement; right movement and rotation. For simplicity, to enable the player to instinctively know which button to press, I believe that the button for moving left should be on the left of the keyboard and on the right for moving right. To avoid a user stretching their hand too far, I believe that the rotating key should be in between the left and right key. Because of this, I have chosen the ‘W’, ‘A’, and ‘D’ keys as they support the hand nicely and the controls are intuitive enough for any user to use. Furthermore, many games on computers use these keys as well, so people may be more familiar to using them.

## Stakeholder Input:

I gave the stakeholder a copy of the design for the user interface of the game and intended user controls to see their opinion on it.

The stakeholder thinks that the play area seems to have good spacing and will be sufficient to allow people to move the shapes from one side to another in time to fill out each line.

He also believed that the controls are as good as possible for what I am limited to, as I has explained to him that, to the extent of my knowledge, the arrow keys on the keyboard were not possible keys to be used in ARMLite.

# Technical Solution

## The interrupts

In order for the game to run, interrupts were used to refresh the page each time the clock hit a certain count. This allowed for greater control of the speed the shapes fell from the top of the screen:

 1|// Set up Interrupt handling

  2|      MOV R12,#repeat

  3|      STR R12,.ClockISR

  4|      MOV R12, #200     //sets the clock interrupt frequency

  5|      STR R12,.ClockInterruptFrequency

  6|      MOV R12, #Move    //Prepares Keyboard interrupt

  7|      STR R12, .KeyboardISR //sets it

  8|      MOV R12, #1

  9|      STR R12, .KeyboardMask

Figure 5: this shows the interrupt handling where the interrupts that will be used are setup and tells the clock interrupt its interrupt frequency (how fast it will refresh). However, the interrupts have not been allowed to be accessed yet as the interrupt register has not been set to one yet. This is so that the interrupts are not called whilst the shape is being drawn and shown on screen. In order to start the interrupt, we allow it in the mainloop function.

10|//starts the program and initialises the pixel screen

 11|      MOV R2, #.grey    //gets colour

 12|      MOV R0, #20       //start position

 13|      BL setPlay        //runs the function to make the entire screen grey

 14|      MOV R4, #0        //sets R4 to 0 array index

 15|      MOV R6, #SpinOne  //sets to the first spin array

 16|      BL randomizedCheck //gets the shape

 17|      MOV r8, #0        //sets R8 to 0

 18|      MOV R0, #31       //resets pixel index to middle

 19|      MOV R1, #0        //sets y to 0

 20|      b mainLoop        //runs the repeating function

 21|mainLoop:

 22|      MOV R12,#0        // stops it from repeating while drawing

 23|      STR R12, .InterruptRegister

 24|      MOV R8, #0        //resets the shape's line counter to 0

 25|      MOV R4, #0        //resets the array count to 0

 26|      MOV R2, #36       //gets the colour index

 27|      LDR R2, [R7+R2]   //sets the colour of the pixel

 28|      SUB R0, R0, #1    //moves x back one

 29|      bl shapeDrawer    //draw shape

 30|      MOV R12,#1

 31|      STR R12,.InterruptRegister //Enable all interrupts

 32|      b mainLoop        //repeating function

Figure 6: This is the mainloop function and some commands that are run before the program starts to make sure that it works. In the pre-program section, it is seen that the set play function is run, this will draw the screen design shown in the design section. Next, we set the spin array in R6, this is so that the shape starts out on the correct spin function. This is followed by the “randomizedCheck” function that can be returned from. This function will get a random shape and store it in the register used for storing the shape array. This will be a permanent register, only used for the shape array, a global variable. Then the drawing and coordinate registers are set to their required positions. At this point we are now in the mainloop function.

This function is repeated throughout the length of the entire program until the game ends upon losing. It may leave the function in an interrupt, but this is temporary. The first 2 lines in the function are used to prevent the user or system calling an interrupt whilst it is still drawing the shapes. Then, the registers are set to their required numbers for creating the shapes and the colour required for the shape is fetched from the shape array. Now, the shape is drawn and interrupts are reactivated to allow the shape to be moved, spun and erased.

165|//

166|//initial screen

167|//

168|setPlay:

169|      MOV R3, #.PixelScreen //sets pixel screen

170|      PUSH {R0-R12,LR}  //pushes stack

171|      CMP R0, #45       //is at edge of play area?

172|      BEQ setterDown    //go down a line

173|      LSL R5, R0, #2    //get x word

174|      LSL R4, R1, #8    //get y word

175|      ADD R5, R4, R5    //add them

176|      STR R2, [R5+R3]   //draw pixel

177|      ADD R0, R0, #1    //move right one pixel

178|      B setPlay         //repeat

179|setterDown:

180|      ADD R1, R1, #1    //move down one

181|      CMP R1, #48       //is R1 at bottom

182|      BEQ barsetter     //create black bars

183|      MOV R0, #20       //move x back to left

184|      B setPlay         //repeat

185|barsetter:

186|      MOV R1, #0        //move y back to top

187|      MOV R0, #19       //move x to left of play area

188|barDown:

189|      MOV R2, #.black   //set colour

190|      LSL R4, R0, #2    //get x word

191|      LSL R5, R1, #8    //get y word

192|      ADD R5, R4, R5    //add them for pixel index

193|      STR R2, [R3+R5]   //draw pixel

194|      ADD R1, R1, #1    //move down one

195|      CMP R1, #48       //is at bottom?

196|      BEQ blackBarDone  //move

197|      B barDown

198|blackBarDone:

199|      MOV R1, #0        //set y to top

200|      CMP R0, #45       //already done both?

201|      BEQ END           //end

202|      MOV R0, #45       //Move to right side of play area

203|      B barDown         //repeat

204|//

205|//end initial screen

206|//

207|//

Figure 7: This dictates the initial screen of the program. The first two function show the drawing of the grey background whilst the other three show the two side bars that are in black on the side of the shape.

207|//

208|//Draw shape

209|//

210|shapeDrawer:

211|      ADD R0, R0, #1    //increases x pos by 1

212|      ADD R8, R8, #1    //vert counter

213|      CMP R8, #4        //sees if vert counter is done

214|      BEQ Down          //moves down

215|      CMP R4, #36       //checks if shape is done

216|      BEQ stop          //stops

217|      LDR R5, [R4 + R6] //gets spin index

218|      LDR R5, [R7 + R5] //loads if there is a pixel required here

219|      ADD R4, R4, #4    //increases word count

220|      CMP R5, #0        //checks if a pixel if required here

221|      BEQ shapeDrawer   //repeats function

222|      PUSH {R3-R12, LR}

223|      BL drawPixel      //draws pixel

224|      POP {R3-R12, LR}

225|      B shapeDrawer     //repeats function

226|      HALT              //stops for error

227|Down:

228|      MOV R8, #0        //resets line counter

229|      ADD R1, R1, #1    //increases y-coord

230|      SUB R0, R0, #4    //sets x-coord back to left

231|      B shapeDrawer     //repeats drawing function

232|stop:

233|      SUB R1, R1, #3    //moves back to top

234|      RET               //returns to the program

235|drawPixel:

236|      MOV R3, #.PixelScreen //sets the pixel screen

237|      LSL R4, R0, #2    //multiplies x-coors by 4

238|      LSL R5, R1, #8    //multiplies y-coords by 256

239|      ADD R5, R5, R4    //adds together to get the position

240|      STR R2, [R3+R5]   //draws pixel

241|      RET               //returns

242|//

243|//End draw shape

244|//

Figure 8: Here is the section of code that draws the shapes. This section may have been the most important as it is, more or less, the whole point of Tetris. It should be noted that the stop function is not solely used by the drawing function but is located here because it was the first place I had needed it. The program starts off with the x coordinate being incremented. This is to allow for the x coordinate to be looped through each pixel on each repeat. This is the reason the x coordinate is decreased every time the shape draw function is called. Next the “vert counter” register is incremented and checked against the number four. This is so that the function knows when it needs to start drawing the next line of the program. Then the word counter checks if the shape is finished. The word counter exists so that the right pixel is chosen and so we know when the shape is finished. Now we see the function whereby the pointed pixel is checked to see if the shape needs a pixel drawn here. It does this by first getting the spin index that tells the program which pixel in the shape array should be drawn. This is then followed by getting said index and if it reads a one, the pixel will be drawn.

Otherwise, it will repeat the program. When drawing the pixel we see the command: MOV R3, #.PixelScreen. This sets register 3 to be the index location of the start of the screen. This is needed because the way a screen works in assembly is by stating what you would like the starting and ending index will be and then, when drawing a pixel, you store a colour value into the corresponding pixel location. We multiply the x coordinates by 4 because we need to account for the fact that each memory location in the main memory requires 4 to be incremented by one. This is the same with the y coordinate, but we multiply by 256 because one line is 64 pixels across, and we must multiply this by four to account for the words as well. It is for this reason as well that that the value in R4 in the original stack is increased by four so that the list is accumulating the correct value.

In the mainloop function, the shape will be drawn over itself over and over, until the system receives a clock interrupt, where the shape will be covered by a grey version of itself and then moved down one line, checked, and then, if no blocks detected, it will return back to the original mainloop function.

150|repeat:

151|      MOV R8, #0        //resets the shape's line counter to 0

152|      MOV R4, #0        //resets the array count to 0

153|      MOV R2, #.grey    //set colour to clear it

154|      SUB R0,R0,#1      //move back one pixel

155|      BL shapeDrawer    //clears the shape

156|      MOV R8, #0        //resets the shape's line counter to 0

157|      MOV R4, #0        //resets the array count to 0

158|      ADD R1,R1,#1      //moves down line

159|      SUB R0, R0, #1    //move back one pixel

160|      PUSH {R0-R12, LR} //push stack

161|      BL checkBelow     //draw "invisible" shape

162|      POP {R0-R12, LR}  //return stack

163|      ADD R0, R0, #1    //move right one

164|      RFE

Figure 9: This function, as stated before, is only run on a clock interrupt. It can be seen that the colour of the shape is set to grey and the drawn over the original and then the registers are all shifted on to a stack, to conserve the values and to return the coordinates once it’s done. The checkBelow does as it says, checks the pixels one below the original. It was decided to be put here so that the shape did not get confused by the presence of itself being there and then cancelling its movement.

 33|//

 34|//checker stuff

 35|//

 36|checkBelow:

 37|      ADD R0, R0, #1    //increases x by 1

 38|      ADD R8, R8, #1    //increases horizontal counter

 39|      CMP R8, #4        //checks if the shape is done with its line

 40|      BEQ DownCheck     //moves down

 41|      CMP R4, #36       //checks if shape is done

 42|      BEQ stop

 43|      LDR R5, [R4 + R6] //loads the rotation index

 44|      LDR R5, [R7 + R5] //loads if there is a pixel required here

 45|      ADD R4,R4,#4      //increases word count

 46|      CMP R5, #0        //checks if there is not supposed to be pixel

 47|      BEQ checkBelow    //repeats

 48|      PUSH {R3-R12, LR} //adds stack

 49|      BL isDone         //checks if it has anything below it

 50|      POP {R3-R12, LR}  //returns stack

 51|      B checkBelow      //repeats

 52|isDone:

 53|      CMP R1, #48       //checks if it has gone below the screen

 54|      BEQ setIt         //sets shape

 55|      MOV R3, #.PixelScreen //sets the pixel screen

 56|      LSL R4, R0, #2    //gets words for x axis

 57|      LSL R5, R1, #8    //gets words for y axis

 58|      ADD R5, R4, R5    //adds x and y to get position

 59|      LDR R6, [R3+R5]   //loads the colour of that pixel

 60|      MOV R2, #.grey    //sets r2 to background

 61|      CMP R6, R2        //checks if fg matches bg

 62|      BNE setIt         //sets it

 63|      RET               //returns

 64|DownCheck:

 65|      MOV R8, #0        //resets line counter

 66|      ADD R1, R1, #1    //increases y-coord

 67|      SUB R0, R0, #4    //sets x-coord back to left

 68|      B checkBelow      //repeats function

Figure 10: This function is designed to detect when the shape can no longer be moved down. The conditions of this are: if the shape has reached the bottom of the screen; if the shape is about to land on another one that is already there. The height detector is done on line 53 where the y coordinate is compared with 48 (one below the bottom row). Only the pixels that are actually going to exist are checked so the ones that, after drawing, will remain blank will not be checked for anything. If it is found to be at the bottom, it will be ‘set’, which is when the shape stops being controlled by the user and is now no longer moving down. Next, it must check if the shape is about to overlap another shape. This is done between lines 53 to 55. The pixel index and rotation index are both collected and its location on the screen is found in its correct word form. It will then load the colour of that pixel into a register. This is now compared to the background colour and if they are not the same then the object is set.

 69|setIt:

 70|      POP {R3-R12, LR}  //returns 2 separate stacks

 71|      POP {R0-R12, LR}

 72|      SUB R1, R1, #1    //moves the shape up 1

 73|      MOV R8,#0         //sets shape counter to 0

 74|      MOV R2, #36       //gets colour of shape

 75|      LDR R2, [R7+R2]

 76|      PUSH {R3-R12, LR} //stack

 77|      BL shapeDrawer    //draws shape

 78|      POP {R3-R12, LR}  //returns stack

 79|      PUSH {R3-R12, LR} //stack

 80|      SUB R1, R1, #2    //move up 2

 81|      MOV R3, #.PixelScreen //sets pixel screen

 82|      MOV R8, #0        //sets the line index

 83|      BL lineDumpCheck  //checks if line is full of pixels

 84|      POP {R3-R12, LR}  //returns stack

 85|      MOV R0, #31       //puts x axis in middle

 86|      MOV R1, #0        //puts y axis to top

 87|      MOV R8, #0        //resets line counter

 88|      MOV R10, #0       //resets rotation counter

 89|      MOV R6, #SpinOne  //resets spin

 90|      bl randomizedCheck //gets new shape

 91|      MOV r8, #0        //sets R8 to 0

 92|      MOV R0, #31       //resets pixel index to middle

 93|      MOV R1, #0        //moves the shape to top

 94|      b mainLoop        //runs the repeating function

Figure 11: This is the function that sets the shapes and makes it so that they are no longer being controlled by the user. The first 2 lines are there to account for the stacks that were left behind in the checking function. This is followed by the drawing of the shape one line above the thing that blocked it. Then it checks each line it edited when it landed. Now it will produce a new random shape and start at the top of the screen to drop from the top all over again. Finally, it will repeat the mainloop program all over again.

This process will keep repeating over and over again until, either a line is filled, or the shape reaches the top of the screen.

 96|///

 97|///checkLine

 98|///

 99|lineDumpCheck:

100|      ADD R0, R0, #1    //moves right one pixel

101|      CMP R0, #44       //checks if it has finnished the line

102|      BEQ LineFull      //removes the line

103|      LSL R5, R0, #2    //gets the pixel index

104|      LSL R4, R1, #8

105|      ADD R5, R5, R4

106|      LDR R5, [R3+R5]   //loads the pixel colour

107|      MOV R2, #.grey    //checks if it is the same as the background

108|      CMP R5, R2

109|      BEQ notFull       //if it is grey then it will move down a line and go again

110|      B lineDumpCheck   //repeats

111|notFull:

112|      MOV R0, #20       //sets x-coordinate to left side of screen

113|      ADD R8, R8, #1    //increases line counter

114|      CMP R8, #4        //has it done every line of the shape just placed

115|      BEQ stop          //stop removing the shape

116|      ADD R1, R1, #1    //increases y coordinate

117|      B lineDumpCheck   //repeats

118|LineFull:

119|      MOV R2, #.white   //sets colour to white

120|      MOV R0, #20       //sets x coordinate to left side of play area

121|removeLine:

122|      LSL R5, R0, #2    //gets pixel index

123|      LSL R4, R1, #8

124|      ADD R5, R5, R4

125|      STR R2, [R3+R5]   //draws pixel as white

126|      ADD R0, R0, #1    //increases x coordinate

127|      CMP R0, #45       //has it reached the other side?

128|      BNE removeLine    //if not it will repeat

129|      B notFull         //move down a line.

130|doneChecking:

131|      HALT

132|//

133|//end checker stuff

134|//

Figure 12: This is the line removal function. This works by checking each pixel on every y-coordinate layer that the latest block was just placed on. It will loop through every pixel on the line until it finds a pixel that is the same as the background. If a pixel is the same as the background colour, then we know that the line hasn’t been finished and we can stop searching through that line. Conversely, if the checking pointer can make it all the way to the other side, we can then move on to erasing the line from the screen. This is done between lines 118 and 129 inclusive, all that is done here is it will loop through the entire line again but instead of loading the colour into a register, the pixel’s colour will be changed to white and then it will move down the rest of the screen.

The shape must be controlled by the user as well. This is done in the following block of code:

264|//

265|//Move

266|//

267|setKey:

268|      MOV R12, #0

269|      STR R12, .InterruptRegister

270|      MOV R8, #0        //resets the shape's line counter to 0

271|      MOV R4, #0        //resets the array count to 0

272|      MOV R2, #.grey

273|      SUB R0, R0, #1

274|      BL shapeDrawer    //deleats the shape

275|      CMP R11, #65      //if key is A it will move left

276|      BEQ leftMove

277|      CMP R11, #68      // if key is D it will move right

278|      BEQ rightMove

279|      CMP R11, #87      // if key is W then it will increase the rotation register

280|      BEQ Spin

281|      RFE

282|Move: LDR R11, .LastKey //loads the last key input from the keyboard

283|      CMP R11, #65      //compares it with letter A

284|      BEQ setKey        //moves to setkey function

285|      CMP R11, #68      //compares key with D

286|      BEQ setKey        //moves to set key function

287|      CMP R11, #87      //compares with W

288|      BEQ setKey

289|      RFE               //returns from interrupt if no relevent keys pressed

290|leftMove:

291|      SUB R0, R0, #1

292|      BL LeftCheck

293|      RFE

294|LeftCheck:

295|      ADD R0, R0, #1    //increases x by 1

296|      ADD R8, R8, #1    //increases horizontal counter

297|      CMP R8, #4        //checks if the shape is done with its line

298|      BEQ DownCheckLeft //moves down

299|      CMP R4, #36       //checks if shape is done

300|      BEQ acceptL

301|      LDR R5, [R4 + R6] //loads the rotation index

302|      LDR R5, [R7 + R5] //loads if there is a pixel required here

303|      ADD R4,R4,#4      //increases word count

304|      CMP R5, #0        //checks if there is not supposed to be pixel

305|      BEQ LeftCheck     //repeats

306|      PUSH {R3-R12, LR} //adds stack

307|      BL isDoneLeft     //checks if it has anything next to it

308|      POP {R3-R12, LR}  //returns stack

309|      B LeftCheck       //repeats

310|isDoneLeft:

311|      CMP R0, #22

312|      BEQ NopeLeft

313|      MOV R3, #.PixelScreen //sets the pixel screen

314|      LSL R4, R0, #2    //gets words for x axis

315|      LSL R5, R1, #8    //gets words for y axis

316|      ADD R5, R4, R5    //adds x and y to get position

317|      LDR R6, [R3+R5]   //loads the colour of that pixel

318|      MOV R2, #.grey    //sets r2 to background

319|      CMP R6, R2        //checks if fg matches bg

320|      BNE NopeLeft      //sets it

321|      RET               //returns

322|DownCheckLeft:

323|      MOV R8, #0        //resets line counter

324|      ADD R1, R1, #1    //increases y-coord

325|      SUB R0, R0, #4    //sets x-coord back to left

326|      B LeftCheck       //repeats function

327|NopeLeft:

328|      POP {R3-R12, LR}

329|      RFE

330|acceptL:

331|      SUB R0, R0, #1

332|      RET

333|//moving right

334|rightMove:

335|      ADD R0, R0, #1

336|      BL RightCheck

337|      RFE

338|RightCheck:

339|      ADD R0, R0, #1    //increases x by 1

340|      ADD R8, R8, #1    //increases horizontal counter

341|      CMP R8, #4        //checks if the shape is done with its line

342|      BEQ DownCheckRight //moves down

343|      CMP R4, #36       //checks if shape is done

344|      BEQ acceptR

345|      LDR R5, [R4 + R6] //loads the rotation index

346|      LDR R5, [R7 + R5] //loads if there is a pixel required here

347|      ADD R4,R4,#4      //increases word count

348|      CMP R5, #0        //checks if there is not supposed to be pixel

349|      BEQ RightCheck    //repeats

350|      PUSH {R3-R12, LR} //adds stack

351|      BL isDoneRight    //checks if it has anything next to it

352|      POP {R3-R12, LR}  //returns stack

353|      B RightCheck      //repeats

354|isDoneRight:

355|      CMP R0, #40

356|      BEQ NopeRight

357|      MOV R3, #.PixelScreen //sets the pixel screen

358|      LSL R4, R0, #2    //gets words for x axis

359|      LSL R5, R1, #8    //gets words for y axis

360|      ADD R5, R4, R5    //adds x and y to get position

361|      LDR R6, [R3+R5]   //loads the colour of that pixel

362|      MOV R2, #.grey    //sets r2 to background

363|      CMP R6, R2        //checks if fg matches bg

364|      BNE NopeRight     //sets it

365|      RET               //returns

366|DownCheckRight:

367|      MOV R8, #0        //resets line counter

368|      ADD R1, R1, #1    //increases y-coord

369|      SUB R0, R0, #4    //sets x-coord back to Rightt

370|      B RightCheck      //repeats function

371|NopeRight:

372|      POP {R3-R12, LR}

373|      RFE

374|acceptR:

375|      SUB R0, R0, #1

376|      RET

377|//

378|//end Move

379|//

380|//

381|//spin

382|//

383|Spin:

384|      ADD R10, R10, #1  //move onto next spin

385|      CMP R10, #1       //checks which spin its on

386|      BEQ changeTwo     //changes to the correct spin

387|      CMP R10,#2

388|      BEQ changeThree

389|      CMP R10, #3

390|      BEQ changeFour

391|      MOV R10, #0       //reset back to 0 when loop finished

392|      MOV R6, #SpinOne  //sets r6 to spin one

393|      RFE

394|changeTwo: MOV R6, #SpinTwo //sets the spin

395|      RFE               //return from exception

396|changeThree: MOV R6, #SpinThree

397|      RFE

398|changeFour: MOV R6, #SpinFour

399|      RFE

400|//

401|//end spin

402|//

322|DownCheckLeft:

323|      MOV R8, #0        //resets line counter

324|      ADD R1, R1, #1    //increases y-coord

325|      SUB R0, R0, #4    //sets x-coord back to left

326|      B LeftCheck       //repeats function

327|NopeLeft:

328|      POP {R3-R12, LR}

329|      RFE

330|acceptL:

331|      SUB R0, R0, #1

332|      RET

333|//moving right

334|rightMove:

335|      ADD R0, R0, #1

336|      BL RightCheck

337|      RFE

338|RightCheck:

339|      ADD R0, R0, #1    //increases x by 1

340|      ADD R8, R8, #1    //increases horizontal counter

341|      CMP R8, #4        //checks if the shape is done with its line

342|      BEQ DownCheckRight //moves down

343|      CMP R4, #36       //checks if shape is done

344|      BEQ acceptR

345|      LDR R5, [R4 + R6] //loads the rotation index

346|      LDR R5, [R7 + R5] //loads if there is a pixel required here

347|      ADD R4,R4,#4      //increases word count

348|      CMP R5, #0        //checks if there is not supposed to be pixel

349|      BEQ RightCheck    //repeats

350|      PUSH {R3-R12, LR} //adds stack

351|      BL isDoneRight    //checks if it has anything next to it

352|      POP {R3-R12, LR}  //returns stack

353|      B RightCheck      //repeats

354|isDoneRight:

355|      CMP R0, #40

356|      BEQ NopeRight

357|      MOV R3, #.PixelScreen //sets the pixel screen

358|      LSL R4, R0, #2    //gets words for x axis

359|      LSL R5, R1, #8    //gets words for y axis

360|      ADD R5, R4, R5    //adds x and y to get position

361|      LDR R6, [R3+R5]   //loads the colour of that pixel

362|      MOV R2, #.grey    //sets r2 to background

363|      CMP R6, R2        //checks if fg matches bg

364|      BNE NopeRight     //sets it

365|      RET               //returns

366|DownCheckRight:

367|      MOV R8, #0        //resets line counter

368|      ADD R1, R1, #1    //increases y-coord

369|      SUB R0, R0, #4    //sets x-coord back to Rightt

370|      B RightCheck      //repeats function

371|NopeRight:

372|      POP {R3-R12, LR}

373|      RFE

374|acceptR:

375|      SUB R0, R0, #1

376|      RET

377|//

378|//end Move

379|//

380|//

381|//spin

382|//

383|Spin:

384|      ADD R10, R10, #1  //move onto next spin

385|      CMP R10, #1       //checks which spin its on

386|      BEQ changeTwo     //changes to the correct spin

387|      CMP R10,#2

388|      BEQ changeThree

389|      CMP R10, #3

390|      BEQ changeFour

391|      MOV R10, #0       //reset back to 0 when loop finished

392|      MOV R6, #SpinOne  //sets r6 to spin one

393|      RFE

394|changeTwo: MOV R6, #SpinTwo //sets the spin

395|      RFE               //return from exception

396|changeThree: MOV R6, #SpinThree

397|      RFE

398|changeFour: MOV R6, #SpinFour

399|      RFE

400|//

401|//end spin

402|//

Figure 13: This is a rather large section of code that shows how the shape moves left and right and most of how it spins. The reason it is so long is because it has so much to do. It must check if it’s at the edge of the screen and if the shape is going to be inside another if the shape is moved. This is done similarly to the checking system before where the shape was checked to see if it was landing on something.

The ‘invisible’ shape on this occurs to the left or the right of the shape and if the pixel is detected on top of another shape or out of bounds then the shape is not moved in that direction. For moving left, this can be seen between lines 290 and 321. The pixels are checked, and it will carry on and maintain the location of the x coordinate in place if the shape is fine.

A similar problem is presented with the rotation where it may rotate into another shape. This is fixed in a similar way.

380|//

381|//spin

382|//

383|Spin:

384|      ADD R10, R10, #1  //move onto next spin

385|      CMP R10, #1       //checks which spin its on

386|      BEQ changeTwo     //changes to the correct spin

387|      CMP R10,#2

388|      BEQ changeThree

389|      CMP R10, #3

390|      BEQ changeFour

391|      MOV R10, #0       //reset back to 0 when loop finished

392|      MOV R6, #SpinOne  //sets r6 to spin one

393|      RFE

394|changeTwo: MOV R6, #SpinTwo //sets the spin

395|      RFE               //return from exception

396|changeThree: MOV R6, #SpinThree

397|      RFE

398|changeFour: MOV R6, #SpinFour

399|      RFE

400|//

401|//end spin

402|//

Figure 14: This is the spin function and shows how, whenever the spin key is pushed, the spin index is increased and therefore, the circular queue of the spin functions is also increased.

Finally, we have the actual arrays that the shapes are drawn from:

405|//

406|//start arrays

407|//

408|arrayLength: 10         //stores the length of the arrays

409|arrayL:0                //start of arrays of the shapes

410|      1

411|      0

412|      0

413|      1

414|      0

415|      0

416|      1

417|      1

418|      .blue

419|arrayr:0

420|      1

421|      0

422|      0

423|      1

424|      0

425|      1

426|      1

427|      0

428|      .yellow

429|arrayT:0

430|      0

431|      0

432|      0

433|      1

434|      0

435|      1

436|      1

437|      1

438|      .purple

439|arrayS: 1

440|      0

441|      0

442|      1

443|      1

444|      0

445|      0

446|      1

447|      0

448|      .green

449|arrayZ: 0

450|      0

451|      1

452|      0

453|      1

454|      1

455|      0

456|      1

457|      0

458|      .red

459|arraySQ: 1

460|      1

461|      1

462|      1

463|      .yellow           //end of arrays of shapes

464|arrayI: 1

465|      1

466|      1

467|      1

468|      .cyan

469|SpinOne: 0              //start of rotation arrays

470|      4

471|      8

472|      12

473|      16

474|      20

475|      24

476|      28

477|      32

478|SpinTwo: 8

479|      20

480|      32

481|      4

482|      16

483|      28

484|      0

485|      12

486|      24

487|SpinThree: 32

488|      28

489|      24

490|      20

491|      16

492|      12

493|      8

494|      4

495|      0

496|SpinFour: 24

497|      12

498|      0

499|      28

500|      16

501|      4

502|      32

503|      20

504|      8                 //end of rotation arrays

469|SpinOne: 0              //start of rotation arrays

470|      4

471|      8

472|      12

473|      16

474|      20

475|      24

476|      28

477|      32

478|SpinTwo: 8

479|      20

480|      32

481|      4

482|      16

483|      28

484|      0

485|      12

486|      24

487|SpinThree: 32

488|      28

489|      24

490|      20

491|      16

492|      12

493|      8

494|      4

495|      0

496|SpinFour: 24

497|      12

498|      0

499|      28

500|      16

501|      4

502|      32

503|      20

504|      8                 //end of rotation arrays

Figure 15: This shows the arrays dictating the shapes that will be drawn and the order that their pixels should be drawn in. It can be seen that the colour of the shapes is in the array as well. This was so that the colour of each shape would be consistent with the shape that is drawn. Furthermore, the values in the spin arrays are all multiplied by 4. This was so that when the shape’s pixel was being found, I wouldn’t have to multiply by 4 and waste runtime in order to get the correct word size.

# Testing

This project is a qualitative or descriptive study rather than quantitative study and therefore lacks numerical data which could produce statistical analysis. Therefore, testing this project was achieved by practically running each section of the program.

ARMLite allows the user to select a line of program at which, you would like it to temporarily halt during runtime. This allows for ease with debugging and a better way of structuring the testing section.

Text, letter

Description automatically generatedFigure 16: I tested to see if the program could draw the initial screen. This would require a grey section in the middle and 2 black bars either side. To do this, I selected line 14 to be a halt line so that it would stop just after drawing the initial screen.

Figure 17: This produced exactly what I expected, with the screen and bars:

A picture containing application

Description automatically generatedThis is the play area described by the design and analysis section of the report, with one minor difference being the size of the play area being smaller due to the fact that the shapes need to actually be able to reach each side of the play area by the time it reaches the bottom.

Graphical user interface, application, Word

Description automatically generatedA picture containing waterfall chart

Description automatically generatedApplication

Description automatically generatedNext, I needed to test and see if it could draw one of the shapes at random. This was done by running the program 3 times and stopping it at line 30:

Figure 18: This demonstrates that the shapes are different in each image as they are chosen at random each time.

Graphical user interface, application

Description automatically generatedIcon

Description automatically generatedThe next test was to see if the user could control movement left, right and the rotation of the shape? To demonstrate this, I used the control keys to move each shape and then paused it to record the image.

Graphical user interface, application, Word

Description automatically generated

Figure 19:The shapes have responded appropriately to the given input keys and interrupts, furthermore, it can be seen that the shapes move down evenly at the given clock speed.

Next is to test if the shapes will land on each other and the ground. To demonstrate this I tested multiple additional shapes in different orientations and observed them move down to the stop position.

Chart, histogram

Description automatically generatedFigure 20: This is a screen shot showing the successful shape positioning.

# Evaluation

In the analysis section of the project report, I placed out 12 objectives that I believed to be necessary to render the project a success. These objectives were:

1. Multiple shapes
   1. Each with their own colour
   2. Randomly chosen
   3. Controllable
      1. Left and right
      2. Rotate
   4. The shapes must stop
      1. At the bottom
      2. On top of other shapes that are already there
2. Lines must be removed upon completion
   1. The lines above must be moved down after
3. The game must end when it reaches the top

As far as these conditions go, I believe them to be complete. However, I believe that there may have been better ways to solve each of them.

## Condition 1 multiple shapes:

Drawing multiple shapes was a success as it worked well, however the way it was designed meant that shapes that did not follow the normal shape of 3x3 then it would require a totally separate function for drawing it. This could be solved in a similar way to the way I solved the shape’s colour. I could store the dimensions of the shape in the array, these values can then be stored in a register when drawing and this can then loop around however many times in the x and y.

Initially, when creating this function, I tried only having one register for the position of the pixel pointer, when moving right I would add 4 to it and when moving down a line I’d add 256. This worked well enough but presented problems with how much thought would have to go in every time I was making this as it was not always easy to know what pixel it was pointing to, and it made debugging tough as I would just have a number in the thousands. Furthermore, I found that this made my functions limited to solo items.

I found my current solution by my assembly language student book, where on page 47 it talks about subroutines and shows that the x and y coordinates can be stored in separate registers. This allowed me to hold these values as global variables, however there were some occasions where I stored the data in these registers in a stack so that after the function was done, it could return to the location it was beforehand.

### 1.1 each shape must have their own colour:

The solution implemented worked out exactly as planned and did not run into any problems. I also couldn’t think of any way to improve this method.

Before I had the idea of storing the shapes as arrays, I had no clue on how I was going to give each shape their own colours. However, luckily after this I had come up with the method used in the program.

### 1.2 the shapes must be randomly chosen:

The random function in ARMLite was confusing as it didn’t give each integer an even chance of being chosen at random. I found this originally confusing as it didn’t make a whole lot of sense and it kept only showing the blue ‘L’ shape and the occasional other shape but vary rarely.

Eventually I figured to just make it so that each shape would have a range of integers to be represented by. This fixed the problem, but I still have a feeling that there was a bias with the selection as it still seems to give the blue shape more times than the others. Now, I believe that it may be better to limit it with a hexadecimal input rather than a denary input as, according to the assembly language student book, it is what they use for their snake game.

### 1.3 controllable left/right and rotation:

Moving left and right in the game works well and gives the user control over the shape, furthermore, it cannot exit the limits of the play area or cross through another shape. I doubt I would change much if I were to do this again.

The rotation works well; however, I think it turns in the opposite direction to how a normal game of Tetris turns. Furthermore, it may turn and clip into another shape. I would seek to fix this the same way I have delt with this sort of checks before.

Initially, the game wouldn’t work in full game speed after turning on the interrupts and this made me worried as to what I had to do. Eventually, I figured out the flaw in my code as it was that instead of removing the shape once moving it down and then repeatedly drawing it, I had it the wrong way round. I had the shape being drawn once and then removed over and over. With a bit of moving around, the code was working, and the shape could move down.

### 1.4 the shapes must stop:

I am happy with the way the shapes stop at the bottom of the screen and on top of each other. I believe that it very well organised and it stops in the correct position every time.

Initially, the shape wouldn’t stop at any point it needed to. This was problematic as it grossly opposed the whole point of the game. My initial reaction to this was anger, but upon calming down I quickly assumed the position of, “I’ll figure it out later.” And moved on to other parts of the program. Eventually, I came up with a better way of drawing the shapes than before and it gave a separate register for the y coordinate. After writing the new function and testing it on a separate file, I had a working function for checking if the shape was over a shape or at the bottom. I was happy with this as it was proving very annoying for ages, only stopping at the bottom.

## Condition 2 lines must be removed upon completion:

In the game currently, the line removal function doesn’t work. I have not yet been able to identify why as I have meticulously run through the code to see why it may not work by checking the lines and removing them. I suspect that his was a failing at the point where the program checks whether or the line was full or not. I imagine that it could be that at the point that the line is checked maybe the shape hasn’t been drawn yet and so there is a blank bit of the line still there.

Currently, it works like this: when a shape has been “set”, each line of the shape that was just placed is checked. If, at any point, a pixel is grey, the line is not considered full, and the function will start again on the next line of the shape that had just been placed. I chose to only check these lines as they would be the only lines that could have just been filled. If, however it went across the whole line without finding any grey, it would loop over every pixel in that line and turn it white (I chose white in order to give a sort of flash effect). This doesn’t seem to work, however.

### 2.1 lines above must be moved down:

As for the moving lines down, I would do this by copying each line above the gap down and then filling in the blank lines at the top.

Potentially, I would loop over every pixel on the top line and copy the pixel from however many lines above as there were removed, I would need to do this for each of the lines that were removed. Then, I would remove those lines and I would have to do the same for them.

## Condition 3 the game must end when the shapes have stacked to the top of the play area:

The game stops playing when the shape reaches the top as it hits an error and pauses the program. This is good enough to achieve my objective, however, if I were to do this again, I wouldn’t do it the way I did it this time as it is not logically sound and does not aesthetically feel like a game has been completed.

Currently, the game stops because it cannot draw another pixel outside of the shape area and so it displays an error and halts the code. This is good as it does stop the game, but it means that if I wanted to add a “game over” screen, I couldn’t.

Upon doing this project again, I would check the position of the shape that has just been placed, if it is above a certain area, it will be game over.

This project has been fun to do, and I feel as if I know a lot more about the architecture of a processor and a computer. I also feel as if I know more about how to write assembly code, and I know that I would write the program very differently structurally if I were to do this again; for example, I would try to write in more of a functional method, trying to take advantage of the changing program counter. This would be a lot closer to how I am used to programming and the rigid structure of functional programming would give less room for errors. Furthermore, I believe I would take more advantage of using stacks as I believe in the program I made, I wasn’t using stacks enough. Finally, I would use more arrays to create and store object. This will allow for better structure and better use of memory and registers.

For better improvement in the future, it may be a good idea upon doing this again, to avoid using an emulator and instead actually program on an ARM processor. This would give me a wider range of commands to use along with its own share of added problems, for example initialising a range to describe where the screen will be saved.

# References

For learning assembly: Richard Pawson and Peter Higginson. Computer Science From The Metal Up.

Tetris wiki <https://en.wikipedia.org/wiki/Tetris> for research section

Used <https://app.diagrams.net/#G1EZhsoFvZ2tXer_lph8vjroVlCr_j3u7s> to produce flow diagrams

<https://www.researchgate.net/figure/The-seven-Tetris-shapes-used-in-the-Tetris-game-Each-block-can-be-rotated-an-fitted_fig26_241802318> was for the image of the different Tetris shapes