VRIJE
UNIVERSITEIT
BRUSSEL

Project Computersystems

Checkers

Group G16

Elliott Octave & Rayane Kouidane

September 26, 2022

Academic year 2022-2023

Bachelor Computerscience

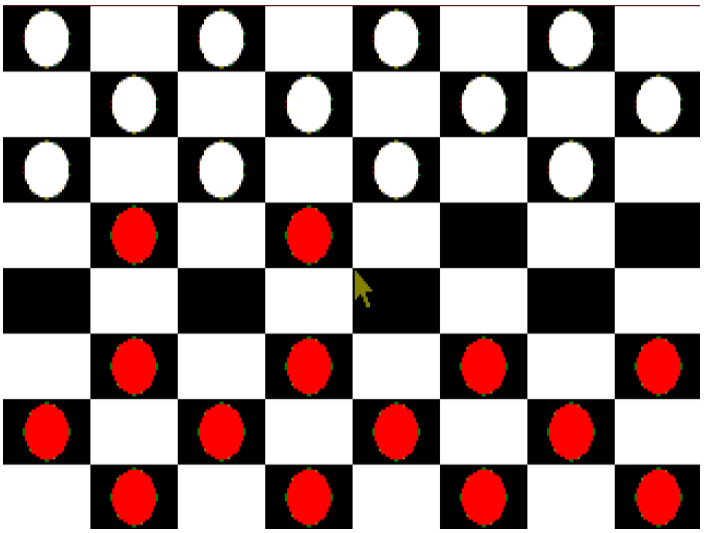

Figure 1: Example screenshot of the game.

1 Introduction

For the Computer Systems course project, we had to work in groups of 2 to create a working interactive and graphical program. This had to be created in 80386 32-bit protected mode assembler language, compiled by the Borland Turbo Assembler (TASM), running in the DOSBox environment. We have chosen to create the game of Checkers, also known as Draughts. At the end, we should be able to see a board with the checkers on it. We should also be able to move the checkers and capture other checkers. At the end of the game there is also a winner (can also finish on draw).

2 Manual

2.1 Starting

To run this game users, must run the ***asm*** file through **DOSBox**.
Once the game is started a title screen (shown below) will appear, like instructed on the title screen users will have to press the ENTER button on their keyboard to start the game.

Figure 2: Title screen of the game.

2.2 Game

Once the game is started the checkers board will be displayed with all the pieces on their starting squares. Red will have the first move. To make a move a player must click once on the piece he/she wants to move and then click a second time on the position he/she wants that piece to move to. If the move is legal it will be played and the turn will change to the opponent.

3 Program features

Our program is an implementation of the English draughts variation of checkers.
This means that normal pieces can only move diagonally forward to adjacent squares. Kings (normal pieces that make it to the other end of the board) can move in all four directions on the same diagonal as their starting square.

If a piece makes a capture and can make a second one in the same turn it has to make that second capture. However in our implementation this doesn't work consistently for king pieces. It will work for the normal pieces, but only in the direction of the player

If a piece can make a capturing move the player is obligated to make that move.
However in our implementation there can only be one obligated move each turn, this not what we wanted but there seems to be a problem that we weren't able to solve in our code. The only input that is needed is the mouse (to move the pieces) and the ESC and ENTER button (to quit and start the game).

4 Program design

The program defines several constants, such as the video memory address, screen width and height, and the size of the board. It also includes a procedure called CheckMove, which takes three arguments: a pointer to an array representing the board, and two positions on the board. The procedure checks if the move from the current position to the next position is valid or not, and returns a value of TRUE in the ax register if it is valid, or FALSE if it is not valid. The procedure checks the move based on the color of the checker at the current position and whether it is a regular checker or a king. It also handles the special case of a king moving diagonally across the board.

The MakeMove procedure is responsible for making a move on a checkers board, represented as an array in memory. It takes three arguments: a pointer to the array, the current position of the piece being moved, and the position it is being moved to. The procedure performs the move by swapping the values at the current and next positions in the array. It also handles capturing pieces by replacing the value of any piece that is "jumped" during a move with the value 'x'. If the move results in a piece reaching the opposite end of the board, that piece is promoted to a king by changing its value to 'WK' or 'BK' depending on its color. The procedure returns after completing the move.

The CheckEatAll procedure is responsible for checking if there are any available moves that involve capturing pieces in the current player's turn on a checkers board. It takes a single argument: a pointer to the board array. It performs this check by looping through each position on the board and calling the 'CheckAnotherEat' procedure for each position. If 'CheckAnotherEat' returns a value other than -1, indicating that a capturing move is possible from the current position, the position is stored in either 'whitepos1' or 'whitepos2'. For a particular reason that we don't understand, there is never a value stored in 'whitepos2'. Normally when we find a pieces that must capture we store it in 'whitepos1' and we continue the loop. If we come across another piece that must capture we save it in 'whitepos2', but it isn't working. If both of these variables are already filled, the loop exits early. Once the loop has completed, the procedure returns.

The rest of the procedures are responsible for all the drawing/input logic of the program.

5 Encountered problems

The main difficulties were:

How to start, the assembly x86 language was new to us both so starting a project was very difficult since we had no idea where to start. Our first procedure was called move this was supposed to make the pieces able to move. First, we had to understand the registers, etc…. This procedure is completely different to how we make our moves now.

The second big issue was displaying our board and the pieces. This took some time, mainly researching. We didn't understand how we could convert an image to use it in our assembly project. This took us a lot of time.

The third main difficulty was allowing users to use the mouse as input. Again, this took quite some time to implement. We tried using the code of the WPO's but it didn't work out. So we looked on the internet and found the information we needed.

We also had organization difficulties. We needed to work in group, so to find a way to work simultaneously on the project. Besides that we had also a lot of other projects to finalize. So we needed to separate our time to work on this project and the others. For the limitations that were mentioned in '3.Program features' we have thought about them, but we didn't have enough time to implement these problems. We also wanted to separate our code in different files, but we lost a lot of time trying to find how to do this. But in the end we couldn't separate it.

For the rest our time was used improving code and adding new code.

6 Conclusion

We think that our program was pretty successful. We tried our best to implement all the rules of checkers. We are proud of this work, since none of us had any experience with assemble x86. If we had more time or less projects we could have made a better game.