VRIJE
UNIVERSITEIT
BRUSSEL

Project Computersystemen

CHESS

Group G24

Stuker Abel & Ruiz Callejo Daniel

December 26, 2023

Academic year 2022-2023

Computerwetenschappen

Contents

1 Project description **1**
1.1 Playing the game . . . 1

2 Program design **2**
2.1 Mouse . . . 2
2.2 Clocks . . . 2
2.3 Pieces and Cell selection . . . 2
2.3.1 Piece data . . . 2
2.3.2 Movement & capturing . . . 2
2.4 Bitmaps . . . 3
2.4.1 En Passant . . . 3
2.5 Bitmaps . . . 3
2.5.1 Castling . . . 3
2.6 Button selection . . . 3
2.6.1 Pawn Promotion . . . 4
2.7 Scenarios and game state . . . 4

3 Encountered problems **4**
3.1 Bitmaps . . . 4
3.1.1 Bitmap generator . . . 4
3.1.2 Font generator . . . 4
3.1.3 Endianness . . . 5
3.2 Writing to files . . . 5
3.3 Drawing the mouse . . . 5
3.4 Storing pieces in memory . . . 5

4 Reflection **5**

1 Project description

This report describes the project for the course Computersystem at the VUB. For this project, we made an implementation of Chess in the 80386 32-bot protected mode assembly language. The code is compiled by TASM and runs in DOSBox.

The chess game is implemented according to the FIDE-rules. This implies that every piece has certain movement and capturing restrictions. Illegal moves are restricted by the program. The most important features are player movement & capturing (incl. en passant, castling and pawn promotion), and the implementation of player clocks. We provide a more detailed description on all the different features in section 2.

1.1 Playing the game

Start the game by executing the `wget` command, and then execute the executable by typing `chess`. You can play the game like any usual chess game. A difference however is that you have to kill the king of your opponent to win. This is because no checkmate detection system has been implemented (yet). To test the pawn promotion, en passant, castling and last move features, you can select a scenario at the right side of the screen. If you wish to restart the game, you can do so by tapping the restart button, and if you want more time, use the "+5 min button". Although the game is automatically saved after every move, there is still a dedicated button if you want to manually save it again. This saved state automatically loads once you restart the game. The use of the rest of the interface is straightforward. Press a piece to select it and press another cell to move the piece. A winner is decided when a king has been killed or when a player is out of time.
Notice that restarting the game again in the same DOSBox session might not work. Simply restarting DOSBox resolves the issue.

2 Program design

2.1 Mouse

Most of the program features rely on input from the user. To interact with the chess board and with the buttons, mouse functionality is provided. To implement this, we used the code provided by Tim Bruylants in `EXAMPLES/MOUSE.ASM`[1]. When the mouse moves or is clicked, the installed `mouseHandler` procedure will be called, where the last mouse positions will be updated (global variables). When the left mouse button is pressed, it will first try to find a button to select (calling `selectButton`); if no such button has been found, it will try to select a cell (calling `selectCell`). The latter procedure handles all the possible cases when selecting a cell (moving, changing selection, capturing etc.).

Finally, all the elements will be redrawn (the buttons and the board with pieces). The only exception is when the Pawn Promotion menu is visible; in that case the board doesn't need to be redrawn since no cells can be selected until a Pawn Promotion selection is made.

2.2 Clocks

The only elements that can not be updated solely when the mouse handler is called, are the clocks. Therefore, the end of the `main` procedure will contain an infinite loop in which `updateInformation` called. This procedure will each time redraw the color of the player and the corresponding clock. Both players each have three global variables: one variable containing the string ("##:##") of the time, one variable containing the time left (in seconds) when the player's turn started (we define this as P_{turn}), and one variable containing the time left at this moment (we define this as P_{now}). A single variable `startTime` (we define this as T_{turn}) is always updated to the current time (in seconds) when the current turn is changed. To update the clocks to show the time left for each player, the `updateTimers` procedure is called. This procedure works as follows. We retrieve the current time (in seconds); let this be T_{now}. P_{now} of the current player is updated with $T_{now} - T_{turn} - P_{turn}$, the time the player had left at the beginning of his turn minus the time elapsed since then. This value is then converted into text format to be displayed. When the time of a player is up, the other player has won the game.

2.3 Pieces and Cell selection

When the mouse handler detects a click, the `selectCell` procedure may be called (cf. section 2.1). This procedure checks whether the selection indicates the player wanting to move a currently selected piece to the newly selected cell, or whether he wants to select a piece to move later.

2.3.1 Piece data

All pieces are stored in memory, referred to in the data section as `pieces`. Each piece is represented as a double word, but only has 12 bytes of useful information:

- **Bit 1: state** (0 = captured, 1 = on field)
- **Bit 2: player** (0 = white, 1 = black)
- **Bit 3 - 6: piece type**
- **Bit 7 - 9: x-coordinate**
- **Bit 10 - 12: y-coordinate**

These pieces are modified when a piece moves or is killed (cf. section 2.3.2). The procedure `drawPieces` iterates over these pieces to draw them (if not captured) on their encoded position. Note that we made the deliberate choice to not store the pieces in an array where every element represents a cell on the board; we elaborate on that choice in section 3.4.

2.3.2 Movement & capturing

When the player wants to move to another cell, the occupation of that destination cell is determined. The possibility of Pawn Promotion and En Passant will also be checked here, after which the `movePiece` procedure can finally be executed. First, the procedure `calculatePossibleMove` determines whether the selected piece

[1]

can move to the destination cell. For every piece type, a dedicated procedure exists. If the movement is possible, this will be executed and, if necessary, a piece will be captured by `killPiece` (either the piece at the destination cell, or the piece passed by en passant).

2.4 Bitmaps

2.4.1 En Passant

A pawn can be captured 'En Passant' in the following scenario. Assume a pawn P_A from player A is still at its original position and moves 2 ranks forward. If a pawn P_B from player B resides at an adjacent cell on the same rank, player B can capture P_A in its next turn by moving P_B to the cell that P_A skipped when moving forward. Figure 1 shows the white piece being vulnerable for en passant capture. If the black piece moves to the cell between the two green cells, the white pawn gets captured.

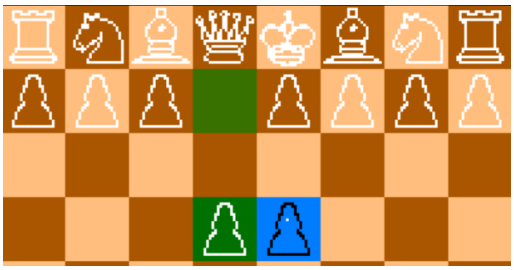

Figure 1: En passant scenario

The program stores an `enPassantState` in memory, which contains the following information:

- **Bit 1-3: x-coordinate** of the en passant cell (jumped over by the player)
- **Bit 4 - 6: y-coordinate** of the en passant cell (jumped over by the player)
- **Bit 7: direction** of the piece that can be captured en passant (0 = above, 1 = below)
- **Bit 8: state** (0 = en passant not possible, 1 = en passant possible)

A piece becomes vulnerable to an en passant capture after moving its initial 2 positions forward, which is when the `enPassentState` is set. As described before, the `selectCell` procedure will check for and execute en passant.

2.5 Bitmaps

2.5.1 Castling

When the king and a rook of a player haven't moved yet, and when the path between them is empty, a move called "castling" can be performed. In this situation, the king moves 2 positions in the direction of the rook, after which the rook jumps over the king to the single cell skipped by the king. Castling is implemented in the `checkKingMove` called from `calculatePossibleMove`.

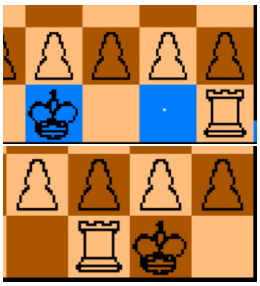

Figure 2: Castling scenario (before & after)

2.6 Button selection

When the mouse handler detects a click, the `selectButton` procedure is called first. Multiple buttons can be specified in the data section, where every button consists of 6 double words:

- **DWORD 1: x-coordinate** of the top left corner of the button
- **DWORD 2: y-coordinate** of the top left corner of the button
- **DWORD 3: width**
- **DWORD 4: height**
- **DWORD 5: address of the string** displayed on the button
- **DWORD 6: address of the procedure** to be called when clicked on the button

Note that the four first double words are set to 0 initially, since their values are still unknown. These values are calculated and set by `makeButton`, called by `initializeButtons` (at the beginning for the 'static' buttons) and by `requestPawnPromotionSelection` (do dynamically position the pawn promotion option buttons).

When trying to find the clicked button, the `selectButton` procedure needs to loop over all arrays in memory that store button addresses (i.e., `ppButtons` for piece promotion buttons, `optionsMenuButtons` and `scenarioButtons`). When the mouse has been clicked within the boundaries of the button, the procedure

that is provided at DWORD 6 in the button array must be executed. Besides that, 1 must be returned in eax to indicate that a button has been selected (0 otherwise).

2.6.1 Pawn Promotion

A pawn must be promoted to a queen, bishop, knight or rook once it reaches the opposite end of the field. To make a selection, a list of options must be displayed. This is initiated in the `requestPawnPromotion` procedure (called from `selectCell`, where the buttons are initialized (`makeButton`) on the correct position and the global flag `ppMenuVisible` is set. This ensures that the pawn promotion menu can be properly drawn while interaction with the board becomes impossible. The `selectButton` procedure may detect when a pawn promotion option is selected, after which the `ppMenuVisible` flag is set to 0 and the `executePawnPromotion` procedure is called. The pawn will be promoted to the selected type based on the selection.

Figure 3: Pawn promotion scenario

2.7 Scenarios and game state

After every move, the state of the game is saved to `scen/saved.bin`. When the game is started again (after closing), it will automatically open from this file. More concretely, the pieces array is exactly copied from and to this file. The `scen/original.bin` file contains the default initial board setup, which allows for easy restart of the game). Besides that, several scenarios are stored that can be loaded into the game.

3 Encountered problems

During the development of the program, we encountered a lot of problems we needed to fix. Some of these will be described in this section.

3.1 Bitmaps

We started with the project very early on. The first thing we wanted to achieve is to display bitmaps on the screen, in order to be able to display chess pieces, letters and numbers. Since we didn't have the provided MATLAB tool yet at that time, we already came up with our own solution to generate and read bitmaps. Our goal was not to incorporate colors into the bitmap, but to make them as generic as possible. Therefore, we chose to use 1-bit bitmaps where 0 would represent transparent or background color, and 1 the foreground. That enabled us to draw letters and numbers in all possible colours, with all possible backgrounds. Also the chess pieces of the same time could be drawn from the same bitmap for both teams.

```
00000000000000000000000000
00000000000000000000000000
00001111001111001111000000
00001001111001111001000000
00001000000000000001000000
00001111111111111111100000
00000110000000000110000000
00000010000000000100000000
00000010000000000100000000
00000010000000000100000000
00000010000000000100000000
00000010000000000100000000
00000010000000000100000000
00000010000000000100000000
00000010000000001100000000
00000010000000000100000000
00000111111111111100000000
00000100000000000100000000
00001100000000000100000000
00011111111111111111000000
00010000000000000011000000
00011111111111111110000000
00000000000000000000000000
00000000000000000000000000
```

Figure 4: Rook drawn in bits

3.1.1 Bitmap generator

The most difficult part was to encode the bits of the sprite into a bitmap. First, we would draw the sprite as zeroes and ones in a text editor, as can be seen in Figure 4. We would then remove all the newlines to move all bits to a single line. To convert the text bits into an actual bitmap, we wrote a C script `stringtobitmap.c` that can be found in `auxcode/bitmap_generator` and with executable named `stb`. This not only allowed us to make piece bitmaps, but also to create bitmaps for letters and numbers.

3.1.2 Font generator

Since we wanted to display text on the screen while in mode `13h`, we encoded all the capital letters, all the numbers and some symbols into bitmaps. Since it would be way too inefficient to use a different bitmap and array for every character, we came up with another solution. We renamed the bitmap of every character to its corresponding ASCII decimal value (+ ".bin"). We then wrote another C script `generate_ascii_bitmap.c` in `auxcode/font_generator` with executable `gen_font`. This script asks for the name of the folder where all the ASCII-character bitmaps are stored. Let S be the size (in bytes) of a single-character bitmap, and let

C be its corresponding decimal ASCII-value. The bitmap is then encoded into the large bitmap starting at position $S * C$. This allows the character bitmap to be easily retrieved by the program, since it offset can be calculated directly from the ASCII-value (C) and the font-size (S).

3.1.3 Endianness

In order for a bitmap to be drawn onto the screen, double words are read from memory. Due to the little endian memory ordering, writing the `drawBitmap` procedure was not straightforward, but after a lot of drafting, we finally managed to implement the correct procedure.

3.2 Writing to files

In order to save the game state, we write the `pieces` array to the file `scen/saved.bin`. We had a lot of trouble with writing the file, but finally discovered that we needed to provide the value 1 in `al` when retrieving the file handle to obtain write (write-only) permission.

3.3 Drawing the mouse

While retrieving the last position of the mouse was easy (since we used the code from Tim Bruylants[2]), drawing that mouse onto the screen was more challenging. We noticed that, obviously, the mouse would not 'undraw' itself from its last position after it moves to a new position. Simply recoloring the previous coordinate too black doesn't suffice either. Before drawing the mouse, we must first temporarily store the underlying color. We can then write the mouse in the `framebuffer` and transfer the framebuffer so that all elements are drawn onto the screen, including the mouse. Immediately after that, we restore the saved color back into the framebuffer, so that the mouse won't interfere with other visual elements.

3.4 Storing pieces in memory

At first, we thought that it was a good idea to create an array where every element corresponds to a cell on the chess board. When an element is 0, that would mean the cell is empty. In the other case the element would be an encoded piece. The reason why we did this is because this allowed for easy access when reading the state of a cell at a certain position (performance: $O(1)$). However, after some considerations, we did change this implementation. Instead of storing every and moving the pieces to their corresponding cell in the array, we would from then on only store an array of all the pieces, where the piece itself now holds it position. Although direct cell state access would now be less efficient, with $O(n_p)$ performance (n_p = amount of pieces), we noticed that the efficiency while drawing the board and pieces actually improved. With the initial implementation, drawing the board and pieces required reading every cell state from memory, resulting in $\theta(n_c)$ performance (n_c = amount of cells). The new implementation only reads the pieces from memory, yielding $\theta(n_p)$ performance. Clearly, $n_p < n_c$ shows that the new implementation actually improves the drawing functionality, since it doesn't need to read the $n_c - n_p$ empty cell states from memory. The drawing functionality is definitely more frequently used than the direct cell state access, from which we conclude that only storing the pieces is more efficient compared to storing every cell state.

4 Reflection

Although it was a very tough project to begin with, we started liking it more throughout the entire journey of developing this program. Because there wasn't a lot of information to be found online, we needed to wait on the classes to get the theory needed to continue working on the project, but most of the time we tried to figure it out by ourselves by trial and error. The steep learning curve of the language, and the amount of time spent on the project with its accompanying problems was definitely worth it at the end.

[2]