Description of Software

This program simulates a game a chess between two users. The program starts off with an intro screen, using clickable buttons to give the user the option to pick between “PLAY”, which starts the game, “RULES”, which goes to the rules screen, and “QUIT”, which exits the program.

The rules screen includes basic instructions for how to play and information on troubleshooting. It also includes an animation for how to move a piece, and has a back button to take the user back to the intro screen

The game begins once the user presses “PLAY” on the intro screen. The first turn is white’s, and the first user can move a piece by clicking on it with the mouse to first see the available (highlighted in blue) moves, and then clicking on the highlighted square to move the piece to that location. Then, it’s black’s turn, and the second user performs the same process. When there is a check (an opposing piece can take out the king), the screen becomes red and displays a “Check! Save your King!” message to alert the user. The game ends when one of the users takes out the opponents’ king or someone forfeits, and the game returns to the intro screen.

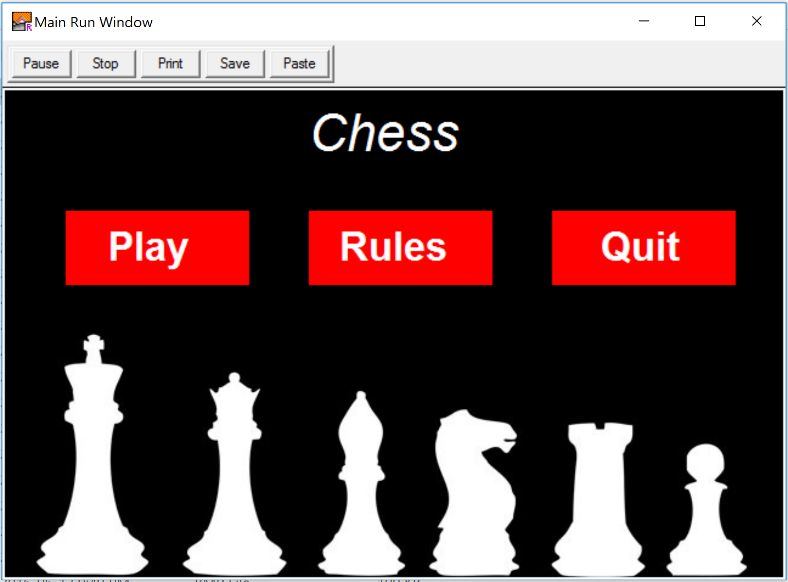
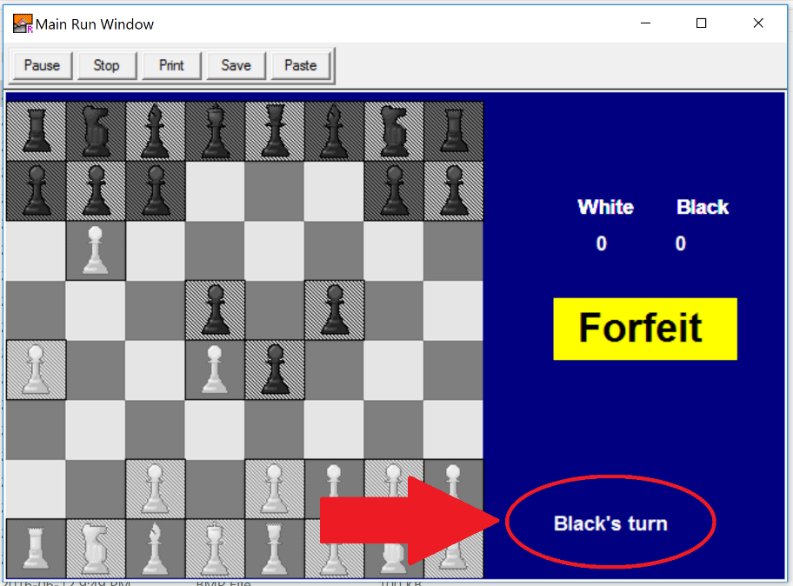
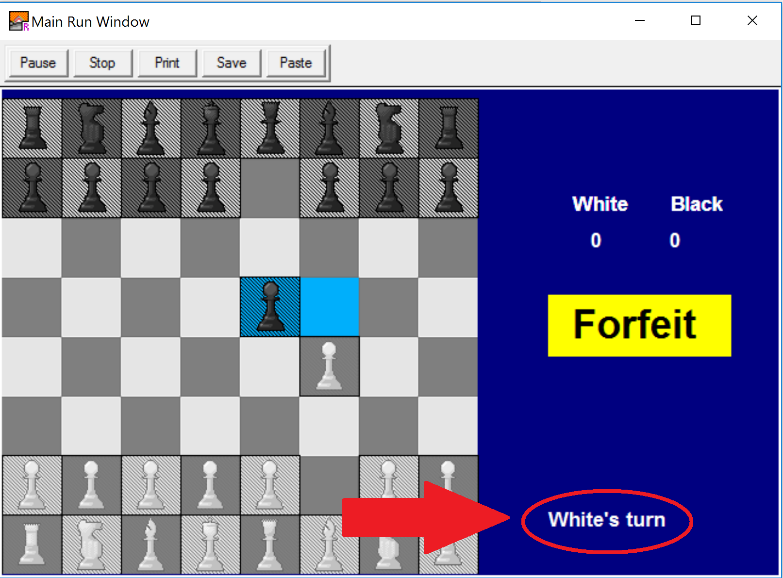
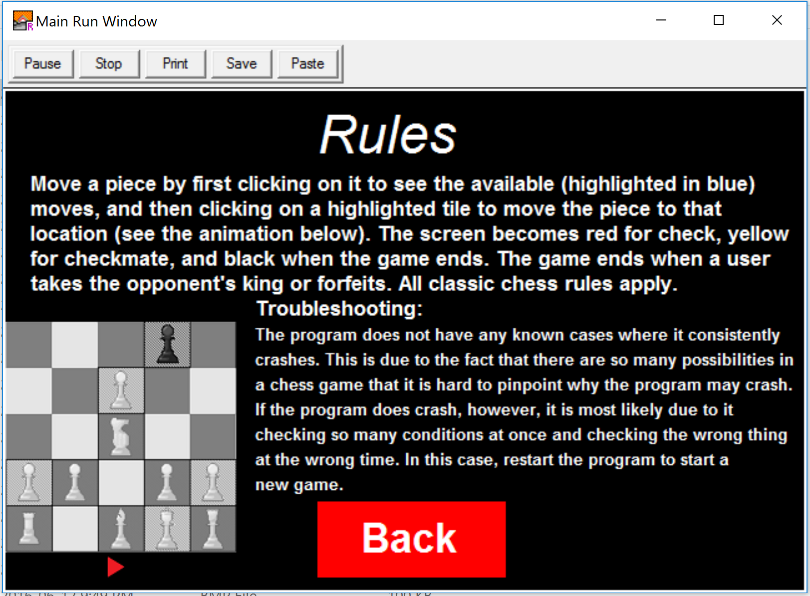
The program is written in Turing version 4.1.1, and potential users include anyone who wants to play chess with another individual.

Program Assumptions and Restrictions

Some assumptions and restrictions include:

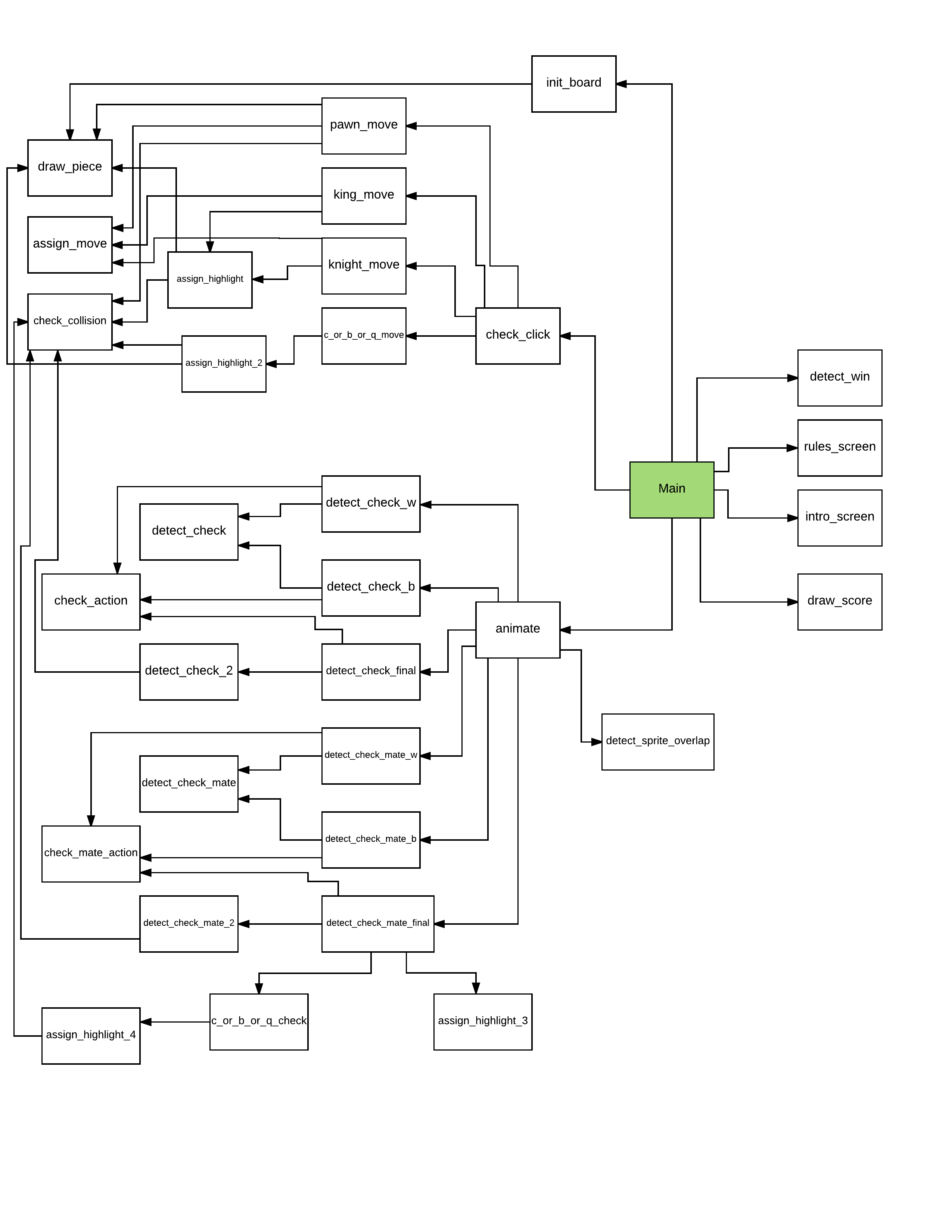
* The user must have some general background knowledge for how to play chess
* The computer must have a disk drive to input the disk and run the file
* The user must have a mouse connected to the computer to use the program

User Interface

Various prompting is used by the program to interact with the user. The program contains various menus, each with buttons to guide the users’ decision. During the game, the program prompts the user by displaying a “White’s turn” or “Black’s turn” message in the lower right hand corner.

*Figures 1 – 4: Examples of menus and prompts in the program*

Diagram of Procedures and Functions Used

Global Variables

|  |  |  |  |
| --- | --- | --- | --- |
| **Variable name** | **Type** | **Initialized?** | **Purpose** |
| SPACE | Constant | 49 | The pixel size (length and width) of one square on the chess board – used to simplify move algorithms |
| sprite | User defined type | No | Contains values of all sprites used. Contains (x,y) location (lower left hand corner), a sprite value to declare the sprite, a name for identifying the sprite, and max vertical, horizontal, and diagonal displacement values. |
| pointers | Array of unchecked pointers to sprite | The address of the variables for all 32 pieces | Makes it so that I can run through a for loop if I need to check a condition for all pieces instead of checking each piece individually |
| move | int | 0 | Variable to tell if the user has moved a piece or not (1 for yes, 0 for no) |
| turn | int | 0 | Represents the current turn (0 for white, 1 for black) |
| w\_score | int | 0 | Keeps track of white’s score |
| b\_score | int | 0 | Keeps track of black’s score |
| pics | Array of int | All pictures used | Stores all external bmp images used in one array |
| x, y, button | Int | No | X and Y are coordinates of the mouse (using Mouse.Where) and button is 1 if the mouse is clicked – used for button click detection |
| font1, font2, font3, font4 | Int | Corresponding Fonts | Different fonts |
| w\_pawns, b\_pawns | Array 0 .. 7 of sprite | Initial pawn locations on the board, image of pawn, name identifier | White and black pawn sprites used in gameplay |
| w\_bishops, b\_bishops | Array 0 .. 1 of sprite | Initial bishop locations on the board, max displacement values, image of bishop, name identifier | White and black bishop sprites used in gameplay |
| w\_knights, b\_knights | Array 0 .. 1 of sprite | Initial knight locations on the board, image of knight, name identifier | White and black knight sprites used in gameplay |
| w\_castles, b\_castles | Array 0 .. 1 of sprite | Initial castle locations on the board, max displacement values, image of castle, name identifier | White and black castle sprites used in gameplay |
| w\_queen, b\_queen | sprite | Initial queen locations on the board, max displacement values, image of queen, name identifier | White and black queen sprites used in gameplay |
| w\_king, b\_king | sprite | Initial king locations on the board, image of king, name identifier | White and black king sprites used in gameplay |
| highlight | Array 0 .. 200 of sprite | Blue highlight image, “null” location value of (10000, 10000), name identifier | Blue highlight used to show user possible moves and used in checkmate detection to store values of tiles with pressure on them from opposing player |
| background | sprite | Background chess board image, location, name identifier | Displays chess board |
| check, w\_forfeit, b\_forfeit | Boolean | False | Indicate if there is a check or if someone has clicked forfeit. |

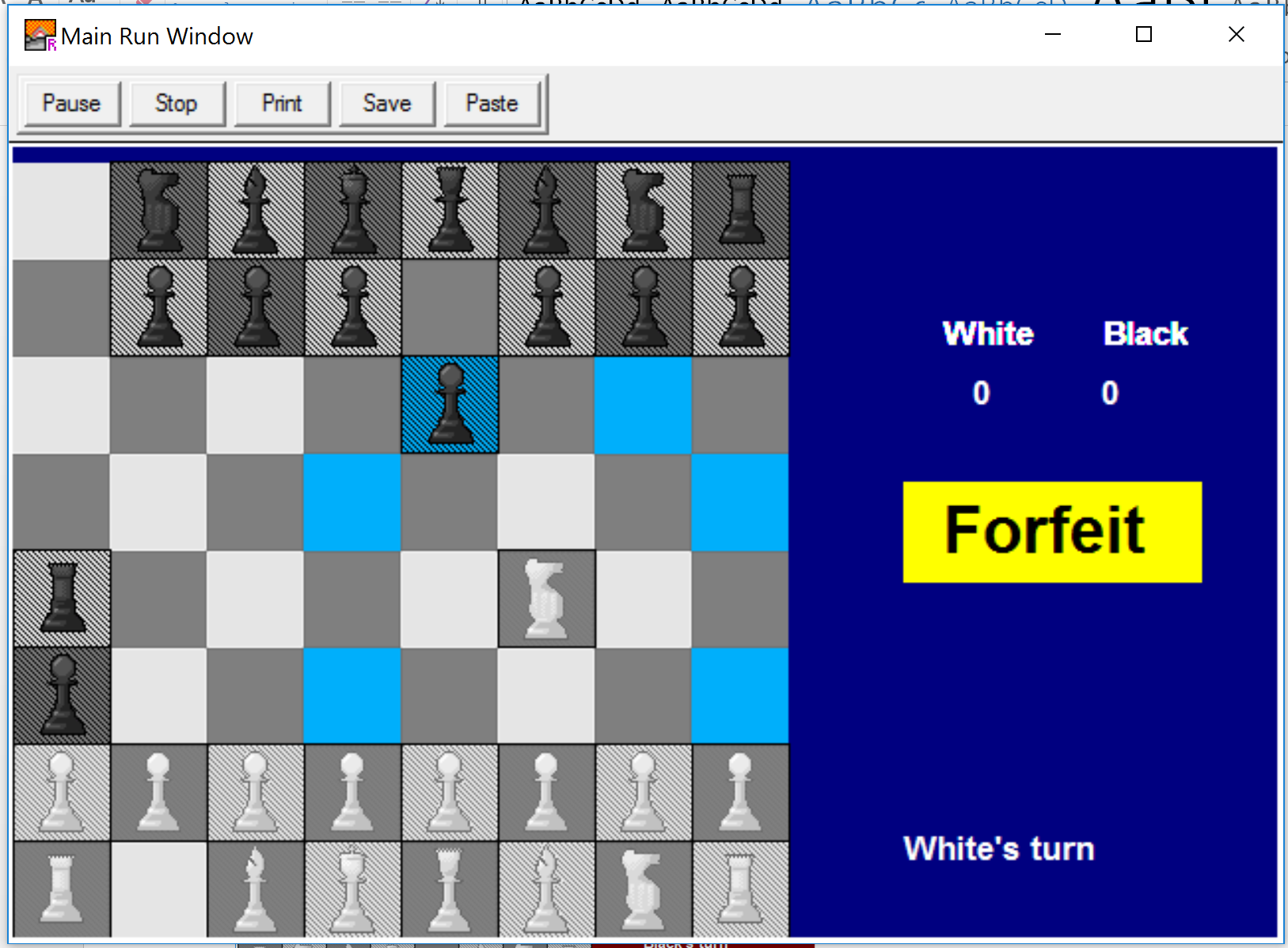
Functions and Procedures

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Name | Fcn/Proc | Parameters | Returns value? | Purpose |
| draw\_piece | P | sprite | No | Draws a sprite on the board |
| intro\_screen | P | No | No | Introduces the user with 3 buttons for play, rules, and quit |
| rules\_screen | P | No | No | Rules screen to explain how to play including an animation |
| init\_board | P | No | No | Initializes all appropriate values including sprite locations and pointer values |
| draw\_score | P | No | No | Displays the current white and black score |
| detect\_sprite\_overlap | P | Sprite and pointer to sprite | No | Checks If the sprite overlaps with the pointer (if two sprites overlap). Updates score and hides overlapped sprite. |
| check\_collison | F | Sprite and 2 ints (vert and horiz displacement) | Int – 0 for white, 1 for black, 2 for vacant | Checks if a position on the board is occupied |
| assgin\_move | P | Sprite | No | If user clicks on a highlighted area, set the move variable and hide highlights. |
| assign\_highlight | P | Sprite and 3 ints – index for highlight, vert and horiz displacement | No | draw highlight if tile is on board and piece can be taken or tile is vacant |
| assign\_highlight\_2 | F | Sprite and 3 ints – index for highlight, vert and horiz displacement | Int - 0 if occupied tile, 1 if vacant, 2 for continue | Used for to ensure pieces cannot jump over other pieces - 0 if occupied tile, 1 if vacant, 2 for continue |
| assign\_highlight\_3 | P | Sprite and 3 ints – index for highlight, vert and horiz displacement | No | Used for checkmate detection with pawns, knights, and king (non-repeated moves) – if an opposing piece can move to a specific tile on the next turn, then store the value of that tile in the highlight array for reference. |
| assign\_highlight\_4 | F | Sprite and 3 ints – index for highlight, vert and horiz displacement | Int - 0 if occupied tile, 1 if vacant, 2 for continue | Used for checkmate detection with castle, bishops, and queen (repeated moves) – if an opposing piece can move to a specific tile on the next turn, then store the value of that tile in the highlight array for reference. |
| c\_or\_b\_or\_q\_move | P | sprite | No | Assigns moves for castle bishop and queen using their predefined maximum displacement values. |
| pawn\_move | P | Sprite, int – 1 for white, -1 for black | No | Used for pawn move - if tile is vacant, draw highlight, and if turn is 1 then draw two highlights |
| knight\_move | P | sprite | No | assign highlights for 8 possible knight moves |
| king\_move | P | Sprite | No | assign highlights for 8 possible king moves |
| check\_click | P | Sprite | No | Check if the user clicked a piece, and pass the sprite to the appropriate procedure. If user clicks forfeit, assign forfeit variable to true |
| detect\_check | F | Sprite, 2 ints – vert and horiz displacement | Int – 0 for white king, 1 for black king, 2 for no check | return 0 if tile has white king, 1 for black king, 2 for no check |
| detect\_check\_2 | F | Sprite, 2 ints – vert and horiz displacement | Int - 3 if a piece is blocking movement and the piece is not a king, 0 for black check, 1 for white check | Return 3 if a piece is blocking movement and the piece is not a king, 0 for black check, 1 for white check |
| detect\_check\_w | P | No | No | detect check for white pawns, king, and knights |
| detect\_check\_b | P | No | No | detect check for black pawns, king, and knights |
| detect\_check\_final | F | Sprite, int – 1 for white, 0 for black | Int – 1 or 0 for white/black check, 3 for piece in the way of the king, 2 for continue checking | If there is a check, then call check action |
| check\_action | P | No | No | Alerts user of check |
| animate | P | Unchecked pointer to sprite | No | hide highlights, animate sprite, detect check and sprite overlap |
| detect\_win | F | No | Int – 0 for black win, 1 for white win, 2 for continue | If either king is taken or someone has forfeited, display a win message, hide everything, and return to the intro screen |
| check\_mate\_action | P | No | No | Alert users of checkmate |
| c\_or\_b\_or\_q\_check | P | Sprite, int for highlight index | No | Checks if the opposing castles, bishops, or queen can move to a certain tile on the next round using max displacement values |
| detect\_check\_mate | F | Sprite, 2 ints – vert and horiz displacement | Int – 0 if tile has white king, 1 for black king, 2 for no checkmate | Used to detect checkmate for non-repeating moves |
| detect\_check\_mate\_2 | F | Sprite, 2 ints – vert and horiz displacement | Int - 3 if a piece is blocking movement and the piece is not a king, 0 for black checkmate, 1 for white checkmate, 2 for continue | Used to detect checkmate for repeating moves |  |  |
| detect\_check\_mate\_w | P | No | No | detect checkmate for white pawns, king, and knights |
| detect\_check\_mate\_b | P | No | No | detect checkmate for black pawns, king, and knights |
| detect\_check\_mate\_final | F | Sprite, 3 ints – vert and horiz displacement, and white or black indicator (1 white, 0 black) |  | assign values for highlights, detect a checkmate, and exit. 1 - checkmate, 0 - exit, 2 - continue checking |

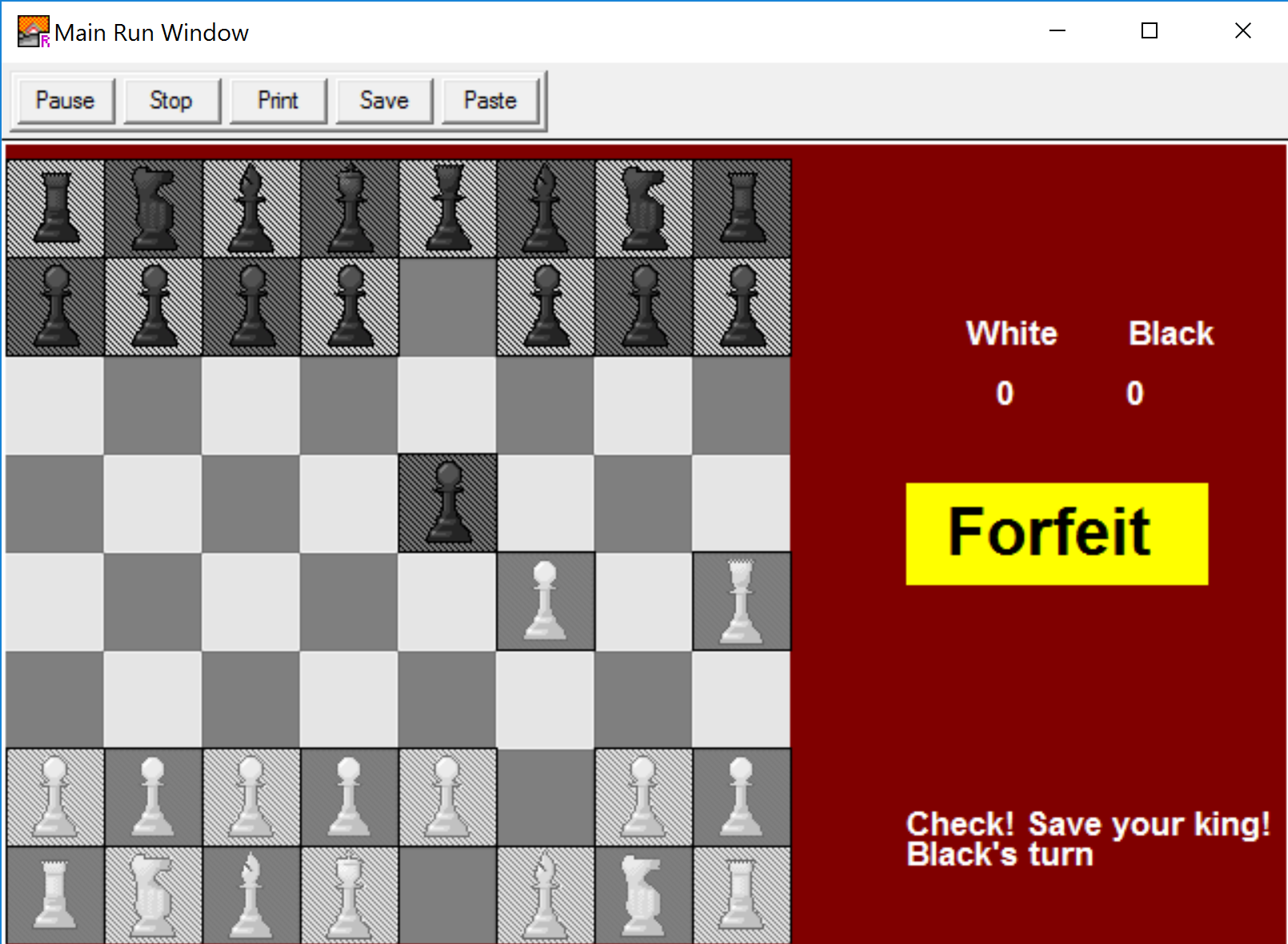
Testing Data

Here are some examples for when the program responds to input correctly:

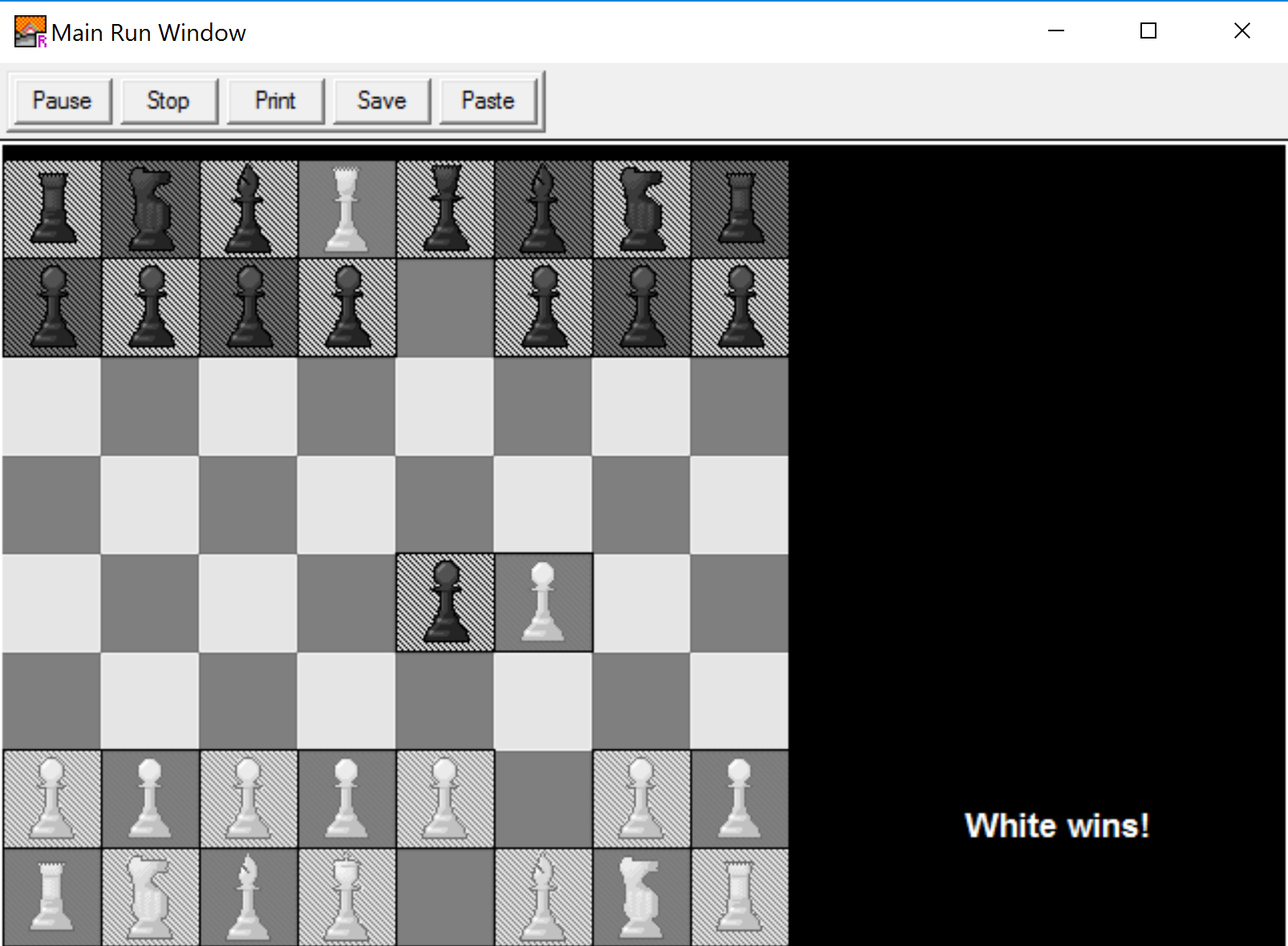
1. The program correctly detects tiles which are occupied by a piece of the same colour and restricts the user from moving there.



1. The program correctly detects a check for either user.

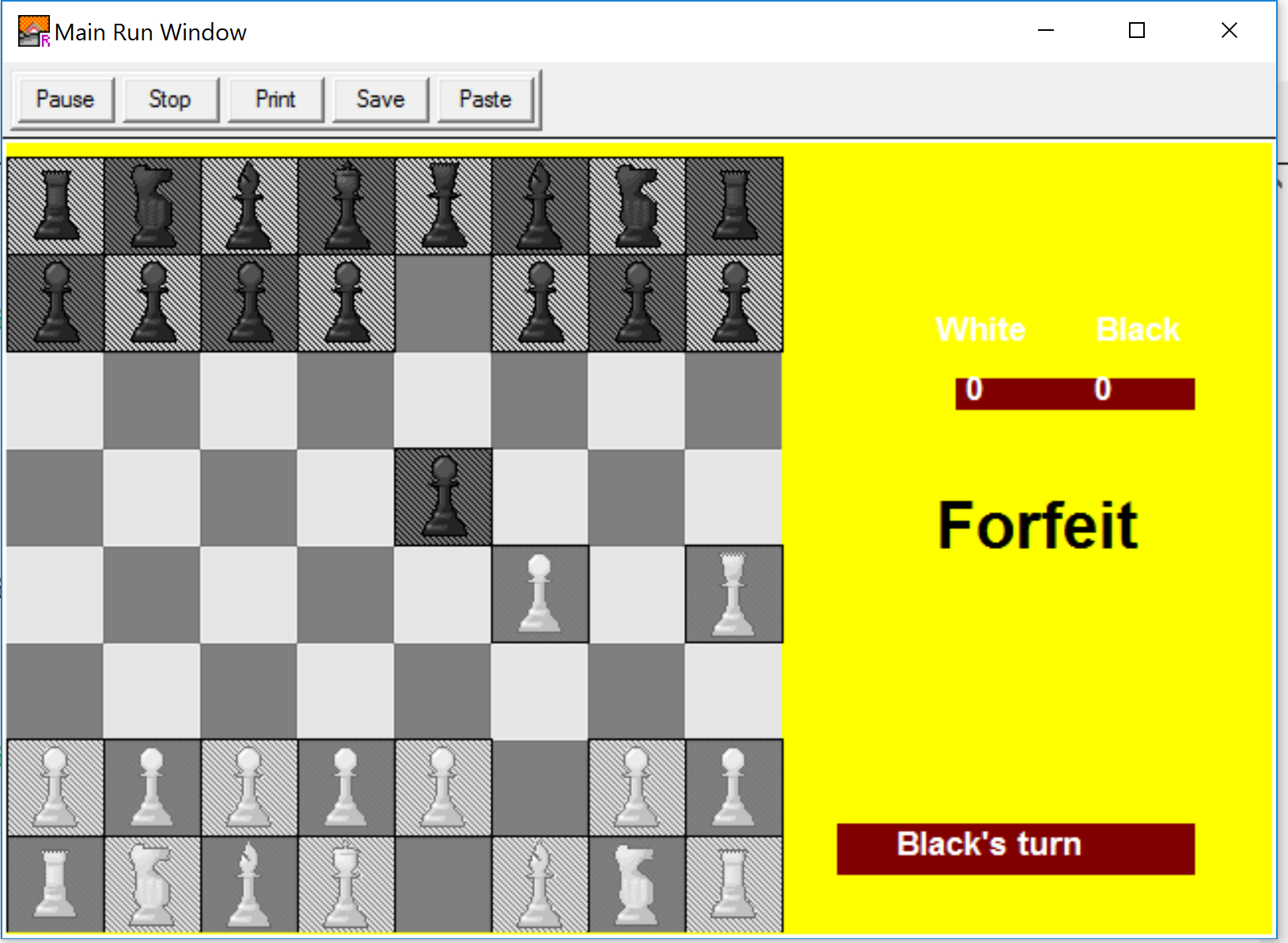


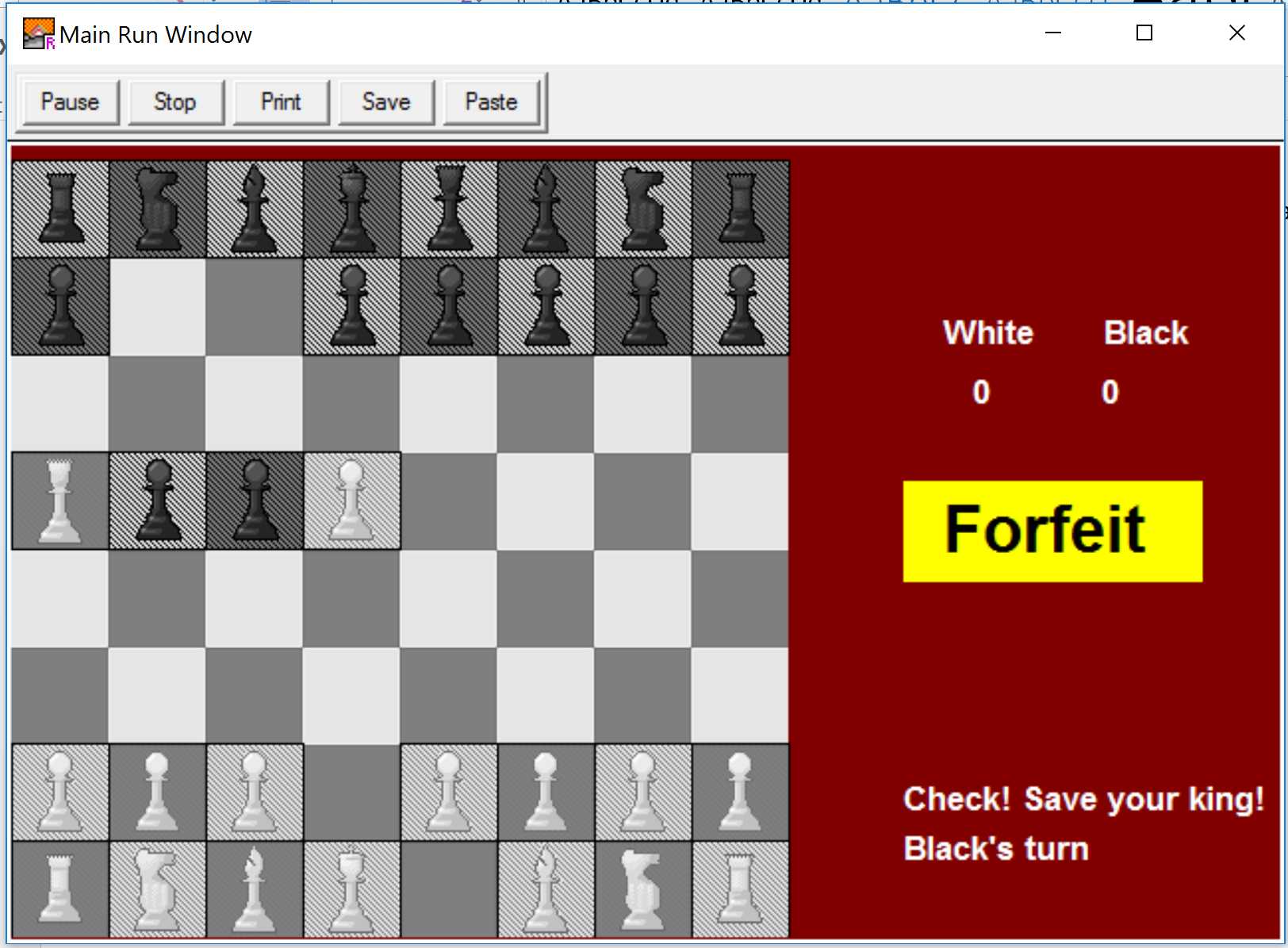
1. The program correctly detects when the game is over (either king is taken)

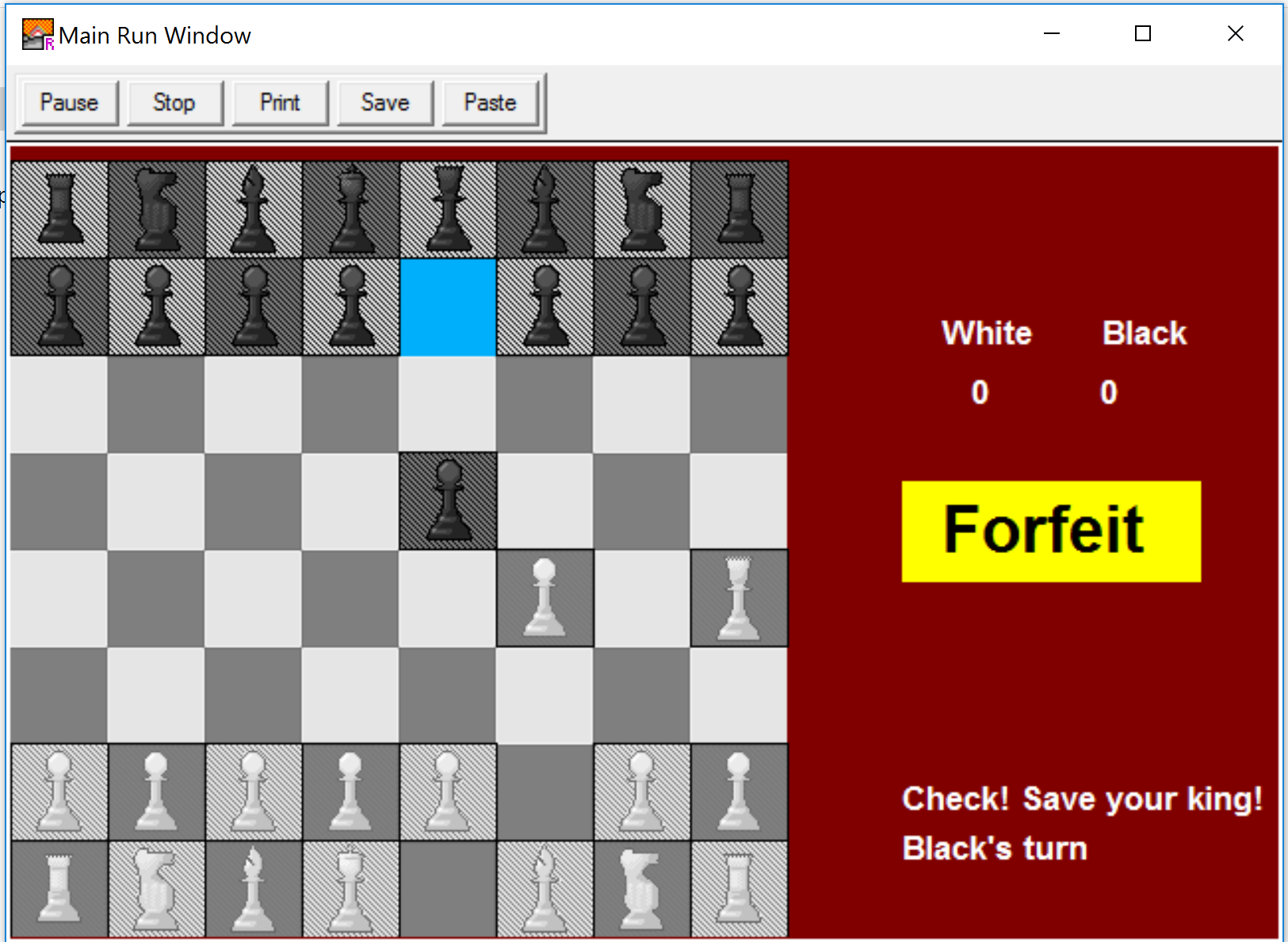


Known Errors

Within the program, there are a few known errors. For example:

1. The program sometimes detects a checkmate when there is no checkmate (in the image below, the black queen can move up one tile to stop the check).
2. The program sometimes does not detect a checkmate when there is a checkmate.



1. The program allows users to move their king into a position that would could end the game on the next turn (the white queen can still take the black king if it moves diagonally one square). 

**Week 1 Project Work Report**

Name: Ibrahim Irfan

Week May 1 to May 7

Name of Project: Chess

Note(s):

For the first week Part 1 does not need to be completed.

For the last week Part 3 does not need to be completed.

**Part 2**

a) Work completed during the current week (include both new tasks and leftover tasks from the previous week)

* Initialize board and piece values: 2 hours
* Click detection for every piece: 30 min
* Move algorithm created for pawn, knight, and king: 2 hours

b) Work not completed this week

* Two-player turn by turn game
* One player AI
* Move algorithm for queen, bishop, and castle
* Taking opponents pieces
* Check/checkmate detection
* Intro menu

**Part 3**

Work to be completed next week (this must include tasks not completed this week as well as new tasks).

* Move algorithm for queen, bishop, and castle
* Two-player turn by turn game

**Week 2 Project Work Report**

Name: Ibrahim Irfan

Week of May 9 to May 13

Name of Project: Chess

Note(s):

For the first week Part 1 does not need to be completed.

For the last week Part 3 does not need to be completed.

**Part 1**

List of tasks not completed during the previous week that were to be completed this week.

- Move algorithm for queen, bishop, and castle

- Two-player turn by turn game

**Part 2**

a) Work completed during the current week (include both new tasks and leftover tasks from the previous week)

- Move algorithm for queen, bishop, and castle (all pieces now completed): 2 hours

- More efficient procedures and functions: 1 hour

b) Work not completed this week

- Two-player turn by turn game

- One player AI

- Taking opponent’s pieces

- Check/checkmate detection

- Intro menu

As I spent some time making my original code more efficient, I was not able to complete the two-player turn by turn game.

**Part 3**

Work to be completed next week (this must include tasks not completed this week as well as new tasks).

- Two-player turn by turn game

- Taking opponent’s pieces

**Week 3 Project Work Report**

Name: Ibrahim Irfan

Week of May 30 to June 3

Name of Project: Chess

Note(s):

For the first week Part 1 does not need to be completed.

For the last week Part 3 does not need to be completed.

**Part 1**

List of tasks not completed during the previous week that were to be completed this week.

- Two-player turn by turn game

- Taking opponent’s pieces

**Part 2**

a) Work completed during the current week (include both new tasks and leftover tasks from the previous week)

- Two-player turn by turn game: 1 hour

- Taking opponent’s pieces including displaying the score: 1 hour

- Check detection: 3 hours

- File divided into sub files: 20 min

b) Work not completed this week

* Intro screen
* Checkmate and stalemate detection
* Preventing user from making a move that would expose the king

I was able to complete everything I wanted to complete this week.

**Part 3**

Work to be completed next week (this must include tasks not completed this week as well as new tasks).

* Intro screen
* Checkmate and stalemate detection
* Preventing user from making a move that would expose the king

**Week 4 Project Work Report**

Name: Ibrahim Irfan

Week of June 6 to June 10

Name of Project: Chess

Note(s):

For the first week Part 1 does not need to be completed.

For the last week Part 3 does not need to be completed.

**Part 1**

List of tasks not completed during the previous week that were to be completed this week.

* Intro screen
* Checkmate and stalemate detection
* Preventing user from making a move that would expose the king

**Part 2**

a) Work completed during the current week (include both new tasks and leftover tasks from the previous week)

- Intro screen: 20 min

- Introduced pointers which shortened code: 2 hours

b) Work not completed this week

* Checkmate and stalemate detection
* Preventing user from making a move that would expose the king

Since I found out a way to shorten my code immensely using pointers, I spent a great deal of time on that rather than some of the other things which were supposed to be completed this week.

**Part 3**

Work to be completed next week (this must include tasks not completed this week as well as new tasks).

* Checkmate and stalemate detection
* Preventing user from making a move that would expose the king

**Week 5 Project Work Report**

Name: Ibrahim Irfan

Week of June 13 to June 17

Name of Project: Chess

Note(s):

For the first week Part 1 does not need to be completed.

For the last week Part 3 does not need to be completed.

**Part 1**

List of tasks not completed during the previous week that were to be completed this week.

- Checkmate and stalemate detection

- Preventing user from making a move that would expose the king

**Part 2**

a) Work completed during the current week (include both new tasks and leftover tasks from the previous week)

- Rules screen: 30 min

- Checkmate detection (inconsistent): 3 hours

- Preventing user from making a move that would expose the king (inconsistent): 30 min

- Finishing touches (comments, burning to disk, etc): 2 hours

b) Work not completed this week

- Stalemate detection

- Checkmate detection (consistently working)

- Preventing user from making a move that would expose the king (consistently working)

The above topics were too difficult to implement with the amount of time I had left, so they had to be left out of the final version of the program.

Conclusion

In May, my goal was to create two-player chess game with the option to play against an AI as well. I recognized from the beginning that creating a chess AI on Turing, or on any programming language, would be extremely difficult, however I wanted to create something that would challenge me in some way. In the end, I ran out of time to include the one player mode, but the two-player mode proved to be more than difficult enough. It was a topic which really made me think and pushed me to write my code in the most efficient way possible. Despite this, I ran into a *countless* number of bugs, but it was interesting to find ways to work around them. My code in the end turned out really well, as (besides the checkmate detection) it was almost bug-free. Also, I was able to try out new things, such as pointers, which I had never used before in Turing, which cut down my code a few hundred lines. Although my program is long, I feel that it’s efficient and as non-repetitive as I could make it. Overall, my program was a success. I feel that I am a better programmer after coding it, and definitely enjoyed the process.

References

Mrs. Barsan’s website < https://sites.google.com/a/hdsb.ca/barsanl/home/courses/ics3u1 > was used to refer to arrays, records, and general Turing syntax help.

Compsci < http://compsci.ca/v3/index.php?h=1&pf=120 > was used for Turing help, specifically with pointers.

Wikipedia’s page on Chess < https://en.wikipedia.org/wiki/Chess > was used for help with clarifying the rules of chess.