CS1006 – ANIMAL CHESS – REPORT

PREFACE

This practical is probably going to be the largest you have marked so far. The last practical received feedback requesting more detail in the report, despite already being way over the advised word limit. As this practical is more complicated than that one the word limit has been ignored. As the output of the program is in colour, the sample-session.txt file does not properly convey it. A coloured version of the sample session is available in the sample-session-with-colour.pdf file. Some UML files, generated by IntelliJ, have been provided to assist with understanding the structure of the program. These can be found in the uml folder. Development for this program was done using the windows command prompt and the Lucida Console font. The pieces on the game board are meant to be square, so if they are not you may wish to mess with the cell spacing options in your terminal emulator until they are. If the blue pieces are hard to see you should increase the green content of the blue colour in the terminal emulator's settings.

SUMMARY OF FUNCTIONALITY

- Arrow key based, coloured, responsive user interface.
- Multi-threaded minmax AI with user controllable depth.
- Ability to pause and quit the game early.
- The optional rule outlined in the specification where a player can win by moving their lion to the other side of the board.
- The ability to play against players on the local network using web sockets.

DESIGN

GAME LOGIC

The first part of the program developed was the Piece class. This represents a piece in the game. It contains an enum containing possible move directions, a boolean to specify which player it belongs to, and a position on the game board. The move directions are dynamic and flip whenever the piece is taken. This class contains methods to check if the piece can move to a certain coordinate and to check if another piece can be captured. These checks do not consider position on the board; only move directions and player types. These will be checked by the game itself. The piece also contains a representation character that can be used by a text-based UI. Each piece also has an id which is used to uniquely identify each piece. This id is retained when the piece is copied or promoted, even though the object is different. This is used in overloaded equals and hashCode methods. Moves are communicated to the pieces using four methods: moved, placed, promoted, and captured. These all return a piece object, in the case of the piece base class always this, that can be used to return a new object when promoted or captured etc. The moved method takes the new position as a parameter and sets the position to the new position. captured changes the piece's player and sets its position to null. A position of null is used when the piece is in a player's hand. placed takes a new position and sets the piece's position to this. Pieces also store a boolean that determines if they are promotable. The promoted method in Piece does not do anything. The five types of piece extend the Piece class and are Chick, Elephant, Giraffe, Lion, and Rooster. Elephants and giraffes do not modify any of the movement methods and just pass in their movements, promotability (false), and representations to the super class in the constructor. Lion is promotable and for its promoted action triggers a victory for the player it belongs to. This is also triggered if the lion reaches the end of the board and cannot be taken by overriding the moved method and storing a reference to the game which is passed in the constructor. These are an interpretation of the rule where the game is won if the lion reaches the opposite end of the board and cannot be taken. promoted only gets called if a piece has been in the last row for at least one turn so even if the lion reaches the end and could be taken, if the other player does not win, the player can still win. The captured method is also overridden to make the other player win if captured. Chick overrides moved and promoted to return a rooster if moved to the last row or promoted. Rooster overrides captured to return a chick.

The Game class contains the game board and orchestrates moves and placements. The position of pieces on the board is represented using a class called Coord. This contains the x and y position on the board. The board itself is stored as a map of Coord to Piece. The equals method of the Coord then had to be overridden to check coordinate equality instead of object equality. As a hash map is used, the hashCode method also had to be overridden. In this case one coordinate is added to a multiple of another that is larger than the first one can become. It contains methods to check if a piece can move to a particular location or be placed in a certain location. Pieces are allowed to move one space off the end of the board if they are promotable and are in the last row. This is relevant, for instance, when a chick is placed in the final row of the board and the player wants to promote it next turn. There are also methods to do a movement or placement. These take the player to move and either a movement direction for a move or a coordinate for a placement. When a piece is captured it is removed from the board and the result of its captured method is added to the player who captured it's hand. The piece to be moved or placed is removed from its old location and the result of its moved or placed method is added at the new position. If the new location is off the board the promoted method is used instead and the piece is placed back in its current position. There are also methods that retrieve all possible moves and all possible placements on the board. These return a map that uses either coordinates or hand indexes as keys and sets of either possible moves or coordinates as values. The play method is the game loop and performs each move in turn until the game is won. For each move a method in the player class; doMove, is called and then another object; the game handler, is told a move has ended. In the case of this program the game handler is the UI and it uses this opportunity to render the board. If a graphical UI or mobile app were created, then different game handlers could be created. The Player class contains the players hand, a boolean for which player it is (true for player 2), and a name. It also contains an abstract method called doMove that, as previously mentioned, is called when it is this player's move.

USER INTERFACE

The standard java console input and output streams are not very capable. The program cannot receive input characters until the enter key has been pressed. This practical instead utilises the JLine library to process console input and output. This allows key presses to be read instantly and also allows the screen to be cleared. This enables a wide array of possibilities which are not possible with the standard java libraries. An example of this is navigation of a menu using the arrow keys. The first class created in the user interface code was the TerminalHandler class. This creates a JLine terminal and buffered input and output readers for it. This originally utilised two interfaces: KeyHandler and PrintHandler. A key handler contains a method called keyPress which handles a single key press by the user. PrintHandler has a method called getDisplay which returns a string which is intended to be written to the terminal. The terminal handler waits for the user to press a key then passes it to the stored key handler. It then clears the terminal window, gets the string from the print handler and prints it to the screen. Later on this was modified to also check the size of the terminal window, and if it has changed the new size is passed to a new interface; ResizeHandler, that contains a method called resize that takes the new size as parameters. The KeyHandler interface was also modified to return a KeyHandler. This is to allow a key handler to return a different key handler for example when choosing a piece to move. As reading from an input reader is a blocking operation, any terminal window resizes would only take effect after a key press. This is fixed by moving the read to a new thread. The BlockingThread class is used to achieve this. It contains a boolean field called complete that is set to true after an operation, which is left abstract, and can be accessed by a public getter. The Reader class is an implementation that reads a character from an input reader provided in the constructor. The field that stores the read value is initialised to -2. When a character is read in by the TerminalHandler a reader is used. While completed is false the main thread will check for terminal window resizes. A 1ms delay is inserted into this loop to stop the program using all cpu power. This is still not ideal but the JLine library is very sparsely documented and a resize event could not be found. To break the program loop, a boolean called exit can be set to true.

DRAWING

To draw to the terminal window a string must be provided. This can contain ANSI escape codes for colouration. In this program two enums are used to store these codes; one for colours and another for background colours. The ColouredChar class is used to represent a coloured character with a foreground and background colour. A ColouredStringBuilder is then used to assemble these into a string containing the appropriate escape sequences. These are only needed when changing colours, so the coloured string builder keeps track of the colour of the previous character. A coloured string is a string of one colour and works similarly to a coloured char. In fact, an interface called ColouredElement is used to simplify the logic of the coloured string builder by providing getters for the colour and background colour. To draw things on the screen, the coloured string builder is used. Everything that can be drawn (including the coloured string and character) implements the Section interface. This contains getters for width and height, and a method that will add a row of the section to a coloured string builder. DrawableSection is an implementation of this and also provides an implementation for PrintHandler that uses a coloured string builder to get each row of the section, separated by the new line string. A buffered section is one that can return itself as a 2D array of coloured chars and is represented by the BufferedSection interface. The DrawableBufferedSection class is an implementation of this that extends DrawableSection and simply contains a 2D array of coloured chars. For getRow it just appends each character from the row to the coloured string builder. Coloured string and char also implement this interface as they have a fixed size and can easily be converted to an array.

A section composer is used to combine multiple sections together. A similar coordinate system to the game board is used to store the location of each section. In a basic section composer, a tree map of coordinates is used, and the entries are stored primarily in order of x coordinate. This means that when getRow is called the map can be traversed in order. The first section in the map that intersects the first character in the row is used to get the character. The getRow method in the Section interface was modified at this point to take a start position and a maximum length. Any characters before the x coordinate of the next section in the map that have not been covered by a previous section must be empty and are therefore set to the filler character, a space with no colour. A buffered section composer works slightly differently. It still contains a map of coordinates to sections (these must be buffered sections), but these coordinates also specify a z-index. Each buffered section in the map is copied, in order, onto the buffer of the composer. The map is sorted primarily by z-index so this can be used to place some elements on top of others. The copy operation is done when an update method is called. This has been added to the section interface and is called by the DrawableSection before assembling the output string. The section composers call the update method of each section in the map when their update method is called. Accessing the sections in a section composer is inconvenient, so the SectionGrid class stores a 2D grid of sections for easier referencing. It also contains a constructor that can create and populate a section composer based on a grid. This works out the maximum size of section in every row and column in the grid and uses these to work out the coordinates for the section composer. To get an instance of the section composer, an abstract method is used that returns a new instance of a section composer. This is implemented for BasicSectionComposer and BufferedSectionComposer.

As the user can resize the terminal window, it would be good if the contents of the screen were responsive to changes in size. The ResponsiveSection interface is used to achieve this and contains a method called setSize that contains a new size in rows and columns. This is implemented by DrawableResponsiveSection, that stores a size inside it and updates it when setSize is called. It also implements ResizeHandler and maps it to the setSize method. The CentredSection class is a responsive section that contains another section and centres it within itself. Its getRow method works out the space before and after the section and pads it so as to make it appear in the centre. Another responsive section is the MinimumSizeSection. This contains two sections and shows a different one when it is smaller than a minimum size. This is used in the game to show a message to the user asking them to make their terminal window bigger if it is too small to fit the game. A BorderSection contains another section and generates a border around it. Initially this was intended to use box drawing characters, but these were found to not show properly in most terminal emulators. Nevertheless, these are still available just unused in the program. An enum containing all the different border character sets is used. This contains the horizontal, vertical, and all four corner characters for each set. The getRow method for this class adds the border characters to both sides of the getRow method of the contained section, except for the top and bottom rows which are all border characters. The filler class is a section that consists all of the same character. Its getRow method simply appends the filler character for the required number of times.

INPUT

User input to the program is implemented using the aforementioned KeyHandler interface. One implementation of this is MovementKeyHandler class. This is an abstract class that translates key presses into movement directions. The WASD keys are mapped directly to underlying abstract methods for each direction. The arrow keys are slightly more complicated in that they use an escape code. In this case the last character in the escape sequence corresponds to the arrow key. This class just checks for these naively in the same way as checking for WASD. The space and enter keys are also given their own abstract methods, while another interface; the escape handler, is used for when the user presses escape. Detecting escape is not trivial however, as the arrow keys also produce an escape as part of their escape sequences. If the escape is part of an arrow key escape sequence the next character will be 91 or 79 depending on operating system. The next character can be read and if it is 91 or 79 an escape is not registered. The issue comes when the user presses the escape key and nothing else. As reading a character blocks until a character is available, the escape will not be processed until the user presses another key. To solve this the method in the terminal handler that waits for the next character was modified to accept a timeout. If this timeout has expired -2 is returned. The Reader object is then stored ready for the next time the method is called and will continue to wait for a key press in the background. As -2 is not 91 or 79 this will still register as an escape. Any other key is passed to an abstract method called other. Another implementation of KeyHandler is TextInputKeyHandler. This contains a regex expression, and any keys input are matched against it. If they match, they are sent to an abstract method called typed. If the backspace key is pressed, a separate abstract method is called. Any other keys are passed to an other method the same as the one in MovementKeyHandler.

The SelectionHandler interface defines something that can handle navigation in a primary and secondary directions, with an action when the user specifically selects it. An extension of this is the SelectableSection interface. This adds the specification that the section can be enabled or disabled, it has a parent SelectionHandler, and an underlying key handler. This underlying key handler is often the MovementSelectionHandler, an extension of MovementKeyHandler, that maps the movement keys on the keyboard to the selection handler. Extension interfaces are extended from this that also implement BufferedSection and ResponsiveSection; SelectableBufferedSection and SelectableResponsiveSection respectively. An implementation of SelectableSection is DrawableSelectableSection; an extension of DrawableSection that stores a boolean for if the section is enabled, a key handler, and a parent section. It implements the directional SelectionHandler methods to call the parent's methods or do nothing if the parent is not defined. It also adds a new abstract method; unfocus, that is called when the parent's movement methods are called. A buffered extension to this; DrawableSelectableBufferedSection, extends DrawableSelectableSection and contains an underlying buffered section that is used to satisfy all the buffered section interface requirements. There are extensions for the bordered, centred, and minimum size sections that specify selectable sections only as the types of section that can be stored. The selectable section methods are all passed through to the stored section (the one that's currently selected in case of the minimum size section).

The SelectableGrid class is an extension of DrawableSelectableSection that contains a SectionGrid and is designed to allow navigation between selectable sections. It uses the section grid's grid to outline the positions of the sections relative to each other. It then creates its own internal representation of the grid to better suit its task. This is a list of treemaps of integer to section. For each row in the grid, if there are any sections there, they are added to a treemap with a key of their column in the grid. The treemap is then added to the list. Two integers; selectedRow and selectedCol, keep track of the selected index in the list and key in the treemap respectively. The reason for using a map instead of another list for rows is that when moving from row to row the closest section in the next row must be found. This would not be possible with a list as the indexes could not align depending on how full each row is in the grid. The secondaryNext and secondaryPrevious methods are overridden to get the next or previous element in the treemap. If none is available, then the selectable grid's parent selection handler's equivalent method is called. This is important, for instance, when nesting selectable grids. When there are no more elements in the nested grid it will move to the next element in the parent grid. The primary directions are slightly more complicated. These check each row above/below them depending on which direction and checks if there is a section in the same position as the currently selected one. If there is not it checks the closest on each side in the map. If none are found that are enabled, it moves to the next row. If the last row is reached and none are found, then the parent's method is called. Due to limitations of java's generics, the underlying section composer can only hold selectable sections. The reason for this is that the type of section in the grid must extend the type of section in the section composer but must obviously also extend SelectableSection. Java does not allow requiring a type parameter to extend both another type parameter and an interface, so the only way to force sections in the grid to extend SelectableSection is to make sections in the section composer also extend SelectableSection. This is a limitation as you may want to have additional background sections in the section composer. To get around this issue, a class called UnselectableSection was created. This contains a normal section which is used for all section interface methods, but also implements SelectionHandler. The selection handler methods are all replaced with dummy methods. Extensions of this for responsive and buffered sections were also created.

GAME AND MENUS

The menus in the game are implemented using selectable grids, and menu elements. These are selectable sections that fulfil certain tasks expected for a menu system. These include buttons, option choosers, and text boxes. Buttons are characterised by square brackets on each side of the button text. When they are focused the background colour is changed, and when they are selected the button action is done. Option choosers contain an array of strings and when focused the secondary move directions are overridden to change the displayed string. This is used, for example, to choose which type each player is such as human player or AI player. The text box makes use of a TextInputKeyHandler to get user input and display it in the section. The game menus let the user pick types for each player and configure them with different configuration options available for each type of player. There is also a pause menu and a game over menu that let the user exit to the menu or exit the game entirely. The Piece class (different to the logic piece class) extends DrawableSelectableBufferedSection and contains a logic piece. It uses the piece's representation for the centre character, and the characters surrounding it are used to represent the piece's moveable directions. All the moveable directions are represented by a bullet point. If the piece can currently move in a direction, the bullet point is lit up in the player's colour. This uses the canMove method of the logic Game class. If the piece is null, then the buffer is filled with the filler character. If the piece is currently enabled, it is highlighted yellow and if the piece is currently focused it is highlighted a different colour.

The GameBoard class is an extension of SelectableBufferedGrid that contains a grid of pieces. It spaces the pieces out in the underlying section composer and sets the composer's buffer to a solid character. The pieces are copied over the top of this, but the border remains around them. As pieces can be promoted from the last row, a selectable section was needed to allow the user to do this. This is a PromotionIndicator and there is a row of them above the pieces in the board. During the game, the board flips so that the human player playing is always at the bottom of the board. This means that promotion indicators are only needed at the top. The game board is updated from scratch every time a change is made. The reason for this is to avoid having to make sure the two boards are in sync. When the game board is updated, the updateBoard method is used. This takes the board from the logic game as a parameter, along with a set containing all the coordinates that are selectable by the user. The player's hands use the same Piece section as the board, but the grid is dynamically created to show the pieces in a player's hand. This is due to the fact that the size of the hand changes. Each piece in the hand is wrapped in a selectable buffered grid that has a background of the solid character. The piece is in the middle of the section. This allows one piece to be drawn and have the correct border, but if two pieces are drawn next to each other, they can be drawn with overlapping borders and the section composer will draw them on top of each other. This may be slightly inefficient but increases the simplicity of the code. The pieces are drawn in rows of 2 and if there is an odd number one is centred. When the pieces are updated there is two ways to enable pieces for selection. If it is the player's turn then all pieces will be enabled, so a boolean called enabled can be set to true. If the user has selected a piece from the hand then the other pieces should be disabled, but this piece should remain enabled to allow the user to cancel their selection.

The Game class is another extension of SelectableGrid. Its grid contains both player's hands and the game board, as well as a pause button. It also implements EscapeHandler and pauses the game when escape is pressed. At the end of every turn board is drawn to the terminal window. This allows the user to, for example, watch two AI players playing against each other. Every piece is disabled for selection so the user can only select the pause button. When it is a human player's move, the doHumanMove method is called. This sets the pieces belonging to the player to be selectable and waits for user input. It also flips the board if it is player 2's turn. When a piece is selected, the piece class's selected method calls the game's selectPiece method with itself as the parameter. It is then stored as the selected piece and only the coordinates where the piece can move and the piece itself are enabled. If the same piece is selected again the move is cancelled and all pieces are enabled again. This also happens if the user presses escape. Once the user chooses a place to move to the move / place method of the game is called. The HumanPlayer class calls the doHumanMove method when its doMove method is called.

AI

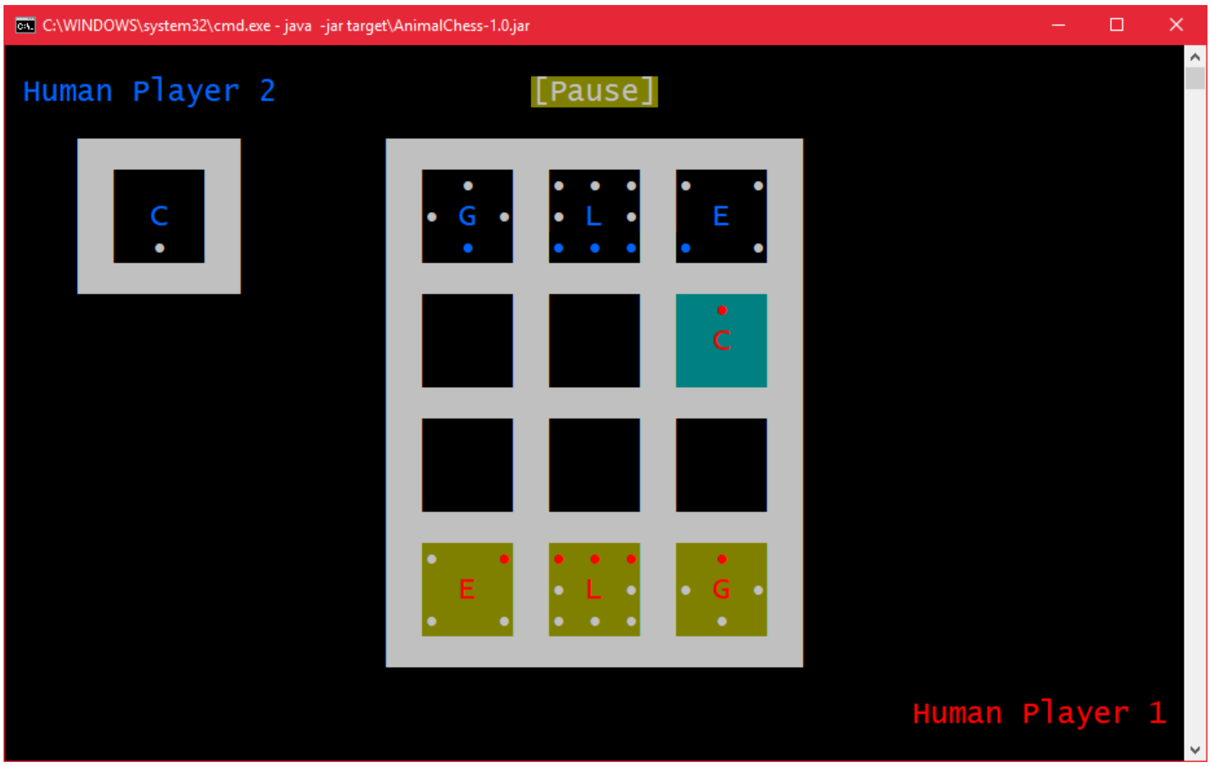
The program uses the minmax algorithm for its AI player. This is multithreaded and utilises alpha beta pruning. The first layer of the minmax is split over a thread for each CPU core of the system and share a maximum value for pruning. A copy of the game is created for each thread, but the copy is of a subclass just for the AI called AIGame. This keeps a stack of moves completed and can roll them back. This allows the AI to operate more efficiently as copying the whole game is a long operation, while rolling back a move only creates copies of the pieces affected. A copy of the piece before the move is stored alongside its old and new positions. When the move is rolled back the piece at the new position is removed and the copy is placed back in the old position. When a piece is taken, a separate move is stored for the take that is rolled back after the move of the taking piece. The static metric for the board takes into account the number of directions a piece can move, whether the piece is in the player's hand or on the board, how far up the board the piece is, whether the piece is promotable or not, and if the piece is in the central column (as it will have more moveable directions in general). The value of each piece is proportional to which player they belong to and are summed to provide a single score. If the game is won a very large or very negative number is returned depending on which player has won. When pruning, if the value is large enough to be a victory, then the branch is pruned. If a victory is detected at any stage in the minmax then the static metric is returned instantly. While the AI operation is going the user should still be able to interact with the UI, so the minmax operation is carried out using a blocking thread. A new method was added to the terminal handler called await that takes a blocking thread as a parameter. It starts the thread then handles user input until the thread is completed. This allows the user to pause the game and exit it while the AI is processing.

ONLINE PLAYER

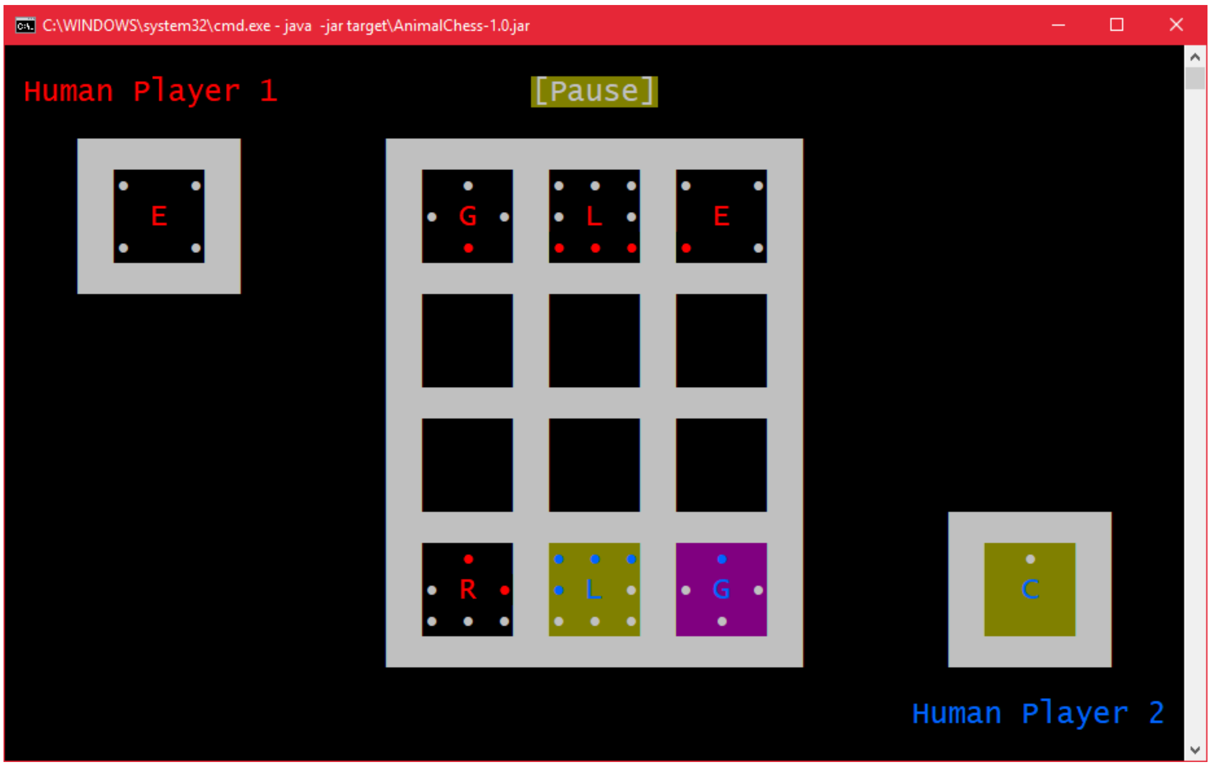
The game includes an additional player type that enables playing with another user on the local network. This was implemented because it adds a lot of potential for minimal code. It was decided that the server should always be player 2 and the client always player 1. For each move this player sends the game state to an established socket and then waits until it receives the new game state from the socket. This process is the same for the client and the server, apart from on the first move where the client skips sending the state. The name of the other player in the game is also sent in an initial handshake. While the server socket waits for new players, the await method of the terminal handler is used to allow the user to cancel the connection while a connection screen shows. There is very minimal error handling implemented that sets the game to won if a connection error occurs. More robust error handling has not been added here due to time constraints and the fact that it is not part of the specification. Each piece on the board is sent with its coordinate represented as two integers, after its id as an integer and a boolean specifying if it belongs to player 2. After this its type is sent as a byte. The DataInputStream and DataOutputStream classes are used for this section of the program.

TESTING

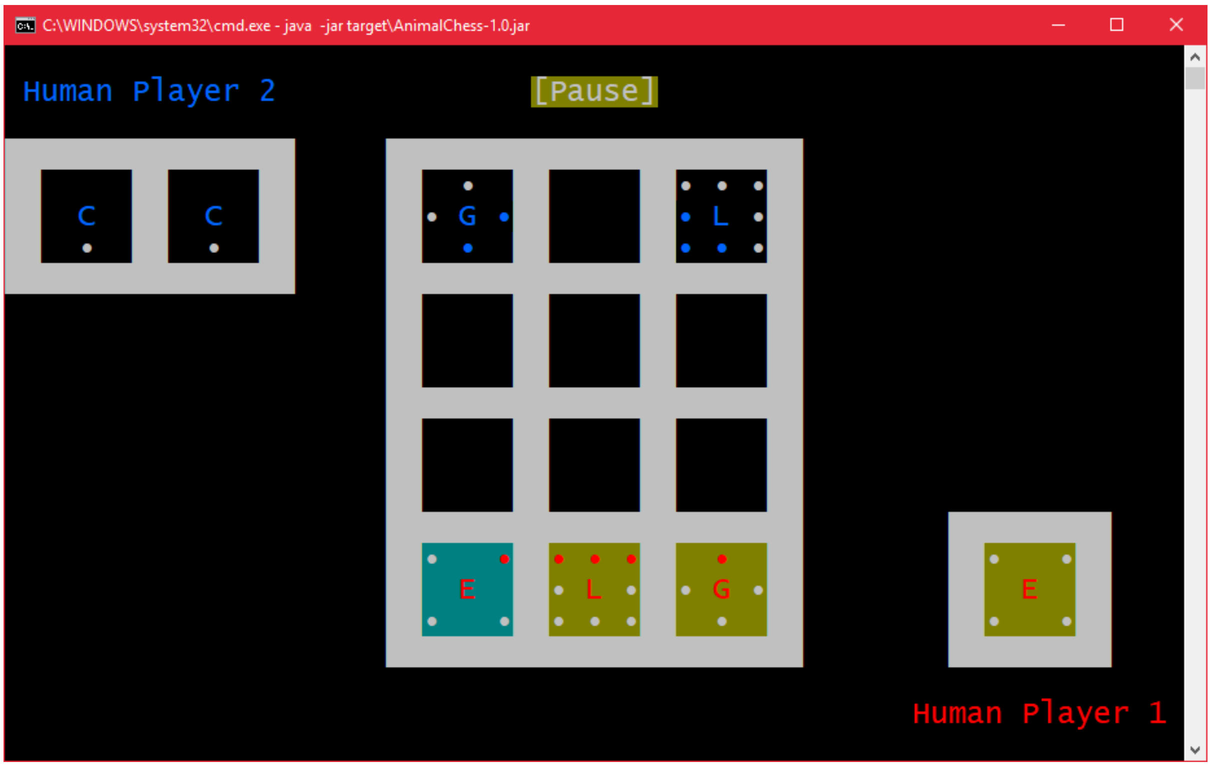
As this program does not have a great deal of input validation, there is not a lot of edge case testing to do. Instead the different game rules will be tested. The first rule to be tested is the promotion of a chick when it reaches the opposite end of the board. The following situation has been set up:


C:\WINDOWS\system32\cmd.exe - java -jar target\AnimalChess-1.0.jar
Human Player 2
[Pause]
C
G
L
E
C
E
L
G
Human Player 1

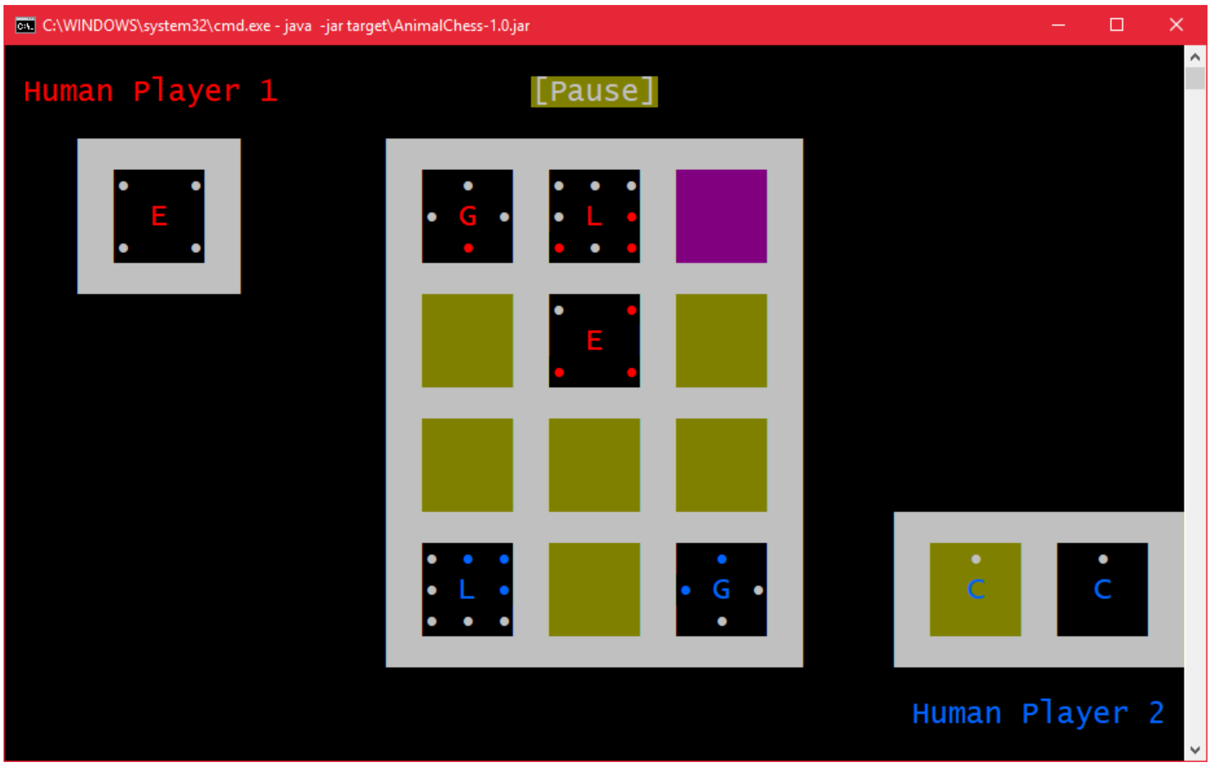
The chick to the top right of the board should be able to take the elephant and be promoted into a rooster. This is what happened:


C:\WINDOWS\system32\cmd.exe - java -jar target\AnimalChess-1.0.jar
Human Player 1
[Pause]
E
G
L
E
R
L
G
C
Human Player 2

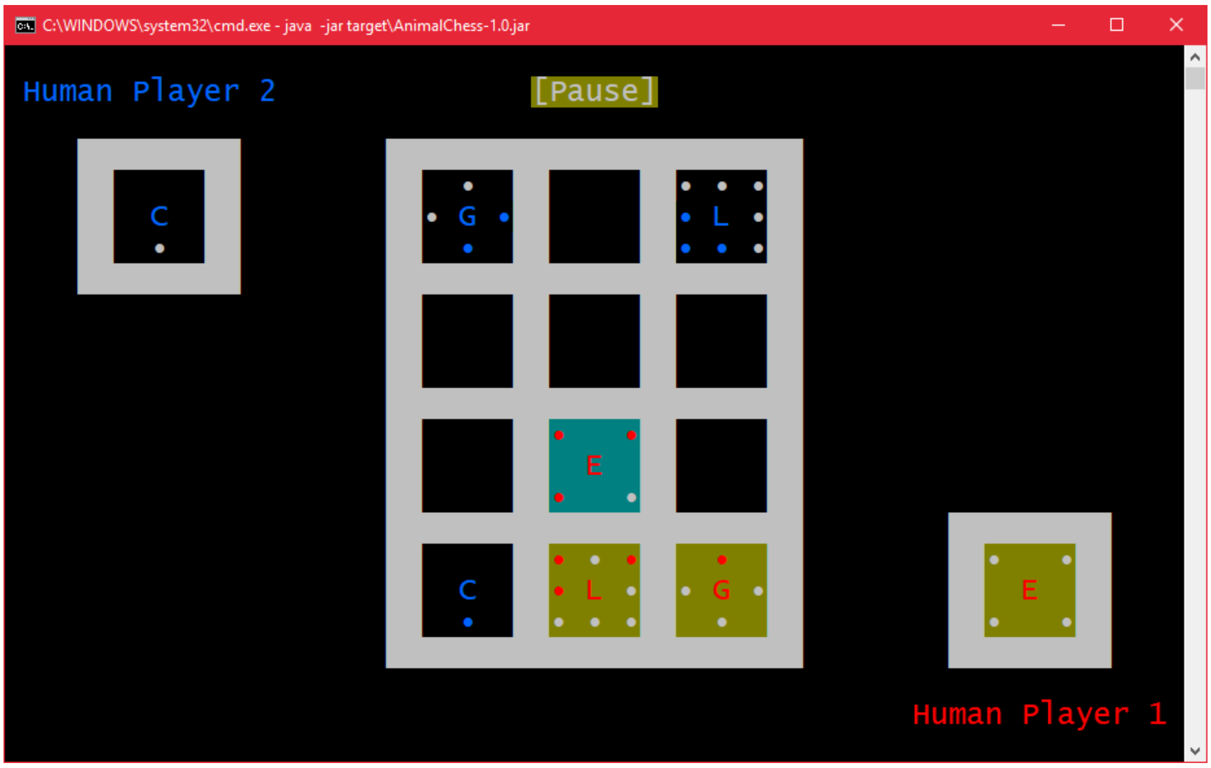
Ignoring the board flipping to the other player's perspective, the chick has been correctly promoted to a rooster. The next test is that the rooster should revert to a chick when taken. The blue lion will take the rooster:


C:\WINDOWS\system32\cmd.exe - java -jar target\AnimalChess-1.0.jar
Human Player 2
[Pause]
C
C
G
L
E
L
G
E
Human Player 1

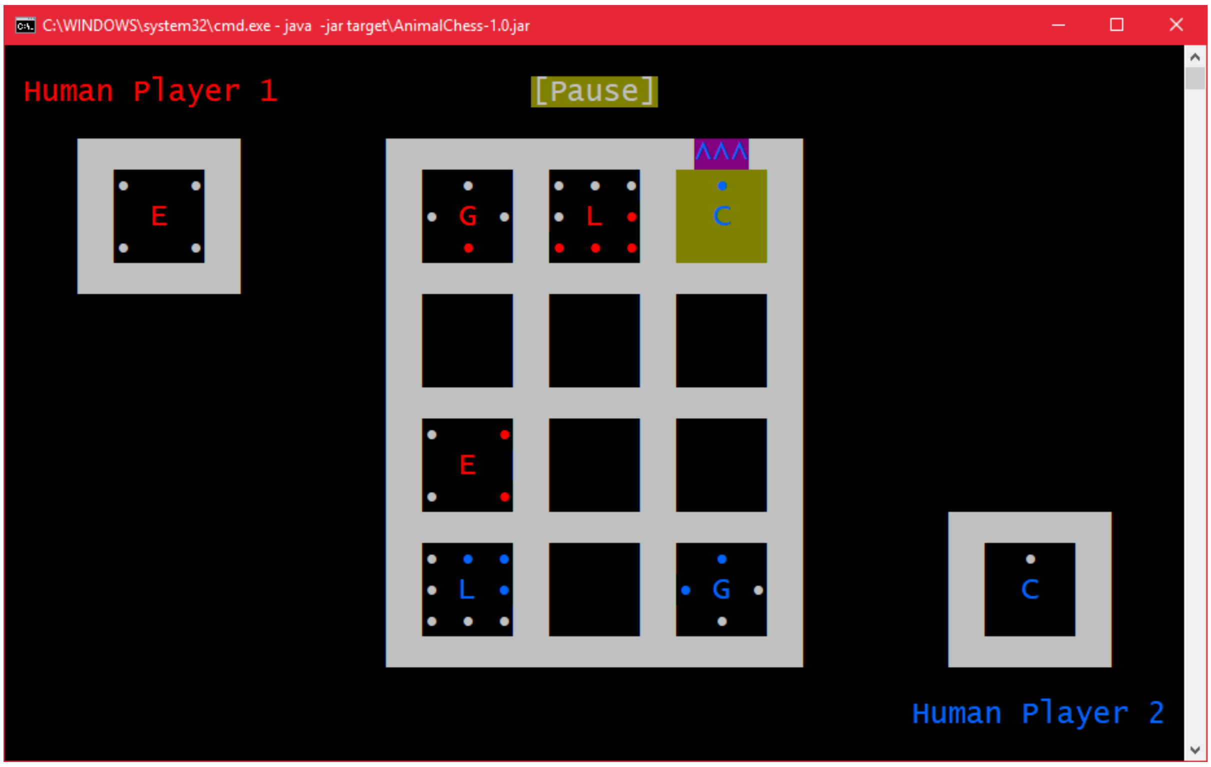
The rooster has been reverted to a chick in the other player's hand. The next test is that a chick placed into the last row from the player's hand should not get promoted. To test this the following condition was set up:


C:\WINDOWS\system32\cmd.exe - java -jar target\AnimalChess-1.0.jar
Human Player 1
[Pause]
E
G
L
E
L
G
C
C
Human Player 2

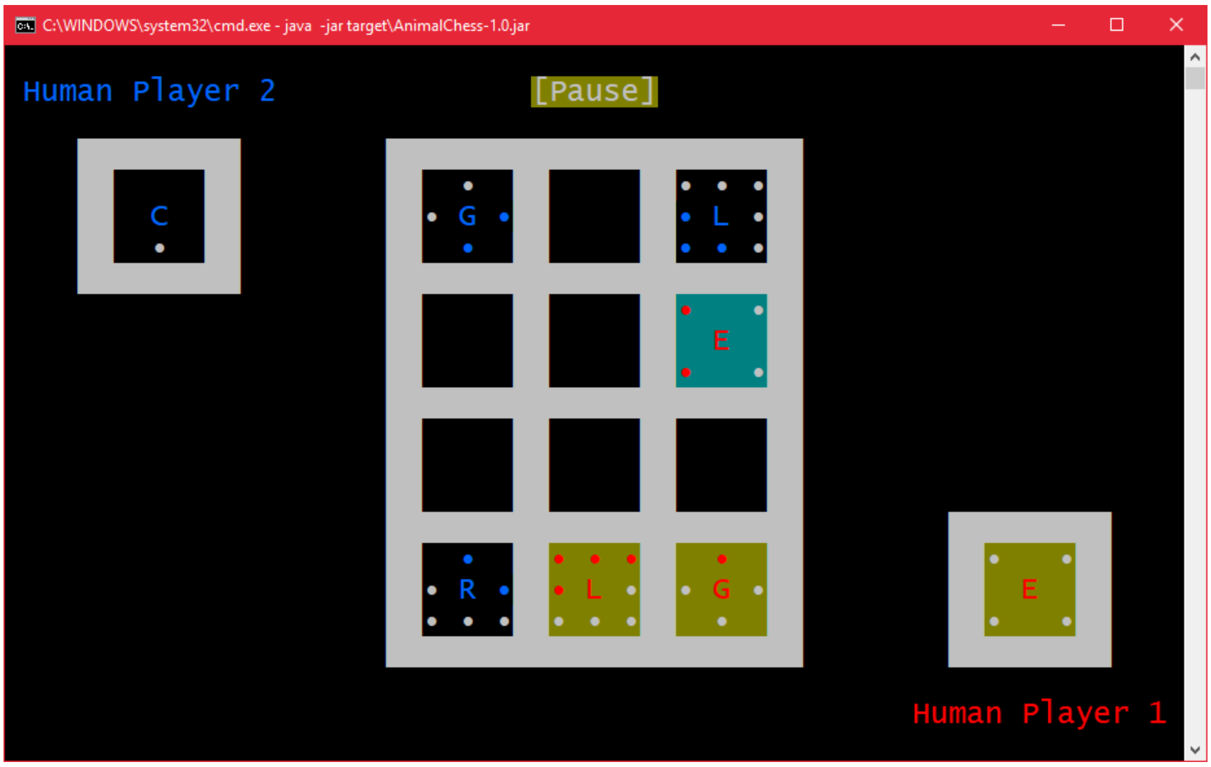
One of the blue chicks is placed into the last row:


C:\WINDOWS\system32\cmd.exe - java -jar target\AnimalChess-1.0.jar
Human Player 2
[Pause]
C
G
L
E
C
L
G
E
Human Player 1

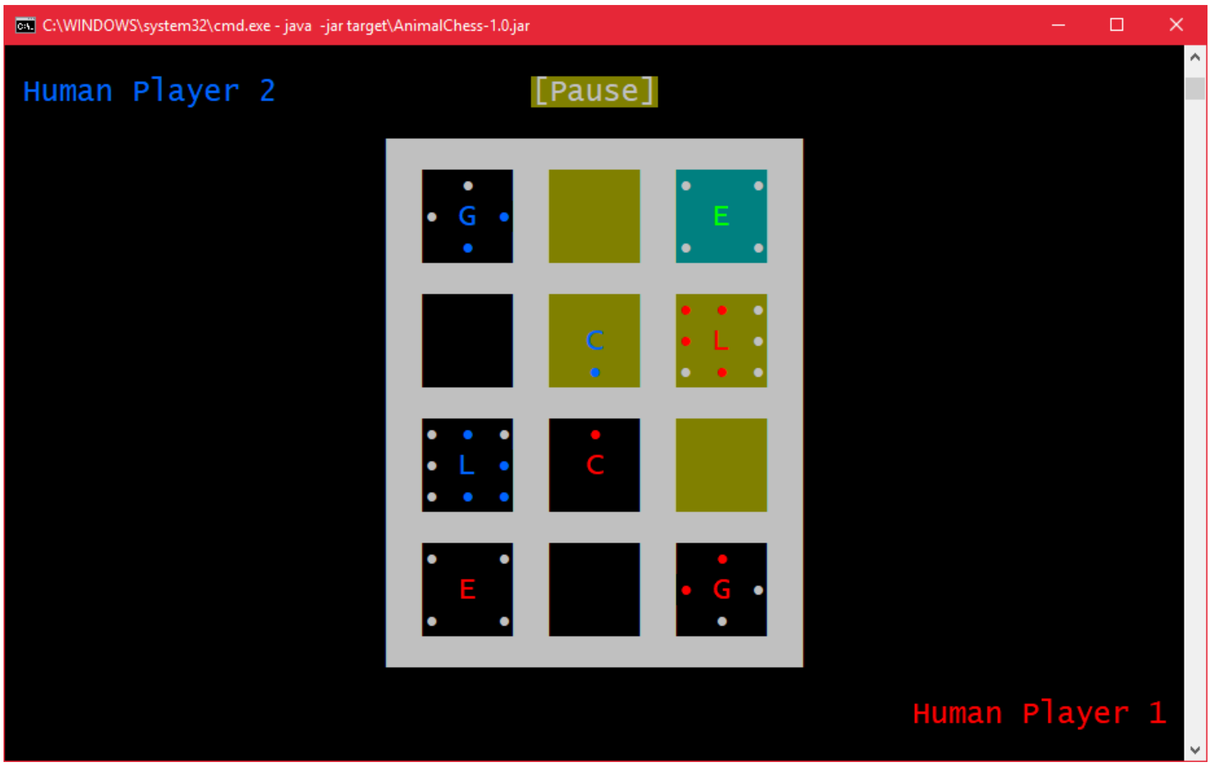
The chick remains a chick and should be promotable next turn:


C:\WINDOWS\system32\cmd.exe - java -jar target\AnimalChess-1.0.jar
Human Player 1
[Pause]
E
^^^
G
L
C
E
L
G
C
Human Player 2

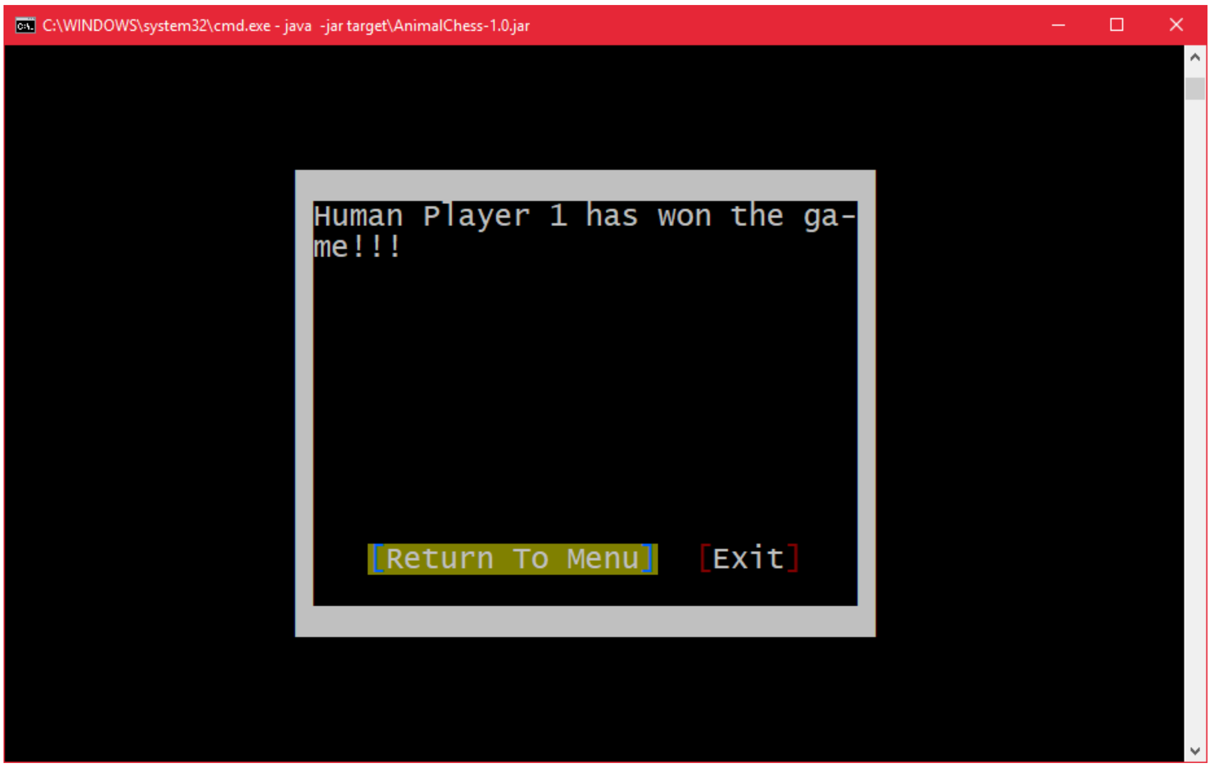
The promotion indicator above the chick has been enabled and should promote the chick to a rooster when selected:


C:\WINDOWS\system32\cmd.exe - java -jar target\AnimalChess-1.0.jar
Human Player 2
[Pause]
C
G
L
E
R
L
G
E
Human Player 1

The chick has been promoted to a rooster. The next test is that a lion that reaches the opposite side of the board should win the game if it can not be taken. The following situation was set up:

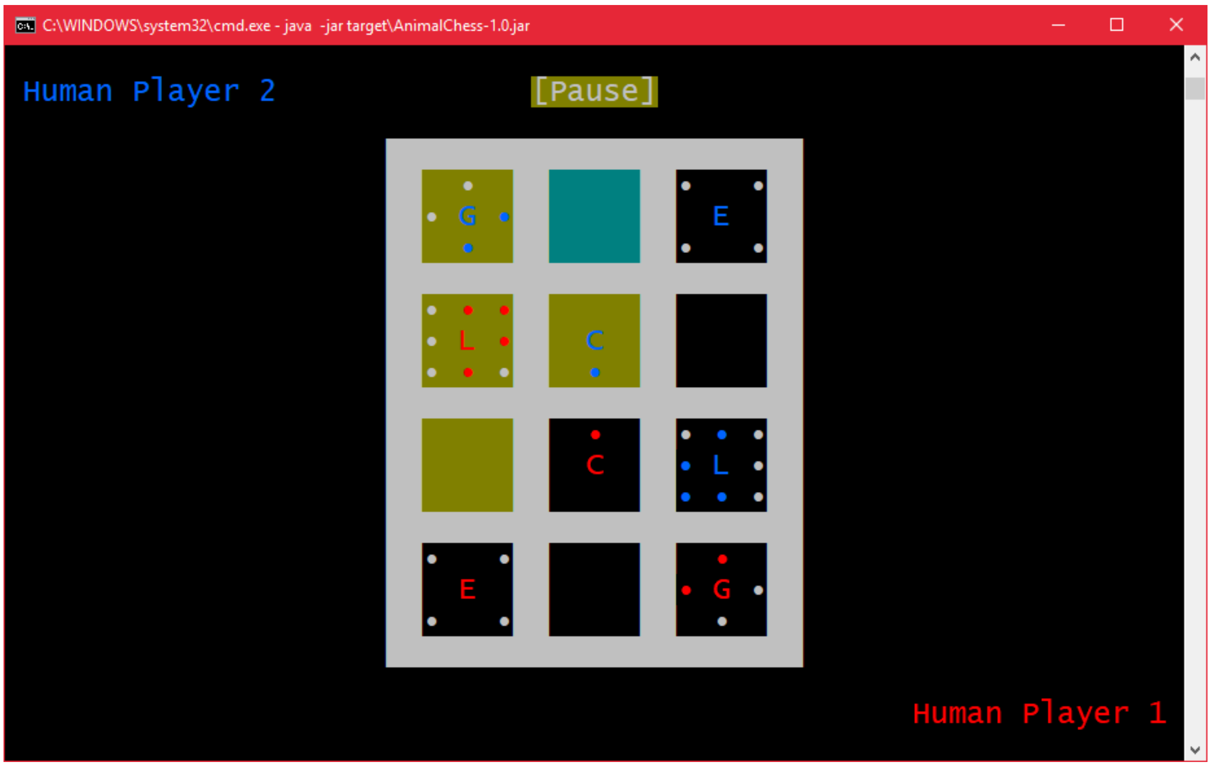

C:\WINDOWS\system32\cmd.exe - java -jar target\AnimalChess-1.0.jar
Human Player 2
[Pause]
G
E
C
L
L
C
E
G
Human Player 1

The red lion should win the game as it cannot be taken in this position, so Human Player 1 should win:


C:\WINDOWS\system32\cmd.exe - java -jar target\AnimalChess-1.0.jar

```
Human Player 1 has won the ga-
me!!!
```

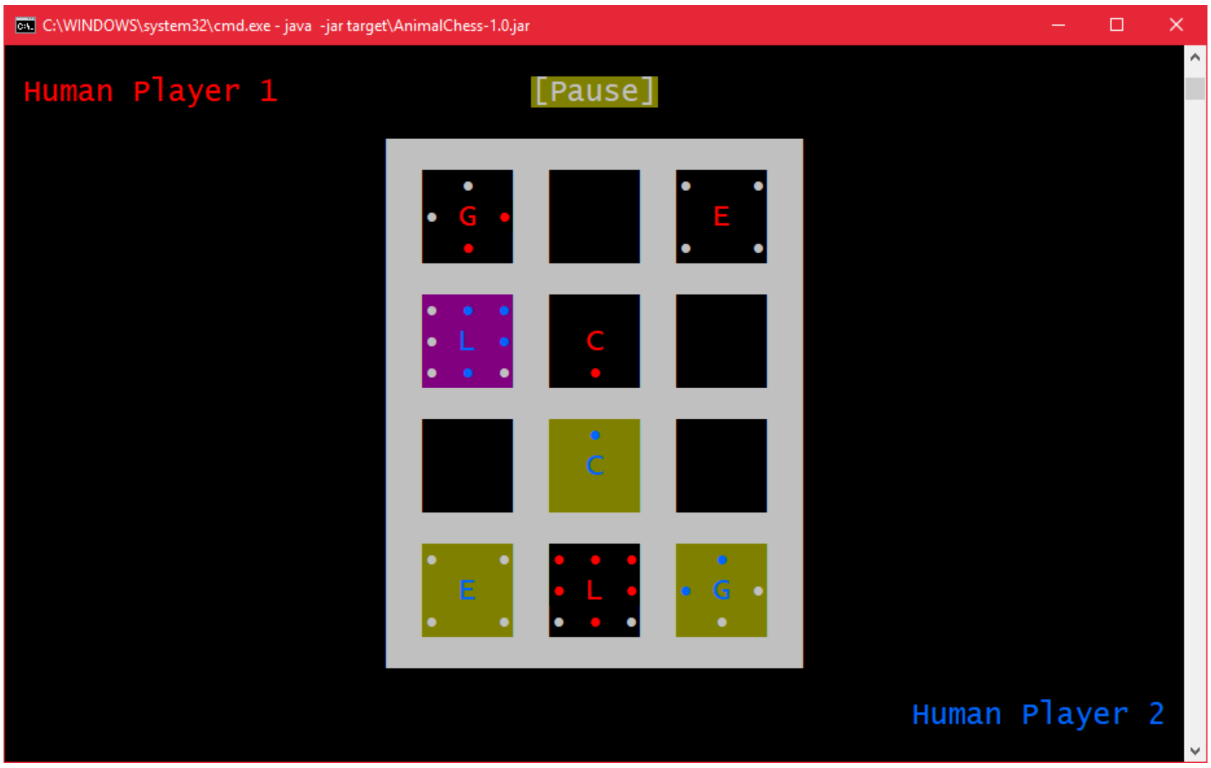
The next test will test when another piece can take the lion. The following condition was set up:



C:\WINDOWS\system32\cmd.exe - java -jar target\AnimalChess-1.0.jar

```
Human Player 2                [Pause]

                     G            E
                     L     C
                           C      L
                     E            G

                                          Human Player 1
```

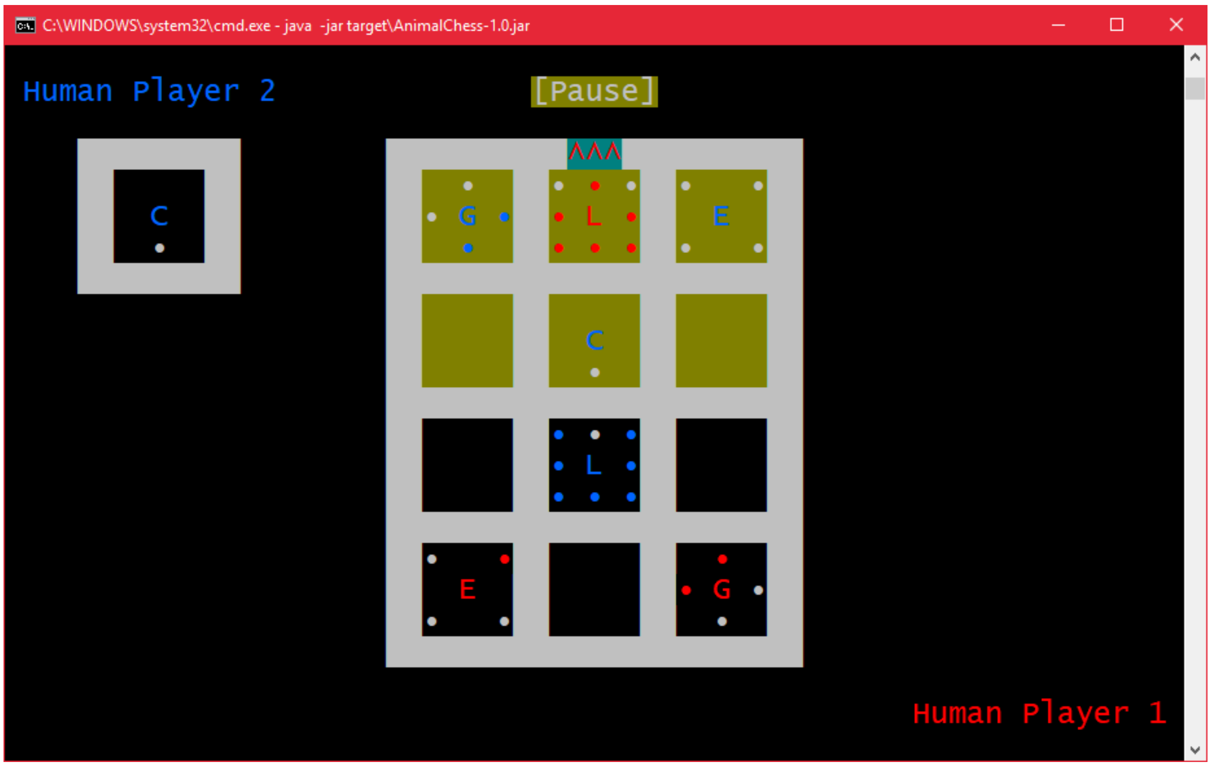
The red lion should not win the game this turn as it could be taken by the blue giraffe:



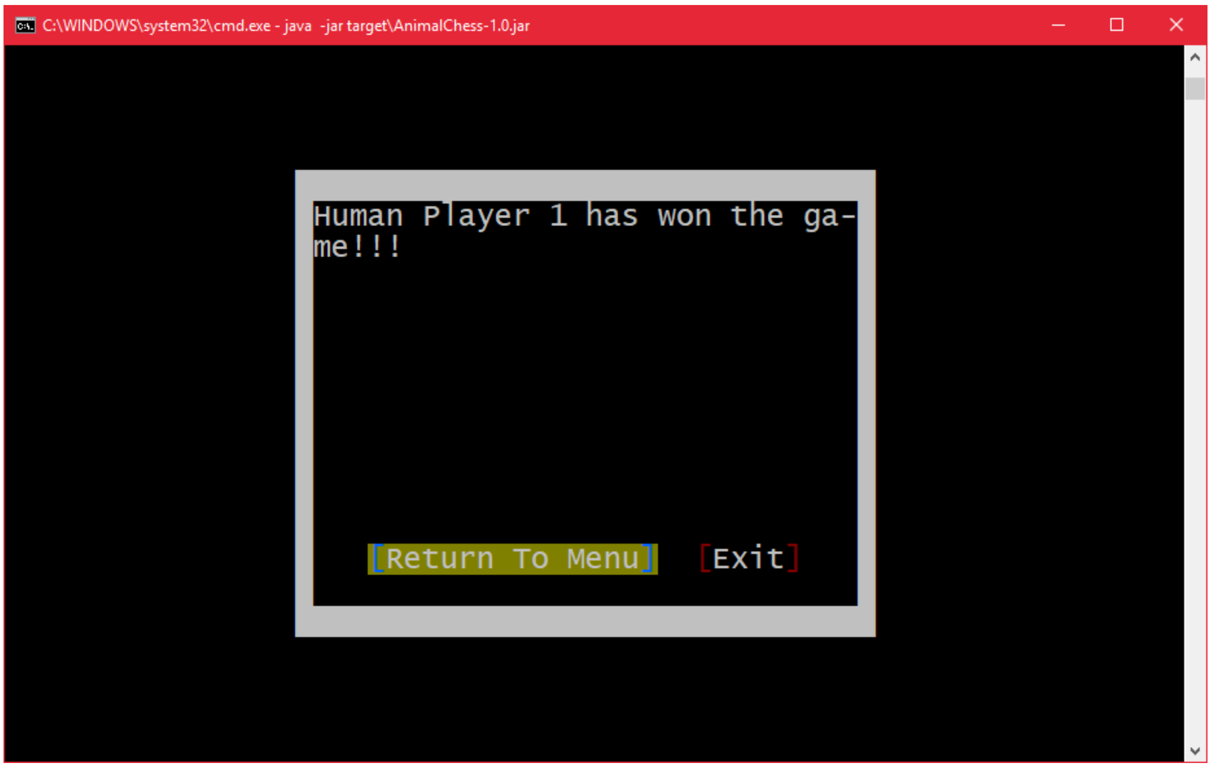
C:\WINDOWS\system32\cmd.exe - java -jar target\AnimalChess-1.0.jar

```
Human Player 1                [Pause]

                     G            E
                     L     C
                           C
                     E     L      G

                                          Human Player 2
```

The lion should then be able to win the next turn by being promoted. This is the set up:


C:\WINDOWS\system32\cmd.exe - java -jar target\AnimalChess-1.0.jar
Human Player 2
[Pause]
C
G
L
E
C
L
E
G
Human Player 1

Human Player 1 should win when the lion is promoted:


C:\WINDOWS\system32\cmd.exe - java -jar target\AnimalChess-1.0.jar
[Return To Menu]
[Exit]

```
Human Player 1 has won the ga-
me!!!
```

BUGS AND DEFECTS

Due to the complexity of the program, there are bound to be some undiscovered bugs and glitches. Here is a list of discovered ones:

- When a piece is selected from a player's hand, no piece appears to be focussed even though the game thinks the piece in the hand is still focused.
- The AI does not always produce the same move for the same condition, probably due to its multi-threaded nature.
- The backspace key does not function correctly in a text box in some terminal emulators.
- There may be some kind of memory leak, which has been kept under control by garbage collecting frequently. Without this the program would rapidly consume multiple gigabytes of memory.

Some defects in the program are as follows:

- Very little user input validation is done, especially for the online section of the game.
- Most errors are just thrown as unchecked exceptions and will crash the program.
- The AI only utilises as many CPU cores as top-level moves are available.
- The code layout is cluttered and unclean, due to not having a solid architecture thought out beforehand.
- The program will not run on all terminal emulators and will crash if the user tries to pipe input into it.
- No accessibility options are available.

EVALUATION

The program matches the specification. It implements the game of animal chess using a text-based user interface and contains an AI for the user to play against. The user interface is intuitive, allowing the user to navigate it using the arrow keys. The pieces are displayed in a grid and the user can select which piece to move (and where to move it) using the arrow keys. The directions a piece can move are clearly indicated. The players are clearly differentiable due to using different coloured pieces. The AI is powerful and even on the hardest labelled difficulty level makes a decision on average before five seconds has passed. On the default difficulty level, the AI moves almost instantly. The user can also connect to another user on the local network or beyond using sockets. If more time were available for this practical, the first thing to be done would be a refactoring of the code. The practical evolved over time and this shows in the code. The resize handler would be merged with the print handler and the selection system would be overhauled. Another change would be to add proper error handling to the online player, with a connection screen that can cancel the connection and retry a failed connection. A section for game options would be added that allows the user to disable the screen flipping between each turn. An instructions dialog would be added.

CONCLUSION

The program matches the specification and is very user friendly. The AI is fast and difficult to beat. This practical has been a success.

WORD COUNT

Words: 5768