New Bulgarian University

Bachelor Faculty

Bachelor programme Network Technologies

NETB 375 Programming Practice

AI Chess Game

Prepared by:

Vasil Yoshev, F 74312

5th semester

Sofia, 2017

Project Description

Using the programming language C++ and programming framework Qt implement a computer program the famous chess game. The game must gave two modes of play: human against human (on the same computer), and human against CPU. The program must have the following features:

- fully functioning GUI that represents the chess board and all pieces;
- white piece can be moved if it is a white player's move, and the same for black pieces;
- each piece is allowed to perform only valid moves according to game rules;
- AI complexity is left to project developers, and it can be the simplest possible -- a random move.

The chess pieces

The Piece class

The chess pieces’ types are distributed in separate classes, which all inherit the Piece base class. The Piece class provides basic common to all piece types methods and fields. The provided fields give information about the piece’s color, whether or not it’s been killed or moved and a PieceType field, providing information about the type of the specific piece. The class also provides setters and getters for the fields.

Every class for a specific piece contains a **constructor**, **getPossibleMoves** function for getting the possible moves of this piece and **getCopy** function to get a copy object of this piece.

In the following lines are described the mechanics of the **getPossibleMoves** functions of all different piece types. For every piece type this function returns a vector of vectors, where each vector represents all moves in a given direction. This is later used in the **specialMovesHandler** by the **filterInvalidMoves**.

The Bishop class

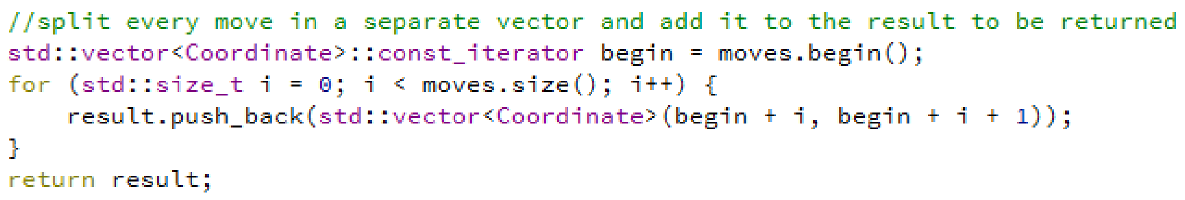
The starting squares are c1 and f1 for White's bishops, and c8 and f8 for Black's bishops. The bishop has no restrictions in distance for each move, but is limited to diagonal movement. Bishops, like all other pieces except the knight, cannot jump over other pieces. A bishop captures by occupying the square on which an enemy piece sits.

In the **getPossibleMoves** function this behaviour is implemented with the help of four vectors, each holding the moves for each diagonal until the end of the board. Vectors are **upperLeft**, **upperRight**, **lowerLeft**, and **lowerRight**. The four vectors are put in a vector of vectors to be returned.

The King class

The white king starts on e1 and the black king on e8. A king can move one square in any direction (horizontally, vertically, or diagonally) unless the square is already occupied by a friendly piece or the move would place the king in check.

In the **getPossibleMoves** function we add all surrounding fields which are inside the borders of the boards to a vector of coordinates. Also in here we check if the king has been moved and also add possible moves for castling – kingside and queenside respectively. Since all **getPossibleMoves** need to return a vector of vectors to be filtered later on in the **specialMovesHandler**, and all the king's moves are in different directions, we need to put every move in a distinct vector and add it to a vector of vectors to be returned:



```
//split every move in a separate vector and add it to the result to be returned
std::vector<Coordinate>::const_iterator begin = moves.begin();
for (std::size_t i = 0; i < moves.size(); i++) {
    result.push_back(std::vector<Coordinate>(begin + i, begin + i + 1));
}
return result;
```

The Kinght class

Each player starts with two knights, which begin on the row closest to the player, between the rooks and bishops. The knight move is unusual among chess pieces. When it moves, it can move to a square that is two squares away horizontally and one square vertically, or two squares vertically and one square horizontally. The complete move therefore looks like the letter L. Unlike all other standard chess pieces, the knight can "jump over" all other pieces (of either color) to its destination square.

In the **getPossibleMoves** function we first generate all possible positions, then we remove those outside the borders of the board. Then, just like with the king, we separate each move in a different vector to be returned later as vector of vectors.

The Pawn class

Each player begins a game of chess with eight pawns, one on each square of the rank immediately in front of the other pieces. (The white pawns start on a2, b2, c2, ..., h2, while black pawns start on a7, b7, c7, ..., h7.) Unlike the other pieces, pawns may not move backwards. Normally a pawn moves by advancing a single square, but the first time a pawn is moved, it has the option of advancing two squares. Pawns may not use the initial two-square advance to jump over an occupied square, or to capture. Any piece directly in front of a pawn, friend or foe, blocks its advance. In the diagram at the right, the pawn on c4 may move to c5, while the pawn on e2 may move to either e3 or e4. Unlike other pieces, the pawn does not capture in the same direction as it moves. A pawn captures diagonally forward one square to the left or right.

In the **getPossibleMoves** function we add both forward moves and diagonal moves (if they are not out of border). Also if a pawn has not been moved a second forward move is added to comply with the game rules. Finally forward moves vector is added to the vector of vectors to be returned and diagonals are split in different vectors and added to the final vector, too.

The Queen class

The white queen starts on d1, while the black queen starts on d8. The queen can be moved any number of unoccupied squares in a straight line vertically, horizontally, or diagonally, thus combining the moves of the rook and bishop. The queen captures by occupying the square on which an enemy piece sits.

In the **getPossibleMoves** function we create a vector for every possible direction – upperLeft, upperRight, lowerLeft, lowerRight, left, right, up, down. After that we fill the vector with coordinates of fields following this direction until the end of the board. The vectors are added to the vector of vectors to be returned.

The Rook class

The white rooks start on squares a1 and h1, while the black rooks start on a8 and h8. The rook moves horizontally or vertically, through any number of unoccupied squares (see diagram). As with captures by other pieces, the rook captures by occupying the square on which the enemy piece sits. The rook also participates, with the king, in a special move called castling.

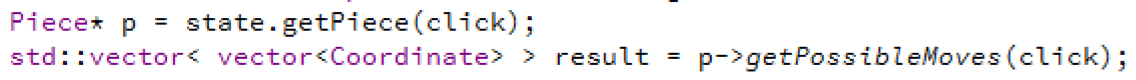
In the **getPossibleMoves** function we create a vector for every possible direction – up, down, left, right. After that we fill the vector with coordinates of fields following this direction until the end of the board. The vectors are added to the vector of vectors to be returned.

The chess logic

The SpecialMovesHandler class

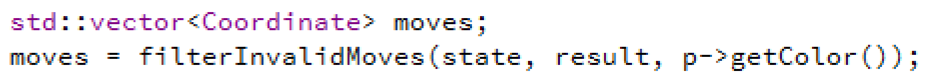
The **SpecialMovesHandler** needs to transform a piece's possible moves to valid for the current game state moves. This is accomplished in the function **getValidMoves**. It does the following things:

1. The function gets the possible moves from a specific piece type, as described in the previous section through the function **getPossibleMoves.**



```
Piece* p = state.getPiece(click);
std::vector< vector<Coordinate> > result = p->getPossibleMoves(click);
```

2. Then it filters these moves through **filterInvalidMoves**.



```
std::vector<Coordinate> moves;
moves = filterInvalidMoves(state, result, p->getColor());
```

3. Finally, it checks for special moves like castling and applies changes, then returns the resulting vector of coordinates.

The **filterInvalidMoves** function receives a vector of vectors as a parameter, where each vector is a collection of possible moves in a given direction. Then it filters each vector depending on the pieces on the board in the current state. The filtering process starts from the specific piece position and goes until the end of the direction vector. If a coordinate is standing on a **friendly** piece it is deleted along with all consecutive coordinates. If a coordinate is standing on an **enemy** piece, all consecutive coordinates are deleted, except for the current coordinate (standing on an enemy piece). The result is returned as a vector of coordinates.

We also have two **getSpecialMoves** functions – one for the king and one for the pawn.

The *king* **getSpecialMoves** function implements fully the castling special move. Castling consists of moving the king two squares towards a rook on the player's first rank, then moving the rook to the square over which the king crossed. Castling may only be done if the king has never moved, the rook involved has never moved, the squares between the king and the rook involved are unoccupied, the king is not in check, and the king does not cross over or end on a square in which it would be in check.

*NOTE: The **getSpecialMoves** does not do the castling. It only checks if all conditions for it are met. The castling is performed in the **Controller** under the*

***movePiece** function, where it is checked if a given moves covers the selected move correspond to a kingside or queenside castle and then it calls the **doCastle** function. The castling is the only move in which two figures are moved with one turn.*

The *pawn* **getSpecialMoves** function acts as a kind of “filter” for the pawn, since it does not behave like the other pieces. It basically does two things:

1. If a forward position is occupied by another piece (enemy or friendly) the pawn can’t take that piece, so if this condition is met the function removes this possible move as it is not a valid move in the current game *state*.
2. If a diagonal possible move position is *NOT* occupied by an enemy figure, the move is erased from the vector of coordinates, as it is not a valid move in the current game *state*.