ChessEngine\ChessEngine.cpp

```
001:  // ChessEngine.cpp : Defines the entry point for the console application.
002:  //
003:
004:  #include "ChessEngine.h"
005:
006:  int main(int argc, char *argv[])
007:  {
008:  	engineLoop();
009:  }
010:
011:  void engineLoop()
012:  {
013:  	TranspositionEntry* transpositionTable = new TranspositionEntry[TTSize];
014:
015:  	Board board = Board();
016:  	board.defaults();
017:  	bool hasSetupEngine = false;
018:
019:  	while (true)
020:  	{
021:  		std::string response;
022:  		std::getline(std::cin, response);
023:  		std::vector<std::string> words = split(response);
024:
025:  		if (response == "uci")
026:  		{
027:  			std::cout << "id name UNSR Chess System\n";
028:  			std::cout << "id author UNSR\n";
029:  			std::cout << "uciok\n";
030:  		}
031:  		if (response == "isready")
032:  		{
033:  			if (!hasSetupEngine)
034:  			{
035:  				hasSetupEngine = true;
036:  				setupEngine();
037:  			}
038:  			std::cout << "readyok\n";
039:  		}
040:  		if (response == "quit")
041:  		{
042:  			break;
043:  		}
044:
045:  		if (words[0] == "position")
046:  		{
047:  			if (words[1] == "startpos")
048:  			{
049:  				if (words.size() == 2)
050:  				{
051:  					board.defaults();
052:  				}
053:  				else if (words[2] == "moves")
054:  				{
055:  					board.defaults();
056:  					Move currentMove;
057:  					for (int x = 3; x < words.size(); x++)
058:  					{
059:  						//Applies each move to calculate the current board.
060:  						currentMove = moveFromNotation(words[x], &board);
061:  						currentMove.applyMove(&board);
062:  					}
063:  				}
064:  			}
065:  			else if (words[1] == "fen")
066:  			{
067:  				std::string fenString = response.substr(13);
068:  				board = Board();
069:  				board.loadFromFen(fenString);
070:
071:  				if (words.size() > 8 && words[8] == "moves") {
072:  					Move currentMove;
073:  					for (int x = 9; x < words.size(); x++)
074:  					{
075:  						//Applies each move to calculate the current board.
076:  						currentMove = moveFromNotation(words[x], &board);
077:  						currentMove.applyMove(&board);
078:
079:  					}
080:  				}
081:  			}
082:  		}
083:
084:  		if (words[0] == "go")
085:  		{
086:  			//Default value of 15 mins
087:  			long int btime = 900000;
088:  			long int wtime = 900000;
089:  			int depth = -1;
090:
091:  			//Retreives the clock values from the message
092:  			if (std::find(words.begin(), words.end(), "btime") != words.end())
093:  			{
094:  				btime = std::stoi(*(std::find(words.begin(), words.end(), "btime") + 1));
095:  			}
096:  			if (std::find(words.begin(), words.end(), "wtime") != words.end())
097:  			{
098:  				wtime = std::stoi(*(std::find(words.begin(), words.end(), "wtime") + 1));
099:  			}
100:
101:  			//Retreives depth from message , if set
102:  			if (std::find(words.begin(), words.end(), "depth") != words.end())
103:  			{
104:  				depth = std::stoi(*(std::find(words.begin(), words.end(), "depth") + 1));
105:  			}
106:
107:  			std::cout << btime << " " << wtime << "\n";
108:
109:  			timeManagement timer;
110:  			if (depth == -1)
```

```
111:                                    timer = timeManagement(btime, wtime, board.nextColour);
112:                            else
113:                                    timer = timeManagement(depth);
114:
115:                            board.printBoard();
116:
117:                            Move pv = startSearch(&board, transpositionTable, &timer);
118:
119:                            std::cout << "bestmove " << notationFromMove(pv) << "\n";
120:                    }
121:            }
122:
123:    }
124:
125:    std::vector<std::string> split(std::string words)
126:    {
127:            std::string temp = "";
128:            std::vector<std::string> wordList;
129:            for (int x = 0; x < words.size(); x++)
130:            {
131:                    if (words[x] == ' ')
132:                    {
133:                            if (temp != "")
134:                            {
135:                                    wordList.push_back(temp);
136:                                    temp = "";
137:                            }
138:                    }
139:                    else
140:                    {
141:                            temp += words[x];
142:                    }
143:                    if (x == words.size() - 1)
144:                    {
145:                            wordList.push_back(temp);
146:                    }
147:
148:            }
149:            return wordList;
150:    }
151:
152:    void setupEngine()
153:    {
154:            magicBitboards::setupMagicBitboards();
155:            ZorbistKeys::initialize();
156:            setupMoveGen();
157:            setupBitboardUtils();
158:    }
159:
```

ChessEngine\ChessEngine.h

```
001:    #pragma once
002:    #include <iostream>
003:    #include <vector>
004:    #include <string>
005:
006:    #include "board.h"
007:    #include "search.h"
008:    #include "utils.h"
009:    #include "transpositionTable.h"
010:    #include "moveGeneration.h"
011:    #include "timeManagement.h"
012:
013:
014:    void engineLoop();
015:    std::vector<std::string> split(std::string words);
016:    void setupEngine();
017:
018:
019:
```

ChessEngineLibrary\bitboard.cpp

```
001:  #include "bitboard.h"
002:
003:  int bitSum(uint64_t bitboard)
004:  {
005:  	int count = 0;
006:  	while (bitboard)
007:  	{
008:  		count++;
009:  		bitboard &= bitboard - 1;
010:  	}
011:  	return count;
012:  }
013:
014:  uint64_t pop(uint64_t& bitboard)
015:  {
016:  	uint64_t lsb = bitboard & -bitboard;
017:  	bitboard -= lsb;
018:  	return lsb;
019:  }
020:
021:  const int index64[64] = {
022:  	0,  1, 48,  2, 57, 49, 28,  3,
023:  	61, 58, 50, 42, 38, 29, 17,  4,
024:  	62, 55, 59, 36, 53, 51, 43, 22,
025:  	45, 39, 33, 30, 24, 18, 12,  5,
026:  	63, 47, 56, 27, 60, 41, 37, 16,
027:  	54, 35, 52, 21, 44, 32, 23, 11,
028:  	46, 26, 40, 15, 34, 20, 31, 10,
029:  	25, 14, 19,  9, 13,  8,  7,  6
030:  };
031:
032:  int bitScanForward(uint64_t bitboard) {
033:  	const uint64_t debruijn64 = 285870213051386505;
034:  	return index64[((bitboard & -bitboard) * debruijn64) >> 58];
035:  }
036:
037:  std::vector<int> getSetBits(uint64_t bitboard)
038:  {
039:  	std::vector<int> setBits;
040:  	while (bitboard)
041:  	{
042:  		uint64_t pos = pop(bitboard);
043:  		setBits.push_back(bitScanForward(pos));
044:  	}
045:  	return setBits;
046:  }
047:
048:  uint64_t inBetweenLookup[64][64];
049:
050:  uint64_t inBetween(int from, int to)
051:  {
052:  	return inBetweenLookup[from][to];
053:  }
054:
055:  void setupBitboardUtils()
056:  {
057:  	for (int from = 0; from < 64; from++)
058:  	{
059:  		for (int to = 0; to < 64; to++)
060:  		{
061:  			inBetweenLookup[from][to] = 0;
062:  			if (from % 8 == to % 8) //Same column
063:  			{
064:  				if (to > from)
065:  				{
066:  					for (int x = from + 8; x < to; x += 8)
067:  					{
068:  						inBetweenLookup[from][to] |= (uint64_t)1 << x;
069:  					}
070:  				}
071:  				else
072:  				{
073:  					for (int x = from - 8; x > to; x -= 8)
074:  					{
075:  						inBetweenLookup[from][to] |= (uint64_t)1 << x;
076:  					}
077:  				}
078:  			}
079:  			else if (from / 8 == to / 8) //Same row
080:  			{
081:  				if (to > from)
082:  				{
083:  					for (int x = from + 1; x < to; x += 1)
084:  					{
085:  						inBetweenLookup[from][to] |= (uint64_t)1 << x;
086:  					}
087:  				}
088:  				else
089:  				{
090:  					for (int x = from - 1; x > to; x -= 1)
091:  					{
092:  						inBetweenLookup[from][to] |= (uint64_t)1 << x;
093:  					}
094:  				}
095:  			}
096:  			else if (std::abs(from - to) % 9 == 0) //Bottom-left to top-right diagonal
097:  			{
098:  				if (to > from)
099:  				{
100:  					for (int x = from + 9; x < to; x += 9)
101:  					{
102:  						inBetweenLookup[from][to] |= (uint64_t)1 << x;
103:  					}
104:  				}
105:  				else
106:  				{
107:  					for (int x = from - 9; x > to; x -= 9)
108:  					{
109:  						inBetweenLookup[from][to] |= (uint64_t)1 << x;
110:  					}
```

```
                                        }
                                }
                                else if (std::abs(from - to) % 7 == 0) //Top-left to Bottom-right diagonal
                                {
                                        if (to > from)
                                        {
                                                for (int x = from + 7; x < to; x += 7)
                                                {
                                                        inBetweenLookup[from][to] |= (uint64_t)1 << x;
                                                }
                                        }
                                        else
                                        {
                                                for (int x = from - 7; x > to; x -= 7)
                                                {
                                                        inBetweenLookup[from][to] |= (uint64_t)1 << x;
                                                }
                                        }
                                }
                        }
                }
        }

uint64_t shift(uint64_t bitboard, int shift)
{
        return (shift > 0) ? bitboard << shift : bitboard >> -shift;
}

```

ChessEngineLibrary\bitboard.h

```
001:  #pragma once
002:  #include <stdint.h>
003:  #include <algorithm>
004:  #include <vector>
005:
006:  #define emptyBitboard 0
007:  #define universalBitboard 18446744073709551615 //2**64 - 1
008:
009:  #define rank1 255
010:  #define rank2 65280
011:  #define rank3 16711680
012:  #define rank4 4278190080
013:  #define rank5 1095216660480
014:  #define rank6 280375465082880
015:  #define rank7 71776119061217280
016:  #define rank8 18374686479671623680
017:
018:  #define fileA 72340172838076673
019:  #define fileB 144680345676153346
020:  #define fileC 289360691352306692
021:  #define fileD 578721382704613384
022:  #define fileE 1157442765409226768
023:  #define fileF 2314885530818453536
024:  #define fileG 4629771061636907072
025:  #define fileH 9259542123273814144
026:
027:  int bitSum(uint64_t bitboard);
028:  uint64_t pop(uint64_t& bitboard);
029:  int bitScanForward(uint64_t bitboard);
030:  std::vector<int> getSetBits(uint64_t bitboard);
031:  uint64_t inBetween(int from, int to);
032:  void setupBitboardUtils();
033:  uint64_t shift(uint64_t bitboard, int shift);
```

ChessEngineLibrary\board.cpp

```
001:  #include "board.h"
002:
003:
004:  Board::Board()
005:  {
006:  	clearBoard();
007:  }
008:
009:  Board::Board(std::string fenString)
010:  {
011:  	loadFromFen(fenString);
012:  }
013:
014:  void Board::clearBoard()
015:  {
016:  	pieceBitboards[white][pawn] = emptyBitboard;
017:  	pieceBitboards[white][rook] = emptyBitboard;
018:  	pieceBitboards[white][knight] = emptyBitboard;
019:  	pieceBitboards[white][bishop] = emptyBitboard;
020:  	pieceBitboards[white][queen] = emptyBitboard;
021:  	pieceBitboards[white][king] = emptyBitboard;
022:
023:  	pieceBitboards[black][pawn] = emptyBitboard;
024:  	pieceBitboards[black][rook] = emptyBitboard;
025:  	pieceBitboards[black][knight] = emptyBitboard;
026:  	pieceBitboards[black][bishop] = emptyBitboard;
027:  	pieceBitboards[black][queen] = emptyBitboard;
028:  	pieceBitboards[black][king] = emptyBitboard;
029:
030:  	nextColour = white;
031:  	enPassantSquare = -1;
032:
033:  	canBlackCastleQueenSide = false;
034:  	canBlackCastleKingSide = false;
035:  	canWhiteCastleQueenSide = false;
036:  	canWhiteCastleKingSide = false;
037:
038:  	kingDangerSquares = 0;
039:  	whiteMaterialScore = 0;
040:  	blackMaterialScore = 0;
041:
042:  	zorbistKey = 0;
043:  	pawnScoreZorbistKey = 0;
044:
045:  	update();
046:  }
047:
048:  void Board::defaults()
049:  {
050:  	pieceBitboards[white][pawn] = rank2;
051:  	pieceBitboards[white][rook] = 129; //2^0 + 2^7
052:  	pieceBitboards[white][knight] = 66; //2^1 + 2^6
053:  	pieceBitboards[white][bishop] = 36; //2^2 + 2^5
054:  	pieceBitboards[white][queen] = 8; //2^3
055:  	pieceBitboards[white][king] = 16; //2^4
056:
057:  	pieceBitboards[black][pawn] = rank7;
058:  	pieceBitboards[black][rook] = 9295429630892703744; //2^56 + 2^63
059:  	pieceBitboards[black][knight] = 4755801206503243776; //2^57 + 2^62
060:  	pieceBitboards[black][bishop] = 2594073385365405696; //2^58 + 2^61
061:  	pieceBitboards[black][queen] = 576460752303423488; //2^59
062:  	pieceBitboards[black][king] = 1152921504606846976; //2^60
063:
064:  	nextColour = white;
065:  	enPassantSquare = -1;
066:
067:  	canBlackCastleQueenSide = true;
068:  	canBlackCastleKingSide = true;
069:  	canWhiteCastleQueenSide = true;
070:  	canWhiteCastleKingSide = true;
071:
072:  	generateZorbistKey();
073:  	updateScoreValues();
074:  	update();
075:  }
076:
077:  void Board::printBoard()
078:  {
079:  	int counter = 56;
080:  	for (int x = 0; x < 8; x++)
081:  	{
082:  		for (int y = 0; y < 8; y++)
083:  		{
084:
085:  			uint64_t currentPosBitboard = (uint64_t)1 << counter;
086:
087:  			std::cout << "|";
088:
089:  			//Checking for white pieces
090:  			if ((pieceBitboards[white][pawn] & currentPosBitboard) != 0) //If the piece is a white pawn;
091:  			{
092:  				std::cout << "WP";
093:  			}
094:  			if ((pieceBitboards[white][rook] & currentPosBitboard) != 0) //If the piece is a white rook;
095:  			{
096:  				std::cout <<  "WR";
097:  			}
098:  			if ((pieceBitboards[white][knight] & currentPosBitboard) != 0) //If the piece is a white knight;
099:  			{
100:  				std::cout << "WN";
101:  			}
102:  			if ((pieceBitboards[white][bishop] & currentPosBitboard) != 0) //If the piece is a white bishop;
103:  			{
104:  				std::cout << "WB";
105:  			}
106:  			if ((pieceBitboards[white][queen] & currentPosBitboard) != 0) //If the piece is a white queen;
107:  			{
108:  				std::cout << "WQ";
109:  			}
110:  			if ((pieceBitboards[white][king] & currentPosBitboard) != 0) //If the piece is a white king;
```

```
111:                            {
112:                                    std::cout << "WK";
113:                            }
114:
115:                            //Checking for black pieces
116:                            if ((pieceBitboards[black][pawn] & currentPosBitboard) != 0) //If the piece is a black pawn;
117:                            {
118:                                    std::cout << "BP";
119:                            }
120:                            if ((pieceBitboards[black][rook] & currentPosBitboard) != 0) //If the piece is a black rook;
121:                            {
122:                                    std::cout << "BR";
123:                            }
124:                            if ((pieceBitboards[black][knight] & currentPosBitboard) != 0) //If the piece is a black knight;
125:                            {
126:                                    std::cout << "BN";
127:                            }
128:                            if ((pieceBitboards[black][bishop] & currentPosBitboard) != 0) //If the piece is a black bishop;
129:                            {
130:                                    std::cout << "BB";
131:                            }
132:                            if ((pieceBitboards[black][queen] & currentPosBitboard) != 0) //If the piece is a black queen;
133:                            {
134:                                    std::cout << "BQ";
135:                            }
136:                            if ((pieceBitboards[black][king] & currentPosBitboard) != 0) //If the piece is a black king;
137:                            {
138:                                    std::cout << "BK";
139:                            }
140:                            if ((allPieces & currentPosBitboard) == 0) //If the piece is empty
141:                            {
142:                                    std::cout << "  ";
143:                            }
144:                            counter++;
145:                    }
146:                    std::cout << "|\n";
147:                    counter -= 16;
148:            }
149:    }
150:
151:    void Board::update()
152:    {
153:            blackPieces = pieceBitboards[black][pawn] | pieceBitboards[black][rook] | pieceBitboards[black][knight] | pieceBitboards[black][bishop] | pieceBitboards[black][king] | pieceBitboards[black][queen];
154:            whitePieces = pieceBitboards[white][pawn] | pieceBitboards[white][rook] | pieceBitboards[white][knight] | pieceBitboards[white][bishop] | pieceBitboards[white][king] | pieceBitboards[white][queen];
155:
156:            allPieces = whitePieces | blackPieces;
157:    }
158:
159:    void Board::nextMove()
160:    {
161:            nextColour = switchColour(nextColour);
162:            kingDangerSquares = 0;
163:            moveHistory.push_back(zorbistKey);
164:            update();
165:    }
166:
167:    void Board::loadFromFen(std::string fen)
168:    {
169:            clearBoard();
170:
171:            int currentPosInBoard = 56;
172:            int currentCharPos = 0;
173:            char currentChar = 'X';
174:
175:            while (!isspace(currentChar))
176:            {
177:                    currentChar = fen[currentCharPos];
178:
179:                    if (currentChar == 'p')
180:                    {
181:                            pieceBitboards[black][pawn] |= (uint64_t)1 << currentPosInBoard;
182:                            currentPosInBoard++;
183:                    }
184:                    else if (currentChar == 'r')
185:                    {
186:                            pieceBitboards[black][rook] |= (uint64_t)1 << currentPosInBoard;
187:                            currentPosInBoard++;
188:                    }
189:                    else if (currentChar == 'n')
190:                    {
191:                            pieceBitboards[black][knight] |= (uint64_t)1 << currentPosInBoard;
192:                            currentPosInBoard++;
193:                    }
194:                    else if (currentChar == 'b')
195:                    {
196:                            pieceBitboards[black][bishop] |= (uint64_t)1 << currentPosInBoard;
197:                            currentPosInBoard++;
198:                    }
199:                    else if (currentChar == 'k')
200:                    {
201:                            pieceBitboards[black][king] |= (uint64_t)1 << currentPosInBoard;
202:                            currentPosInBoard++;
203:                    }
204:                    else if (currentChar == 'q')
205:                    {
206:                            pieceBitboards[black][queen] |= (uint64_t)1 << currentPosInBoard;
207:                            currentPosInBoard++;
208:                    }
209:                    else if (currentChar == 'P')
210:                    {
211:                            pieceBitboards[white][pawn] |= (uint64_t)1 << currentPosInBoard;
212:                            currentPosInBoard++;
213:                    }
214:                    else if (currentChar == 'R')
215:                    {
216:                            pieceBitboards[white][rook] |= (uint64_t)1 << currentPosInBoard;
217:                            currentPosInBoard++;
218:                    }
219:                    else if (currentChar == 'N')
220:                    {
221:                            pieceBitboards[white][knight] |= (uint64_t)1 << currentPosInBoard;
222:                            currentPosInBoard++;
223:                    }
224:                    else if (currentChar == 'B')
```

```
225:                    {
226:                            pieceBitboards[white][bishop] |= (uint64_t)1 << currentPosInBoard;
227:                            currentPosInBoard++;
228:                    }
229:                    else if (currentChar == 'K')
230:                    {
231:                            pieceBitboards[white][king] |= (uint64_t)1 << currentPosInBoard;
232:                            currentPosInBoard++;
233:                    }
234:                    else if (currentChar == 'Q')
235:                    {
236:                            pieceBitboards[white][queen] |= (uint64_t)1 << currentPosInBoard;
237:                            currentPosInBoard++;
238:                    }
239:                    else if (currentChar == '/')
240:                    {
241:                            currentPosInBoard -= 16;
242:                    }
243:                    else
244:                    {
245:                            currentPosInBoard += (currentChar - '0');
246:                    }
247:                    currentCharPos++;
248:            }
249:
250:            if (fen[currentCharPos] == 'w')
251:            {
252:                    nextColour = white;
253:            }
254:            else if (fen[currentCharPos] == 'b')
255:            {
256:                    nextColour = black;
257:            }
258:
259:            currentCharPos += 2;
260:            currentChar = fen[currentCharPos];
261:
262:            while (!isspace(currentChar))
263:            {
264:                    if (currentChar == 'K')
265:                    {
266:                            canWhiteCastleKingSide = true;
267:                    }
268:                    else if (currentChar == 'Q')
269:                    {
270:                            canWhiteCastleQueenSide = true;
271:                    }
272:                    else if (currentChar == 'k')
273:                    {
274:                            canBlackCastleKingSide = true;
275:                    }
276:                    else if (currentChar == 'q')
277:                    {
278:                            canBlackCastleQueenSide = true;
279:                    }
280:                    currentCharPos++;
281:                    currentChar = fen[currentCharPos];
282:            }
283:
284:
285:            currentCharPos++;
286:            currentChar = fen[currentCharPos];
287:            if (currentChar != '-')
288:            {
289:                    //En passant target square
290:                    int column = currentChar - 'a';
291:                    int row = fen[currentCharPos + 1] - '1';
292:                    int pos = row * 8 + column;
293:
294:                    enPassantSquare = pos;
295:            }
296:            else
297:            {
298:                    enPassantSquare = -1;
299:            }
300:
301:            currentCharPos += 2;
302:
303:            //Halfmove clock
304:            //Fullmove clock
305:
306:            updateScoreValues();
307:            generateZorbistKey();
308:            update();
309:
310:    }
311:
312:    //Outputs the state of the board in the fen format.
313:    //see this link for more details https://en.wikipedia.org/wiki/Forsyth%E2%80%93Edwards_Notation
314:    std::string Board::exportAsFen()
315:    {
316:            std::string fenString;
317:
318:            int emptySquaresCounter = 0;
319:
320:            //Outputs the position of all pieces.
321:            int counter = 56;
322:            while (counter >= 0)
323:            {
324:                    uint64_t currentPieceBitboard = (uint64_t)1 << counter;
325:                    pieceType currentSquarePiece = getPieceTypeInSquare(currentPieceBitboard);
326:
327:                    if (currentSquarePiece == blank)
328:                    {
329:                            emptySquaresCounter++;
330:                    }
331:                    else
332:                    {
333:                            if (emptySquaresCounter > 0)
334:                            {
335:                                    fenString += std::to_string(emptySquaresCounter);
336:                                    emptySquaresCounter = 0;
337:                            }
338:
```

```
339:
340:                             char pieceChar;
341:                             switch (currentSquarePiece)
342:                             {
343:                             case pawn:
344:                                     pieceChar = 'p';
345:                                     break;
346:                             case knight:
347:                                     pieceChar = 'n';
348:                                     break;
349:                             case bishop:
350:                                     pieceChar = 'b';
351:                                     break;
352:                             case rook:
353:                                     pieceChar = 'r';
354:                                     break;
355:                             case queen:
356:                                     pieceChar = 'q';
357:                                     break;
358:                             case king:
359:                                     pieceChar = 'k';
360:                                     break;
361:                             }
362:
363:                             //Converts to uppercase if the piece is white
364:                             if ((whitePieces & currentPieceBitboard) > 0) pieceChar = toupper(pieceChar);
365:
366:                             fenString += pieceChar;
367:
368:                     }
369:
370:                     counter++;
371:                     if (counter % 8 == 0)
372:                     {
373:                             if (emptySquaresCounter > 0)
374:                             {
375:                                     char emptySquareCounterChar = '0' + emptySquaresCounter;
376:                                     fenString += emptySquareCounterChar;
377:                                     emptySquaresCounter = 0;
378:                             }
379:                             if(counter != 8) fenString += "/";
380:                             counter -= 16;
381:                     }
382:             }
383:
384:             fenString += ' ';
385:
386:             //Next player to move
387:             if (nextColour == white) fenString += "w";
388:             else fenString += "b";
389:
390:             fenString += ' ';
391:
392:             //Castling Rights
393:             if (canWhiteCastleKingSide) fenString += "K";
394:             if (canWhiteCastleQueenSide) fenString += "Q";
395:             if (canBlackCastleKingSide) fenString += "k";
396:             if (canBlackCastleQueenSide) fenString += "q";
397:             if (!canWhiteCastleKingSide && !canWhiteCastleQueenSide && !canBlackCastleKingSide && !canBlackCastleQueenSide)
398:                     fenString += "-";
399:
400:             fenString += " ";
401:
402:             if (enPassantSquare != -1)
403:             {
404:                     std::string notation;
405:
406:                     int column = (enPassantSquare % 8);
407:                     int row = 1 + enPassantSquare / 8;
408:
409:                     char rowChar = row + '0';
410:                     char columnChar = column + 'a';
411:                     notation += columnChar;
412:                     notation += rowChar;
413:
414:                     fenString += notation;
415:             }
416:             else fenString += "-";
417:
418:             //Halfmove clock and fullmove number are not currently tracked by this class.
419:             fenString += " 0 0";
420:
421:             return fenString;
422:     }
423:
424:     uint64_t Board::getPieceBitboard(colours colour, pieceType piece)
425:     {
426:             return pieceBitboards[colour][piece];
427:     }
428:
429:     void Board::setBitboard(colours colour, pieceType piece, uint64_t bitboard)
430:     {
431:             pieceBitboards[colour][piece] = bitboard;
432:     }
433:
434:     void Board::removePiece(uint64_t bitboard)
435:     {
436:             colours colour;
437:             if (bitboard & whitePieces)
438:                     colour = white;
439:             else
440:                     colour = black;
441:
442:             for (int piece = 0; piece <= 6; piece++)
443:             {
444:                     if ((pieceBitboards[colour][piece] & bitboard) != 0)
445:                     {
446:                             pieceBitboards[colour][piece] = pieceBitboards[colour][piece] & ~bitboard;
447:                             return;
448:                     }
449:             }
450:     }
451:
452:     pieceType Board::getPieceTypeInSquare(uint64_t bitboard)
```

```
453:  {
454:  	for (int piece = 0; piece <= 6; piece++)
455:  		if (bitboard & (pieceBitboards[black][piece] | pieceBitboards[white][piece])) return (pieceType)piece;
456:  	return blank;
457:  }
458:  
459:  bool Board::isPieceAttacked(int piecePos, colours colour)
460:  {
461:  	uint64_t pieceBitboard = (uint64_t)1 << piecePos;
462:  
463:  	if (colour == white)
464:  	{
465:  		if ((pieceBitboard << 7) & pieceBitboards[black][pawn] & ~fileH || (pieceBitboard << 9) & pieceBitboards[black][pawn] & ~fileA)
466:  		{
467:  			return true;
468:  		}
469:  	}
470:  	else
471:  	{
472:  		if ((pieceBitboard >> 9) & pieceBitboards[white][pawn] & ~fileH || (pieceBitboard >> 7) & pieceBitboards[white][pawn] & ~fileA)
473:  		{
474:  			return true;
475:  		}
476:  	}
477:  
478:  	//KnightMoves
479:  	uint64_t knightMoves = knightMovesArray[piecePos];
480:  
481:  	if (colour == white)
482:  	{
483:  		if (knightMoves & pieceBitboards[black][knight])
484:  		{
485:  			return true;
486:  		}
487:  	}
488:  	else
489:  	{
490:  		if (knightMoves & pieceBitboards[white][knight])
491:  		{
492:  			return true;
493:  		}
494:  	}
495:  
496:  	uint64_t kingMoves = kingMovesArray[piecePos];
497:  
498:  	if (colour == white)
499:  	{
500:  		if (knightMoves & pieceBitboards[black][king])
501:  		{
502:  			return true;
503:  		}
504:  	}
505:  	else
506:  	{
507:  		if (knightMoves & pieceBitboards[white][king])
508:  		{
509:  			return true;
510:  		}
511:  	}
512:  
513:  	//Rook and half of queen moves
514:  	uint64_t occupancy = magicBitboards::rookMask[piecePos] & allPieces;
515:  	uint64_t magicResult = occupancy * magicBitboards::magicNumberRook[piecePos];
516:  	int arrayIndex = magicResult >> magicBitboards::magicNumberShiftRook[piecePos];
517:  	uint64_t magicMoves = magicBitboards::magicMovesRook[piecePos][arrayIndex];
518:  
519:  	if (colour == white)
520:  	{
521:  		if (magicMoves & (pieceBitboards[black][rook] | pieceBitboards[black][queen]))
522:  		{
523:  			return true;
524:  		}
525:  	}
526:  	else
527:  	{
528:  		if (magicMoves & (pieceBitboards[white][rook] | pieceBitboards[white][queen]))
529:  		{
530:  			return true;
531:  		}
532:  	}
533:  
534:  	//Bishop and half of queen moves
535:  	occupancy = magicBitboards::bishopMask[piecePos] & allPieces;
536:  	magicResult = occupancy * magicBitboards::magicNumberBishop[piecePos];
537:  	arrayIndex = magicResult >> magicBitboards::magicNumberShiftBishop[piecePos];
538:  	magicMoves = magicBitboards::magicMovesBishop[piecePos][arrayIndex];
539:  
540:  	if (colour == white)
541:  	{
542:  		if (magicMoves & (pieceBitboards[black][bishop] | pieceBitboards[black][queen]))
543:  		{
544:  			return true;
545:  		}
546:  	}
547:  	else
548:  	{
549:  		if (magicMoves & (pieceBitboards[white][bishop] | pieceBitboards[white][queen]))
550:  		{
551:  			return true;
552:  		}
553:  	}
554:  	return false;
555:  }
556:  
557:  void Board::generateZorbistKey()
558:  {
559:  	update();
560:  
561:  	uint64_t hash = 0;
562:  	uint64_t pawnHash = 0;
563:  
564:  	for (int x = 0; x < 64; x++)
565:  	{
566:  		uint64_t currentPosBitboard = (uint64_t)1 << x;
```

```
567:                    if (currentPosBitboard & allPieces) //If their is a piece at the square.
568:                    {
569:                            colours colour;
570:                            if (whitePieces & currentPosBitboard) colour = white;
571:                            else colour = black;
572:
573:                            for (int piece = 0; piece < 6; piece++)
574:                            {
575:                                    if ((pieceBitboards[colour][piece] & currentPosBitboard) > 0)
576:                                    {
577:                                            hash ^= ZorbistKeys::pieceKeys[x][colour * 6 + piece];
578:
579:                                            if (piece == pawn)
580:                                            {
581:                                                    pawnHash ^= ZorbistKeys::pieceKeys[x][colour * 6 + piece];
582:                                            }
583:                                    }
584:                            }
585:                    }
586:            }
587:            if (nextColour == black)
588:                    hash ^= ZorbistKeys::blackMoveKey;
589:            if(canBlackCastleQueenSide)
590:                    hash ^= ZorbistKeys::blackQueenSideCastlingKey;
591:            if (canBlackCastleKingSide)
592:                    hash ^= ZorbistKeys::blackKingSideCastlingKey;
593:            if (canWhiteCastleQueenSide)
594:                    hash ^= ZorbistKeys::whiteQueenSideCastlingKey;
595:            if (canWhiteCastleKingSide)
596:                    hash ^= ZorbistKeys::whiteKingSideCastlingKey;
597:            if (enPassantSquare != -1)
598:                    hash ^= ZorbistKeys::enPassantKeys[enPassantSquare % 8]; //Adds the hash for the column the of en passant square
599:
600:            zorbistKey = hash;
601:            pawnScoreZorbistKey = pawnHash;
602:    }
603:
604:    bool Board::isMaterialDraw()
605:    {
606:            const uint64_t bothKingsBitboard = getPieceBitboard(white, king) | getPieceBitboard(black, king);
607:
608:            //If their are only kings on the board , its a draw.
609:            if (bothKingsBitboard == allPieces)
610:            {
611:                    return true;
612:            }
613:            //If King + knight vs King , its a draw
614:            else if ((bothKingsBitboard | getPieceBitboard(white, knight)) == allPieces && bitSum(getPieceBitboard(white, knight)) == 1
615:                     || (bothKingsBitboard | getPieceBitboard(black, knight)) == allPieces && bitSum(getPieceBitboard(black, knight)) == 1)
616:            {
617:                    return true;
618:            }
619:            //If their are only kings and bishops , and the bishops are on the same , colour square , its a draw
620:            else if ((bothKingsBitboard | getPieceBitboard(white, bishop) | getPieceBitboard(black, bishop)) == allPieces)
621:            {
622:                    uint64_t bishopsBitboard = getPieceBitboard(white, bishop) | getPieceBitboard(black, bishop);
623:                    const colours firstColour = getPieceColour(bitScanForward(pop(bishopsBitboard)));
624:
625:                    while (bishopsBitboard)
626:                    {
627:                            if (firstColour != getPieceColour(bitScanForward(pop(bishopsBitboard))))
628:                            {
629:                                    return false;
630:                            }
631:                    }
632:                    return true;
633:            }
634:
635:            return false;
636:
637:    }
638:
639:    colours Board::getPieceColour(int pos)
640:    {
641:            const colours coloursArray[] = { black, white, black, white, black, white, black, white, white, black, white, black, white, black, white, black };
642:            return coloursArray[pos % 16];
643:    }
644:
645:    bool Board::isInCheck()
646:    {
647:            if (kingDangerSquares == 0)         generateKingDangerSquares();
648:            return kingDangerSquares & getPieceBitboard(nextColour, king);
649:    }
650:
651:    uint64_t Board::getKingDangerSquares()
652:    {
653:            if (kingDangerSquares == 0)         generateKingDangerSquares();
654:            return kingDangerSquares;
655:    }
656:
657:    void Board::generateKingDangerSquares()
658:    {
659:            const uint64_t allPiecesExceptKing = allPieces & ~getPieceBitboard(nextColour, king);
660:            const colours oppositeColour = switchColour(nextColour);
661:
662:            uint64_t attackSet = 0;
663:            uint64_t currentPos;
664:
665:            uint64_t pawnBitboard = getPieceBitboard(oppositeColour, pawn);
666:            if (oppositeColour == white)
667:            {
668:                    while (pawnBitboard)
669:                    {
670:                            currentPos = pop(pawnBitboard);
671:                            attackSet |= currentPos << 7 & ~fileH | currentPos << 9 & ~fileA;
672:                    }
673:            }
674:            else
675:            {
676:                    while (pawnBitboard)
677:                    {
678:                            currentPos = pop(pawnBitboard);
679:                            attackSet |= (currentPos >> 9) & ~fileH | (currentPos >> 7) & ~fileA;
680:                    }
```

```
681:            }
682:
683:            uint64_t knightBitboard = getPieceBitboard(oppositeColour, knight);
684:            while (knightBitboard)
685:            {
686:                    currentPos = pop(knightBitboard);
687:                    attackSet |= knightMovesArray[bitScanForward(currentPos)];
688:            }
689:
690:            uint64_t kingBitboard = getPieceBitboard(oppositeColour, king);
691:            while (kingBitboard)
692:            {
693:                    currentPos = pop(kingBitboard);
694:                    attackSet |= kingMovesArray[bitScanForward(currentPos)];
695:            }
696:
697:            uint64_t rookBitboard = getPieceBitboard(oppositeColour, rook) | getPieceBitboard(oppositeColour, queen);
698:            while (rookBitboard)
699:            {
700:                    currentPos = pop(rookBitboard);
701:                    int piecePos = bitScanForward(currentPos);
702:
703:                    uint64_t occupancy = magicBitboards::rookMask[piecePos] & allPiecesExceptKing;
704:                    uint64_t magicResult = occupancy * magicBitboards::magicNumberRook[piecePos];
705:                    int arrayIndex = magicResult >> magicBitboards::magicNumberShiftRook[piecePos];
706:                    attackSet |= magicBitboards::magicMovesRook[piecePos][arrayIndex];
707:            }
708:
709:            uint64_t bishopBitboard = getPieceBitboard(oppositeColour, bishop) | getPieceBitboard(oppositeColour, queen);
710:            while (bishopBitboard)
711:            {
712:                    currentPos = pop(bishopBitboard);
713:                    int piecePos = bitScanForward(currentPos);
714:
715:                    uint64_t occupancy = magicBitboards::bishopMask[piecePos] & allPiecesExceptKing;
716:                    uint64_t magicResult = occupancy * magicBitboards::magicNumberBishop[piecePos];
717:                    int arrayIndex = magicResult >> magicBitboards::magicNumberShiftBishop[piecePos];
718:                    attackSet |= magicBitboards::magicMovesBishop[piecePos][arrayIndex];
719:            }
720:
721:            kingDangerSquares = attackSet;
722:    }
723:
724:    int Board::getMaterialScore(colours colour)
725:    {
726:            if (colour == white) return whitePawnScore + whiteMaterialScore - blackPawnScore - blackMaterialScore;
727:            else return blackPawnScore + blackMaterialScore - whitePawnScore - whiteMaterialScore;
728:    }
729:
730:    int Board::getOnlyMaterialScore(colours colour)
731:    {
732:            if (colour == white) return whiteMaterialScore;
733:            else return blackMaterialScore;
734:    }
735:
736:    int Board::getPositionalScore(colours colour)
737:    {
738:            if (colour == white) return whitePositionalScore - blackPositionalScore;
739:            else return whitePositionalScore - blackPositionalScore;
740:    }
741:
742:    int Board::getMidGameKingPositionalScore(colours colour)
743:    {
744:            if (colour == white) return whiteMidGameKingPositionalScore;
745:            else return blackMidGameKingPositionalScore;
746:    }
747:
748:    int Board::getLateGameKingPositionalScore(colours colour)
749:    {
750:            if (colour == white) return whiteLateGameKingPositionalScore;
751:            else return blackLateGameKingPositionalScore;
752:    }
753:
754:    void Board::updateScoreValues()
755:    {
756:            whitePawnScore = bitSum(getPieceBitboard(white, pawn)) * 100;
757:
758:            whiteMaterialScore = 0;
759:            whiteMaterialScore += bitSum(getPieceBitboard(white, knight)) * 300;
760:            whiteMaterialScore += bitSum(getPieceBitboard(white, bishop)) * 300;
761:            whiteMaterialScore += bitSum(getPieceBitboard(white, rook)) * 500;
762:            whiteMaterialScore += bitSum(getPieceBitboard(white, queen)) * 900;
763:
764:            blackPawnScore = bitSum(getPieceBitboard(black, pawn)) * 100;
765:
766:            blackMaterialScore = 0;
767:            blackMaterialScore += bitSum(getPieceBitboard(black, knight)) * 300;
768:            blackMaterialScore += bitSum(getPieceBitboard(black, bishop)) * 300;
769:            blackMaterialScore += bitSum(getPieceBitboard(black, rook)) * 500;
770:            blackMaterialScore += bitSum(getPieceBitboard(black, queen)) * 900;
771:
772:            whitePositionalScore = 0;
773:            whitePositionalScore += pieceSquareData::pawnSquare.calcScore(getPieceBitboard(white, pawn), white);
774:            whitePositionalScore += pieceSquareData::knightSquare.calcScore(getPieceBitboard(white, knight), white);
775:            whitePositionalScore += pieceSquareData::bishopSquare.calcScore(getPieceBitboard(white, bishop), white);
776:            whiteMidGameKingPositionalScore = pieceSquareData::midGameKingSquare.calcScore(getPieceBitboard(white, king), white);
777:            whiteLateGameKingPositionalScore = pieceSquareData::lateGameKingSquare.calcScore(getPieceBitboard(white, king), white);
778:
779:            blackPositionalScore = 0;
780:            blackPositionalScore += pieceSquareData::pawnSquare.calcScore(getPieceBitboard(black, pawn), black);
781:            blackPositionalScore += pieceSquareData::knightSquare.calcScore(getPieceBitboard(black, knight), black);
782:            blackPositionalScore += pieceSquareData::bishopSquare.calcScore(getPieceBitboard(black, bishop), black);
783:            blackMidGameKingPositionalScore = pieceSquareData::midGameKingSquare.calcScore(getPieceBitboard(black, king), black);
784:            blackLateGameKingPositionalScore = pieceSquareData::lateGameKingSquare.calcScore(getPieceBitboard(black, king), black);
785:    }
786:
787:    void Board::addMaterialScore(colours colour, pieceType piece)
788:    {
789:            const int materialValues[6] = { 100,300,300,500,900,0 };
790:            if (colour == white)
791:            {
792:                    if (piece == pawn)
793:                    {
794:                            whitePawnScore += materialValues[pawn];
```

```
795:                    }
796:                    else
797:                    {
798:                            whiteMaterialScore += materialValues[piece];
799:                    }
800:
801:            }
802:            else
803:            {
804:                    if (piece == pawn)
805:                    {
806:                            blackPawnScore += materialValues[pawn];
807:                    }
808:                    else
809:                    {
810:                            blackMaterialScore += materialValues[piece];
811:                    }
812:            }
813:    }
814:
815:    void Board::removeMaterialScore(colours colour, pieceType piece)
816:    {
817:            const int materialValues[6] = { 100,300,300,500,900,0 };
818:            if (colour == white)
819:            {
820:                    if (piece == pawn)
821:                    {
822:                            whitePawnScore -= materialValues[pawn];
823:                    }
824:                    else
825:                    {
826:                            whiteMaterialScore -= materialValues[piece];
827:                    }
828:            }
829:            else
830:            {
831:                    if (piece == pawn)
832:                    {
833:                            blackPawnScore -= materialValues[pawn];
834:                    }
835:                    else
836:                    {
837:                            blackMaterialScore -= materialValues[piece];
838:                    }
839:            }
840:    }
841:
842:    void Board::addPositionalScore(colours colour, pieceType piece, int piecePos)
843:    {
844:            if (colour == white)
845:            {
846:                    if (piece == pawn)
847:                    {
848:                            whitePositionalScore += pieceSquareData::pawnSquare.getScoreFromPos(piecePos, white);
849:                    }
850:                    else if (piece == knight)
851:                    {
852:                            whitePositionalScore += pieceSquareData::knightSquare.getScoreFromPos(piecePos, white);
853:                    }
854:                    else if (piece == bishop)
855:                    {
856:                            whitePositionalScore += pieceSquareData::bishopSquare.getScoreFromPos(piecePos, white);
857:                    }
858:                    else if (piece == king)
859:                    {
860:                            whiteMidGameKingPositionalScore += pieceSquareData::midGameKingSquare.getScoreFromPos(piecePos, white);
861:                            whiteLateGameKingPositionalScore += pieceSquareData::lateGameKingSquare.getScoreFromPos(piecePos, white);
862:                    }
863:            }
864:            else
865:            {
866:                    if (piece == pawn)
867:                    {
868:                            blackPositionalScore += pieceSquareData::pawnSquare.getScoreFromPos(piecePos, black);
869:                    }
870:                    else if (piece == knight)
871:                    {
872:                            blackPositionalScore += pieceSquareData::knightSquare.getScoreFromPos(piecePos, black);
873:                    }
874:                    else if (piece == bishop)
875:                    {
876:                            blackPositionalScore += pieceSquareData::bishopSquare.getScoreFromPos(piecePos, black);
877:                    }
878:                    else if (piece == king)
879:                    {
880:                            blackMidGameKingPositionalScore += pieceSquareData::midGameKingSquare.getScoreFromPos(piecePos, black);
881:                            blackLateGameKingPositionalScore += pieceSquareData::lateGameKingSquare.getScoreFromPos(piecePos, black);
882:                    }
883:            }
884:    }
885:
886:    void Board::removePositionalScore(colours colour, pieceType piece, int piecePos)
887:    {
888:            if (colour == white)
889:            {
890:                    if (piece == pawn)
891:                    {
892:                            whitePositionalScore -= pieceSquareData::pawnSquare.getScoreFromPos(piecePos, white);
893:                    }
894:                    else if (piece == knight)
895:                    {
896:                            whitePositionalScore -= pieceSquareData::knightSquare.getScoreFromPos(piecePos, white);
897:                    }
898:                    else if (piece == bishop)
899:                    {
900:                            whitePositionalScore -= pieceSquareData::bishopSquare.getScoreFromPos(piecePos, white);
901:                    }
902:                    else if (piece == king)
903:                    {
904:                            whiteMidGameKingPositionalScore -= pieceSquareData::midGameKingSquare.getScoreFromPos(piecePos, white);
905:                            whiteLateGameKingPositionalScore -= pieceSquareData::lateGameKingSquare.getScoreFromPos(piecePos, white);
906:                    }
907:            }
908:            else
```

```
909:            {
910:                    if (piece == pawn)
911:                    {
912:                            blackPositionalScore -= pieceSquareData::pawnSquare.getScoreFromPos(piecePos, black);
913:                    }
914:                    else if (piece == knight)
915:                    {
916:                            blackPositionalScore -= pieceSquareData::knightSquare.getScoreFromPos(piecePos, black);
917:                    }
918:                    else if (piece == bishop)
919:                    {
920:                            blackPositionalScore -= pieceSquareData::bishopSquare.getScoreFromPos(piecePos, black);
921:                    }
922:                    else if (piece == king)
923:                    {
924:                            blackMidGameKingPositionalScore -= pieceSquareData::midGameKingSquare.getScoreFromPos(piecePos, black);
925:                            blackLateGameKingPositionalScore -= pieceSquareData::lateGameKingSquare.getScoreFromPos(piecePos, black);
926:                    }
927:            }
928:    }
929:
930:
931:
```

ChessEngineLibrary\board.h

```
001:  #pragma once
002:  #include <stdint.h>
003:  #include <cmath>
004:  #include <iostream>
005:  #include <vector>
006:
007:  #include "bitboard.h"
008:  #include "magicBitboards.h"
009:  #include "piece.h"
010:  #include "transpositionTable.h"
011:  #include "moveGenerationTables.h"
012:  #include "pieceSquare.h"
013:
014:  class Board
015:  {
016:  public:
017:          Board();
018:          Board(std::string fenString);
019:
020:          void clearBoard();
021:          void defaults();
022:          void printBoard();
023:          void update();
024:          void nextMove();
025:
026:          void loadFromFen(std::string fen);
027:          std::string exportAsFen();
028:
029:          uint64_t getPieceBitboard(colours colour , pieceType piece);
030:          void setBitboard(colours colour, pieceType piece, uint64_t bitboard);
031:          //returns the value of the piece lost.
032:          void removePiece(uint64_t bitboard);
033:          pieceType getPieceTypeInSquare(uint64_t bitboard);
034:          bool isPieceAttacked(int piecePos, colours colour);
035:
036:          void generateZorbistKey();
037:
038:          //Checks whether the game is drawn by lack of material
039:          bool isMaterialDraw();
040:
041:          //Gets the colour of a piece on the board
042:          colours  getPieceColour(int pos);
043:
044:          //Used for detecting three-fold repitition.
045:          std::vector<uint64_t> moveHistory;
046:
047:          int enPassantSquare;
048:          colours nextColour;
049:
050:          bool canBlackCastleQueenSide;
051:          bool canBlackCastleKingSide;
052:          bool canWhiteCastleQueenSide;
053:          bool canWhiteCastleKingSide;
054:
055:          //Used for for indexing the main transposition table.
056:          uint64_t zorbistKey;
057:
058:          //Used for indexing pawn structure scores hash table.
059:          uint64_t pawnScoreZorbistKey;
060:
061:          uint64_t whitePieces;
062:          uint64_t blackPieces;
063:          uint64_t allPieces;
064:
065:
066:          uint64_t kingDangerSquares;
067:
068:          bool isInCheck();
069:          uint64_t getKingDangerSquares();
070:          void generateKingDangerSquares();
071:
072:          //returns the total material score.
073:          int getMaterialScore(colours colour);
074:          //returns the material score (no pawns) , for only one colour.
075:          int getOnlyMaterialScore(colours colour);
076:
077:          //returns total positional score (not including kings)
078:          int getPositionalScore(colours colour);
079:          //returns the current positional score for the king (mid game)
080:          int getMidGameKingPositionalScore(colours colour);
081:          //returns the current positional score for the king (late game)
082:          int getLateGameKingPositionalScore(colours colour);
083:
084:          //updates the values for material and positional scores.
085:          //Only run once , values are updated incrementally from then on.
086:          void updateScoreValues();
087:
088:          //Adds the material value of a piece.
089:          void addMaterialScore(colours colour, pieceType piece);
090:          //Removes the material value of the piece .
091:          void removeMaterialScore(colours colour, pieceType piece);
092:
093:          //Adds the material value of a piece.
094:          void addPositionalScore(colours colour, pieceType piece, int piecePos);
095:          //Removes the material value of the piece .
096:          void removePositionalScore(colours colour, pieceType piece, int piecePos);
097:
098:          //Material score not including pawns. (used for phase changed in scoring.cpp)
099:          int whiteMaterialScore;
100:          int blackMaterialScore;
101:
102:          //Material score of pawns
103:          int whitePawnScore;
104:          int blackPawnScore;
105:
106:          //Positional scores based on piece-square tables.
107:          int whitePositionalScore;
108:          int blackPositionalScore;
109:
110:          //Positional scores of kings. (usage changes based on game phase)
```

```
111:            int whiteMidGameKingPositionalScore;
112:            int blackMidGameKingPositionalScore;
113:            int whiteLateGameKingPositionalScore;
114:            int blackLateGameKingPositionalScore;
115:
116:    private:
117:            uint64_t pieceBitboards[2][6];
118:    };
```

ChessEngineLibrary\magicBitboards.cpp

```
001:  #include "magicBitboards.h"
002:
003:  void magicBitboards::generateMagicMovesRook()
004:  {
005:  	for (int square = 0; square < 64; square++)
006:  	{
007:  		uint64_t mask = rookMask[square];
008:  		std::vector<int> setBitsInMask = getSetBits(mask);
009:  		int variationCount = 1 << bitSum(mask);
010:  		std::vector<uint64_t> variations;
011:
012:  		for (int i = 0; i < variationCount; i++) //Generation Occupancy Variations
013:  		{
014:  			uint64_t variation = 0;
015:  			std::vector<int> setBitsInIndex = getSetBits(i);
016:  			for (int j = 0; j < setBitsInIndex.size(); j++)
017:  			{
018:  				variation |= ((uint64_t)1 << setBitsInMask[setBitsInIndex[j]]);
019:  			}
020:
021:  			uint64_t possibleMoves = 0;
022:  			for (int x = square + 8; x <= 63; x += 8) //Move up
023:  			{
024:  				possibleMoves |= (uint64_t)1 << x;
025:  				if ((variation & (uint64_t)1 << x) > 0) //If Their is a blocker on this square;
026:  				{
027:  					break;
028:  				}
029:  			}
030:  			for (int x = square - 8; x >= 0; x -= 8) //Move down
031:  			{
032:  				possibleMoves |= (uint64_t)1 << x;
033:  				if ((variation & (uint64_t)1 << x) > 0) //If Their is a blocker on this square;
034:  				{
035:  					break;
036:  				}
037:  			}
038:  			for (int x = square -1; (x % 8) != 7 && x >= 0; x--) //Move left
039:  			{
040:  				possibleMoves |= (uint64_t)1 << x;
041:  				if ((variation & (uint64_t)1 << x) > 0) //If Their is a blocker on this square;
042:  				{
043:  					break;
044:  				}
045:  			}
046:  			for (int x = square + 1; x % 8 != 0; x++) //Move right
047:  			{
048:  				possibleMoves |= (uint64_t)1 << x;
049:  				if ((variation & (uint64_t)1 << x) > 0) //If Their is a blocker on this square;
050:  				{
051:  					break;
052:  				}
053:  			}
054:
055:  			int magicIndex = (int)((uint64_t)(variation * magicNumberRook[square]) >> magicNumberShiftRook[square]);
056:
057:  			magicMovesRook[square][magicIndex] = possibleMoves;
058:
059:  		}
060:  	}
061:  }
062:
063:  void magicBitboards::generateMagicMovesBishop()
064:  {
065:  	for (int square = 0; square < 64; square++)
066:  	{
067:  		uint64_t mask = bishopMask[square];
068:  		std::vector<int> setBitsInMask = getSetBits(mask);
069:  		int variationCount = 1 << bitSum(mask);
070:  		std::vector<uint64_t> variations;
071:
072:  		for (int i = 0; i < variationCount; i++) //Generation Occupancy Variations
073:  		{
074:  			uint64_t variation = 0;
075:  			std::vector<int> setBitsInIndex = getSetBits(i);
076:  			for (int j = 0; j < setBitsInIndex.size(); j++)
077:  			{
078:  				variation |= ((uint64_t)1 << setBitsInMask[setBitsInIndex[j]]);
079:  			}
080:
081:  			uint64_t possibleMoves = 0;
082:  			for (int x = square + 9; x % 8 != 0 && x <= 63; x += 9) //Move Top-Right
083:  			{
084:  				possibleMoves |= (uint64_t)1 << x;
085:  				if ((variation & (uint64_t)1 << x) > 0) //If Their is a blocker on this square;
086:  				{
087:  					break;
088:  				}
089:  			}
090:  			for (int x = square - 9; x % 8 != 7 && x >= 0; x -= 9) //Move Bottom-Left
091:  			{
092:  				possibleMoves |= (uint64_t)1 << x;
093:  				if ((variation & (uint64_t)1 << x) > 0) //If Their is a blocker on this square;
094:  				{
095:  					break;
096:  				}
097:  			}
098:  			for (int x = square +7 ; x % 8 != 7 && x <= 63; x += 7) //Move Top-Left
099:  			{
100:  				possibleMoves |= (uint64_t)1 << x;
101:  				if ((variation & (uint64_t)1 << x) > 0) //If Their is a blocker on this square;
102:  				{
103:  					break;
104:  				}
105:  			}
106:  			for (int x = square - 7; x % 8 != 0 && x >=0; x -= 7) //Move Bottom-Right
107:  			{
108:  				possibleMoves |= (uint64_t)1 << x;
109:  				if ((variation & (uint64_t)1 << x) > 0) //If Their is a blocker on this square;
110:  				{
```

```
                                        break;
                                }
                        }

                        int magicIndex = (int)((uint64_t)(variation * magicNumberBishop[square]) >> magicNumberShiftBishop[square]);

                        magicMovesBishop[square][magicIndex] = possibleMoves;

                }
        }
}

void magicBitboards::setupMagicBitboards()
{
        generateMagicMovesRook();
        generateMagicMovesBishop();
}

std::array<uint64_t, 64> magicBitboards::rookMask = { 0x101010101017e, 0x202020202027c, 0x404040404047a, 0x8080808080876, 0x1010101010106e, 0x2020202020205e, 0x4040404040403e, 0x8080808080807e, 0x1010
std::array<uint64_t, 64> magicBitboards::bishopMask = { 0x40201008040200, 0x402010080400, 0x4020100a00, 0x40221400, 0x2442800, 0x204085000, 0x20408102000, 0x2040810204000, 0x20100804020000, 0x402010080

std::array<int, 64> magicBitboards::magicNumberShiftBishop = { 58,59,59,59,59,59,59,58,59,59,59,59,59,59,59,59,59,57,57,57,57,59,59,59,59,57,55,55,57,59,59,59,59,57,55,55,57,59,59,59,59,57,57,57,59,59,59,59,59,5
std::array<int, 64> magicBitboards::magicNumberShiftRook = { 52,53,53,53,53,53,53,52,53,54,54,54,54,54,54,53,53,54,54,54,54,54,54,53,53,54,54,54,54,54,54,53,53,54,54,54,54,54,54,53,53,54,54,54,54,54,54,53,53,54,54,54,

std::array<uint64_t, 64> magicBitboards::magicNumberBishop = { 0x2910054208004104, 0x2100630a7020180, 0x5822022042000000, 0x2ca804a100200020, 0x204042200000900, 0x2002121024000002, 0x80404104202000e
std::array<uint64_t, 64> magicBitboards::magicNumberRook = { 0xa180022080400230, 0x40100040022000, 0x80088020001002, 0x80080280841000, 0x4200042010460008, 0x4800a0003040080, 0x400110082041008, 0x800

std::array<std::unordered_map<int, uint64_t>, 64> magicBitboards::magicMovesRook;
std::array<std::unordered_map<int, uint64_t>, 64> magicBitboards::magicMovesBishop;
```

ChessEngineLibrary\magicBitboards.h

```
001:    #pragma once
002:    #include <stdint.h>
003:    #include <array>
004:    #include <vector>
005:    #include <unordered_map>
006:
007:    #include "bitboard.h"
008:
009:    class magicBitboards
010:    {
011:    public:
012:            static std::array<uint64_t, 64> bishopMask;
013:            static std::array<uint64_t, 64> rookMask;
014:
015:            static std::array<int, 64> magicNumberShiftBishop;
016:            static std::array<int, 64> magicNumberShiftRook;
017:
018:            static std::array<uint64_t, 64> magicNumberRook;
019:            static std::array<uint64_t, 64> magicNumberBishop;
020:
021:            static std::array<std::unordered_map<int, uint64_t>, 64> magicMovesRook;
022:            static std::array<std::unordered_map<int, uint64_t>, 64> magicMovesBishop;
023:
024:            static void setupMagicBitboards();
025:
026:    private:
027:            static void generateMagicMovesRook();
028:            static void generateMagicMovesBishop();
029:    };
030:
031:
```

ChessEngineLibrary\move.cpp

```
001:  #include "move.h"
002:
003:  Move::Move()
004:  {
005:  }
006:
007:  Move::Move(int newFrom, int newTo, MoveType newMoveType, pieceType newPieceType, Board* board)
008:  {
009:  	moveType = newMoveType;
010:  	to = newTo;
011:  	from = newFrom;
012:  	piece = newPieceType;
013:  	moveRating = 0;
014:
015:  	if (board->allPieces & (uint64_t)1 << to)
016:  	{
017:  		capturedPiece = board->getPieceTypeInSquare((uint64_t)1 << to);
018:  	}
019:  	//En passant capture
020:  	else if (to == board->enPassantSquare && piece == pawn)
021:  	{
022:  		capturedPiece = pawn;
023:  	}
024:  	else
025:  	{
026:  		capturedPiece = blank;
027:  	}
028:
029:  	canBlackCastleQueenSide = board->canBlackCastleQueenSide;
030:  	canBlackCastleKingSide = board->canBlackCastleKingSide;
031:  	canWhiteCastleQueenSide = board->canWhiteCastleQueenSide;
032:  	canWhiteCastleKingSide = board->canWhiteCastleKingSide;
033:  	enPassantSquare = board->enPassantSquare;
034:  	hash = board->zorbistKey;
035:  	pawnHash = board->pawnScoreZorbistKey;
036:  }
037:
038:  void Move::applyMove(Board * board)
039:  {
040:  	colours opponentColour = switchColour(board->nextColour);
041:
042:  	updateCastlingRights(board, this);
043:  	updateZorbistKeys(board, opponentColour);
044:
045:  	board->removePositionalScore(board->nextColour, piece, from);
046:
047:  	if (moveType != capture) board->enPassantSquare = -1;
048:
049:  	//Updates materialScore for removed pieces
050:  	if (capturedPiece != blank)
051:  	{
052:  		board->removeMaterialScore(opponentColour, capturedPiece);
053:  		if (piece != pawn || to != board->enPassantSquare) board->removePositionalScore(opponentColour, capturedPiece, to);
054:  	}
055:
056:  	switch (moveType)
057:  	{
058:  	case quietMove:
059:  	{
060:  		//Moves the piece
061:  		uint64_t bitboard = board->getPieceBitboard(board->nextColour, piece);
062:  		bitboard = (bitboard & ~((uint64_t)1 << from)) | ((uint64_t)1 << to);
063:  		board->setBitboard(board->nextColour, piece, bitboard);
064:
065:  		board->addPositionalScore(board->nextColour, piece, to);
066:  	}
067:  	break;
068:  	case capture:
069:  	{
070:  		if (board->enPassantSquare == to && piece == pawn)
071:  		{
072:  			//Removes the captued piece under en passent
073:  			if (board->nextColour == white)
074:  			{
075:  				board->removePiece((uint64_t)1 << (to - 8));
076:  				board->removePositionalScore(opponentColour, pawn, to - 8);
077:  			}
078:  			else
079:  			{
080:  				board->removePiece((uint64_t)1 << (to + 8));
081:  				board->removePositionalScore(opponentColour, pawn, to + 8);
082:  			}
083:  		}
084:  		else
085:  		{
086:  			//Removes the captued piece
087:  			board->removePiece((uint64_t)1 << to);
088:  		}
089:
090:  		//Moves the piece
091:  		uint64_t bitboard = board->getPieceBitboard(board->nextColour, piece);
092:  		bitboard = (bitboard & ~((uint64_t)1 << from)) | ((uint64_t)1 << to);
093:  		board->setBitboard(board->nextColour, piece, bitboard);
094:  		board->enPassantSquare = -1;
095:
096:  		board->addPositionalScore(board->nextColour, piece, to);
097:  	}
098:  	break;
099:  	case knightPromotion:
100:  	{
101:  		//Removes the captued piece
102:  		board->removePiece((uint64_t)1 << to);
103:
104:  		//Removes the moved Piece
105:  		board->removePiece((uint64_t)1 << from);
106:
107:  		//Creates the promoted piece
108:  		uint64_t bitboard = board->getPieceBitboard(board->nextColour, knight);
109:  		bitboard |= ((uint64_t)1 << to);
110:  		board->setBitboard(board->nextColour, knight, bitboard);
```

```
111:
112:                board->removeMaterialScore(board->nextColour, pawn);
113:                board->addMaterialScore(board->nextColour, knight);
114:                board->addPositionalScore(board->nextColour, knight, to);
115:        }
116:        break;
117:        case bishopPromotion:
118:        {
119:                //Removes the captued piece
120:                board->removePiece((uint64_t)1 << to);
121:
122:                //Removes the moved Piece
123:                board->removePiece((uint64_t)1 << from);
124:
125:                //Creates the promoted piece
126:                uint64_t bitboard = board->getPieceBitboard(board->nextColour, bishop);
127:                bitboard |= ((uint64_t)1 << to);
128:                board->setBitboard(board->nextColour, bishop, bitboard);
129:
130:                board->removeMaterialScore(board->nextColour, pawn);
131:                board->addMaterialScore(board->nextColour, bishop);
132:                board->addPositionalScore(board->nextColour, bishop, to);
133:        }
134:        break;
135:        case rookPromotion:
136:        {
137:                //Removes the captued piece
138:                board->removePiece((uint64_t)1 << to);
139:
140:                //Removes the moved Piece
141:                board->removePiece((uint64_t)1 << from);
142:
143:                //Creates the promoted piece
144:                uint64_t bitboard = board->getPieceBitboard(board->nextColour, rook);
145:                bitboard |= ((uint64_t)1 << to);
146:                board->setBitboard(board->nextColour, rook, bitboard);
147:
148:                board->removeMaterialScore(board->nextColour, pawn);
149:                board->addMaterialScore(board->nextColour, rook);
150:                board->addPositionalScore(board->nextColour, rook, to);
151:        }
152:        break;
153:        case queenPromotion:
154:        {
155:                //Removes the captued piece
156:                board->removePiece((uint64_t)1 << to);
157:
158:                //Removes the moved Piece
159:                board->removePiece((uint64_t)1 << from);
160:
161:                //Creates the promoted piece
162:                uint64_t bitboard = board->getPieceBitboard(board->nextColour, queen);
163:                bitboard |= ((uint64_t)1 << to);
164:                board->setBitboard(board->nextColour, queen, bitboard);
165:
166:                board->removeMaterialScore(board->nextColour, pawn);
167:                board->addMaterialScore(board->nextColour, queen);
168:                board->addPositionalScore(board->nextColour, queen, to);
169:        }
170:        break;
171:        case pawnDoubleMove:
172:        {
173:                //Moves the piece
174:                uint64_t bitboard = board->getPieceBitboard(board->nextColour, piece);
175:                bitboard = (bitboard & ~((uint64_t)1 << from)) | ((uint64_t)1 << to);
176:                board->setBitboard(board->nextColour, piece, bitboard);
177:
178:                //Sets En passant target square and hash.
179:                if (board->nextColour == white)
180:                {
181:                        board->enPassantSquare = to - 8;
182:                }
183:                else
184:                {
185:                        board->enPassantSquare = to + 8;
186:                }
187:
188:                board->addPositionalScore(board->nextColour, piece, to);
189:        }
190:        break;
191:        case kingSideCastling:
192:        {
193:                //Moves the king
194:                uint64_t bitboard = board->getPieceBitboard(board->nextColour, piece);
195:                bitboard = (bitboard & ~((uint64_t)1 << from)) | ((uint64_t)1 << to);
196:                board->setBitboard(board->nextColour, piece, bitboard);
197:                if (board->nextColour == white)
198:                {
199:                        //Moves the rook
200:                        board->setBitboard(white, rook ,(board->getPieceBitboard(white, rook) & ~128) | 32);
201:
202:                        board->removePositionalScore(white, rook, 7);
203:                        board->addPositionalScore(white, rook, 5);
204:                }
205:                else
206:                {
207:                        //Moves the rook
208:                        board->setBitboard(black, rook, (board->getPieceBitboard(black, rook) & ~9223372036854775808) | 2305843009213693952);
209:
210:                        board->removePositionalScore(white, rook, 63);
211:                        board->addPositionalScore(white, rook, 61);
212:                }
213:
214:                board->addPositionalScore(board->nextColour, piece, to);
215:        }
216:        break;
217:        case queenSideCastling:
218:        {
219:                //Moves the king
220:                uint64_t bitboard = board->getPieceBitboard(board->nextColour, piece);
221:                bitboard = (bitboard & ~((uint64_t)1 << from)) | ((uint64_t)1 << to);
222:                board->setBitboard(board->nextColour, piece, bitboard);
223:                if (board->nextColour == white)
224:                {
```

```
225:                            //Moves the rook
226:                            board->setBitboard(white, rook, (board->getPieceBitboard(white, rook) & ~1) | 8);
227:
228:                            board->removePositionalScore(white, rook, 0);
229:                            board->addPositionalScore(white, rook, 3);
230:                    }
231:                    else
232:                    {
233:                            //Moves the rook
234:                            board->setBitboard(black, rook, (board->getPieceBitboard(black, rook) & ~72057594037927936) | 576460752303423488);
235:
236:                            board->removePositionalScore(white, rook, 56);
237:                            board->addPositionalScore(white, rook, 59);
238:                    }
239:
240:                    board->addPositionalScore(board->nextColour, piece, to);
241:            }
242:            break;
243:            }
244:            board->nextMove();
245:    }
246:
247:    //Updates castling rights and the zorbist hash keys for castling rights.
248:    void updateCastlingRights(Board * newBoard, Move * move)
249:    {
250:            if (move->piece == king)
251:            {
252:                    if (newBoard->nextColour == white)
253:                    {
254:                            if(newBoard->canWhiteCastleKingSide)
255:                            {
256:                                    newBoard->canWhiteCastleKingSide = false;
257:                                    newBoard->zorbistKey ^= ZorbistKeys::whiteKingSideCastlingKey;
258:                            }
259:                            if (newBoard->canWhiteCastleQueenSide)
260:                            {
261:                                    newBoard->canWhiteCastleQueenSide = false;
262:                                    newBoard->zorbistKey ^= ZorbistKeys::whiteQueenSideCastlingKey;
263:                            }
264:                    }
265:                    else
266:                    {
267:                            if (newBoard->canBlackCastleKingSide)
268:                            {
269:                                    newBoard->canBlackCastleKingSide = false;
270:                                    newBoard->zorbistKey ^= ZorbistKeys::blackKingSideCastlingKey;
271:                            }
272:                            if (newBoard->canBlackCastleQueenSide)
273:                            {
274:                                    newBoard->canBlackCastleQueenSide = false;
275:                                    newBoard->zorbistKey ^= ZorbistKeys::blackQueenSideCastlingKey;
276:                            }
277:                    }
278:            }
279:            else if (move->piece == rook)
280:            {
281:                    if (newBoard->nextColour == white)
282:                    {
283:                            if (newBoard->canWhiteCastleQueenSide && move->from == 0)
284:                            {
285:                                    newBoard->canWhiteCastleQueenSide = false;
286:                                    newBoard->zorbistKey ^= ZorbistKeys::whiteQueenSideCastlingKey;
287:                            }
288:                            else if (newBoard->canWhiteCastleKingSide && move->from == 7)
289:                            {
290:                                    newBoard->canWhiteCastleKingSide = false;
291:                                    newBoard->zorbistKey ^= ZorbistKeys::whiteKingSideCastlingKey;
292:                            }
293:                    }
294:                    else
295:                    {
296:                            if (newBoard->canBlackCastleQueenSide && move->from == 56)
297:                            {
298:                                    newBoard->canBlackCastleQueenSide = false;
299:                                    newBoard->zorbistKey ^= ZorbistKeys::blackQueenSideCastlingKey;
300:                            }
301:                            else if (newBoard->canBlackCastleKingSide && move->from == 63)
302:                            {
303:                                    newBoard->canBlackCastleKingSide = false;
304:                                    newBoard->zorbistKey ^= ZorbistKeys::blackKingSideCastlingKey;
305:                            }
306:                    }
307:            }
308:            else if(move->moveType != quietMove)
309:            {
310:                    //Capturing a rook
311:                    if (((uint64_t)1 << move->to & newBoard->getPieceBitboard(white,rook)) > 0)
312:                    {
313:                            if (move->to == 0)
314:                            {
315:                                    if (newBoard->canWhiteCastleQueenSide)
316:                                    {
317:                                            newBoard->canWhiteCastleQueenSide = false;
318:                                            newBoard->zorbistKey ^= ZorbistKeys::whiteQueenSideCastlingKey;
319:                                    }
320:                            }
321:                            else if (move->to == 7)
322:                            {
323:                                    if (newBoard->canWhiteCastleKingSide)
324:                                    {
325:                                            newBoard->canWhiteCastleKingSide = false;
326:                                            newBoard->zorbistKey ^= ZorbistKeys::whiteKingSideCastlingKey;
327:                                    }
328:                            }
329:                    }
330:                    else if (((uint64_t)1 << move->to & newBoard->getPieceBitboard(black, rook)) > 0)
331:                    {
332:                            if (move->to == 56)
333:                            {
334:                                    if (newBoard->canBlackCastleQueenSide)
335:                                    {
336:                                            newBoard->canBlackCastleQueenSide = false;
337:                                            newBoard->zorbistKey ^= ZorbistKeys::blackQueenSideCastlingKey;
338:                                    }
```

```
339:                            }
340:                            else if (move->to == 63)
341:                            {
342:                                    if (newBoard->canBlackCastleKingSide)
343:                                    {
344:                                            newBoard->canBlackCastleKingSide = false;
345:                                            newBoard->zorbistKey ^= ZorbistKeys::blackKingSideCastlingKey;
346:                                    }
347:                            }
348:                    }
349:            }
350:    }
351:
352:    void Move::undoMove(Board * board)
353:    {
354:            board->moveHistory.pop_back();
355:
356:            colours opponentColour = board->nextColour;
357:            board->nextColour = switchColour(board->nextColour);
358:            board->kingDangerSquares = 0;
359:            board->canBlackCastleQueenSide = canBlackCastleQueenSide;
360:            board->canBlackCastleKingSide = canBlackCastleKingSide;
361:            board->canWhiteCastleQueenSide = canWhiteCastleQueenSide;
362:            board->canWhiteCastleKingSide = canWhiteCastleKingSide;
363:            board->zorbistKey = hash;
364:            board->pawnScoreZorbistKey = pawnHash;
365:            board->enPassantSquare = enPassantSquare;
366:
367:            board->addPositionalScore(board->nextColour, piece, from);
368:
369:            //Updates materialScore for removed pieces
370:            if (capturedPiece != blank)
371:            {
372:                    board->addMaterialScore(opponentColour, capturedPiece);
373:                    if(capturedPiece != blank && (piece != pawn || to != board->enPassantSquare))
374:                    board->addPositionalScore(opponentColour, capturedPiece, to);
375:            }
376:
377:            switch (moveType)
378:            {
379:            case quietMove:
380:            {
381:                    //Moves the piece
382:                    uint64_t bitboard = board->getPieceBitboard(board->nextColour, piece);
383:                    bitboard = (bitboard & ~((uint64_t)1 << to)) | ((uint64_t)1 << from);
384:                    board->setBitboard(board->nextColour, piece, bitboard);
385:
386:                    board->removePositionalScore(board->nextColour, piece, to);
387:            }
388:            break;
389:            case capture:
390:            {
391:                    if (board->enPassantSquare == to)
392:                    {
393:                            //Adds the captured piece under en passent
394:                            if (board->nextColour == white)
395:                            {
396:                                    board->setBitboard(opponentColour, pawn, board->getPieceBitboard(opponentColour, pawn) | ((uint64_t)1 << (to - 8)));
397:                                    board->addPositionalScore(opponentColour, pawn, to - 8);
398:                            }
399:                            else
400:                            {
401:                                    board->setBitboard(opponentColour, pawn, board->getPieceBitboard(opponentColour, pawn) | ((uint64_t)1 << (to + 8)));
402:                                    board->addPositionalScore(opponentColour, pawn, to + 8);
403:                            }
404:                    }
405:                    else
406:                    {
407:                            //Adds the captured piece
408:                            board->setBitboard(opponentColour, capturedPiece, board->getPieceBitboard(opponentColour, capturedPiece) | ((uint64_t)1 << to ));
409:                    }
410:
411:                    //Moves the piece
412:                    uint64_t bitboard = board->getPieceBitboard(board->nextColour, piece);
413:                    bitboard = (bitboard & ~((uint64_t)1 << to)) | ((uint64_t)1 << from);
414:                    board->setBitboard(board->nextColour, piece, bitboard);
415:
416:                    board->removePositionalScore(board->nextColour, piece, to);
417:            }
418:            break;
419:            case knightPromotion:
420:            {
421:                    //Adds the captued piece
422:                    if(capturedPiece != blank)
423:                            board->setBitboard(opponentColour, capturedPiece, board->getPieceBitboard(opponentColour, capturedPiece) | ((uint64_t)1 << to));
424:
425:                    //Adds the moved Piece
426:                    board->setBitboard(board->nextColour, piece, board->getPieceBitboard(board->nextColour, piece) | ((uint64_t)1 << from));
427:
428:                    //Removes the promoted piece
429:                    uint64_t bitboard = board->getPieceBitboard(board->nextColour, knight);
430:                    bitboard &= ~((uint64_t)1 << to);
431:                    board->setBitboard(board->nextColour, knight, bitboard);
432:
433:                    board->addMaterialScore(board->nextColour, pawn);
434:                    board->removeMaterialScore(board->nextColour, knight);
435:                    board->removePositionalScore(board->nextColour, knight, to);
436:            }
437:            break;
438:            case bishopPromotion:
439:            {
440:                    //Adds the captued piece
441:                    if (capturedPiece != blank)
442:                            board->setBitboard(opponentColour, capturedPiece, board->getPieceBitboard(opponentColour, capturedPiece) | ((uint64_t)1 << to));
443:
444:                    //Adds the moved Piece
445:                    board->setBitboard(board->nextColour, piece, board->getPieceBitboard(board->nextColour, piece) | ((uint64_t)1 << from));
446:
447:                    //Removes the promoted piece
448:                    uint64_t bitboard = board->getPieceBitboard(board->nextColour, bishop);
449:                    bitboard &= ~((uint64_t)1 << to);
450:                    board->setBitboard(board->nextColour, bishop, bitboard);
451:
452:                    board->addMaterialScore(board->nextColour, pawn);
```

```
453:                    board->removeMaterialScore(board->nextColour, bishop);
454:                    board->removePositionalScore(board->nextColour, bishop, to);
455:            }
456:            break;
457:            case rookPromotion:
458:            {
459:                    //Adds the captued piece
460:                    if (capturedPiece != blank)
461:                            board->setBitboard(opponentColour, capturedPiece, board->getPieceBitboard(opponentColour, capturedPiece) | ((uint64_t)1 << to));
462:
463:                    //Adds the moved Piece
464:                    board->setBitboard(board->nextColour, piece, board->getPieceBitboard(board->nextColour, piece) | ((uint64_t)1 << from));
465:
466:                    //Removes the promoted piece
467:                    uint64_t bitboard = board->getPieceBitboard(board->nextColour, rook);
468:                    bitboard &= ~((uint64_t)1 << to);
469:                    board->setBitboard(board->nextColour, rook, bitboard);
470:
471:                    board->addMaterialScore(board->nextColour, pawn);
472:                    board->removeMaterialScore(board->nextColour, rook);
473:                    board->removePositionalScore(board->nextColour, rook, to);
474:            }
475:            break;
476:            case queenPromotion:
477:            {
478:                    //Adds the captued piece
479:                    if (capturedPiece != blank)
480:                            board->setBitboard(opponentColour, capturedPiece, board->getPieceBitboard(opponentColour, capturedPiece) | ((uint64_t)1 << to));
481:
482:                    //Adds the moved Piece
483:                    board->setBitboard(board->nextColour, piece, board->getPieceBitboard(board->nextColour, piece) | ((uint64_t)1 << from));
484:
485:                    //Removes the promoted piece
486:                    uint64_t bitboard = board->getPieceBitboard(board->nextColour, queen);
487:                    bitboard &= ~((uint64_t)1 << to);
488:                    board->setBitboard(board->nextColour, queen, bitboard);
489:
490:                    board->addMaterialScore(board->nextColour, pawn);
491:                    board->removeMaterialScore(board->nextColour, queen);
492:                    board->removePositionalScore(board->nextColour, queen, to);
493:            }
494:            break;
495:            case pawnDoubleMove:
496:            {
497:                    //Moves the piece
498:                    uint64_t bitboard = board->getPieceBitboard(board->nextColour, piece);
499:                    bitboard = (bitboard & ~((uint64_t)1 << to)) | ((uint64_t)1 << from);
500:                    board->setBitboard(board->nextColour, piece, bitboard);
501:
502:                    board->removePositionalScore(board->nextColour, piece, to);
503:            }
504:            break;
505:            case kingSideCastling:
506:            {
507:                    //Moves the king
508:                    uint64_t bitboard = board->getPieceBitboard(board->nextColour, piece);
509:                    bitboard = (bitboard & ~((uint64_t)1 << to)) | ((uint64_t)1 << from);
510:                    board->setBitboard(board->nextColour, piece, bitboard);
511:                    if (board->nextColour == white)
512:                    {
513:                            board->setBitboard(white, rook, (board->getPieceBitboard(white, rook) & ~32) | 128); //Moves the rook
514:
515:                            board->addPositionalScore(white, rook, 7);
516:                            board->removePositionalScore(white, rook, 5);
517:                    }
518:                    else
519:                    {
520:                            board->setBitboard(black, rook, (board->getPieceBitboard(black, rook) & ~2305843009213693952) | 9223372036854775808); //Moves the rook
521:
522:                            board->addPositionalScore(white, rook, 63);
523:                            board->removePositionalScore(white, rook, 61);
524:                    }
525:
526:                    board->removePositionalScore(board->nextColour, piece, to);
527:            }
528:            break;
529:            case queenSideCastling:
530:            {
531:                    //Moves the king
532:                    uint64_t bitboard = board->getPieceBitboard(board->nextColour, piece);
533:                    bitboard = (bitboard & ~((uint64_t)1 << to)) | ((uint64_t)1 << from);
534:                    board->setBitboard(board->nextColour, piece, bitboard);
535:                    if (board->nextColour == white)
536:                    {
537:                            board->setBitboard(white, rook, (board->getPieceBitboard(white, rook) & ~8) | 1); //Moves the rook
538:                            board->addPositionalScore(white, rook, 0);
539:                            board->removePositionalScore(white, rook, 3);
540:                    }
541:                    else
542:                    {
543:                            board->setBitboard(black, rook, (board->getPieceBitboard(black, rook) & ~576460752303423488) | 72057594037927936); //Moves the rook
544:                            board->addPositionalScore(white, rook, 56);
545:                            board->removePositionalScore(white, rook, 59);
546:                    }
547:                    board->removePositionalScore(board->nextColour, piece, to);
548:            }
549:            break;
550:            }
551:            board->update();
552:    }
553:
554:    void Move::updateZorbistKeys(Board * board, colours opponentColour)
555:    {
556:            board->zorbistKey ^= ZorbistKeys::blackMoveKey;
557:
558:            //Removes en passant file from hash.
559:            if (board->enPassantSquare != -1)
560:            {
561:                    board->zorbistKey ^= ZorbistKeys::enPassantKeys[board->enPassantSquare % 8];
562:            }
563:
564:            //moves piece in hash
565:            board->zorbistKey ^= ZorbistKeys::pieceKeys[from][piece + 6 * board->nextColour];
566:            board->zorbistKey ^= ZorbistKeys::pieceKeys[to][piece + 6 * board->nextColour];
```

```
567:
568:        //updates pawn structure hash
569:        if (piece == pawn)
570:        {
571:            board->pawnScoreZorbistKey ^= ZorbistKeys::pieceKeys[from][piece + 6 * board->nextColour];
572:            board->pawnScoreZorbistKey ^= ZorbistKeys::pieceKeys[to][piece + 6 * board->nextColour];
573:        }
574:
575:        //Removed captured piece from hash when capture is not enpassant
576:        if (capturedPiece != blank && (to != enPassantSquare || piece != pawn))
577:        {
578:            board->zorbistKey ^= ZorbistKeys::pieceKeys[to][capturedPiece + 6 * opponentColour];
579:
580:            //Updates pawn structure hash
581:            if (capturedPiece == pawn)
582:            {
583:                board->pawnScoreZorbistKey ^= ZorbistKeys::pieceKeys[to][capturedPiece + 6 * opponentColour];
584:            }
585:        }
586:
587:        switch (moveType)
588:        {
589:        case capture:
590:        {
591:            if (board->enPassantSquare == to && piece == pawn)
592:            {
593:                //Removes the captured piece's hash under en passent
594:                if (board->nextColour == white)
595:                {
596:                    board->zorbistKey ^= ZorbistKeys::pieceKeys[to - 8][pawn + 6 * opponentColour];
597:                    board->pawnScoreZorbistKey ^= ZorbistKeys::pieceKeys[to - 8][pawn + 6 * opponentColour];
598:                }
599:                else
600:                {
601:                    board->zorbistKey ^= ZorbistKeys::pieceKeys[to + 8][pawn + 6 * opponentColour];
602:                    board->pawnScoreZorbistKey ^= ZorbistKeys::pieceKeys[to + 8][pawn + 6 * opponentColour];
603:                }
604:            }
605:        }
606:        break;
607:        case knightPromotion:
608:        {
609:            //Switches hash to promoted pieceType
610:            board->zorbistKey ^= ZorbistKeys::pieceKeys[to][piece + 6 * board->nextColour];
611:            board->zorbistKey ^= ZorbistKeys::pieceKeys[to][knight + 6 * board->nextColour];
612:
613:            //Removes piece from pawn hash
614:            board->pawnScoreZorbistKey ^= ZorbistKeys::pieceKeys[to][pawn + 6 * board->nextColour];
615:        }
616:        break;
617:        case bishopPromotion:
618:        {
619:            //Switches hash to promoted pieceType
620:            board->zorbistKey ^= ZorbistKeys::pieceKeys[to][piece + 6 * board->nextColour];
621:            board->zorbistKey ^= ZorbistKeys::pieceKeys[to][bishop + 6 * board->nextColour];
622:
623:            //Removes piece from pawn hash
624:            board->pawnScoreZorbistKey ^= ZorbistKeys::pieceKeys[to][pawn + 6 * board->nextColour];
625:        }
626:        break;
627:        case rookPromotion:
628:        {
629:            //Switches hash to promoted pieceType
630:            board->zorbistKey ^= ZorbistKeys::pieceKeys[to][piece + 6 * board->nextColour];
631:            board->zorbistKey ^= ZorbistKeys::pieceKeys[to][rook + 6 * board->nextColour];
632:
633:            //Removes piece from pawn hash
634:            board->pawnScoreZorbistKey ^= ZorbistKeys::pieceKeys[to][pawn + 6 * board->nextColour];
635:        }
636:        break;
637:        case queenPromotion:
638:        {
639:            //Switches hash to promoted pieceType
640:            board->zorbistKey ^= ZorbistKeys::pieceKeys[to][piece + 6 * board->nextColour];
641:            board->zorbistKey ^= ZorbistKeys::pieceKeys[to][queen + 6 * board->nextColour];
642:
643:            //Removes piece from pawn hash
644:            board->pawnScoreZorbistKey ^= ZorbistKeys::pieceKeys[to][pawn + 6 * board->nextColour];
645:        }
646:        break;
647:        case pawnDoubleMove:
648:        {
649:            //Sets En passant target hash.
650:            if (board->nextColour == white)
651:            {
652:                board->zorbistKey ^= ZorbistKeys::enPassantKeys[to % 8];
653:            }
654:            else
655:            {
656:                board->zorbistKey ^= ZorbistKeys::enPassantKeys[to % 8];
657:            }
658:        }
659:        break;
660:        case kingSideCastling:
661:        {
662:            if (board->nextColour == white)
663:            {
664:                //Updates hash for rook
665:                board->zorbistKey ^= ZorbistKeys::pieceKeys[7][rook + 6 * board->nextColour];
666:                board->zorbistKey ^= ZorbistKeys::pieceKeys[5][rook + 6 * board->nextColour];
667:            }
668:            else
669:            {
670:                //Updates hash for rook
671:                board->zorbistKey ^= ZorbistKeys::pieceKeys[63][rook + 6 * board->nextColour];
672:                board->zorbistKey ^= ZorbistKeys::pieceKeys[61][rook + 6 * board->nextColour];
673:            }
674:        }
675:        break;
676:        case queenSideCastling:
677:        {
678:            if (board->nextColour == white)
679:            {
680:                //Updates hash for rook
```

```
681:                            board->zorbistKey ^= ZorbistKeys::pieceKeys[0][rook + 6 * board->nextColour];
682:                            board->zorbistKey ^= ZorbistKeys::pieceKeys[3][rook + 6 * board->nextColour];
683:                    }
684:                    else
685:                    {
686:                            //Updates hash for rook
687:                            board->zorbistKey ^= ZorbistKeys::pieceKeys[56][rook + 6 * board->nextColour];
688:                            board->zorbistKey ^= ZorbistKeys::pieceKeys[59][rook + 6 * board->nextColour];
689:                    }
690:            }
691:            break;
692:            }
693:    }
694:
```

ChessEngineLibrary\move.h

```
001:  #pragma once
002:  #include <string>
003:  #include <stdint.h>
004:
005:  #include "piece.h"
006:  #include "board.h"
007:
008:  #include "transpositionTable.h"
009:
010:  enum MoveType{quietMove, capture, knightPromotion, bishopPromotion, rookPromotion, queenPromotion, pawnDoubleMove, kingSideCastling, queenSideCastling};
011:
012:  struct Move
013:  {
014:          Move();
015:          Move(int newFrom, int newTo, MoveType newMoveType, pieceType newPieceType, Board* board);
016:          int from;
017:          int to;
018:          pieceType piece;
019:          pieceType capturedPiece;
020:          MoveType moveType;
021:
022:          int moveRating;
023:
024:          //The flags of the board prior to applyMove. Used in undoMove
025:          bool canBlackCastleQueenSide;
026:          bool canBlackCastleKingSide;
027:          bool canWhiteCastleQueenSide;
028:          bool canWhiteCastleKingSide;
029:          int enPassantSquare;
030:          uint64_t hash;
031:          uint64_t pawnHash;
032:
033:          void applyMove(Board* board);
034:          void undoMove(Board* board);
035:
036:          //Updates all keys used for hash tables.
037:          void updateZorbistKeys(Board* board, colours opponentColour);
038:
039:          bool operator==(const Move& b)
040:          {
041:                  return (from == b.from) && (to == b.to) && (piece == b.piece) && (moveType == b.moveType);
042:          }
043:  };
044:
045:  void updateCastlingRights(Board* newBoard, Move* move);
046:
047:
048:
049:
```

ChessEngineLibrary\moveGeneration.cpp

```
001:  #include "moveGeneration.h"
002:
003:  uint64_t knightMovesArray[64] = {0};
004:  uint64_t kingMovesArray[64] = { 0 };
005:  uint64_t pawnWhiteAttacksArray[64] = { 0 };
006:  uint64_t pawnBlackAttacksArray[64] = { 0 };
007:
008:  int searchForMoves(Board * board, std::array<Move,150>* moveList)
009:  {
010:      int arraySize = 0;
011:
012:      magicBitboards magicData;
013:
014:      uint64_t friendlyPieces, enemyPieces;
015:      if (board->nextColour == white)
016:      {
017:          friendlyPieces = board->whitePieces;
018:          enemyPieces = board->blackPieces;
019:      }
020:      else
021:      {
022:          friendlyPieces = board->blackPieces;
023:          enemyPieces = board->whitePieces;
024:      }
025:
026:      uint64_t captureMask = 0xFFFFFFFFFFFFFFFF;
027:      uint64_t pushMask = 0xFFFFFFFFFFFFFFFF;
028:
029:      uint64_t kingDangerSquares = board->getKingDangerSquares();
030:      uint64_t pinnedPieces = getPinnedPieces(board);
031:
032:      uint64_t kingAttackers;
033:      int numOfKingAttackers;
034:
035:      //If in check
036:      if (kingDangerSquares & board->getPieceBitboard(board->nextColour, king))
037:      {
038:          kingAttackers = getAttackers(board, board->nextColour, board->getPieceBitboard(board->nextColour, king));
039:          numOfKingAttackers = bitSum(kingAttackers);
040:      }
041:      else
042:      {
043:          kingAttackers = 0;
044:          numOfKingAttackers = 0;
045:      }
046:
047:      if (numOfKingAttackers == 1)
048:      {
049:          captureMask = kingAttackers;
050:
051:          //If the attacker is a sliding piece
052:          if (kingAttackers & (board->getPieceBitboard(switchColour(board->nextColour), rook) | board->getPieceBitboard(switchColour(board->nextColour), queen) | board->getPieceBitboard(switchColour(board->nextColour
053:          {
054:              pushMask = inBetween(bitScanForward(board->getPieceBitboard(board->nextColour, king)), bitScanForward(kingAttackers));
055:          }
056:          else
057:          {
058:              pushMask = 0;
059:          }
060:      }
061:      if(numOfKingAttackers > 1) arraySize = generateKingMoves(board, moveList, friendlyPieces, enemyPieces, kingDangerSquares, arraySize);
062:      else
063:      {
064:          arraySize = generatePawnMoves(board, moveList, pinnedPieces, pushMask, captureMask, arraySize);
065:          arraySize = generateKingMoves(board, moveList, friendlyPieces, enemyPieces, kingDangerSquares, arraySize);
066:          arraySize = generateKnightMoves(board, moveList, friendlyPieces, enemyPieces, pinnedPieces, pushMask, captureMask, arraySize);
067:          arraySize = generateRookMoves(board, moveList, friendlyPieces, enemyPieces, pinnedPieces, pushMask, captureMask, arraySize);
068:          arraySize = generateBishopMoves(board, moveList, friendlyPieces, enemyPieces, pinnedPieces, pushMask, captureMask, arraySize);
069:          arraySize = generateQueenMoves(board, moveList, friendlyPieces, enemyPieces, pinnedPieces, pushMask, captureMask, arraySize);
070:          arraySize = generateCastlingMoves(board, moveList, friendlyPieces, enemyPieces, kingDangerSquares, arraySize);
071:      }
072:
073:      return arraySize;
074:  }
075:
076:  int generatePawnMoves(Board* board, std::array<Move,150>* Movelist, uint64_t pinnedPieces, uint64_t pushMask, uint64_t captureMask, int arraySize)
077:  {
078:      uint64_t pawnPos, pawnMoves, pawnAttacks, pawnDoubleMoves, legalMoves;
079:      bool isPinned;
080:
081:      //Calculates constants used later
082:      const int forwards = (board->nextColour == white) ? 8 : -8;
083:      const uint64_t rank3BB = (board->nextColour == white) ? rank3 : rank6;
084:      const uint64_t emptySquares = ~board->allPieces;
085:      //const uint64_t emptySquares = ~board->allPieces & pushMask;
086:      const int leftAttack = (board->nextColour == white) ? 7 : -9;
087:      const int rightAttack = (board->nextColour == white) ? 9 : -7;
088:      const uint64_t enemyPieces = ((board->nextColour == white) ? board->blackPieces : board->whitePieces) | (uint64_t)1 << board->enPassantSquare;
089:
090:      //The bitboard of all pawns that are not pinned. (pinned pieces need to be calculated seperatly)
091:      uint64_t pawnBitboard = board->getPieceBitboard(board->nextColour, pawn) & ~pinnedPieces;
092:      uint64_t pinnedPawnBitboard = board->getPieceBitboard(board->nextColour, pawn) & pinnedPieces;
093:
094:      //Adds en-passant position to capture moves (if not pinned)
095:      if (board->enPassantSquare != -1)
096:      {
097:          const uint64_t enemyEnPassantTarget = captureMask & shift((uint64_t)1 << board->enPassantSquare,-forwards);
098:          if (enemyEnPassantTarget)
099:          {
100:              //The two pieces that will be (re)moved by an enpassant capture
101:              uint64_t leftPos = (uint64_t)1 << (board->enPassantSquare - forwards - 1) & ~fileH & board->getPieceBitboard(board->nextColour, pawn);
102:              uint64_t rightPos = (uint64_t)1 << (board->enPassantSquare - forwards + 1) & ~fileA & board->getPieceBitboard(board->nextColour, pawn);
103:
104:              if (leftPos && rightPos)
105:              {
106:                  captureMask |= (uint64_t)1 << board->enPassantSquare;
107:              }
108:              else if (leftPos && !isPinnedEnPassant(board, leftPos | enemyEnPassantTarget))
109:              {
110:                  captureMask |= (uint64_t)1 << board->enPassantSquare;
```

```
111:                            }
112:                            else if (rightPos && !isPinnedEnPassant(board, rightPos | enemyEnPassantTarget))
113:                            {
114:                                    captureMask |= (uint64_t)1 << board->enPassantSquare;
115:                            }
116:                            else
117:                            {
118:                                    captureMask &= ~((uint64_t)1 << board->enPassantSquare);
119:                            }
120:                    }
121:            }
122:
123:
124:            //Adds quiet moves to moveList
125:            pawnMoves = shift(pawnBitboard, forwards) & emptySquares;
126:            pawnDoubleMoves = shift(pawnMoves & rank3BB, forwards) & emptySquares & pushMask;
127:            pawnMoves &= pushMask; //Done after calculating double moves to allow double moves , where the single move would be illegal (in check).
128:            while (pawnMoves)
129:            {
130:                    uint64_t currentMove = pop(pawnMoves);
131:                    int to = bitScanForward(currentMove);
132:                    arraySize = addPawnMovesPromotions(to - forwards, to, currentMove, quietMove, board, Movelist, arraySize);
133:            }
134:            while (pawnDoubleMoves)
135:            {
136:                    uint64_t currentMove = pop(pawnDoubleMoves);
137:                    int to = bitScanForward(currentMove);
138:                    (*Movelist)[arraySize] = Move(to - forwards * 2, to, pawnDoubleMove, pawn, board);
139:                    arraySize++;
140:            }
141:
142:            //Adds captures moves to moveList. (right and left attacks are handled seperatly)
143:            pawnAttacks = shift(pawnBitboard, rightAttack) & enemyPieces & ~fileA & captureMask;
144:            while (pawnAttacks)
145:            {
146:                    uint64_t currentMove = pop(pawnAttacks);
147:                    int to = bitScanForward(currentMove);
148:                    arraySize = addPawnMovesPromotions(to - rightAttack, to, currentMove, capture, board, Movelist, arraySize);
149:
150:            }
151:            pawnAttacks = shift(pawnBitboard, leftAttack) & enemyPieces & ~fileH & captureMask;
152:            while (pawnAttacks)
153:            {
154:                    uint64_t currentMove = pop(pawnAttacks);
155:                    int to = bitScanForward(currentMove);
156:                    arraySize = addPawnMovesPromotions(to - leftAttack, to, currentMove, capture, board, Movelist, arraySize);
157:            }
158:
159:
160:            while (pinnedPawnBitboard)
161:            {
162:                    pawnPos = pop(pinnedPawnBitboard);
163:                    legalMoves = generateLegalFilterForPinnedPiece(board, pawnPos);
164:
165:                    int pawnPosIndex = bitScanForward(pawnPos);
166:
167:                    pawnMoves = shift(pawnPos, forwards) & ~board->allPieces & legalMoves; //Move forward
168:                    pawnDoubleMoves = shift(pawnMoves & rank3BB,  forwards) & ~board->allPieces & legalMoves & pushMask; //Move twice on first turn if first is clear
169:                    pawnMoves &= pushMask;
170:
171:                    pawnAttacks = ((board->nextColour == white) ? pawnWhiteAttacksArray[pawnPosIndex] : pawnBlackAttacksArray[pawnPosIndex]) & enemyPieces;
172:                    pawnAttacks &= legalMoves & captureMask;
173:
174:                    if (pawnDoubleMoves)
175:                    {
176:                            (*Movelist)[arraySize] = Move(pawnPosIndex, pawnPosIndex + forwards*2, pawnDoubleMove, pawn, board);
177:                            arraySize++;
178:                    }
179:
180:                    arraySize = addPawnMoves(pawnPosIndex, pawnMoves, pawnAttacks, board, Movelist, arraySize);
181:            }
182:
183:            return arraySize;
184:    }
185:
186:    int generateKingMoves(Board * board, std::array<Move, 150>* Movelist, uint64_t friendlyPieces, uint64_t enemyPieces, uint64_t kingDangerSquares, int arraySize)
187:    {
188:            uint64_t kingBitboard;
189:            if (board->nextColour == white)
190:            {
191:                    kingBitboard = board->getPieceBitboard(white, king);
192:            }
193:            else
194:            {
195:                    kingBitboard = board->getPieceBitboard(black, king);
196:            }
197:            if (kingBitboard)
198:            {
199:                    uint64_t moves = kingMovesArray[bitScanForward(kingBitboard)] & ~friendlyPieces;
200:
201:                    //Filters out moves that would move the king into check.
202:                    moves &= ~kingDangerSquares;
203:
204:                    int kingPosIndex = bitScanForward(kingBitboard);
205:                    while (moves)
206:                    {
207:                            uint64_t kingPos = pop(moves);
208:                            arraySize = addMoves(kingPosIndex, bitScanForward(kingPos), king, Movelist, enemyPieces, board, arraySize);
209:                    }
210:            }
211:            return arraySize;
212:    }
213:
214:    int generateKnightMoves(Board * board, std::array<Move, 150>* Movelist, uint64_t friendlyPieces, uint64_t enemyPieces, uint64_t pinnedPieces, uint64_t pushMask, uint64_t captureMask, int arraySize)
215:    {
216:            uint64_t legalMoves;
217:            //Gets the knight bitboard , filtering out pieces that cannot move due to being pinned
218:            uint64_t knightBitboard = board->getPieceBitboard(board->nextColour, knight) & ~pinnedPieces;
219:
220:        while(knightBitboard)
221:        {
222:            uint64_t currentKnight = pop(knightBitboard);
223:                    int knightPosIndex = bitScanForward(currentKnight);
224:
```

```
225:                    uint64_t moveBitboard = knightMovesArray[knightPosIndex] & ~friendlyPieces;
226:
227:                    //Filters out invalid moves while in check
228:                    moveBitboard &= (pushMask | captureMask);
229:
230:                while (moveBitboard)
231:                {
232:                        uint64_t knightMove = pop(moveBitboard);
233:                    arraySize = addMoves(knightPosIndex, bitScanForward(knightMove), knight, Movelist, enemyPieces, board, arraySize);
234:                }
235:        }
236:        return arraySize;
237:    }
238:
239:    int generateRookMoves(Board * board, std::array<Move, 150>* Movelist, uint64_t friendlyPieces, uint64_t enemyPieces, uint64_t pinnedPieces, uint64_t pushMask, uint64_t captureMask, int arraySize)
240:    {
241:        uint64_t rookBitboard, legalMoves;
242:        if (board->nextColour == white)
243:        {
244:            rookBitboard = board->getPieceBitboard(white, rook);
245:        }
246:        else
247:        {
248:            rookBitboard = board->getPieceBitboard(black, rook);
249:        }
250:        while (rookBitboard)
251:        {
252:            uint64_t currentRook = pop(rookBitboard);
253:            int currentPos = bitScanForward(currentRook);
254:
255:            if (currentRook & pinnedPieces) legalMoves = generateLegalFilterForPinnedPiece(board, currentRook);
256:            else legalMoves = ~0;
257:
258:            uint64_t occupancy = magicBitboards::rookMask[currentPos] & board->allPieces;
259:            uint64_t magicResult = occupancy * magicBitboards::magicNumberRook[currentPos];
260:            int arrayIndex = magicResult >> magicBitboards::magicNumberShiftRook[currentPos];
261:            uint64_t moves = magicBitboards::magicMovesRook[currentPos][arrayIndex] & ~friendlyPieces & legalMoves & (pushMask | captureMask);
262:
263:            while (moves)
264:            {
265:                uint64_t rookPos = pop(moves);
266:                arraySize = addMoves(currentPos, bitScanForward(rookPos), rook, Movelist, enemyPieces, board, arraySize);
267:            }
268:        }
269:        return arraySize;
270:    }
271:
272:    int generateBishopMoves(Board * board, std::array<Move, 150>* Movelist, uint64_t friendlyPieces, uint64_t enemyPieces, uint64_t pinnedPieces, uint64_t pushMask, uint64_t captureMask, int arraySize)
273:    {
274:        uint64_t bishopBitboard, legalMoves;
275:        if (board->nextColour == white)
276:        {
277:            bishopBitboard = board->getPieceBitboard(white, bishop);
278:        }
279:        else
280:        {
281:            bishopBitboard = board->getPieceBitboard(black, bishop);
282:        }
283:        while (bishopBitboard)
284:        {
285:            uint64_t currentBishop = pop(bishopBitboard);
286:            int currentPos = bitScanForward(currentBishop);
287:
288:            if (currentBishop & pinnedPieces) legalMoves = generateLegalFilterForPinnedPiece(board, currentBishop);
289:            else legalMoves = ~0;
290:
291:            uint64_t occupancy = magicBitboards::bishopMask[currentPos] & board->allPieces;
292:            uint64_t magicResult = occupancy * magicBitboards::magicNumberBishop[currentPos];
293:            int arrayIndex = magicResult >> magicBitboards::magicNumberShiftBishop[currentPos];
294:            uint64_t moves = magicBitboards::magicMovesBishop[currentPos][arrayIndex] & ~friendlyPieces & legalMoves & (pushMask | captureMask);
295:
296:            while (moves)
297:            {
298:                uint64_t bishopPos = pop(moves);
299:                arraySize = addMoves(currentPos, bitScanForward(bishopPos), bishop, Movelist, enemyPieces, board, arraySize);
300:            }
301:        }
302:        return arraySize;
303:    }
304:
305:    int generateQueenMoves(Board * board, std::array<Move,150>* Movelist, uint64_t friendlyPieces, uint64_t enemyPieces, uint64_t pinnedPieces, uint64_t pushMask, uint64_t captureMask, int arraySize)
306:    {
307:        uint64_t queenBitboard, legalMoves;
308:        if (board->nextColour == white)
309:        {
310:            queenBitboard = board->getPieceBitboard(white, queen);
311:        }
312:        else
313:        {
314:            queenBitboard = board->getPieceBitboard(black, queen);
315:        }
316:        while (queenBitboard)
317:        {
318:            uint64_t currentQueen = pop(queenBitboard);
319:            int currentPos = bitScanForward(currentQueen);
320:
321:            if (currentQueen & pinnedPieces) legalMoves = generateLegalFilterForPinnedPiece(board, currentQueen);
322:            else legalMoves = ~0;
323:
324:            //Moves bishop moves
325:            uint64_t occupancy = magicBitboards::bishopMask[currentPos] & board->allPieces;
326:            uint64_t magicResult = occupancy * magicBitboards::magicNumberBishop[currentPos];
327:            int arrayIndex = magicResult >> magicBitboards::magicNumberShiftBishop[currentPos];
328:            uint64_t moves = magicBitboards::magicMovesBishop[currentPos][arrayIndex] & ~friendlyPieces & legalMoves & (pushMask | captureMask);
329:
330:            //Rook moves
331:            occupancy = magicBitboards::rookMask[currentPos] & board->allPieces;
332:            magicResult = occupancy * magicBitboards::magicNumberRook[currentPos];
333:            arrayIndex = magicResult >> magicBitboards::magicNumberShiftRook[currentPos];
334:            moves |= magicBitboards::magicMovesRook[currentPos][arrayIndex] & ~friendlyPieces & legalMoves & (pushMask | captureMask);
335:
336:            while (moves)
337:            {
338:                uint64_t queenPos = pop(moves);
```

```
339:                        arraySize = addMoves(currentPos, bitScanForward(queenPos), queen, Movelist, enemyPieces, board,arraySize);
340:                }
341:        }
342:        return arraySize;
343: }
344:
345: int generateCastlingMoves(Board * board, std::array<Move,150>* Movelist, uint64_t friendlyPieces, uint64_t enemyPieces, uint64_t kingDangerSquares, int arraySize)
346: {
347:        if (board->nextColour == white)
348:        {
349:                if (board->canWhiteCastleKingSide && (board->allPieces & 96) == 0)//Kingside castling
350:                {
351:                        if ((112 & kingDangerSquares) == 0)
352:                        {
353:                                (*Movelist)[arraySize] = Move(4, 6, kingSideCastling, king, board);
354:                                arraySize++;
355:                        }
356:                }
357:                if(board->canWhiteCastleQueenSide && (board->allPieces & 14) == 0)//Queenside castling
358:                {
359:                        if ((28 & kingDangerSquares) == 0)
360:                        {
361:                                (*Movelist)[arraySize] = Move(4, 2, queenSideCastling, king, board);
362:                                arraySize++;
363:                        }
364:                }
365:        }
366:        else
367:        {
368:                if (board->canBlackCastleKingSide && (board->allPieces & 6917529027641081856) == 0)//Kingside castling
369:                {
370:                        if ((8070450532247928832 & kingDangerSquares) == 0)
371:                        {
372:                                (*Movelist)[arraySize] = Move(60, 62, kingSideCastling, king, board);
373:                                arraySize++;
374:                        }
375:                }
376:                if (board->canBlackCastleQueenSide && (board->allPieces & 1008806316530991104) == 0)//Queenside castling
377:                {
378:                        if ((2017612633061982208 & kingDangerSquares) == 0)
379:                        {
380:                                (*Movelist)[arraySize] = Move(60, 58, queenSideCastling, king, board);
381:                                arraySize++;
382:                        }
383:                }
384:        }
385:        return arraySize;
386: }
387:
388: int addPawnMoves(int start, uint64_t quietMoves, uint64_t captureMoves, Board* board, std::array<Move,150>* Movelist, int arraySize)
389: {
390:        uint64_t currentMove;
391:        int currentPos;
392:        while (quietMoves)
393:        {
394:                currentMove = pop(quietMoves);
395:                currentPos = bitScanForward(currentMove);
396:                //Pawn promotion
397:                if ((board->nextColour == white && currentMove & rank8) || (board->nextColour == black && currentMove & rank1))
398:                {
399:                        (*Movelist)[arraySize] = Move(start, currentPos, rookPromotion, pawn, board);
400:                        (*Movelist)[arraySize + 1] = Move(start, currentPos, knightPromotion, pawn, board);
401:                        (*Movelist)[arraySize + 2] = Move(start, currentPos, queenPromotion, pawn, board);
402:                        (*Movelist)[arraySize + 3] = Move(start, currentPos, bishopPromotion, pawn, board);
403:                        arraySize += 4;
404:                }
405:                else
406:                {
407:                        (*Movelist)[arraySize] = Move(start, currentPos, quietMove, pawn, board);
408:                        arraySize++;
409:                }
410:        }
411:        while (captureMoves)
412:        {
413:                currentMove = pop(captureMoves);
414:                currentPos = bitScanForward(currentMove);
415:                //Pawn promotion
416:                if ((board->nextColour == white && currentMove & rank8) || (board->nextColour == black && currentMove & rank1))
417:                {
418:                        (*Movelist)[arraySize] = Move(start, currentPos, rookPromotion, pawn, board);
419:                        (*Movelist)[arraySize + 1] = Move(start, currentPos, knightPromotion, pawn, board);
420:                        (*Movelist)[arraySize + 2] = Move(start, currentPos, queenPromotion, pawn, board);
421:                        (*Movelist)[arraySize + 3] = Move(start, currentPos, bishopPromotion, pawn, board);
422:                        arraySize += 4;
423:                }
424:                else
425:                {
426:                        (*Movelist)[arraySize] = Move(start, currentPos, capture, pawn, board);
427:                        arraySize++;
428:                }
429:        }
430:        return arraySize;
431: }
432:
433: int addMoves(int start, int end, pieceType piece, std::array<Move,150>* Movelist, uint64_t enemyPieces, Board* board, int arraySize)
434: {
435:        if ((((uint64_t)1 << end) & enemyPieces )!= 0) //If the move is a capture
436:        {
437:                (*Movelist)[arraySize] = Move(start, end, capture, piece, board);
438:                arraySize++;
439:        }
440:        else
441:        {
442:                (*Movelist)[arraySize] = Move(start, end, quietMove, piece, board);
443:                arraySize++;
444:        }
445:        return arraySize;
446: }
447:
448: int addPawnMovesPromotions(int from, int to, uint64_t move, MoveType type, Board* board, std::array<Move, 150>* Movelist, int arraySize)
449: {
450:        if ((board->nextColour == white && (move & rank8)) || (board->nextColour == black && (move & rank1)))
451:        {
452:                (*Movelist)[arraySize] = Move(from, to, rookPromotion, pawn, board);
```

```
453:                    (*Movelist)[arraySize + 1] = Move(from, to, knightPromotion, pawn, board);
454:                    (*Movelist)[arraySize + 2] = Move(from, to, queenPromotion, pawn, board);
455:                    (*Movelist)[arraySize + 3] = Move(from, to, bishopPromotion, pawn, board);
456:                    arraySize += 4;
457:            }
458:            else
459:            {
460:                    (*Movelist)[arraySize] = Move(from, to, type, pawn, board);
461:                    arraySize++;
462:            }
463:            return arraySize;
464:    }
465:
466:    uint64_t getPinnedPieces(Board * board)
467:    {
468:            uint64_t pinnedPieces = 0;
469:
470:            uint64_t kingBitBoard = board->getPieceBitboard(board->nextColour, king);
471:            int kingPos = bitScanForward(kingBitBoard);
472:
473:            uint64_t enemyPieces, friendlyPieces;
474:            if (board->nextColour == white)
475:            {
476:                    enemyPieces = board->blackPieces;
477:                    friendlyPieces = board->whitePieces;
478:            }
479:            else
480:            {
481:                    enemyPieces = board->whitePieces;
482:                    friendlyPieces = board->blackPieces;
483:            }
484:
485:            uint64_t occupancy = magicBitboards::rookMask[kingPos] & enemyPieces;
486:            uint64_t magicResult = occupancy * magicBitboards::magicNumberRook[kingPos];
487:            int arrayIndex = magicResult >> magicBitboards::magicNumberShiftRook[kingPos];
488:            uint64_t kingRaysRook = magicBitboards::magicMovesRook[kingPos][arrayIndex] & (board->getPieceBitboard(switchColour(board->nextColour), rook) | board->getPieceBitboard(switchColour(board->nextColour), queen))
489:            while (kingRaysRook)
490:            {
491:                    uint64_t pinner = pop(kingRaysRook);
492:                    uint64_t pinnedPiece = inBetween(kingPos, bitScanForward(pinner)) & friendlyPieces;
493:                    if (bitSum(pinnedPiece) == 1) pinnedPieces |= pinnedPiece;
494:            }
495:
496:
497:            occupancy = magicBitboards::bishopMask[kingPos] & enemyPieces;
498:            magicResult = occupancy * magicBitboards::magicNumberBishop[kingPos];
499:            arrayIndex = magicResult >> magicBitboards::magicNumberShiftBishop[kingPos];
500:            uint64_t kingRaysBishop = magicBitboards::magicMovesBishop[kingPos][arrayIndex] & (board->getPieceBitboard(switchColour(board->nextColour), bishop) | board->getPieceBitboard(switchColour(board->nextColour), qu
501:            while (kingRaysBishop)
502:            {
503:                    uint64_t pinner = pop(kingRaysBishop);
504:                    uint64_t pinnedPiece = inBetween(kingPos, bitScanForward(pinner)) & friendlyPieces;
505:                    if (bitSum(pinnedPiece) == 1) pinnedPieces |= pinnedPiece;
506:            }
507:
508:            return pinnedPieces;
509:    }
510:
511:    //Calculates the moves a pinned piece could move and stay out of check.
512:    uint64_t generateLegalFilterForPinnedPiece(Board* board, uint64_t pinnedPiece)
513:    {
514:            uint64_t allPiecesWithoutPiece = board->allPieces & ~pinnedPiece;
515:            uint64_t kingBitBoard = board->getPieceBitboard(board->nextColour, king);
516:            int kingPos = bitScanForward(kingBitBoard);
517:
518:            uint64_t currentPos, currentRay, rookRays, bishopRays;
519:            uint64_t rookBitboard = board->getPieceBitboard(switchColour(board->nextColour), rook) | board->getPieceBitboard(switchColour(board->nextColour), queen);
520:            while (rookBitboard)
521:            {
522:                    currentPos = pop(rookBitboard);
523:                    int piecePos = bitScanForward(currentPos);
524:
525:                    uint64_t occupancy = magicBitboards::rookMask[piecePos] & allPiecesWithoutPiece;
526:                    uint64_t magicResult = occupancy * magicBitboards::magicNumberRook[piecePos];
527:                    int arrayIndex = magicResult >> magicBitboards::magicNumberShiftRook[piecePos];
528:                    rookRays = magicBitboards::magicMovesRook[piecePos][arrayIndex] & kingBitBoard;
529:                    if (inBetween(kingPos, piecePos) & pinnedPiece)
530:                    {
531:                            return inBetween(kingPos, piecePos) | currentPos;
532:                    }
533:
534:            }
535:
536:            uint64_t bishopBitboard = board->getPieceBitboard(switchColour(board->nextColour), bishop) | board->getPieceBitboard(switchColour(board->nextColour), queen);
537:            while (bishopBitboard)
538:            {
539:                    currentPos = pop(bishopBitboard);
540:                    int piecePos = bitScanForward(currentPos);
541:
542:                    uint64_t occupancy = magicBitboards::bishopMask[piecePos] & allPiecesWithoutPiece;
543:                    uint64_t magicResult = occupancy * magicBitboards::magicNumberBishop[piecePos];
544:                    int arrayIndex = magicResult >> magicBitboards::magicNumberShiftBishop[piecePos];
545:                    bishopRays = magicBitboards::magicMovesBishop[piecePos][arrayIndex] & kingBitBoard;
546:                    if (inBetween(kingPos, piecePos) & pinnedPiece)
547:                    {
548:                            return inBetween(kingPos, piecePos) | currentPos;
549:                    }
550:
551:            }
552:
553:            return 0;
554:    }
555:
556:    uint64_t getAttackers(Board * board, colours colour, uint64_t targetBitboard)
557:    {
558:            colours opponentColour = switchColour(colour);
559:            int targetPos = bitScanForward(targetBitboard);
560:            uint64_t attackers = 0;
561:
562:            if (colour == white)
563:            {
564:                    attackers |= board->getPieceBitboard(opponentColour, pawn) & (((targetBitboard << 7) & ~fileH) | ((targetBitboard << 9) & ~fileA));
565:            }
566:            else
```

```
567:        {
568:            attackers |= board->getPieceBitboard(opponentColour, pawn) & (((targetBitboard >> 9) & ~fileH) | ((targetBitboard >> 7) & ~fileA));
569:        }
570:
571:        //KnightMoves
572:        uint64_t knightMoves = knightMovesArray[targetPos];
573:        attackers |= knightMoves & board->getPieceBitboard(opponentColour, knight);
574:
575:        uint64_t moves = kingMovesArray[targetPos];
576:        attackers |= moves & board->getPieceBitboard(opponentColour, king);
577:
578:        //Rook and half of queen moves
579:        uint64_t occupancy = magicBitboards::rookMask[targetPos] & board->allPieces;
580:        uint64_t magicResult = occupancy * magicBitboards::magicNumberRook[targetPos];
581:        int arrayIndex = magicResult >> magicBitboards::magicNumberShiftRook[targetPos];
582:        uint64_t magicMoves = magicBitboards::magicMovesRook[targetPos][arrayIndex];
583:
584:        attackers |= magicMoves & (board->getPieceBitboard(opponentColour, rook) | (board->getPieceBitboard(opponentColour, queen)));
585:
586:        //Bishop and half of queen moves
587:        occupancy = magicBitboards::bishopMask[targetPos] & board->allPieces;
588:        magicResult = occupancy * magicBitboards::magicNumberBishop[targetPos];
589:        arrayIndex = magicResult >> magicBitboards::magicNumberShiftBishop[targetPos];
590:        magicMoves = magicBitboards::magicMovesBishop[targetPos][arrayIndex];
591:
592:        attackers |= magicMoves & (board->getPieceBitboard(opponentColour, bishop) | (board->getPieceBitboard(opponentColour, queen)));
593:
594:        return attackers;
595:    }
596:
597:    bool isPinnedEnPassant(Board* board, uint64_t pieces)
598:    {
599:        uint64_t kingBitBoard = board->getPieceBitboard(board->nextColour, king);
600:        int kingPos = bitScanForward(kingBitBoard);
601:
602:        uint64_t magicResult = 0 * magicBitboards::magicNumberRook[kingPos];
603:        int arrayIndex = magicResult >> magicBitboards::magicNumberShiftRook[kingPos];
604:        uint64_t kingRaysRook = magicBitboards::magicMovesRook[kingPos][arrayIndex];
605:        kingRaysRook &= (board->getPieceBitboard(switchColour(board->nextColour), rook) | board->getPieceBitboard(switchColour(board->nextColour), queen));
606:        while (kingRaysRook)
607:        {
608:            uint64_t pinner = pop(kingRaysRook);
609:            uint64_t pinnedPiece = inBetween(kingPos, bitScanForward(pinner)) & board->allPieces;
610:            if (pinnedPiece ==  pieces) return true;
611:        }
612:        /*
613:        magicResult = 0 * magicBitboards::magicNumberBishop[kingPos];
614:        arrayIndex = magicResult >> magicBitboards::magicNumberShiftBishop[kingPos];
615:        uint64_t kingRaysBishop = magicBitboards::magicMovesBishop[kingPos][arrayIndex];
616:        kingRaysBishop &= (board->getPieceBitboard(switchColour(board->nextColour), bishop) | board->getPieceBitboard(switchColour(board->nextColour), queen));
617:        while (kingRaysBishop)
618:        {
619:            uint64_t pinner = pop(kingRaysBishop);
620:            uint64_t pinnedPiece = inBetween(kingPos, bitScanForward(pinner)) & board->allPieces;
621:            if (pinnedPiece & pieces > 0 && bitSum(pinnedPiece) == bitSum(pinnedPiece & pieces)) return true;
622:        }
623:        */
624:        return false;
625:    }
626:
```

ChessEngineLibrary\moveGeneration.h

```
001:    #pragma once
002:    #include <vector>
003:    #include <array>
004:    #include <iostream>
005:
006:    #include "piece.h"
007:    #include "move.h"
008:    #include "board.h"
009:    #include "magicBitboards.h"
010:    #include "moveGenerationTables.h"
011:
012:    int searchForMoves(Board* board, std::array<Move,150>* moveList);
013:
014:    int generatePawnMoves(Board* board, std::array<Move, 150>* Movelist, uint64_t pinnedPieces, uint64_t pushMask, uint64_t captureMask, int arraySize);
015:    int generateKingMoves(Board* board, std::array<Move, 150>* Movelist, uint64_t friendlyPieces, uint64_t enemyPieces, uint64_t kingDangerSquares, int arraySize);
016:    int generateKnightMoves(Board * board, std::array<Move, 150>* Movelist, uint64_t friendlyPieces, uint64_t enemyPieces, uint64_t pinnedPieces, uint64_t pushMask, uint64_t captureMask, int arraySize);
017:    int generateRookMoves(Board* board, std::array<Move, 150>* Movelist, uint64_t friendlyPieces, uint64_t enemyPieces, uint64_t pinnedPieces, uint64_t pushMask, uint64_t captureMask, int arraySize);
018:    int generateBishopMoves(Board * board, std::array<Move, 150>* Movelist, uint64_t friendlyPieces, uint64_t enemyPieces, uint64_t pinnedPieces, uint64_t pushMask, uint64_t captureMask, int arraySize);
019:    int generateQueenMoves(Board * board, std::array<Move, 150>* Movelist, uint64_t friendlyPieces, uint64_t enemyPieces, uint64_t pinnedPieces, uint64_t pushMask, uint64_t captureMask, int arraySize);
020:    int generateCastlingMoves(Board * board, std::array<Move, 150>* Movelist, uint64_t friendlyPieces, uint64_t enemyPieces, uint64_t kingDangerSquares, int arraySize);
021:
022:    int addPawnMoves(int start, uint64_t quietMoves, uint64_t captureMoves, Board* board, std::array<Move, 150>* Movelist, int arraySize);
023:    int addMoves(int start, int end, pieceType piece, std::array<Move, 150>*, uint64_t enemyPieces, Board* board, int arraySize);
024:    int addPawnMovesPromotions(int from, int to, uint64_t move, MoveType type, Board* board, std::array<Move, 150>* Movelist, int arraySize);
025:
026:    uint64_t getPinnedPieces(Board* board);
027:    uint64_t generateLegalFilterForPinnedPiece(Board* board, uint64_t pinnedPiece);
028:    uint64_t getAttackers(Board* board, colours colour, uint64_t targetBitboard);
029:    bool isPinnedEnPassant(Board* board, uint64_t pieces);
030:
```

ChessEngineLibrary\moveGenerationTables.cpp

```
001:    #include "moveGenerationTables.h"
002:    #include "bitboard.h"
003:
004:    void setupMoveGen()
005:    {
006:            uint64_t currentPos, knightMoves, kingMoves;
007:            for (int x = 0; x < 64; x++)
008:            {
009:                    currentPos = (uint64_t)1 << x;
010:
011:                    knightMoves = 0;
012:                    knightMoves |= currentPos << 15 & ~fileH;
013:                    knightMoves |= currentPos << 17 & ~fileA;
014:                    knightMoves |= currentPos << 06 & ~fileH & ~fileG;
015:                    knightMoves |= currentPos << 10 & ~fileA & ~fileB;
016:                    knightMoves |= currentPos >> 10 & ~fileH & ~fileG;
017:                    knightMoves |= currentPos >> 06 & ~fileA & ~fileB;
018:                    knightMoves |= currentPos >> 17 & ~fileH;
019:                    knightMoves |= currentPos >> 15 & ~fileA;
020:                    knightMovesArray[x] = knightMoves;
021:
022:                    kingMoves = 0;
023:                    kingMoves |= currentPos << 8;
024:                    kingMoves |= currentPos << 9 & ~fileA;
025:                    kingMoves |= currentPos << 1 & ~fileA;
026:                    kingMoves |= currentPos >> 7 & ~fileA;
027:                    kingMoves |= currentPos >> 8;
028:                    kingMoves |= currentPos >> 9 & ~fileH;
029:                    kingMoves |= currentPos >> 1 & ~fileH;
030:                    kingMoves |= currentPos << 7 & ~fileH;
031:                    kingMovesArray[x] = kingMoves;
032:
033:                    pawnWhiteAttacksArray[x] = currentPos << 7 & ~fileH | currentPos << 9 & ~fileA;
034:                    pawnBlackAttacksArray[x] = currentPos >> 9 & ~fileH | currentPos >> 7 & ~fileA;
035:            }
036:    }
```

ChessEngineLibrary\moveGenerationTables.h

```
001:    #pragma once
002:    #include <stdint.h>
003:
004:    extern uint64_t knightMovesArray[64];
005:    extern uint64_t kingMovesArray[64];
006:    extern uint64_t pawnWhiteAttacksArray[64];
007:    extern uint64_t pawnBlackAttacksArray[64];
008:
009:    void setupMoveGen();
```

ChessEngineLibrary\moveOrdering.cpp

```
001:  #include "moveOrdering.h"
002:
003:  int moveRatingComparisonFunc(Move move1, Move move2)
004:  {
005:  	return move1.moveRating > move2.moveRating;
006:  }
007:
008:  void orderSearch(std::array<Move, 150>* moveList, Board* board, int arraySize, Move* TTMove, bool isBestMove, killerEntry killerMoves, std::array<std::array<std::array<Move, 64>, 64>, 2>* counterMoves, Move* prevMove, st
009:  {
010:  	std::vector<Move> killerMoveVector = killerMoves.getKillerMoves();
011:  	int MVVLVAScore;
012:  	int seeScore;
013:
014:  	//Move order scheme:
015:  	//1. Move from hash table (score 5000)
016:  	//2. Promotions (Score ~ 4000)
017:  	//3. Winning or equal captures (SEE >= 0) , (Score 3000 + SEE)
018:  	//4. Strong non-captures (killer move , then countermove heuristic), (Score 2500, 2400)
019:  	//5. Other non-captures sorted by history heuristic, (Score 2000 + history)
020:  	//6. Losing captures, ( Score 2000 - SEE penalty)
021:
022:  	for (int x = 0; x < arraySize; x++)
023:  	{
024:  		if ((*moveList)[x] == *TTMove)
025:  		{
026:  			(*moveList)[x].moveRating = 5000;
027:  		}
028:  		else
029:  		{
030:  			if ((*moveList)[x].capturedPiece != blank)
031:  			{
032:  				seeScore = SEE(&(*moveList)[x], board);
033:  			}
034:
035:  			if ((*moveList)[x].moveType == queenPromotion)
036:  				(*moveList)[x].moveRating = 4000;
037:  			else if ((*moveList)[x].moveType == rookPromotion)
038:  				(*moveList)[x].moveRating = 3999;
039:  			else if ((*moveList)[x].moveType == bishopPromotion)
040:  				(*moveList)[x].moveRating = 3998;
041:  			else if ((*moveList)[x].moveType == knightPromotion)
042:  				(*moveList)[x].moveRating = 3997;
043:  			else if ((*moveList)[x].capturedPiece != blank && seeScore >= 0)
044:  				(*moveList)[x].moveRating = 3000 + seeScore;
045:  			//If the move is in the killerMoveTable
046:  			else if (std::find(killerMoveVector.begin(), killerMoveVector.end(), (*moveList)[x]) != killerMoveVector.begin())
047:  				(*moveList)[x].moveRating = 2500;
048:  			//If the move is in the countermove table
049:  			else if (prevMove != nullptr && (*moveList)[x] == (*counterMoves)[board->nextColour][prevMove->from][prevMove->to])
050:  					(*moveList)[x].moveRating = 2400;
051:  			else if ((*moveList)[x].capturedPiece != blank)
052:  			{
053:  				(*moveList)[x].moveRating = 2000 + seeScore;
054:  			}
055:  			else
056:  			{
057:  				(*moveList)[x].moveRating = 2000 + (*historyMoves)[board->nextColour][(*moveList)[x].from][(*moveList)[x].to];
058:  			}
059:  		}
060:  	}
061:
062:  	std::sort(moveList->begin(), moveList->begin() + arraySize, moveRatingComparisonFunc);
063:  }
064:
065:  int orderQuiescentSearch(std::array<Move, 150>* moveList, Board * board, int arraySize)
066:  {
067:  	//Filters out non-capture moves
068:  	int counter = 0;
069:  	for (int x = 0; x < arraySize; x++)
070:  	{
071:  		if ((*moveList)[x].capturedPiece != blank)
072:  		{
073:  			if (counter != x) (*moveList)[counter] = (*moveList)[x];
074:  			counter++;
075:  		}
076:  	}
077:  	arraySize = counter;
078:
079:  	const int materialValues[6] = { 100,300,300,500,900,0 };
080:
081:  	for (int x = 0; x < arraySize; x++)
082:  	{
083:  		//When weaker pieces capture stronger pieces, the move is probably strong , no need to waste time in SEE
084:  		if (materialValues[(*moveList)[x].piece] < materialValues[(*moveList)[x].capturedPiece])
085:  			(*moveList)[x].moveRating = materialValues[(*moveList)[x].capturedPiece];
086:  		else
087:  			(*moveList)[x].moveRating = SEE(&(*moveList)[x], board);
088:  	}
089:
090:  	std::sort(moveList->begin(), moveList->begin() + arraySize, moveRatingComparisonFunc);
091:
092:  	return arraySize;
093:  }
094:  int getMVVLVAScore(Move* move)
095:  {
096:  	// order is[victim][attacker] , higher scores are better
097:  	int tradeScores[6][6] = {
098:  		{ 6, 5, 4, 3, 2, 1 },		//Pawn Captured   1
099:  		{ 12, 11, 10, 9, 8, 7 },	//Knight Captured 2
100:  		{ 18, 17, 16, 15, 14, 13 },	//Bishop Captured 3
101:  		{ 24, 23, 22, 21, 20, 19 },	//Rook Captured   4
102:  		{ 30, 29, 28, 27, 26, 25 },	//Queen Captured  5
103:  		{ 36, 35, 34, 33, 32, 31 } }; //King Captured   6
104:
105:  	return tradeScores[move->capturedPiece][move->piece];
106:  }
107:
108:  bool MVVLVAComparisonFunc(Move move1, Move move2)
109:  {
110:  	return getMVVLVAScore(&move1) > getMVVLVAScore(&move2);
```

```
111:  }
112:
113:  //Most valuable victim , least valuable attacker
114:  void MVVLVA(std::array<Move, 150>* moveList, Board * board, int arraySize)
115:  {
116:  	std::sort(moveList->begin(), moveList->begin() + arraySize, MVVLVAComparisonFunc);
117:  }
118:
119:  int SEE(Move * move, Board * board)
120:  {
121:  	if (move->capturedPiece == blank) return 0;
122:
123:  	const int materialValues[6] = { 100,300,300,500,900,0 };
124:
125:  	//Keeps track of all pieces (other than those that we have considered moved or captured)
126:  	uint64_t allPieces = board->allPieces & ~((uint64_t)1 << move->from);
127:
128:  	std::array<SEEPiece,20> whiteAttackers;
129:  	std::array<SEEPiece, 20> blackAttackers;
130:  	int whiteAttackersArraySize = 0;
131:  	int blackAttackersArraySize = 0;
132:
133:  	//The total score off all piece captured
134:  	int score = materialValues[move->capturedPiece];
135:  	//The value of the current piece in the target square
136:  	int tempScore = materialValues[move->piece];
137:
138:  	colours currentColour = switchColour(board->nextColour);
139:
140:  	///Adds all the piece that can attack the target square
141:  	uint64_t whitePawnBitboard = pawnBlackAttacksArray[move->to] & board->getPieceBitboard(white, pawn) & allPieces;
142:  	while (whitePawnBitboard)
143:  	{
144:  		whiteAttackers[whiteAttackersArraySize] = SEEPiece(pop(whitePawnBitboard), pawn);
145:  		whiteAttackersArraySize++;
146:  	}
147:
148:  	uint64_t blackPawnBitboard = pawnWhiteAttacksArray[move->to] & board->getPieceBitboard(black, pawn) & allPieces;
149:  	while (blackPawnBitboard)
150:  	{
151:  		blackAttackers[blackAttackersArraySize] = SEEPiece(pop(blackPawnBitboard), pawn);
152:  		blackAttackersArraySize++;
153:  	}
154:
155:  	uint64_t whiteKnightBitboard = knightMovesArray[move->to] & board->getPieceBitboard(white, knight) & allPieces;
156:  	while (whiteKnightBitboard)
157:  	{
158:  		whiteAttackers[whiteAttackersArraySize] = SEEPiece(pop(whiteKnightBitboard), knight);
159:  		whiteAttackersArraySize++;
160:  	}
161:
162:  	uint64_t blackKnightBitboard = knightMovesArray[move->to] & board->getPieceBitboard(black, knight) & allPieces;
163:  	while (blackKnightBitboard)
164:  	{
165:  		blackAttackers[blackAttackersArraySize] = SEEPiece(pop(blackKnightBitboard), knight);
166:  		blackAttackersArraySize++;
167:  	}
168:
169:  	uint64_t whiteKingBitboard = kingMovesArray[move->to] & board->getPieceBitboard(white, king) & allPieces;
170:  	while (whiteKingBitboard)
171:  	{
172:  		whiteAttackers[whiteAttackersArraySize] = SEEPiece(pop(whiteKingBitboard), king);
173:  		whiteAttackersArraySize++;
174:  	}
175:
176:  	uint64_t blackKingBitboard = kingMovesArray[move->to] & board->getPieceBitboard(black, king) & allPieces;
177:  	while (blackKingBitboard)
178:  	{
179:  		blackAttackers[blackAttackersArraySize] = SEEPiece(pop(blackKingBitboard), king);
180:  		blackAttackersArraySize++;
181:  	}
182:
183:  	SEEAddSliders(&whiteAttackers, &blackAttackers, board, move->to, allPieces, &whiteAttackersArraySize, &blackAttackersArraySize);
184:
185:  	while (true)
186:  	{
187:  		std::array<SEEPiece, 20>* currentAttackers = (currentColour == white) ? &whiteAttackers : &blackAttackers;
188:  		int* currentAttackersSize = (currentColour == white) ? &whiteAttackersArraySize : &blackAttackersArraySize;
189:
190:  		if (*currentAttackersSize == 0) break;
191:
192:  		const SEEPiece smallestPiece = getLeastValuablePiece(currentAttackers, currentAttackersSize);
193:
194:  		if (currentColour == board->nextColour) score += tempScore;
195:  		else score -= tempScore;
196:
197:  		tempScore = materialValues[smallestPiece.type];
198:  		allPieces &= ~smallestPiece.pieceBitboard;
199:  		SEEAddSliders(&whiteAttackers, &blackAttackers, board, move->to, allPieces, &whiteAttackersArraySize, &blackAttackersArraySize);
200:
201:  		currentColour = switchColour(currentColour);
202:  	}
203:
204:  	return score;
205:  }
206:
207:  //Adds any sliding pieces that can attack the target square to the respective vectors
208:  void SEEAddSliders(std::array<SEEPiece,20>* whiteAttackers, std::array<SEEPiece,20>* blackAttackers, Board* board, int targetSquare, uint64_t occupancyMask, int* whiteAttackersArraySize, int* blackAttackersArraySize)
209:  {
210:  	const int pieceBitboard = (uint64_t)1 << targetSquare;
211:
212:  	uint64_t occupancy = magicBitboards::rookMask[targetSquare] & occupancyMask;
213:  	uint64_t magicResult = occupancy * magicBitboards::magicNumberRook[targetSquare];
214:  	int arrayIndex = magicResult >> magicBitboards::magicNumberShiftRook[targetSquare];
215:  	uint64_t horizontalSliderPositionsBitboard = magicBitboards::magicMovesRook[targetSquare][arrayIndex] & occupancyMask;
216:
217:  	uint64_t whiteRookPositionsBitboad = horizontalSliderPositionsBitboard & board->getPieceBitboard(white, rook);
218:  	while(whiteRookPositionsBitboad)
219:  	{
220:  		const SEEPiece seePiece = SEEPiece(pop(whiteRookPositionsBitboad),rook);
221:  		if (std::find(whiteAttackers->begin(), whiteAttackers->begin() + *whiteAttackersArraySize, seePiece) == whiteAttackers->begin() + *whiteAttackersArraySize)
222:  		{
223:  			(*whiteAttackers)[*whiteAttackersArraySize] = seePiece;
224:  			(*whiteAttackersArraySize)++;
```

```
225:                    }
226:            }
227:            uint64_t blackRookPositionsBitboad = horizontalSliderPositionsBitboard & board->getPieceBitboard(black, rook);
228:            while (blackRookPositionsBitboad)
229:            {
230:                    const SEEPiece seePiece = SEEPiece(pop(blackRookPositionsBitboad), rook);
231:                    if (std::find(blackAttackers->begin(), blackAttackers->begin() + *blackAttackersArraySize, seePiece) == blackAttackers->begin() + *blackAttackersArraySize)
232:                    {
233:                            (*blackAttackers)[*blackAttackersArraySize] = seePiece;
234:                            (*blackAttackersArraySize)++;
235:                    }
236:            }
237:            uint64_t whiteQueenPositionsBitboad = horizontalSliderPositionsBitboard & board->getPieceBitboard(white, queen);
238:            while (whiteQueenPositionsBitboad)
239:            {
240:                    const SEEPiece seePiece = SEEPiece(pop(whiteQueenPositionsBitboad), queen);
241:                    if (std::find(whiteAttackers->begin(), whiteAttackers->begin() + *whiteAttackersArraySize, seePiece) == whiteAttackers->begin() + *whiteAttackersArraySize)
242:                    {
243:                            (*whiteAttackers)[*whiteAttackersArraySize] = seePiece;
244:                            (*whiteAttackersArraySize)++;
245:                    }
246:            }
247:            uint64_t blackQueenPositionsBitboad = horizontalSliderPositionsBitboard & board->getPieceBitboard(black, queen);
248:            while (blackQueenPositionsBitboad)
249:            {
250:                    const SEEPiece seePiece = SEEPiece(pop(blackQueenPositionsBitboad), queen);
251:                    if (std::find(blackAttackers->begin(), blackAttackers->begin() + *blackAttackersArraySize, seePiece) == blackAttackers->begin() + *blackAttackersArraySize)
252:                    {
253:                            (*blackAttackers)[*blackAttackersArraySize] = seePiece;
254:                            (*blackAttackersArraySize)++;
255:                    }
256:            }
257:
258:            occupancy = magicBitboards::bishopMask[targetSquare] & occupancyMask;
259:            magicResult = occupancy * magicBitboards::magicNumberBishop[targetSquare];
260:            arrayIndex = magicResult >> magicBitboards::magicNumberShiftBishop[targetSquare];
261:            const uint64_t diagonalSliderPositionsBitboard = magicBitboards::magicMovesBishop[targetSquare][arrayIndex] & occupancyMask;
262:
263:            uint64_t whiteBishopPositionsBitboad = diagonalSliderPositionsBitboard& board->getPieceBitboard(white, bishop);
264:            while (whiteBishopPositionsBitboad)
265:            {
266:                    const SEEPiece seePiece = SEEPiece(pop(whiteBishopPositionsBitboad), bishop);
267:                    if (std::find(whiteAttackers->begin(), whiteAttackers->begin() + *whiteAttackersArraySize, seePiece) == whiteAttackers->begin() + *whiteAttackersArraySize)
268:                    {
269:                            (*whiteAttackers)[*whiteAttackersArraySize] = seePiece;
270:                            (*whiteAttackersArraySize)++;
271:                    }
272:            }
273:            uint64_t blackBishopPositionsBitboad = diagonalSliderPositionsBitboard& board->getPieceBitboard(black, bishop);
274:            while (blackBishopPositionsBitboad)
275:            {
276:                    const SEEPiece seePiece = SEEPiece(pop(blackBishopPositionsBitboad), bishop);
277:                    if (std::find(blackAttackers->begin(), blackAttackers->begin() + *blackAttackersArraySize, seePiece) == blackAttackers->begin() + *blackAttackersArraySize)
278:                    {
279:                            (*blackAttackers)[*blackAttackersArraySize] = seePiece;
280:                            (*blackAttackersArraySize)++;
281:                    }
282:            }
283:            whiteQueenPositionsBitboad = diagonalSliderPositionsBitboard& board->getPieceBitboard(white, queen);
284:            while (whiteQueenPositionsBitboad)
285:            {
286:                    const SEEPiece seePiece = SEEPiece(pop(whiteQueenPositionsBitboad), queen);
287:                    if (std::find(whiteAttackers->begin(), whiteAttackers->begin() + *whiteAttackersArraySize, seePiece) == whiteAttackers->begin() + *whiteAttackersArraySize)
288:                    {
289:                            (*whiteAttackers)[*whiteAttackersArraySize] = seePiece;
290:                            (*whiteAttackersArraySize)++;
291:                    }
292:            }
293:            blackQueenPositionsBitboad = diagonalSliderPositionsBitboard& board->getPieceBitboard(black, queen);
294:            while (blackQueenPositionsBitboad)
295:            {
296:                    const SEEPiece seePiece = SEEPiece(pop(blackQueenPositionsBitboad), queen);
297:                    if (std::find(blackAttackers->begin(), blackAttackers->begin() + *blackAttackersArraySize, seePiece) == blackAttackers->begin() + *blackAttackersArraySize)
298:                    {
299:                            (*blackAttackers)[*blackAttackersArraySize] = seePiece;
300:                            (*blackAttackersArraySize)++;
301:                    }
302:            }
303:    }
304:
305:    SEEPiece getLeastValuablePiece(std::array<SEEPiece, 20>* attackers, int* arraySize)
306:    {
307:            const int materialValues[6] = { 100,300,300,500,900,0 };
308:            SEEPiece min;
309:            int minMaterial = 9999;
310:            int minPos;
311:            for (int x = 0; x < *arraySize; x++)
312:            {
313:                    if (materialValues[(*attackers)[x].type] < minMaterial)
314:                    {
315:                            minPos = x;
316:                            min = (*attackers)[x];
317:                            minMaterial = materialValues[(*attackers)[x].type];
318:                    }
319:            }
320:
321:            (*attackers)[minPos] = (*attackers)[*arraySize - 1];
322:            (*arraySize)--;
323:
324:            return min;
325:    }
326:
327:    killerEntry::killerEntry()
328:    {
329:            numKillers = 3;
330:            killerMoves.reserve(numKillers);
331:    }
332:
333:    void killerEntry::addKillerMove(Move move)
334:    {
335:            if (killerMoves.size() < numKillers)
336:            {
337:                    killerMoves.push_back(move);
338:            }
```

```
339:    }
340:
341:    SEEPiece::SEEPiece(uint64_t newPieceBitboard, pieceType newPieceType)
342:    {
343:            pieceBitboard = newPieceBitboard;
344:            type = newPieceType;
345:    }
346:
```

ChessEngineLibrary\moveOrdering.h

```
001:  #pragma once
002:  #include <vector>
003:  #include <algorithm>
004:
005:  #include <array>
006:  #include "move.h"
007:  #include "moveGenerationTables.h"
008:  #include "board.h"
009:
010:  class killerEntry
011:  {
012:  public:
013:          killerEntry();
014:          std::vector<Move> getKillerMoves() { return killerMoves; };
015:          void addKillerMove(Move move);
016:
017:  private:
018:          std::vector<Move> killerMoves;
019:          int numKillers;
020:  };
021:
022:  struct SEEPiece
023:  {
024:          SEEPiece() {};
025:          SEEPiece(uint64_t newPieceBitboard, pieceType newPieceType);
026:
027:          uint64_t pieceBitboard;
028:          pieceType type;
029:
030:          bool operator==(const SEEPiece& b)
031:          {
032:                  return (b.pieceBitboard == pieceBitboard) && (b.type == type);
033:          }
034:  };
035:
036:  void orderSearch(std::array<Move, 150>* moveList, Board* board, int arraySize, Move* TTMove, bool isBestMove, killerEntry killerMoves, std::array<std::array<std::array<Move, 64>, 64>, 2>* counterMoves, Move* prevMove, st
037:  int orderQuiescentSearch(std::array<Move, 150>* moveList, Board * board, int arraySize);
038:
039:  int getMVVLVAScore(Move* move);
040:  bool MVVLVAComparisonFunc(Move move1, Move move2);
041:  void MVVLVA(std::array<Move, 150>* moveList, Board * board, int arraySize);
042:
043:  int SEE(Move* move, Board* board);
044:  void SEEAddSliders(std::array<SEEPiece, 20>* whiteAttackers, std::array<SEEPiece, 20>* blackAttackers, Board* board, int targetSquare, uint64_t occupancyMask, int* whiteAttackersArraySize, int* blackAttackersArraySize);
045:  SEEPiece getLeastValuablePiece(std::array<SEEPiece, 20>* attackers, int* arraySize);
046:
047:
```

ChessEngineLibrary\piece.cpp

```
001:    #include "piece.h"
002:
003:    colours switchColour(colours colour)
004:    {
005:            switch (colour)
006:            {
007:            case white:
008:                    return black;
009:            case black:
010:                    return white;
011:            }
012:    }
013:
014:
```

ChessEngineLibrary\piece.h

```
001:    #pragma once
002:    #include <string>
003:    #include <iostream>
004:
005:    enum pieceType { pawn, knight, bishop, rook, queen, king, blank };
006:    enum colours : bool { white = true, black = false };
007:
008:    colours switchColour(colours colour);
009:
```

ChessEngineLibrary\pieceSquare.cpp

```
001:  #include "pieceSquare.h"
002:
003:
004:  pieceSquare::pieceSquare()
005:  {
006:  }
007:
008:  pieceSquare::pieceSquare(std::string filename, pieceType typeNew, colours defaultColourNew)
009:  {
010:          defaultColour = defaultColourNew;
011:          type = typeNew;
012:          loadFromFile(filename);
013:  }
014:
015:  void pieceSquare::loadFromFile(std::string filename)
016:  {
017:          std::ifstream tableFile;
018:          tableFile.open(filename);
019:          std::string line;
020:          int lineNum = 0;
021:
022:          if (tableFile.is_open())
023:          {
024:                  while (std::getline(tableFile, line))
025:                  {
026:                          int colNum = 0;
027:                          std::string temp = "";
028:                          for (int counter = 0; counter < line.size(); counter++)
029:                          {
030:                                  if (line[counter] != ',')
031:                                  {
032:                                          temp += line[counter];
033:                                  }
034:                                  else
035:                                  {
036:                                          square[colNum][lineNum] = std::stoi(temp);
037:                                          temp = "";
038:                                          colNum++;
039:                                  }
040:                          }
041:                          lineNum++;
042:                  }
043:                  tableFile.close();
044:          }
045:          else
046:          {
047:                  std::cout << "Failed to find " + filename + "\n";
048:          }
049:
050:  }
051:
052:  int pieceSquare::calcScore(uint64_t bitboard,colours targetColour)
053:  {
054:
055:          int score = 0;
056:          int bitPos;
057:          while (bitboard)
058:          {
059:                  bitPos = bitScanForward(pop(bitboard));
060:                  score += getScoreFromPos(bitPos, targetColour);
061:          }
062:          return score;
063:  }
064:
065:  int pieceSquare::getScoreFromPos(int pos, colours targetColour)
066:  {
067:          int x = pos % 8;
068:          int y = (pos - x) / 8;
069:          if (targetColour == defaultColour)
070:          {
071:                  y = 7 - y;
072:          }
073:          return square[x][y];
074:  }
075:
076:  pieceSquare pieceSquareData::pawnSquare = pieceSquare("WPSquareTable.txt", pawn, white);
077:  pieceSquare pieceSquareData::knightSquare = pieceSquare("WNSquareTable.txt", knight, white);
078:  pieceSquare pieceSquareData::bishopSquare = pieceSquare("WBSquareTable.txt", bishop, white);
079:  pieceSquare pieceSquareData::midGameKingSquare = pieceSquare("WKMiddleSquareTable.txt", king, white);
080:  pieceSquare pieceSquareData::lateGameKingSquare = pieceSquare("WKEndSquareTable.txt", king, white);
081:
082:
```

ChessEngineLibrary\pieceSquare.h

```
001:  #pragma once
002:  #include <string>
003:  #include <iostream>
004:  #include <fstream>
005:  #include <iostream>
006:  #include <array>
007:  #include <stdint.h>
008:
009:  #include "piece.h"
010:  #include "bitboard.h"
011:
012:
013:  class pieceSquare
014:  {
015:  public:
016:          pieceSquare();
017:          pieceSquare(std::string filename , pieceType typeNew , colours defaultColourNew);
018:          void loadFromFile(std::string filename);
019:
020:          int calcScore(uint64_t bitboard, colours targetColour);
021:          int getScoreFromPos(int pos, colours targetColour);
022:
023:          int square[8][8];
024:          colours defaultColour;
025:          pieceType type;
026:  };
027:
028:  struct pieceSquareData
029:  {
030:          static pieceSquare pawnSquare, knightSquare, bishopSquare, rookSquare, queenSquare, midGameKingSquare, lateGameKingSquare;
031:  };
032:
```

ChessEngineLibrary\scoring.cpp

```
001:  #include "scoring.h"
002:
003:  //Pawn structure hash table
004:  PawnStructureTableEntry pawnHashTable[2048];
005:
006:  int calculateScoreDiff(Board* board)
007:  {
008:  	int score = calculatePawnStructureScore(board);
009:  	score += calculateMaterialScore(board);
010:  	score += calculateRookPositionScore(board);
011:  	score += calculateKingSafetyScore(board);
012:
013:  	return score;
014:  }
015:
016:  int calculatePawnStructureScore(Board* board)
017:  {
018:  	//If this pawn structure has already been calculated.
019:  	if (pawnHashTable[board->pawnScoreZorbistKey % 2048].zorbistKey == board->pawnScoreZorbistKey && board->pawnScoreZorbistKey != 0)
020:  	{
021:  		//std::cout << "\n\nmatch found\n\n";
022:  		//board->printBoard();
023:  		//std::cout << "\n\n";
024:  		//pawnHashTable[board->pawnScoreZorbistKey % 2048].prevBoard.printBoard();
025:  		//std::cout << "\n\n" << pawnHashTable[board->pawnScoreZorbistKey % 2048].score << "\n\n";
026:
027:  		if (board->nextColour == white) return pawnHashTable[board->pawnScoreZorbistKey % 2048].score;
028:  		else return -pawnHashTable[board->pawnScoreZorbistKey % 2048].score;
029:  	}
030:
031:  	int blackScore = 0;
032:  	int whiteScore = 0;
033:  	const uint64_t fileMasks[8] = { fileA, fileB, fileC, fileD, fileE, fileF, fileG, fileH };
034:
035:  	//Checks for double and triple pawns.
036:  	for (int x = 0; x < 8; x++)
037:  	{
038:  		int pawnsInRank = bitSum(board->getPieceBitboard(white,pawn) & fileMasks[x]);
039:  		if (pawnsInRank > 1)
040:  		{
041:  			whiteScore -= 10 * (pawnsInRank - 1);
042:  		}
043:
044:  		pawnsInRank = bitSum(board->getPieceBitboard(black,pawn) & fileMasks[x]);
045:  		if (pawnsInRank > 1)
046:  		{
047:  			blackScore -= 10 * (pawnsInRank - 1);
048:  		}
049:  	}
050:
051:  	uint64_t currentPawn;
052:  	int currentPos;
053:
054:  	const uint64_t whitePassedPawnMasks[8] = { 0xffffffffffffff00, 0xfffffffffffff0000, 0xffffffffff000000,
055:  		0xfffffff00000000, 0xffffff0000000000, 0xffff000000000000, 0xff00000000000000, 0 };
056:  	const uint64_t neighbouringFileMasks[8] = { fileB, fileA + fileC, fileB + fileD, fileC + fileE, fileD + fileF, fileE + fileG, fileF + fileH, fileG };
057:  	const uint64_t whiteBackwardsPawnMasks[8] = { 0xff, 0xffff, 0xffffff, 0xffffffff, 0xffffffffff,
058:  		0xffffffffffff, 0xffffffffffff, 0xffffffffffffff };
059:  	const uint64_t blackPassedPawnMasks[8] = { 0, 0xff, 0xffff, 0xffffff, 0xffffffff, 0xffffffffff,
060:  		0xffffffffffff, 0xffffffffffffff};
061:  	const uint64_t blackBackwardsPawnMasks[8] = { 0xffffffffffffffff, 0xffffffffffffff00, 0xffffffffffff0000, 0xffffffffff000000,
062:  		0xfffffff00000000,  0xffffff0000000000, 0xffff000000000000, 0xff00000000000000 };
063:
064:  	uint64_t whitePawnBitboard = board->getPieceBitboard(white,pawn);
065:  	while (whitePawnBitboard)
066:  	{
067:  		currentPawn = pop(whitePawnBitboard);
068:  		currentPos = bitScanForward(currentPawn);
069:
070:  		//Passed pawns
071:  		if ((board->getPieceBitboard(black,pawn) & whitePassedPawnMasks[currentPos / 8] & (neighbouringFileMasks[currentPos % 8] | fileMasks[currentPos % 8])) == 0)
072:  		{
073:  			whiteScore += 20 * (currentPos / 8);
074:  		}
075:
076:  		//Isolated Pawns
077:  		if ((board->getPieceBitboard(white,pawn) & neighbouringFileMasks[currentPos % 8]) == 0)
078:  		{
079:  			whiteScore -= 20;
080:  		}
081:  		//Backwards Pawns
082:  		else if((board->getPieceBitboard(white,pawn) & neighbouringFileMasks[currentPos % 8] & whiteBackwardsPawnMasks[currentPos / 8]) == 0)
083:  		{
084:  			whiteScore -= 8;
085:  		}
086:  	}
087:
088:  	uint64_t blackPawnBitboard = board->getPieceBitboard(black,pawn);
089:  	while (blackPawnBitboard)
090:  	{
091:  		currentPawn = pop(blackPawnBitboard);
092:  		currentPos = bitScanForward(currentPawn);
093:
094:  		//Passed pawns
095:
096:  		if ((board->getPieceBitboard(white,pawn) & blackPassedPawnMasks[currentPos / 8] & (neighbouringFileMasks[currentPos % 8] | fileMasks[currentPos % 8])) == 0)
097:  		{
098:  			blackScore += 20 * (7 - (currentPos / 8));
099:  		}
100:
101:  		//Isolated Pawns
102:  		if ((board->getPieceBitboard(black,pawn) & neighbouringFileMasks[currentPos % 8]) == 0)
103:  		{
104:  			blackScore -= 20;
105:  		}
106:  		//Backwards Pawns
107:  		else if ((board->getPieceBitboard(black,pawn) & neighbouringFileMasks[currentPos % 8] & blackBackwardsPawnMasks[currentPos / 8]) == 0)
108:  		{
109:  			blackScore -= 8;
110:  		}
```

```
111:            }
112:
113:            //Caches results into hash table.
114:            pawnHashTable[board->pawnScoreZorbistKey % 2048].zorbistKey = board->pawnScoreZorbistKey;
115:            pawnHashTable[board->pawnScoreZorbistKey % 2048].score = whiteScore - blackScore;
116:            //pawnHashTable[board->pawnScoreZorbistKey % 2048].prevBoard = *board;
117:
118:            if (board->nextColour == black)
119:            {
120:                    return blackScore - whiteScore;
121:            }
122:            else
123:            {
124:                    return whiteScore - blackScore;
125:            }
126:    }
127:
128:    int calculateRookPositionScore(Board * board)
129:    {
130:            int whiteScore = 0;
131:            int blackScore = 0;
132:
133:            uint64_t currentRook;
134:            const uint64_t fileMasks[8] = { fileA, fileB, fileC, fileD, fileE, fileF, fileG, fileH };
135:            int currentPos;
136:
137:            uint64_t whiteRookBitboard = board->getPieceBitboard(white, rook);
138:            while (whiteRookBitboard)
139:            {
140:                    currentRook = pop(whiteRookBitboard);
141:                    currentPos = bitScanForward(currentRook);
142:
143:                    //If the file of the rook contains no pawns.
144:                    if (((board->getPieceBitboard(white,pawn) | board->getPieceBitboard(black,pawn)) & fileMasks[currentPos % 8]) == 0)
145:                    {
146:                            whiteScore += 15;
147:                    }
148:                    //If the file has black pawns but no white pawns.
149:                    else if ((board->getPieceBitboard(white,pawn) & fileMasks[currentPos % 8]) == 0)
150:                    {
151:                            whiteScore += 10;
152:                    }
153:
154:                    if ((currentPos / 8) == 6)
155:                    {
156:                            whiteScore += 20;
157:                    }
158:            }
159:
160:            uint64_t blackRookBitboard = board->getPieceBitboard(black, rook);
161:            while (blackRookBitboard)
162:            {
163:                    currentRook = pop(blackRookBitboard);
164:                    currentPos = bitScanForward(currentRook);
165:
166:                    //If the file of the rook contains no pawns.
167:                    if (((board->getPieceBitboard(white,pawn) | board->getPieceBitboard(black,pawn)) & fileMasks[currentPos % 8]) == 0)
168:                    {
169:                            blackScore += 15;
170:                    }
171:                    //If the file has white pawns but no black pawns.
172:                    else if ((board->getPieceBitboard(black,pawn) & fileMasks[currentPos % 8]) == 0)
173:                    {
174:                            blackScore += 10;
175:                    }
176:
177:                    if ((currentPos / 8) == 1)
178:                    {
179:                            blackScore += 20;
180:                    }
181:            }
182:
183:            if (board->nextColour == black)
184:            {
185:                    return blackScore - whiteScore;
186:            }
187:            else
188:            {
189:                    return whiteScore - blackScore;
190:            }
191:    }
192:
193:    int calculateMaterialScore(Board * board)
194:    {
195:            int whiteScore = board->getPositionalScore(white) + board->getMaterialScore(white);
196:            //If late game
197:            if (board->getOnlyMaterialScore(black) <= 1200)
198:            {
199:                    whiteScore += board->getLateGameKingPositionalScore(white);
200:            }
201:
202:            int blackScore = 0;
203:            //If late game
204:            if (board->getOnlyMaterialScore(white) <= 1200)
205:            {
206:                    blackScore = board->getLateGameKingPositionalScore(black);
207:            }
208:
209:            if (board->nextColour == black)
210:            {
211:                    return blackScore - whiteScore;
212:            }
213:            else
214:            {
215:                    return whiteScore - blackScore;
216:            }
217:    }
218:
219:    int calculateKingSafetyScore(Board * board)
220:    {
221:            int score = 0;
222:            //If midgame for white
223:            if (board->getOnlyMaterialScore(black) > 1200)
224:            {
```

```
225:                    const float whiteMultiplier = ((float)board->getOnlyMaterialScore(black) / 3100.0);
226:                    score += whiteMultiplier * (float)(calculateKingSafetyScoreForColour(board, white) + board->getMidGameKingPositionalScore(white));
227:            }
228:            //If midgame for black
229:            if (board->getOnlyMaterialScore(white) > 1200)
230:            {
231:                    const float blackMultiplier = ((float)board->getOnlyMaterialScore(white) / 3100.0);
232:                    score -= blackMultiplier * (float)(calculateKingSafetyScoreForColour(board, black) + board->getMidGameKingPositionalScore(black));
233:            }
234:
235:            if (board->nextColour == white) return score;
236:            else return -score;
237:    }
238:
239:    int calculateKingSafetyScoreForColour(Board * board, colours colour)
240:    {
241:            const uint64_t fileMasks[8] = { fileA, fileB, fileC, fileD, fileE, fileF, fileG, fileH };
242:            const uint64_t startingRankMask = colour == white ? rank2 : rank7;
243:            const uint64_t movedOnceMask = colour == white ? rank3 : rank6;
244:            const uint64_t movedTwiceMask = colour == white ? rank4 : rank5;
245:
246:            int score = 0;
247:            const uint64_t kingBitboard = board->getPieceBitboard(colour, king);
248:            const uint64_t pawnBitboard = board->getPieceBitboard(colour, pawn);
249:            const uint64_t enemyPawnBitboard = board->getPieceBitboard(switchColour(colour), pawn);
250:
251:            //If the king is in the three left hand files.
252:            if (kingBitboard & 506381209866536711)
253:            {
254:                    //Iterate through the three files.
255:                    for (int x = 0; x < 3; x++)
256:                    {
257:                            uint64_t fileMask = fileMasks[x];
258:
259:                            //Half the scores for files 2 and 5
260:                            const float scoreMultiplier = x != 2 ? 1 : 0.5;
261:                            int fileScore = 0;
262:
263:                            //File is not open
264:                            if (pawnBitboard & fileMask)
265:                            {
266:                                    //If the pawn has moved
267:                                    if ((pawnBitboard & fileMask & startingRankMask) == 0)
268:                                    {
269:                                            //if the pawn has moved more than once, give a penalty
270:                                            if ((pawnBitboard & fileMask & movedOnceMask)  == 0)
271:                                            {
272:                                                    fileScore -= 20;
273:                                            }
274:                                            else
275:                                            {
276:                                                    fileScore -= 10;
277:                                            }
278:                                    }
279:                            }
280:                            //Give a penalty for an open file.
281:                            else
282:                            {
283:                                    fileScore -= 25;
284:
285:                            }
286:
287:                            //Give a penalty for their being no enemy pawns on the file. Semi-open
288:                            if ((enemyPawnBitboard & fileMask) == 0)
289:                            {
290:                                    fileScore -= 15;
291:                            }
292:                            else
293:                            {
294:                                    //If the pawn is in front of your starting rank
295:                                    if (enemyPawnBitboard & movedOnceMask & fileMask)
296:                                    {
297:                                            fileScore -= 10;
298:                                    }
299:                                    //If the pawn is on your front rank.
300:                                    else if (enemyPawnBitboard & movedTwiceMask & fileMask)
301:                                    {
302:                                            fileScore -= 5;
303:                                    }
304:                            }
305:
306:                            score += fileScore * scoreMultiplier;
307:                    }
308:
309:            }
310:            //If the king is in the three right hand files.
311:            else if (kingBitboard & 16204198715729174752)
312:            {
313:                    //Iterate through the three files.
314:                    for (int x = 5; x < 8; x++)
315:                    {
316:                            uint64_t fileMask = fileMasks[x];
317:
318:                            //Half the scores for files 2 and 5
319:                            const float scoreMultiplier = x != 5 ? 1 : 0.5;
320:                            int fileScore = 0;
321:
322:                            //File is not open
323:                            if (pawnBitboard & fileMask)
324:                            {
325:                                    //If the pawn has moved
326:                                    if ((pawnBitboard & fileMask & startingRankMask) == 0)
327:                                    {
328:                                            //if the pawn has moved more than once, give a penalty
329:                                            if ((pawnBitboard & fileMask & movedOnceMask) == 0)
330:                                            {
331:                                                    fileScore -= 20;
332:                                            }
333:                                            else
334:                                            {
335:                                                    fileScore -= 10;
336:                                            }
337:                                    }
338:                            }
```

```
339:                        //Give a penalty for an open file.
340:                        else
341:                        {
342:                                fileScore -= 25;
343:
344:                        }
345:
346:                        //Give a penalty for their being no enemy pawns on the file. Semi-open
347:                        if ((enemyPawnBitboard & fileMask) == 0)
348:                        {
349:                                fileScore -= 15;
350:                        }
351:                        else
352:                        {
353:                                //If the pawn is in front of your starting rank
354:                                if (enemyPawnBitboard & movedOnceMask & fileMask)
355:                                {
356:                                        fileScore -= 10;
357:                                }
358:                                //If the pawn is on your front rank.
359:                                else if (enemyPawnBitboard & movedTwiceMask & fileMask)
360:                                {
361:                                        fileScore -= 5;
362:                                }
363:                        }
364:                        score += fileScore * scoreMultiplier;
365:                }
366:        }
367:
368:        //King in the middle two files.
369:        else
370:        {
371:                const int kingFile = bitScanForward(kingBitboard) % 8;
372:
373:                //Iterate through the three files on and adjacent to the king.
374:                for (int x = kingFile - 1; x < kingFile + 2; x++)
375:                {
376:                        uint64_t fileMask = fileMasks[x];
377:
378:                        //If the file is fully open
379:                        if ((fileMask & (board->getPieceBitboard(white,pawn) | board->getPieceBitboard(black,pawn))) == 0)
380:                        {
381:                                score -= 10;
382:                        }
383:                }
384:        }
385:        return score;
386:    }
```

ChessEngineLibrary\scoring.h

```
001:    #pragma once
002:    #include "pieceSquare.h"
003:    #include "board.h"
004:
005:    int calculateScoreDiff(Board* board);
006:
007:    int calculatePawnStructureScore(Board* board);
008:    int calculateRookPositionScore(Board* board);
009:    int calculateMaterialScore(Board * board);
010:    int calculateKingSafetyScore(Board* board);
011:    int calculateKingSafetyScoreForColour(Board * board, colours colour);
012:
013:    struct PawnStructureTableEntry
014:    {
015:    public:
016:            uint64_t zorbistKey;
017:            //Board prevBoard;
018:
019:            //The score, from whites perspective.
020:            int score;
021:    };
022:
023:
024:
```

ChessEngineLibrary\search.cpp

```
001:  #include "search.h"
002:
003:  void updateUI(searchData * data, Move currentMove, int currentMoveNumber, std::string pvLine)
004:  {
005:  	std::cout << "info depth " << data->depth;
006:  	std::cout << " nodes " << data->nodes;
007:  	std::cout << " nps " << (uint64_t)((data->nodes) / difftime(time(NULL), data->startTime));
008:  	std::cout << " pv " << pvLine;
009:  	std::cout << " currmove " << notationFromMove(currentMove);
010:  	std::cout << " currmovenumber " << currentMoveNumber << "\n";
011:  }
012:
013:  void finalUIUpdate(searchData * data, std::string pvLine)
014:  {
015:  	std::cout << "info depth " << data->depth;
016:  	std::cout << " nodes " << data->nodes;
017:  	std::cout << " nps " << (uint64_t)((data->nodes) / difftime(time(NULL), data->startTime));
018:  	std::cout << " pv " << pvLine << "\n";
019:  }
020:
021:  Move startSearch(Board* board, TranspositionEntry* transpositionTable, timeManagement* timer)
022:  {
023:  	searchData data;
024:  	data.startTime = time(0);
025:  	data.nodes = 0;
026:
027:  	//Tables used for history and counter move heuristics.
028:  	std::array<std::array<std::array<Move, 64>, 64>, 2> counterMoves;
029:  	std::array<std::array<std::array<int, 64>, 64>, 2> historyMoves = {};
030:
031:  	int score = 0;
032:
033:  	PVData bestMove;
034:  	for (int x = 1; x <= 50; x++)
035:  	{
036:  		data.depth = x;
037:  		std::vector<killerEntry>* killerMoveTable = new std::vector<killerEntry>();
038:  		killerMoveTable->resize(x + 1);
039:
040:  		if (x == 1)
041:  		{
042:  			score = negascout(-30000, 30000, x, board, &data, transpositionTable, killerMoveTable, &counterMoves, nullptr, &historyMoves);
043:  		}
044:  		else
045:  		{
046:  			//Search with a narrow (half a pawn width) aspiration window.
047:  			int alpha = score - 25;
048:  			int beta = score + 25;
049:  			score = negascout(alpha, beta, x, board, &data, transpositionTable, killerMoveTable, &counterMoves, nullptr, &historyMoves);
050:
051:  			//Score outside range , therefore full re-search needed
052:  			if (score <= alpha || score >= beta)
053:  			{
054:  				score = negascout(-30000, 30000, x, board, &data, transpositionTable, killerMoveTable, &counterMoves, nullptr, &historyMoves);
055:  			}
056:  		}
057:
058:  		bestMove = extractPVLine(board, transpositionTable, x);
059:
060:  		delete killerMoveTable;
061:  		if (!timer->isMoreTime(x))
062:  			break;
063:  	}
064:  	finalUIUpdate(&data, bestMove.line);
065:  	return bestMove.bestMove;
066:  }
067:
068:  int negascout(int alpha, int beta, int depthLeft, Board* board, searchData* data, TranspositionEntry* transpositionTable, std::vector<killerEntry>* killerMoveTable, std::array<std::array<std::array<Move, 64>, 64>, 2>* counterMove
069:  {
070:  	if (depthLeft == 0) return quiescence(alpha, beta, 3, board, data, false);
071:
072:
073:  	//The score assigned to draws (slightly negative to try to avoid premature draws)
074:  	const int DRAWSCORE = -25;
075:
076:  	data->nodes++;
077:  	int alphaOriginal = alpha;
078:  	Move bestMove;
079:  	bool isBestMove = false;
080:
081:  	TranspositionEntry entry = transpositionTable[board->zorbistKey % TTSize];
082:
083:  	std::array<Move,150> moveList;
084:  	int arraySize = searchForMoves(board, &moveList);
085:  	if (entry.zorbistKey == board->zorbistKey && std::find(moveList.begin(), moveList.end(), entry.bestMove) != moveList.end())
086:  	{
087:  		if (entry.depth >= depthLeft)
088:  		{
089:  			if (entry.flag == Exact)
090:  			{
091:  				return entry.score;
092:  			}
093:  			else if (entry.flag == lowerBound)
094:  			{
095:  				alpha = std::max(alpha, entry.score);
096:  			}
097:  			else if (entry.flag == upperBound)
098:  			{
099:  				beta = std::min(beta, entry.score);
100:  			}
101:  			if (alpha >= beta)
102:  			{
103:  				return entry.score;
104:  			}
105:  		}
106:  		isBestMove = true;
107:  		bestMove = entry.bestMove;
108:  	}
109:
110:  	//Draws by insufficient material for mating
```

```
111:        if (board->isMaterialDraw())
112:        {
113:            return DRAWSCORE;
114:        }
115:
116:        //CheckMate / StaleMate
117:        if (arraySize == 0)
118:        {
119:            //Checkmate
120:            if (board->isInCheck())
121:            {
122:                //Adds the distance to the root node onto the checkmate score.
123:                //This is to ensure the search algorithm prioritizes the fastest checkmate
124:                return -25000 + (data->depth - depthLeft);
125:            }
126:            //Stalemate
127:            else
128:            {
129:                return DRAWSCORE;
130:            }
131:        }
132:
133:        //Futility pruning
134:        if (depthLeft == 1 || depthLeft == 2 && !board->isInCheck())
135:        {
136:            int score = calculateScoreDiff(board);
137:
138:            if (depthLeft == 1)
139:            {
140:                //If the score is a lot lower than alpha , the chance of the one remaining move
141:                //being able to raise alpha is quite low.
142:                if (score + 125 < alpha)
143:                {
144:                    return quiescence(alpha, beta, 3, board, data, false);
145:                }
146:            }
147:            else if (depthLeft == 2)
148:            {
149:                //If the score is a lot lower than alpha , the chance of the one remaining move
150:                //being able to raise alpha is quite low.
151:                if (score + 600 < alpha)
152:                {
153:                    return quiescence(alpha, beta, 3, board, data, false);
154:                }
155:            }
156:        }
157:
158:
159:        //Threefold repetition
160:        if (board->moveHistory.size() > 0)
161:        {
162:            if (std::count(board->moveHistory.begin(), board->moveHistory.end(), board->moveHistory.back()) >= 3)
163:                return DRAWSCORE;
164:        }
165:
166:        orderSearch(&moveList, board, arraySize, &bestMove, isBestMove,(*killerMoveTable)[depthLeft], counterMoves, prevMove, historyMoves);
167:
168:        int score;
169:        for (int x = 0; x < arraySize; x++)
170:        {
171:            //If First node
172:            if (depthLeft == data->depth)
173:            {
174:                PVData bestMove = extractPVLine(board, transpositionTable, depthLeft);
175:                updateUI(data, moveList[x], x + 1, bestMove.line);
176:            }
177:
178:            moveList[x].applyMove(board);
179:            if (x == 0)
180:            {
181:                score = -negascout(-beta, -alpha, depthLeft - 1, board, data, transpositionTable, killerMoveTable, counterMoves, &moveList[x], historyMoves);
182:            }
183:            //Late Move Reductions
184:            else if (x > 4 && depthLeft >= 3 &&
185:                moveList[x].moveType != capture &&
186:                moveList[x].moveType != queenPromotion &&
187:                moveList[x].moveType != rookPromotion &&
188:                moveList[x].moveType != knightPromotion &&
189:                moveList[x].moveType != bishopPromotion)
190:            {
191:                //Try a null window search at a reduced depth
192:                score = -negascout(-alpha - 1, -alpha, depthLeft - 2, board, data, transpositionTable, killerMoveTable, counterMoves, &moveList[x], historyMoves);
193:
194:                //If the score is within the bounds , the first child was not the principle variation
195:                if (alpha < score && score < beta)
196:                    //Therefore do a full re-search
197:                    score = -negascout(-beta, -alpha, depthLeft - 1, board, data, transpositionTable, killerMoveTable, counterMoves, &moveList[x], historyMoves);
198:            }
199:            else
200:            {
201:                //Try a null window search
202:                score = -negascout(-alpha - 1, -alpha, depthLeft - 1, board, data, transpositionTable, killerMoveTable, counterMoves, &moveList[x], historyMoves);
203:
204:                //If the score is within the bounds , the first child was not the principle variation
205:                if (alpha < score && score < beta)
206:                    //Therefore do a full re-search
207:                    score = -negascout(-beta, -alpha, depthLeft - 1, board, data, transpositionTable, killerMoveTable, counterMoves, &moveList[x], historyMoves);
208:            }
209:            moveList[x].undoMove(board);
210:
211:            if (score > alpha)
212:            {
213:                alpha = score; // alpha acts like max in MiniMax
214:                bestMove = moveList[x];
215:            }
216:            if (score >= beta)
217:            {
218:                if (moveList[x].capturedPiece == blank)
219:                {
220:                    (*killerMoveTable)[depthLeft].addKillerMove(moveList[x]);
221:                    if (data->depth - depthLeft > 0)
222:                    {
223:                        (*counterMoves)[board->nextColour][prevMove->from][prevMove->to] = moveList[x];
224:                    }
```

```
                                (*historyMoves)[board->nextColour][moveList[x].from][moveList[x].to] += depthLeft * depthLeft;

                        }
                        break;   //  fail hard beta-cutoff
                }

        }

        TranspositionEntry newEntry = TranspositionEntry();
        newEntry.score = alpha;
        if (alpha <= alphaOriginal)
        {
                newEntry.flag = upperBound;
        }
        else if (alpha >= beta)
        {
                newEntry.flag = lowerBound;
        }
        else
        {
                newEntry.flag = Exact;
        }
        newEntry.depth = depthLeft;
        newEntry.bestMove = bestMove;
        newEntry.zorbistKey = board->zorbistKey;

        transpositionTable[board->zorbistKey % TTSize] = newEntry;
        return alpha;
}

int quiescence(int alpha, int beta, int depthLeft, Board* board, searchData * data, bool isQuiet)
{
        data->nodes++;

        if (isQuiet)
        {
                return calculateScoreDiff(board);
        }

        int score = calculateScoreDiff(board);
        if (score >= beta) return beta;
        if (alpha < score) alpha = score;
        if (depthLeft == 0) return alpha;

        std::array<Move, 150> moveList;
        int arraySize = searchForMoves(board, &moveList);
        arraySize = orderQuiescentSearch(&moveList,board, arraySize);

        for (int x = 0; x < arraySize; x++)
        {
                moveList[x].applyMove(board);
                score = -quiescence(-beta, -alpha, depthLeft - 1, board, data, !continueQuiescence( board, &moveList[x]));
                moveList[x].undoMove(board);
                if (score >= beta)
                {
                        return beta;   //  fail hard beta-cutoff
                }
                if (score > alpha)
                {
                        alpha = score; // alpha acts like max in MiniMax
                }
        }

        return alpha;
}

bool continueQuiescence(Board * board, Move * nextMove)
{
        if (board->isInCheck()) return true;
        if (nextMove->moveType == capture || nextMove->moveType == rookPromotion ||
                nextMove->moveType == queenPromotion || bishopPromotion || queenPromotion) return true;
        return false;
}

PVData extractPVLine(Board * board, TranspositionEntry * transpositionTable, int expectedDepth)
{
        PVData pv;

        if (expectedDepth == 0) return pv;

        TranspositionEntry entry = transpositionTable[board->zorbistKey % TTSize];
        if (entry.zorbistKey == board->zorbistKey) //The move has previously been searched
        {

                std::array<Move, 150> moveList;
                int arraySize = searchForMoves(board, &moveList);

                //If the move is invalid
                if (std::find(moveList.begin(), moveList.begin() + arraySize, entry.bestMove) == moveList.begin() + arraySize)
                {
                        return pv;
                }

                pv.bestMove = entry.bestMove;
                pv.line += notationFromMove(entry.bestMove) + " ";

                entry.bestMove.applyMove(board);

                pv.line += extractPVLine(board, transpositionTable, expectedDepth - 1).line;
                entry.bestMove.undoMove(board);
        }

        return pv;
}
```

ChessEngineLibrary\search.h

```
001:  #pragma once
002:  #include <vector>
003:  #include <array>
004:  #include <algorithm>
005:  #include <time.h>
006:
007:  #include "piece.h"
008:  #include "moveGeneration.h"
009:  #include "board.h"
010:  #include "scoring.h"
011:  #include "move.h"
012:  #include "utils.h"
013:  #include "timeManagement.h"
014:  #include "transpositionTable.h"
015:  #include "transpositionEntry.h"
016:  #include "moveOrdering.h"
017:
018:  struct searchData
019:  {
020:      int nodes;
021:      int depth;
022:      time_t startTime;
023:  };
024:
025:  struct PVData
026:  {
027:      Move bestMove;
028:      std::string line;
029:  };
030:
031:  void updateUI(searchData * data, Move currentMove, int currentMoveNumber, std::string pvLine);
032:  void finalUIUpdate(searchData * data, std::string pvLine);
033:
034:  Move startSearch(Board* board, TranspositionEntry* transpositionTable, timeManagement* timer);
035:  int negascout(int alpha, int beta, int depthLeft, Board* board, searchData* data, TranspositionEntry* transpositionTable, std::vector<killerEntry>* killerMoveTable, std::array<std::array<std::array<Move, 64>, 64>, 2>* counterMove
036:  int quiescence(int alpha, int beta, int depthLeft, Board* board, searchData* data, bool isQuiet);
037:  bool continueQuiescence(Board* board, Move* nextMove);
038:
039:  PVData extractPVLine(Board* board, TranspositionEntry* transpositionTable, int expectedDepth);
```

ChessEngineLibrary\timeManagement.cpp

```
001:  #include "timeManagement.h"
002:
003:
004:
005:  timeManagement::timeManagement()
006:  {
007:  }
008:
009:  timeManagement::timeManagement(long int wtime,long int btime, colours colour)
010:  {
011:          //Assume their is always 30 moves left.
012:          const int movesLeft = 30;
013:
014:          switch (colour)
015:          {
016:          case white:
017:                  targetTime = wtime / movesLeft;
018:                  break;
019:          case black:
020:                  targetTime = btime / movesLeft;
021:                  break;
022:          }
023:
024:          startTime = std::chrono::high_resolution_clock::now();
025:          searchMode = timed;
026:  }
027:
028:  timeManagement::timeManagement(int depth)
029:  {
030:          targetDepth = depth;
031:          searchMode = fixedDepth;
032:  }
033:
034:  timeManagement::~timeManagement()
035:  {
036:  }
037:
038:  bool timeManagement::isMoreTime(int searchDepth)
039:  {
040:          if (searchMode == timed)
041:          {
042:                  //Their is enough time for another iteration if we have used less than half of the target time;
043:                  std::chrono::duration<double, std::milli> time_span = std::chrono::high_resolution_clock::now() - startTime;
044:
045:                  //Divides by 10 as it is assuming each depth is 5 times harder than the previous.
046:                  return time_span.count() < targetTime / 5;
047:          }
048:          else if (searchMode == fixedDepth)
049:          {
050:                  return searchDepth < targetDepth;
051:          }
052:  }
053:
```

ChessEngineLibrary\timeManagement.h

```
001:    #pragma once
002:    #include <ctime>
003:    #include <chrono>
004:
005:    #include "piece.h"
006:
007:    enum Mode { fixedDepth, timed };
008:
009:    class timeManagement
010:    {
011:    public:
012:            timeManagement();
013:            timeManagement(long int wtime, long int btime, colours colour);
014:            timeManagement(int depth);
015:
016:            ~timeManagement();
017:
018:            bool isMoreTime(int searchDepth);
019:
020:    private:
021:            std::chrono::high_resolution_clock::time_point startTime;
022:            long int targetTime;
023:            Mode searchMode;
024:            int targetDepth;
025:    };
026:
027:
028:
```

ChessEngineLibrary\transpositionEntry.cpp

```
001:    #include "transpositionEntry.h"
002:
```

ChessEngineLibrary\transpositionEntry.h

```
001:    #pragma once
002:    #include "move.h"
003:
004:    enum nodeType { Exact, lowerBound, upperBound };
005:
006:    struct TranspositionEntry
007:    {
008:    public:
009:            uint64_t zorbistKey;
010:            Move bestMove;
011:            nodeType flag;
012:            int score;
013:            int depth;
014:    };
015:
```

ChessEngineLibrary\transpositionTable.cpp

```
001:  #include "transpositionTable.h"
002:
003:
004:  void ZorbistKeys::initialize()
005:  {
006:  	srand(25461);
007:
008:  	for (int i = 0; i < 64; i++)
009:  	{
010:  		for (int j = 0; j < 12; j++)
011:  		{
012:  			pieceKeys[i][j] = get64rand();
013:  		}
014:  	}
015:  	for (int i = 0; i < 8; i++)
016:  	{
017:  		enPassantKeys[i] = get64rand();
018:  	}
019:
020:  	blackMoveKey = get64rand();
021:  	blackQueenSideCastlingKey = get64rand();
022:  	blackKingSideCastlingKey = get64rand();
023:  	whiteQueenSideCastlingKey = get64rand();
024:  	whiteKingSideCastlingKey = get64rand();
025:  }
026:
027:  uint64_t get64rand() {
028:  	bool isCollision = true;
029:  	uint64_t num;
030:
031:  	while (isCollision)
032:  	{
033:  		num = (((uint64_t)rand() << 0) & 0x000000000000FFFFull) |
034:  			(((uint64_t)rand() << 16) & 0x00000000FFFF0000ull) |
035:  			(((uint64_t)rand() << 32) & 0x0000FFFF00000000ull) |
036:  			(((uint64_t)rand() << 48) & 0xFFFF000000000000ull);
037:
038:  		isCollision = false;
039:  		for (int i = 0; i < 64; i++)
040:  		{
041:  			for (int j = 0; j < 12; j++)
042:  			{
043:  				if (num == ZorbistKeys::pieceKeys[i][j]) isCollision = true;
044:  			}
045:  		}
046:  		if (num == 0) isCollision = true;
047:  		if (num == ZorbistKeys::blackMoveKey) isCollision = true;
048:  		if (num == ZorbistKeys::blackQueenSideCastlingKey) isCollision = true;
049:  		if (num == ZorbistKeys::blackKingSideCastlingKey) isCollision = true;
050:  		if (num == ZorbistKeys::whiteQueenSideCastlingKey) isCollision = true;
051:  		if (num == ZorbistKeys::whiteKingSideCastlingKey) isCollision = true;
052:  	}
053:
054:  	return num;
055:  }
056:
057:  uint64_t ZorbistKeys::pieceKeys[64][12];
058:  uint64_t ZorbistKeys::blackMoveKey;
059:  uint64_t ZorbistKeys::blackQueenSideCastlingKey;
060:  uint64_t ZorbistKeys::blackKingSideCastlingKey;
061:  uint64_t ZorbistKeys::whiteQueenSideCastlingKey;
062:  uint64_t ZorbistKeys::whiteKingSideCastlingKey;
063:  uint64_t ZorbistKeys::enPassantKeys[8];
```

ChessEngineLibrary\transpositionTable.h

```
001:  #pragma once
002:  #include <cstdlib>
003:  #include <stdint.h>
004:  #include <vector>
005:  #include <unordered_map>
006:
007:  #include "transpositionTable.h"
008:
009:  #define TTSize 16777216
010:
011:  struct TranspositionEntry;
012:
013:  struct ZorbistKeys
014:  {
015:          //A key for each square. WP = 0
016:  public:
017:          static uint64_t pieceKeys[64][12];
018:          static uint64_t enPassantKeys[8];
019:          static uint64_t blackMoveKey;
020:          static uint64_t blackQueenSideCastlingKey;
021:          static uint64_t blackKingSideCastlingKey;
022:          static uint64_t whiteQueenSideCastlingKey;
023:          static uint64_t whiteKingSideCastlingKey;
024:
025:          static void initialize();
026:  };
027:
028:  uint64_t get64rand();
029:
030:
031:
032:
033:
034:
```

ChessEngineLibrary\utils.cpp

```
001:  #include "utils.h"
002:
003:  std::string notationFromMove(const Move & move)
004:  {
005:  	std::string notation = notationFromPiecePos(move.from) + notationFromPiecePos(move.to);
006:
007:  	if (move.moveType == knightPromotion)
008:  	{
009:  		notation += "n";
010:  	}
011:  	else if (move.moveType == rookPromotion)
012:  	{
013:  		notation += "r";
014:  	}
015:  	else if (move.moveType == bishopPromotion)
016:  	{
017:  		notation += "b";
018:  	}
019:  	else if (move.moveType == queenPromotion)
020:  	{
021:  		notation += "q";
022:  	}
023:
024:  	return notation;
025:  }
026:
027:  Move moveFromNotation(std::string moveNotation, Board * board)
028:  {
029:  	int column = moveNotation[0] - 'a';
030:  	int row = moveNotation[1] - '1';
031:  	int from = row * 8 + column;
032:
033:  	column = moveNotation[2] - 'a';
034:  	row = moveNotation[3] - '1';
035:  	int to = row * 8 + column;
036:  	uint64_t fromBitboard = (uint64_t)1 << from;
037:  	colours aiColour;
038:  	pieceType piece = blank;
039:  	if ((fromBitboard & board->whitePieces) != 0) //Is a white piece
040:  	{
041:  		aiColour = white;
042:  		if ((fromBitboard & board->getPieceBitboard(white, bishop)) != 0)
043:  		{
044:  			piece = bishop;
045:  		}
046:  		else if ((fromBitboard & board->getPieceBitboard(white, queen)) != 0)
047:  		{
048:  			piece = queen;
049:  		}
050:  		else if ((fromBitboard & board->getPieceBitboard(white, king)) != 0)
051:  		{
052:  			piece = king;
053:  		}
054:  		else if ((fromBitboard & board->getPieceBitboard(white, rook)) != 0)
055:  		{
056:  			piece = rook;
057:  		}
058:  		else if ((fromBitboard & board->getPieceBitboard(white, knight)) != 0)
059:  		{
060:  			piece = knight;
061:  		}
062:  		else if ((fromBitboard & board->getPieceBitboard(white, pawn)) != 0)
063:  		{
064:  			piece = pawn;
065:  		}
066:  	}
067:  	else if ((fromBitboard & board->blackPieces) != 0)
068:  	{
069:  		aiColour = black;
070:  		if ((fromBitboard & board->getPieceBitboard(black, bishop)) != 0)
071:  		{
072:  			piece = bishop;
073:  		}
074:  		else if ((fromBitboard & board->getPieceBitboard(black, queen)) != 0)
075:  		{
076:  			piece = queen;
077:  		}
078:  		else if ((fromBitboard & board->getPieceBitboard(black, king)) != 0)
079:  		{
080:  			piece = king;
081:  		}
082:  		else if ((fromBitboard & board->getPieceBitboard(black, rook)) != 0)
083:  		{
084:  			piece = rook;
085:  		}
086:  		else if ((fromBitboard & board->getPieceBitboard(black, knight)) != 0)
087:  		{
088:  			piece = knight;
089:  		}
090:  		else if ((fromBitboard & board->getPieceBitboard(black, pawn)) != 0)
091:  		{
092:  			piece = pawn;
093:  		}
094:  	}
095:  	else
096:  	{
097:  		throw std::runtime_error("moveFromNotation failed. Piece not on board.");
098:  	}
099:
100:  	if (std::abs(from - to) == 16 && piece == pawn)//Pawn double move
101:  	{
102:  		return Move(from, to, pawnDoubleMove, piece, board);
103:  	}
104:  	else if ((((uint64_t)1 << to) & board->allPieces) != 0 && moveNotation.length() == 4)//Capture
105:  	{
106:  		return Move(from, to, capture, piece, board);
107:  	}
108:  	else if (to == board->enPassantSquare && piece == pawn)//En passant capture
109:  	{
110:  		return Move(from, to, capture, piece, board);
```

```
111:            }
112:            else if (std::abs(from - to) == 2 && piece == king) //Castling
113:            {
114:                    if (aiColour == white)
115:                    {
116:                            if (from < to) //KingSide castling
117:                            {
118:                                    return Move(from, to, kingSideCastling, piece, board);
119:                            }
120:                            else //QueenSide castling
121:                            {
122:                                    return Move(from, to, queenSideCastling, piece, board);
123:                            }
124:                    }
125:                    else
126:                    {
127:                            if (from < to) //KingSide castling
128:                            {
129:                                    return Move(from, to, kingSideCastling, piece, board);
130:                            }
131:                            else //QueenSide castling
132:                            {
133:                                    return Move(from, to, queenSideCastling, piece, board);
134:                            }
135:                    }
136:            }
137:            else if (moveNotation.length() == 5) //Promotion moves. Format is a7a8q , where the last piece is the piece to be promoted to.
138:            {
139:                    if (moveNotation.back() == 'q' || moveNotation.back() == 'Q')
140:                    {
141:                            return Move(from, to, queenPromotion, piece, board);
142:                    }
143:                    else if (moveNotation.back() == 'r' || moveNotation.back() == 'R')
144:                    {
145:                            return Move(from, to, rookPromotion, piece, board);
146:                    }
147:                    else if (moveNotation.back() == 'n' || moveNotation.back() == 'N')
148:                    {
149:                            return Move(from, to, knightPromotion, piece, board);
150:                    }
151:                    else if (moveNotation.back() == 'b' || moveNotation.back() == 'B')
152:                    {
153:                            return Move(from, to, bishopPromotion, piece, board);
154:                    }
155:
156:            }
157:            else//Quiet Move
158:            {
159:                    return Move(from, to, quietMove, piece, board);
160:            }
161:    }
162:
163:    std::string notationFromPiecePos(int piecePos)
164:    {
165:            std::string notation;
166:
167:            int column = (piecePos % 8);
168:            int row = 1 + piecePos / 8;
169:
170:            char rowChar = row + '0';
171:            char columnChar = column + 'a';
172:            notation += columnChar;
173:            notation += rowChar;
174:
175:            return notation;
176:    }
177:
```

ChessEngineLibrary\utils.h

```
001:    #pragma once
002:    #include "board.h"
003:    #include "move.h"
004:    #include "piece.h"
005:    #include "moveGeneration.h"
006:
007:    #include <string>
008:    #include <stdint.h>
009:    #include <stdlib.h>
010:    #include <vector>
011:
012:    std::string notationFromMove(const Move& move);
013:    Move moveFromNotation(std::string moveNotation, Board* board);
014:
015:    std::string notationFromPiecePos(int piecePos);
```

UnitTesting\Board.cpp

```
001:  #pragma once
002:  #include "gtest/gtest.h"
003:  #include "board.h"
004:
005:  TEST(Board, UpdateFunction)
006:  {
007:  	Board board;
008:  	board.setBitboard(white,pawn, 2);
009:  	board.setBitboard(white, bishop, 4);
010:  	board.update();
011:  	EXPECT_EQ(board.allPieces, 6);
012:
013:  	board.setBitboard(black, bishop, 8);
014:  	board.update();
015:  	EXPECT_EQ(board.allPieces, 14);
016:  	EXPECT_EQ(board.whitePieces, 6);
017:  	EXPECT_EQ(board.blackPieces, 8);
018:  }
019:  TEST(Board, Defaults)
020:  {
021:  	Board board;
022:  	board.defaults();
023:  	EXPECT_EQ(board.getPieceBitboard(white, pawn), 65280);
024:  	EXPECT_EQ(board.getPieceBitboard(white, rook), 129);
025:  	EXPECT_EQ(board.getPieceBitboard(white, knight), 66);
026:  	EXPECT_EQ(board.getPieceBitboard(white, bishop), 36);
027:  	EXPECT_EQ(board.getPieceBitboard(white, queen), 8);
028:  	EXPECT_EQ(board.getPieceBitboard(white, king), 16);
029:
030:  	EXPECT_EQ(board.getPieceBitboard(black, pawn), 71776119061217280);
031:  	EXPECT_EQ(board.getPieceBitboard(black, rook), 9295429630892703744);
032:  	EXPECT_EQ(board.getPieceBitboard(black, knight), 4755801206503243776);
033:  	EXPECT_EQ(board.getPieceBitboard(black, bishop), 2594073385365405696);
034:  	EXPECT_EQ(board.getPieceBitboard(black, queen), 576460752303423488);
035:  	EXPECT_EQ(board.getPieceBitboard(black, king), 1152921504606846976);
036:
037:  }
038:
039:  TEST(Board, loadFromFen)
040:  {
041:  	Board defaultBoard;
042:  	defaultBoard.defaults();
043:  	Board fenBoard;
044:  	fenBoard.loadFromFen("rnbqkbnr/pppppppp/8/8/8/8/PPPPPPPP/RNBQKBNR w KQkq - 0 1");
045:
046:  	EXPECT_EQ(defaultBoard.allPieces, fenBoard.allPieces);
047:  	EXPECT_EQ(defaultBoard.getPieceBitboard(black, bishop), fenBoard.getPieceBitboard(black, bishop));
048:  	EXPECT_EQ(defaultBoard.getPieceBitboard(black, king), fenBoard.getPieceBitboard(black, king));
049:  	EXPECT_EQ(defaultBoard.getPieceBitboard(black, knight), fenBoard.getPieceBitboard(black, knight));
050:  	EXPECT_EQ(defaultBoard.getPieceBitboard(black, pawn), fenBoard.getPieceBitboard(black, pawn));
051:  	EXPECT_EQ(defaultBoard.blackPieces, fenBoard.blackPieces);
052:  	EXPECT_EQ(defaultBoard.getPieceBitboard(black, queen), fenBoard.getPieceBitboard(black, queen));
053:  	EXPECT_EQ(defaultBoard.getPieceBitboard(black, rook), fenBoard.getPieceBitboard(black, rook));
054:  	EXPECT_EQ(defaultBoard.getPieceBitboard(white, bishop), fenBoard.getPieceBitboard(white, bishop));
055:  	EXPECT_EQ(defaultBoard.getPieceBitboard(white, king), fenBoard.getPieceBitboard(white, king));
056:  	EXPECT_EQ(defaultBoard.getPieceBitboard(white, knight), fenBoard.getPieceBitboard(white, knight));
057:  	EXPECT_EQ(defaultBoard.getPieceBitboard(white, pawn), fenBoard.getPieceBitboard(white, pawn));
058:  	EXPECT_EQ(defaultBoard.whitePieces, fenBoard.whitePieces);
059:  	EXPECT_EQ(defaultBoard.getPieceBitboard(white, queen), fenBoard.getPieceBitboard(white, queen));
060:  	EXPECT_EQ(defaultBoard.getPieceBitboard(white, rook), fenBoard.getPieceBitboard(white, rook));
061:  	EXPECT_EQ(fenBoard.enPassantSquare, -1);
062:  	EXPECT_EQ(fenBoard.canBlackCastleKingSide, true);
063:  	EXPECT_EQ(fenBoard.canBlackCastleQueenSide, true);
064:  	EXPECT_EQ(fenBoard.canWhiteCastleQueenSide, true);
065:  	EXPECT_EQ(fenBoard.canWhiteCastleKingSide, true);
066:
067:  	Board board = Board();
068:  	board.setBitboard(white,pawn, 1);
069:  	board.update();
070:  	fenBoard = Board();
071:  	fenBoard.loadFromFen("8/8/8/8/8/8/8/P7 w KQkq - 0 1");
072:
073:  	EXPECT_EQ(board.allPieces, fenBoard.allPieces);
074:  	EXPECT_EQ(board.getPieceBitboard(black, bishop), fenBoard.getPieceBitboard(black, bishop));
075:  	EXPECT_EQ(board.getPieceBitboard(black, king), fenBoard.getPieceBitboard(black, king));
076:  	EXPECT_EQ(board.getPieceBitboard(black, knight), fenBoard.getPieceBitboard(black, knight));
077:  	EXPECT_EQ(board.getPieceBitboard(black, pawn), fenBoard.getPieceBitboard(black, pawn));
078:  	EXPECT_EQ(board.blackPieces, fenBoard.blackPieces);
079:  	EXPECT_EQ(board.getPieceBitboard(black, queen), fenBoard.getPieceBitboard(black, queen));
080:  	EXPECT_EQ(board.getPieceBitboard(black, rook), fenBoard.getPieceBitboard(black, rook));
081:  	EXPECT_EQ(board.getPieceBitboard(white, bishop), fenBoard.getPieceBitboard(white, bishop));
082:  	EXPECT_EQ(board.getPieceBitboard(white, king), fenBoard.getPieceBitboard(white, king));
083:  	EXPECT_EQ(board.getPieceBitboard(white, knight), fenBoard.getPieceBitboard(white, knight));
084:  	EXPECT_EQ(board.getPieceBitboard(white, pawn), fenBoard.getPieceBitboard(white, pawn));
085:  	EXPECT_EQ(board.whitePieces, fenBoard.whitePieces);
086:  	EXPECT_EQ(board.getPieceBitboard(white, queen), fenBoard.getPieceBitboard(white, queen));
087:  	EXPECT_EQ(board.getPieceBitboard(white, rook), fenBoard.getPieceBitboard(white, rook));
088:  	EXPECT_EQ(fenBoard.enPassantSquare, -1);
089:  	EXPECT_EQ(fenBoard.canBlackCastleKingSide, true);
090:  	EXPECT_EQ(fenBoard.canBlackCastleQueenSide, true);
091:  	EXPECT_EQ(fenBoard.canWhiteCastleQueenSide, true);
092:  	EXPECT_EQ(fenBoard.canWhiteCastleKingSide, true);
093:
094:  	board = Board();
095:  	board.setBitboard(white, pawn, 16777216);
096:  	board.setBitboard(black, pawn, 33554432);
097:  	board.update();
098:  	fenBoard = Board();
099:  	fenBoard.loadFromFen("8/8/8/8/Pp6/8/8/8 b - a3 0 1");
100:
101:  	EXPECT_EQ(board.allPieces, fenBoard.allPieces);
102:  	EXPECT_EQ(board.getPieceBitboard(black, bishop), fenBoard.getPieceBitboard(black, bishop));
103:  	EXPECT_EQ(board.getPieceBitboard(black, king), fenBoard.getPieceBitboard(black, king));
104:  	EXPECT_EQ(board.getPieceBitboard(black, knight), fenBoard.getPieceBitboard(black, knight));
105:  	EXPECT_EQ(board.getPieceBitboard(black, pawn), fenBoard.getPieceBitboard(black, pawn));
106:  	EXPECT_EQ(board.blackPieces, fenBoard.blackPieces);
107:  	EXPECT_EQ(board.getPieceBitboard(black, queen), fenBoard.getPieceBitboard(black, queen));
108:  	EXPECT_EQ(board.getPieceBitboard(black, rook), fenBoard.getPieceBitboard(black, rook));
109:  	EXPECT_EQ(board.getPieceBitboard(white, bishop), fenBoard.getPieceBitboard(white, bishop));
110:  	EXPECT_EQ(board.getPieceBitboard(white, king), fenBoard.getPieceBitboard(white, king));
```

```
111:            EXPECT_EQ(board.getPieceBitboard(white, knight), fenBoard.getPieceBitboard(white, knight));
112:            EXPECT_EQ(board.getPieceBitboard(white, pawn), fenBoard.getPieceBitboard(white, pawn));
113:            EXPECT_EQ(board.whitePieces, fenBoard.whitePieces);
114:            EXPECT_EQ(board.getPieceBitboard(white, queen), fenBoard.getPieceBitboard(white, queen));
115:            EXPECT_EQ(board.getPieceBitboard(white, rook), fenBoard.getPieceBitboard(white, rook));
116:            EXPECT_EQ(fenBoard.enPassantSquare, 16);
117:            EXPECT_EQ(fenBoard.canBlackCastleKingSide, false);
118:            EXPECT_EQ(fenBoard.canBlackCastleQueenSide, false);
119:            EXPECT_EQ(fenBoard.canWhiteCastleQueenSide, false);
120:            EXPECT_EQ(fenBoard.canWhiteCastleKingSide, false);
121:    }
122:
123:    TEST(Board, exportAsFen)
124:    {
125:            Board board;
126:            board.defaults();
127:
128:            EXPECT_EQ(board.exportAsFen().substr(0,53), "rnbqkbnr/pppppppp/8/8/8/8/PPPPPPPP/RNBQKBNR w KQkq - ");
129:
130:            board.loadFromFen("rnbqkbnr/pp1ppppp/8/2p5/4P3/5N2/PPPP1PPP/RNBQKB1R b KQkq - 1 2");
131:            board.printBoard();
132:
133:            EXPECT_EQ(board.exportAsFen().substr(0, 59), "rnbqkbnr/pp1ppppp/8/2p5/4P3/5N2/PPPP1PPP/RNBQKB1R b KQkq - ");
134:
135:    }
136:
137:    TEST(Board, IsPieceAttacked)
138:    {
139:            Board board;
140:
141:            board.loadFromFen("r7/8/8/8/8/8/8/K7 w - - 0 1 ");
142:            EXPECT_EQ(board.isPieceAttacked(0, white), true);
143:
144:            board = Board();
145:            board.loadFromFen("1r6/8/8/8/8/8/8/K7 w - - 0 1 ");
146:            EXPECT_EQ(board.isPieceAttacked(0, white), false);
147:
148:            board = Board();
149:            board.loadFromFen("8/8/8/8/8/8/1p6/K7 w - - 0 1 ");
150:            EXPECT_EQ(board.isPieceAttacked(0, white), true);
151:
152:            board = Board();
153:            board.loadFromFen("8/8/8/8/8/8/p7/K7 w - - 0 1 ");
154:            EXPECT_EQ(board.isPieceAttacked(0, white), false);
155:    }
156:
157:    TEST(Board, isMaterialDraw)
158:    {
159:            Board board;
160:
161:            board.defaults();
162:            EXPECT_EQ(board.isMaterialDraw(), false);
163:
164:            board = Board("K7/8/k7/8/8/8/8/8 w -- 0 1");
165:            EXPECT_EQ(board.isMaterialDraw(), true);
166:
167:            board = Board("K7/8/k7/8/N7/8/8/8 w -- 0 1");
168:            EXPECT_EQ(board.isMaterialDraw(), true);
169:
170:            board = Board("K7/8/k7/8/NN6/8/8/8 w -- 0 1");
171:            EXPECT_EQ(board.isMaterialDraw(), false);
172:
173:            board = Board("K7/8/k7/8/8/8/8/B7 w -- 0 1");
174:            EXPECT_EQ(board.isMaterialDraw(), true);
175:
176:            board = Board("K7/8/k7/8/8/8/8/B1B5 w -- 0 1");
177:            EXPECT_EQ(board.isMaterialDraw(), true);
178:
179:            board = Board("K7/8/k7/8/8/8/8/B1B1b1b w -- 0 1");
180:            EXPECT_EQ(board.isMaterialDraw(), true);
181:
182:            board = Board("K7/8/k7/8/8/8/8/B1B1bb1 w -- 0 1");
183:            EXPECT_EQ(board.isMaterialDraw(), false);
184:    }
185:
186:    TEST(Bitboard, bitsum)
187:    {
188:            EXPECT_EQ(bitSum(15), 4);
189:            EXPECT_EQ(bitSum(17), 2);
190:            EXPECT_EQ(bitSum(emptyBitboard), 0);
191:            EXPECT_EQ(bitSum(universalBitboard), 64);
192:            EXPECT_EQ(bitSum(255), 8);
193:    }
194:
195:    TEST(Bitboard, pop)
196:    {
197:            uint64_t b = 15;
198:            EXPECT_EQ(pop(b), 1);
199:            EXPECT_EQ(pop(b), 2);
200:            EXPECT_EQ(pop(b), 4);
201:            EXPECT_EQ(pop(b), 8);
202:            EXPECT_EQ(pop(b), 0);
203:
204:            b = 263;
205:            EXPECT_EQ(pop(b), 1);
206:            EXPECT_EQ(pop(b), 2);
207:            EXPECT_EQ(pop(b), 4);
208:            EXPECT_EQ(pop(b), 256);
209:            EXPECT_EQ(pop(b), 0);
210:    }
211:
212:    TEST(Bitboard, bitScanForward)
213:    {
214:            EXPECT_EQ(bitScanForward(2), 1);
215:            EXPECT_EQ(bitScanForward(256), 8);
216:            EXPECT_EQ(bitScanForward(64), 6);
217:    }
218:
219:
220:
```

UnitTesting\Move.cpp

```
001:  #pragma once
002:  #include "gtest/gtest.h"
003:  #include "board.h"
004:  #include "move.h"
005:  #include "utils.h"
006:
007:  TEST(Move, ApplyQuietMoves)
008:  {
009:      Board board;
010:      board.setBitboard(white, pawn, 2);
011:      board.nextColour = white;
012:      Move move = Move(1, 9, quietMove, pawn, &board);
013:      move.applyMove(&board);
014:      EXPECT_EQ(board.getPieceBitboard(white,pawn), 1 << 9);
015:      EXPECT_EQ(board.getPieceBitboard(white, pawn), 1 << 9);
016:
017:      board = Board();
018:      board.setBitboard(white, queen, 2);
019:      board.nextColour = white;
020:      move = Move(1, 7, quietMove, queen, &board);
021:      move.applyMove(&board);
022:      EXPECT_EQ(board.getPieceBitboard(white, queen), 1 << 7);
023:      EXPECT_EQ(board.getPieceBitboard(white, queen), 1 << 7);
024:  }
025:
026:  TEST(Move, ApplyPawnDoubleMoves)
027:  {
028:      Board board;
029:      board.setBitboard(white, pawn, (uint64_t)1 << 8);
030:      board.nextColour = white;
031:      Move move = Move(8, 24, pawnDoubleMove, pawn, &board);
032:      move.applyMove(&board);
033:      EXPECT_EQ(board.getPieceBitboard(white, pawn), (uint64_t)1 << 24);
034:      EXPECT_EQ(board.allPieces, (uint64_t)1 << 24);
035:      EXPECT_EQ(board.enPassantSquare, 16);
036:
037:      board = Board();
038:      board.setBitboard(black, pawn, (uint64_t)1 << 48);
039:      board.nextColour = black;
040:      move = Move(48, 32, pawnDoubleMove, pawn, &board);
041:      move.applyMove(&board);
042:      EXPECT_EQ(board.getPieceBitboard(black, pawn), (uint64_t)1 << 32);
043:      EXPECT_EQ(board.allPieces, (uint64_t)1 << 32);
044:      EXPECT_EQ(board.enPassantSquare, 40);
045:  }
046:
047:  TEST(Move, ApplyEnPassantMoves)
048:  {
049:      Board board;
050:      board.setBitboard(white, pawn, 4294967296);
051:      board.setBitboard(black, pawn, 8589934592);
052:      board.enPassantSquare = 41;
053:      board.nextColour = white;
054:      Move move = Move(32, 41, capture, pawn, &board);
055:      move.applyMove(&board);
056:      EXPECT_EQ(board.getPieceBitboard(white, pawn), 2199023255552);
057:      EXPECT_EQ(board.allPieces, 2199023255552);
058:      EXPECT_EQ(board.enPassantSquare, -1);
059:
060:      board = Board();
061:      board.setBitboard(white, pawn, 16777216);
062:      board.setBitboard(black, pawn, 33554432);
063:      board.enPassantSquare = 16;
064:      board.nextColour = black;
065:      move = Move(25, 16, capture, pawn, &board);
066:      move.applyMove(&board);
067:      EXPECT_EQ(board.getPieceBitboard(black, pawn), 65536);
068:      EXPECT_EQ(board.allPieces, 65536);
069:      EXPECT_EQ(board.enPassantSquare, -1);
070:
071:      board = Board();
072:      board.setBitboard(white, pawn, 16908288);
073:      board.setBitboard(black, pawn, 33554432);
074:      board.enPassantSquare = 16;
075:      board.nextColour = black;
076:      move = Move(25, 16, capture, pawn, &board);
077:      move.applyMove(&board);
078:      EXPECT_EQ(board.getPieceBitboard(black, pawn), 65536);
079:      EXPECT_EQ(board.getPieceBitboard(white, pawn), 131072);
080:      EXPECT_EQ(board.allPieces, 196608);
081:      EXPECT_EQ(board.enPassantSquare, -1);
082:
083:
084:  }
085:
086:  TEST(Move, ApplyCaptureMoves)
087:  {
088:      Board board = Board();
089:      board.setBitboard(white, queen, 2);
090:      board.setBitboard(black, pawn, 128);
091:      board.update();
092:      board.nextColour = white;
093:      Move move = Move(1, 7, capture, queen, &board);
094:      move.applyMove(&board);
095:      EXPECT_EQ(board.getPieceBitboard(white,queen), 1 << 7);
096:      EXPECT_EQ(board.allPieces, 1 << 7);
097:
098:      board = Board();
099:      board.setBitboard(white, pawn, 2);
100:      board.setBitboard(black, pawn, 256);
101:      board.update();
102:      board.nextColour = white;
103:      move = Move(1, 8, capture, pawn, &board);
104:      move.applyMove(&board);
105:      EXPECT_EQ(board.getPieceBitboard(white, pawn), 1 << 8);
106:      EXPECT_EQ(board.getPieceBitboard(black, pawn), 0);
107:      EXPECT_EQ(board.allPieces, 1 << 8);
108:
109:      board = Board();
110:      board.setBitboard(white, pawn, 34359738368);
```

```
111:            board.setBitboard(black, pawn, 4398046511104);
112:            board.update();
113:            board.nextColour = white;
114:            move = Move(35, 42, capture, pawn, &board);
115:            move.applyMove(&board);
116:            EXPECT_EQ(board.getPieceBitboard(white, pawn), 4398046511104);
117:            EXPECT_EQ(board.getPieceBitboard(black, pawn), 0);
118:            EXPECT_EQ(board.allPieces, 4398046511104);
119:
120:
121:            board = Board();
122:            board.setBitboard(white, rook, 9223372036854775808);
123:            board.setBitboard(black, pawn, 72057594037927936);
124:            board.update();
125:            board.nextColour = white;
126:            move = Move(63, 56, capture, rook, &board);
127:            move.applyMove(&board);
128:            EXPECT_EQ(board.getPieceBitboard(white,rook), 72057594037927936);
129:            EXPECT_EQ(board.getPieceBitboard(black, pawn), 0);
130:            EXPECT_EQ(board.allPieces, 72057594037927936);
131:
132:            board = Board();
133:            board.setBitboard(white, queen, 4194304);
134:            board.setBitboard(black, king, 64);
135:            board.update();
136:            board.nextColour = white;
137:            move = Move(22, 6, capture, queen, &board);
138:            move.applyMove(&board);
139:            EXPECT_EQ(board.getPieceBitboard(white, queen), 64);
140:            EXPECT_EQ(board.getPieceBitboard(black, king), 0);
141:            EXPECT_EQ(board.allPieces, 64);
142:    }
143:
144:    TEST(Move, ApplyPromotionMoves)
145:    {
146:            Board board = Board();
147:            board.setBitboard(white, pawn, 281474976710656);
148:            board.update();
149:            board.nextColour = white;
150:            Move move = Move(48, 56, queenPromotion, pawn, &board);
151:            move.applyMove(&board);
152:            EXPECT_EQ(board.getPieceBitboard(white, queen), 72057594037927936);
153:            EXPECT_EQ(board.getPieceBitboard(white, pawn), 0);
154:            EXPECT_EQ(board.allPieces, 72057594037927936);
155:
156:            board = Board();
157:            board.setBitboard(black, pawn, 256);
158:            board.update();
159:            board.nextColour = black;
160:            move = Move(8, 0, knightPromotion, pawn, &board);
161:            move.applyMove(&board);
162:            EXPECT_EQ(board.getPieceBitboard(black, knight), 1);
163:            EXPECT_EQ(board.getPieceBitboard(black, pawn), 0);
164:            EXPECT_EQ(board.allPieces, 1);
165:    }
166:
167:    TEST(Move, ApplyCastlingMoves)
168:    {
169:            Board board = Board();
170:            board.setBitboard(white, king, 16);
171:            board.setBitboard(white, rook, 1);
172:            board.update();
173:            board.nextColour = white;
174:            Move move = Move(4, 2, queenSideCastling, king, &board);
175:            move.applyMove(&board);
176:            EXPECT_EQ(board.getPieceBitboard(white, king), 4);
177:            EXPECT_EQ(board.getPieceBitboard(white, rook), 8);
178:            EXPECT_EQ(board.allPieces, 12);
179:
180:            board = Board();
181:            board.setBitboard(white, king, 16);
182:            board.setBitboard(white, rook, 128);
183:            board.update();
184:            board.nextColour = white;
185:            move = Move(4, 6, kingSideCastling, king, &board);
186:            move.applyMove(&board);
187:            EXPECT_EQ(board.getPieceBitboard(white, king), 64);
188:            EXPECT_EQ(board.getPieceBitboard(white, rook), 32);
189:            EXPECT_EQ(board.allPieces, 96);
190:
191:            board = Board();
192:            board.setBitboard(black, king, 1152921504606846976);
193:            board.setBitboard(black, rook, 72057594037927936);
194:            board.update();
195:            board.nextColour = black;
196:            move = Move(60, 58, queenSideCastling, king, &board);
197:            move.applyMove(&board);
198:            EXPECT_EQ(board.getPieceBitboard(black, king), 28823037615171744);
199:            EXPECT_EQ(board.getPieceBitboard(black, rook), 576460752303423488);
200:            EXPECT_EQ(board.allPieces, 86469128455135232);
201:
202:            board = Board();
203:            board.setBitboard(black, king, 1152921504606846976);
204:            board.setBitboard(black, rook, 9223372036854775808);
205:            board.update();
206:            board.nextColour = black;
207:            move = Move(60, 62, kingSideCastling, king, &board);
208:            move.applyMove(&board);
209:            EXPECT_EQ(board.getPieceBitboard(black, king), 4611686018427387904);
210:            EXPECT_EQ(board.getPieceBitboard(black, rook), 2305843009213693952);
211:            EXPECT_EQ(board.allPieces, 6917529027641081856);
212:    }
213:
214:    TEST(Move, UpdateCastlingAvaliability)
215:    {
216:            Board board = Board();
217:            board.setBitboard(white, rook, 1);
218:            board.setBitboard(white, king, 16);
219:            board.canWhiteCastleQueenSide = true;
220:            board.update();
221:            board.nextColour = white;
222:            Move move = Move(0, 8, quietMove, rook, &board);
223:            move.applyMove(&board);
224:            EXPECT_EQ(board.canWhiteCastleQueenSide, false);
```

```
225:
226:            board = Board();
227:            board.setBitboard(white, rook, 128);
228:            board.setBitboard(white, king, 16);
229:            board.canWhiteCastleKingSide = true;
230:            board.update();
231:            board.nextColour = white;
232:            move = Move(7, 15, quietMove, rook, &board);
233:            move.applyMove(&board);
234:            EXPECT_EQ(board.canWhiteCastleKingSide, false);
235:
236:            board = Board();
237:            board.setBitboard(white, rook, 129);
238:            board.setBitboard(white, king, 16);
239:            board.canWhiteCastleQueenSide = true;
240:            board.canWhiteCastleKingSide = true;
241:            board.update();
242:            board.nextColour = white;
243:            move = Move(4, 5, quietMove, king, &board);
244:            move.applyMove(&board);
245:            EXPECT_EQ(board.canWhiteCastleKingSide, false);
246:            EXPECT_EQ(board.canWhiteCastleQueenSide, false);
247:
248:            board = Board();
249:            board.setBitboard(white, rook, 72057594037927936);
250:            board.setBitboard(white, king, 1152921504606846976);
251:            board.canBlackCastleQueenSide = true;
252:            board.update();
253:            board.nextColour = black;
254:            move = Move(56, 57, quietMove, rook, &board);
255:            move.applyMove(&board);
256:            EXPECT_EQ(board.canBlackCastleQueenSide, false);
257:
258:            board = Board();
259:            board.setBitboard(white, rook, 9223372036854775808);
260:            board.setBitboard(white, king, 1152921504606846976);
261:            board.canBlackCastleKingSide = true;
262:            board.update();
263:            board.nextColour = black;
264:            move = Move(63, 62, quietMove, rook, &board);
265:            move.applyMove(&board);
266:            EXPECT_EQ(board.canBlackCastleKingSide, false);
267:
268:            board = Board();
269:            board.setBitboard(white, rook, 9295429630892703744);
270:            board.setBitboard(white, king, 1152921504606846976);
271:            board.canBlackCastleKingSide = true;
272:            board.canBlackCastleQueenSide = true;
273:            board.update();
274:            board.nextColour = black;
275:            move = Move(60, 59, quietMove, king, &board);
276:            move.applyMove(&board);
277:            EXPECT_EQ(board.canBlackCastleKingSide, false);
278:            EXPECT_EQ(board.canBlackCastleQueenSide, false);
279:    }
280:
281:    TEST(Move, IncrementingScores)
282:    {
283:            Board board = Board("8/8/8/8/4r3/3P4/8/8 w - - 0 1 ");
284:            EXPECT_EQ(board.getMaterialScore(white), -400);
285:            EXPECT_EQ(board.getMaterialScore(black), 400);
286:            EXPECT_EQ(board.getOnlyMaterialScore(black), 500);
287:            EXPECT_EQ(board.getOnlyMaterialScore(white), 0);
288:
289:            Move move = moveFromNotation("d3e4", &board);
290:            move.applyMove(&board);
291:
292:            EXPECT_EQ(board.getMaterialScore(white), 100);
293:            EXPECT_EQ(board.getMaterialScore(black), -100);
294:            EXPECT_EQ(board.getOnlyMaterialScore(black), 0);
295:            EXPECT_EQ(board.getOnlyMaterialScore(white), 0);
296:
297:            move.undoMove(&board);
298:
299:            EXPECT_EQ(board.getMaterialScore(white), -400);
300:            EXPECT_EQ(board.getMaterialScore(black), 400);
301:            EXPECT_EQ(board.getOnlyMaterialScore(black), 500);
302:            EXPECT_EQ(board.getOnlyMaterialScore(white), 0);
303:    }
304:
305:    TEST(Move, IncrementingZorbistKeys)
306:    {
307:            Board board;
308:            board.defaults();
309:
310:            Move move = moveFromNotation("a2a3", &board);
311:            move.applyMove(&board);
312:            uint64_t key1 = board.zorbistKey;
313:            board.generateZorbistKey();
314:            uint64_t key2 = board.zorbistKey;
315:            EXPECT_EQ(key1, key2);
316:
317:
318:            board.defaults();
319:            move = moveFromNotation("a2a4", &board);
320:            move.applyMove(&board);
321:            key1 = board.zorbistKey;
322:            board.generateZorbistKey();
323:            key2 = board.zorbistKey;
324:            EXPECT_EQ(key1, key2);
325:
326:            board.loadFromFen("8/P7/8/8/8/8/8/8 w - -");
327:            move = moveFromNotation("a7a8q", &board);
328:            move.applyMove(&board);
329:
330:            key1 = board.zorbistKey;
331:            board.generateZorbistKey();
332:            key2 = board.zorbistKey;
333:            EXPECT_EQ(key1, key2);
334:    }
335:
```

UnitTesting\MoveGeneration.cpp

```
001:  #pragma once
002:  #include "gtest/gtest.h"
003:  #include "move.h"
004:  #include "moveGeneration.h"
005:  #include "magicBitboards.h"
006:  #include "utils.h"
007:
008:  TEST(MoveGeneration, PawnMoves)
009:  {
010:          Board board = Board();
011:          board.setBitboard(white, pawn, 2);
012:          board.update();
013:          board.nextColour = white;
014:          std::array<Move, 150> Movelist;
015:          int arraySize = generatePawnMoves(&board, &Movelist, 0, ~0, ~0, 0);
016:          EXPECT_EQ(arraySize, 1);
017:          EXPECT_EQ(Movelist[0].from, 1);
018:          EXPECT_EQ(Movelist[0].to, 9);
019:          EXPECT_EQ(Movelist[0].piece, pawn);
020:          EXPECT_EQ(Movelist[0].moveType, quietMove);
021:
022:          board = Board();
023:          board.setBitboard(white, pawn, 256);
024:          board.update();
025:          board.nextColour = white;
026:          arraySize = generatePawnMoves(&board, &Movelist, 0, ~0, ~0, 0);
027:          EXPECT_EQ(arraySize, 2);
028:          EXPECT_EQ(Movelist[0].from, 8);
029:          EXPECT_EQ(Movelist[1].from, 8);
030:          if (Movelist[0].to == 16)
031:          {
032:                  EXPECT_EQ(Movelist[0].piece, pawn);
033:                  EXPECT_EQ(Movelist[0].moveType, quietMove);
034:
035:                  EXPECT_EQ(Movelist[1].piece, pawn);
036:                  EXPECT_EQ(Movelist[1].moveType, pawnDoubleMove);
037:                  EXPECT_EQ(Movelist[1].to, 24);
038:          }
039:          else
040:          {
041:                  EXPECT_EQ(Movelist[1].piece, pawn);
042:                  EXPECT_EQ(Movelist[1].moveType, quietMove);
043:
044:                  EXPECT_EQ(Movelist[0].piece, pawn);
045:                  EXPECT_EQ(Movelist[0].moveType, pawnDoubleMove);
046:                  EXPECT_EQ(Movelist[0].to, 24);
047:          }
048:
049:          board = Board();
050:          board.setBitboard(white, pawn, 1);
051:          board.setBitboard(white, king, 256);
052:          board.setBitboard(black, pawn, 512);
053:          board.update();
054:          board.nextColour = white;
055:          arraySize = generatePawnMoves(&board, &Movelist, 0, ~0, ~0, 0);
056:          EXPECT_EQ(arraySize, 1);
057:          EXPECT_EQ(Movelist[0].from, 0);
058:          EXPECT_EQ(Movelist[0].to, 9);
059:          EXPECT_EQ(Movelist[0].piece, pawn);
060:          EXPECT_EQ(Movelist[0].moveType, capture);
061:
062:          board = Board();
063:          board.setBitboard(white, pawn, 32768);
064:          board.setBitboard(black, king, 4194304);
065:          board.setBitboard(black, pawn, 8388608);
066:          board.update();
067:          board.nextColour = white;
068:          arraySize = generatePawnMoves(&board, &Movelist, 0, ~0, ~0, 0);
069:          EXPECT_EQ(arraySize, 1);
070:          EXPECT_EQ(Movelist[0].from, 15);
071:          EXPECT_EQ(Movelist[0].to, 22);
072:          EXPECT_EQ(Movelist[0].piece, pawn);
073:          EXPECT_EQ(Movelist[0].moveType, capture);
074:
075:          board = Board();
076:          board.setBitboard(white, pawn, 281474976710656);
077:          board.update();
078:          board.nextColour = white;
079:          arraySize = generatePawnMoves(&board, &Movelist, 0, ~0, ~0, 0);
080:          EXPECT_EQ(arraySize, 4);
081:          EXPECT_EQ(Movelist[0].from, 48);
082:          EXPECT_EQ(Movelist[0].to, 56);
083:          EXPECT_EQ(Movelist[0].piece, pawn);
084:
085:          board = Board();
086:          board.setBitboard(black, pawn, 256);
087:          board.update();
088:          board.nextColour = black;
089:          arraySize = generatePawnMoves(&board, &Movelist, 0, ~0, ~0, 0);
090:          EXPECT_EQ(arraySize, 4);
091:          EXPECT_EQ(Movelist[0].from, 8);
092:          EXPECT_EQ(Movelist[0].to, 0);
093:          EXPECT_EQ(Movelist[0].piece, pawn);
094:
095:  }
096:
097:  TEST(MoveGeneration, KingMoves)
098:  {
099:          std::array<Move,150> movelist;
100:          Board board = Board();
101:          board.setBitboard(white, king, 1);
102:          board.setBitboard(white, pawn, 768);
103:          board.setBitboard(black, pawn, 2);
104:          board.update();
105:          board.nextColour = white;
106:          int arraySize = generateKingMoves(&board, &movelist, board.whitePieces, board.blackPieces, 0,0);
107:          EXPECT_EQ(arraySize, 1);
108:          EXPECT_EQ(movelist[0].from, 0);
109:          EXPECT_EQ(movelist[0].to, 1);
110:          EXPECT_EQ(movelist[0].piece, king);
```

```
111:            EXPECT_EQ(movelist[0].moveType, capture);
112:
113:            board = Board();
114:            board.setBitboard(white, king, 1);
115:            board.setBitboard(white, pawn, 512);
116:            board.setBitboard(white, rook, 2);
117:            board.update();
118:            board.nextColour = white;
119:            arraySize = generateKingMoves(&board, &movelist, board.whitePieces, board.blackPieces, 0, 0);
120:            EXPECT_EQ(arraySize, 1);
121:            EXPECT_EQ(movelist[0].from, 0);
122:            EXPECT_EQ(movelist[0].to, 8);
123:            EXPECT_EQ(movelist[0].piece, king);
124:            EXPECT_EQ(movelist[0].moveType, quietMove);
125:
126:            board = Board();
127:            board.setBitboard(black, king, 922337203685477580 8);
128:            board.setBitboard(black, pawn, 464771481544635 1872);
129:            board.setBitboard(white, rook, 1801439850948 1984);
130:            board.update();
131:            board.nextColour = black;
132:            arraySize = generateKingMoves(&board, &movelist, board.blackPieces, board.whitePieces, 0, 0);
133:            EXPECT_EQ(arraySize, 1);
134:            EXPECT_EQ(movelist[0].from, 63);
135:            EXPECT_EQ(movelist[0].to, 54);
136:            EXPECT_EQ(movelist[0].piece, king);
137:            EXPECT_EQ(movelist[0].moveType, capture);
138:    }
139:
140:    TEST(MoveGeneration, KnightMoves)
141:    {
142:            std::array<Move, 150> movelist;
143:            Board board = Board();
144:            board.setBitboard(white, knight, 1);
145:            board.setBitboard(white, pawn, 1024);
146:            board.update();
147:            board.nextColour = white;
148:            int arraySize = generateKnightMoves(&board, &movelist, board.whitePieces, board.blackPieces, 0, ~0, ~0,0);
149:            EXPECT_EQ(arraySize, 1);
150:            EXPECT_EQ(movelist[0].from, 0);
151:            EXPECT_EQ(movelist[0].to, 17);
152:            EXPECT_EQ(movelist[0].piece, knight);
153:            EXPECT_EQ(movelist[0].moveType, quietMove);
154:
155:            board = Board();
156:            board.setBitboard(white, knight, 922337203685477580 8);
157:            board.setBitboard(white, pawn, 70368744177664);
158:            board.setBitboard(black, rook, 900719925474 0992);
159:            board.update();
160:            board.nextColour = white;
161:            arraySize = generateKnightMoves(&board, &movelist, board.whitePieces, board.blackPieces, 0, ~0, ~0, 0);
162:            EXPECT_EQ(arraySize, 1);
163:            EXPECT_EQ(movelist[0].from, 63);
164:            EXPECT_EQ(movelist[0].to, 53);
165:            EXPECT_EQ(movelist[0].piece, knight);
166:            EXPECT_EQ(movelist[0].moveType, capture);
167:    }
168:
169:    TEST(MoveGeneration, RookMoves)
170:    {
171:            std::array<Move, 150> movelist;
172:            Board board = Board();
173:            board.setBitboard(white, rook, 1);
174:            board.setBitboard(white, pawn, 260);
175:            board.update();
176:            board.nextColour = white;
177:            int arraySize = generateRookMoves(&board, &movelist, board.whitePieces, board.blackPieces, 0, ~0, ~0, 0);
178:            EXPECT_EQ(arraySize, 1);
179:            EXPECT_EQ(movelist[0].from, 0);
180:            EXPECT_EQ(movelist[0].to, 1);
181:            EXPECT_EQ(movelist[0].piece, rook);
182:            EXPECT_EQ(movelist[0].moveType, quietMove);
183:
184:            board = Board();
185:            board.setBitboard(white, rook, 1);
186:            board.setBitboard(white, pawn, 2);
187:            board.setBitboard(black, pawn, 256);
188:            board.update();
189:            board.nextColour = white;
190:            arraySize = generateRookMoves(&board, &movelist, board.whitePieces, board.blackPieces, 0, ~0, ~0, 0);
191:            EXPECT_EQ(arraySize, 1);
192:            EXPECT_EQ(movelist[0].from, 0);
193:            EXPECT_EQ(movelist[0].to, 8);
194:            EXPECT_EQ(movelist[0].piece, rook);
195:            EXPECT_EQ(movelist[0].moveType, capture);
196:    }
197:
198:    TEST(MoveGeneration, BishopMoves)
199:    {
200:            std::array<Move, 150> movelist;
201:            Board board = Board();
202:            board.setBitboard(white, bishop, 1);
203:            board.setBitboard(white, pawn, 262144);
204:            board.update();
205:            board.nextColour = white;
206:            int arraySize = generateBishopMoves(&board, &movelist, board.whitePieces, board.blackPieces, 0, ~0, ~0, 0);
207:            EXPECT_EQ(arraySize, 1);
208:            EXPECT_EQ(movelist[0].from, 0);
209:            EXPECT_EQ(movelist[0].to, 9);
210:            EXPECT_EQ(movelist[0].piece, bishop);
211:            EXPECT_EQ(movelist[0].moveType, quietMove);
212:
213:            board = Board();
214:            board.setBitboard(white,bishop, 128);
215:            board.setBitboard(white, pawn, 2097216);
216:            board.update();
217:            board.nextColour = white;
218:            arraySize = generateBishopMoves(&board, &movelist, board.whitePieces, board.blackPieces, 0, ~0, ~0, 0);
219:            EXPECT_EQ(arraySize, 1);
220:            EXPECT_EQ(movelist[0].from, 7);
221:            EXPECT_EQ(movelist[0].to, 14);
222:            EXPECT_EQ(movelist[0].piece, bishop);
223:            EXPECT_EQ(movelist[0].moveType, quietMove);
224:
```

```
225:            board = Board();
226:            board.setBitboard(white, bishop, 134217728);
227:            board.setBitboard(white, pawn, 68720787456);
228:            board.setBitboard(black, pawn, 17179869184);
229:            board.update();
230:            board.nextColour = white;
231:            arraySize = generateBishopMoves(&board, &movelist, board.whitePieces, board.blackPieces, 0, ~0, ~0, 0);
232:            EXPECT_EQ(arraySize, 1);
233:            EXPECT_EQ(movelist[0].from, 27);
234:            EXPECT_EQ(movelist[0].to, 34);
235:            EXPECT_EQ(movelist[0].piece, bishop);
236:            EXPECT_EQ(movelist[0].moveType, capture);
237:
238:            board = Board();
239:            board.setBitboard(white, bishop, 4);
240:            board.setBitboard(white, pawn, 1536);
241:            board.setBitboard(black, knight, 1048576);
242:            board.update();
243:            board.nextColour = white;
244:            arraySize = generateBishopMoves(&board, &movelist, board.whitePieces, board.blackPieces, 0, ~0, ~0, 0);
245:            EXPECT_EQ(arraySize, 2);
246:            if (movelist[0].to == 11)
247:            {
248:                    EXPECT_EQ(movelist[1].from, 2);
249:                    EXPECT_EQ(movelist[1].to, 20);
250:                    EXPECT_EQ(movelist[1].piece, bishop);
251:                    EXPECT_EQ(movelist[1].moveType, capture);
252:            }
253:            else
254:            {
255:                    EXPECT_EQ(movelist[0].from, 2);
256:                    EXPECT_EQ(movelist[0].to, 20);
257:                    EXPECT_EQ(movelist[0].piece, bishop);
258:                    EXPECT_EQ(movelist[0].moveType, capture);
259:            }
260:    }
261:
262:    TEST(MoveGeneration, QueenMoves)
263:    {
264:            std::array<Move, 150> movelist;
265:            Board board = Board();
266:            board.setBitboard(white, queen, 1);
267:            board.setBitboard(white, pawn, 262402);
268:            board.update();
269:            board.nextColour = white;
270:            int arraySize = generateQueenMoves(&board, &movelist, board.whitePieces, board.blackPieces, 0, ~0, ~0, 0);
271:            EXPECT_EQ(arraySize, 1);
272:            EXPECT_EQ(movelist[0].from, 0);
273:            EXPECT_EQ(movelist[0].to, 9);
274:            EXPECT_EQ(movelist[0].piece, queen);
275:            EXPECT_EQ(movelist[0].moveType, quietMove);
276:
277:            board = Board();
278:            board.setBitboard(white, queen, 128);
279:            board.setBitboard(white, pawn, 2129984);
280:            board.update();
281:            board.nextColour = white;
282:            arraySize = generateQueenMoves(&board, &movelist, board.whitePieces, board.blackPieces, 0, ~0, ~0, 0);
283:            EXPECT_EQ(arraySize, 1);
284:            EXPECT_EQ(movelist[0].from, 7);
285:            EXPECT_EQ(movelist[0].to, 14);
286:            EXPECT_EQ(movelist[0].piece, queen);
287:            EXPECT_EQ(movelist[0].moveType, quietMove);
288:
289:            board = Board();
290:            board.setBitboard(white, queen, 1);
291:            board.setBitboard(white, pawn, 772);
292:            board.update();
293:            board.nextColour = white;
294:            arraySize = generateQueenMoves(&board, &movelist, board.whitePieces, board.blackPieces, 0, ~0, ~0, 0);
295:            EXPECT_EQ(arraySize, 1);
296:            EXPECT_EQ(movelist[0].from, 0);
297:            EXPECT_EQ(movelist[0].to, 1);
298:            EXPECT_EQ(movelist[0].piece, queen);
299:            EXPECT_EQ(movelist[0].moveType, quietMove);
300:
301:            board = Board();
302:            board.setBitboard(white, queen, 2097152);
303:            board.setBitboard(white, pawn, 1615884288);
304:            board.setBitboard(black, knight, 268435456);
305:            board.update();
306:            board.nextColour = white;
307:            arraySize = generateQueenMoves(&board, &movelist, board.whitePieces, board.blackPieces, 0, ~0, ~0, 0);
308:            EXPECT_EQ(arraySize, 1);
309:            EXPECT_EQ(movelist[0].from, 21);
310:            EXPECT_EQ(movelist[0].to, 28);
311:            EXPECT_EQ(movelist[0].piece, queen);
312:            EXPECT_EQ(movelist[0].moveType, capture);
313:
314:            board = Board();
315:            board.setBitboard(white, queen, 67108864);
316:            board.setBitboard(white, pawn, 60264677376);
317:            board.setBitboard(black, knight, 16777216);
318:            board.update();
319:            board.nextColour = white;
320:            arraySize = generateQueenMoves(&board, &movelist, board.whitePieces, board.blackPieces, 0, ~0, ~0, 0);
321:            EXPECT_EQ(arraySize, 2);
322:            if (movelist[0].to == 24)
323:            {
324:                    EXPECT_EQ(movelist[0].from, 26);
325:                    EXPECT_EQ(movelist[0].to, 24);
326:                    EXPECT_EQ(movelist[0].piece, queen);
327:                    EXPECT_EQ(movelist[0].moveType, capture);
328:            }
329:            else
330:            {
331:                    EXPECT_EQ(movelist[1].from, 26);
332:                    EXPECT_EQ(movelist[1].to, 24);
333:                    EXPECT_EQ(movelist[1].piece, queen);
334:                    EXPECT_EQ(movelist[1].moveType, capture);
335:            }
336:
337:
338:            board = Board();
```

```
339:            board.setBitboard(white, queen, 4194304);
340:            board.setBitboard(white, pawn, 3768623104);
341:            board.setBitboard(black, queen, 64);
342:            board.update();
343:            board.nextColour = white;
344:            arraySize = generateQueenMoves(&board, &movelist, board.whitePieces, board.blackPieces, 0, ~0, ~0, 0);
345:            EXPECT_EQ(arraySize, 2);
346:            if (movelist[0].to == 6)
347:            {
348:                    EXPECT_EQ(movelist[0].from, 22);
349:                    EXPECT_EQ(movelist[0].to, 6);
350:                    EXPECT_EQ(movelist[0].piece, queen);
351:                    EXPECT_EQ(movelist[0].moveType, capture);
352:            }
353:            else
354:            {
355:                    EXPECT_EQ(movelist[1].from, 22);
356:                    EXPECT_EQ(movelist[1].to, 6);
357:                    EXPECT_EQ(movelist[1].piece, queen);
358:                    EXPECT_EQ(movelist[1].moveType, capture);
359:            }
360:    }
361:
362:    TEST(MoveGeneration, CastlingMoves)
363:    {
364:            std::array<Move, 150> movelist;
365:            Board board = Board();
366:            board.setBitboard(white, king, 16);
367:            board.setBitboard(white, rook, 128);
368:            board.canWhiteCastleKingSide = true;
369:            board.update();
370:            board.nextColour = white;
371:            int arraySize = generateCastlingMoves(&board, &movelist, board.whitePieces, board.blackPieces, 0, 0);
372:            EXPECT_EQ(arraySize, 1);
373:            EXPECT_EQ(movelist[0].from, 4);
374:            EXPECT_EQ(movelist[0].to, 6);
375:            EXPECT_EQ(movelist[0].piece, king);
376:            EXPECT_EQ(movelist[0].moveType, kingSideCastling);
377:
378:            board = Board();
379:            board.setBitboard(white, king, 16);
380:            board.setBitboard(white, rook, 1);
381:            board.canWhiteCastleQueenSide = true;
382:            board.update();
383:            board.nextColour = white;
384:            arraySize = generateCastlingMoves(&board, &movelist, board.whitePieces, board.blackPieces, 0, 0);
385:            EXPECT_EQ(arraySize, 1);
386:            EXPECT_EQ(movelist[0].from, 4);
387:            EXPECT_EQ(movelist[0].to, 2);
388:            EXPECT_EQ(movelist[0].piece, king);
389:            EXPECT_EQ(movelist[0].moveType, queenSideCastling);
390:
391:            board = Board();
392:            board.setBitboard(black, king, 1152921504606846976);
393:            board.setBitboard(black, rook, 9223372036854775808);
394:            board.canBlackCastleKingSide = true;
395:            board.update();
396:            board.nextColour = black;
397:            arraySize = generateCastlingMoves(&board, &movelist, board.whitePieces, board.blackPieces, 0, 0);
398:            EXPECT_EQ(arraySize, 1);
399:            EXPECT_EQ(movelist[0].from, 60);
400:            EXPECT_EQ(movelist[0].to, 62);
401:            EXPECT_EQ(movelist[0].piece, king);
402:            EXPECT_EQ(movelist[0].moveType, kingSideCastling);
403:
404:            board = Board();
405:            board.setBitboard(black, king, 1152921504606846976);
406:            board.setBitboard(black, rook, 72057594037927936);
407:            board.canBlackCastleQueenSide = true;
408:            board.update();
409:            board.nextColour = black;
410:            arraySize = generateCastlingMoves(&board, &movelist, board.whitePieces, board.blackPieces, 0, 0);
411:            EXPECT_EQ(arraySize, 1);
412:            EXPECT_EQ(movelist[0].from, 60);
413:            EXPECT_EQ(movelist[0].to, 58);
414:            EXPECT_EQ(movelist[0].piece, king);
415:            EXPECT_EQ(movelist[0].moveType, queenSideCastling);
416:    }
417:
418:    uint64_t perft(int depth, Board* board, int divide)
419:    {
420:            std::array<Move,150> moveList;
421:
422:            int arraySize = searchForMoves(board,&moveList);
423:            int perftVal;
424:            uint64_t nodes = 0;
425:
426:            if (depth == 0) return 1;
427:
428:            for (int x = 0; x < arraySize; x++)
429:            {
430:                    moveList[x].applyMove(board);
431:                    perftVal = perft(depth - 1, board, divide);
432:                    nodes += perftVal;
433:                    if (depth == divide) std::cout << notationFromMove(moveList[x]) << " " << perftVal << "\n";
434:                    moveList[x].undoMove(board);
435:            }
436:            return nodes;
437:    }
438:
439:    TEST(MoveGeneration, Perft)
440:    {
441:            Board board;
442:
443:            //time_t startTime = time(0);
444:            //board.defaults();
445:            //perft(6, &board, -1);
446:            //std::cout << (uint64_t)((119060324) / difftime(time(NULL), startTime)) << "\n";
447:
448:            board.defaults();
449:            EXPECT_EQ(perft(0, &board, -1), 1);
450:            EXPECT_EQ(perft(1, &board, -1), 20);
451:            EXPECT_EQ(perft(2, &board, -1), 400);
452:            EXPECT_EQ(perft(3, &board, -1), 8902);
```

```
453:            EXPECT_EQ(perft(4, &board, -1), 197281);
454:
455:            board = Board();
456:            board.loadFromFen("r3k2r/p1ppqpb1/bn2pnp1/3PN3/1p2P3/2N2Q1p/PPPBBPPP/R3K2R w KQkq -");
457:            EXPECT_EQ(perft(0, &board, -1), 1);
458:            EXPECT_EQ(perft(1, &board, -1), 48);
459:            EXPECT_EQ(perft(2, &board, -1), 2039);
460:            EXPECT_EQ(perft(3, &board, -1), 97862);
461:            EXPECT_EQ(perft(4, &board, -1), 4085603);
462:
463:            board = Board();
464:            board.loadFromFen("8/2p5/3p4/KP5r/1R3p1k/8/4P1P1/8 w - -");
465:            EXPECT_EQ(perft(0, &board, -1), 1);
466:            EXPECT_EQ(perft(1, &board, -1), 14);
467:            EXPECT_EQ(perft(2, &board, -1), 191);
468:            EXPECT_EQ(perft(3, &board, -1), 2812);
469:            EXPECT_EQ(perft(4, &board, -1), 43238);
470:
471:            board = Board();
472:            board.loadFromFen("r3k2r/Pppp1ppp/1b3nbN/nP6/BBP1P3/q4N2/Pp1P2PP/R2Q1RK1 w kq - 0 1");
473:            EXPECT_EQ(perft(0, &board, -1), 1);
474:            EXPECT_EQ(perft(1, &board, -1), 6);
475:            EXPECT_EQ(perft(2, &board, -1), 264);
476:            EXPECT_EQ(perft(3, &board, -1), 9467);
477:            EXPECT_EQ(perft(4, &board, -1), 422333);
478:
479:            board = Board();
480:            board.loadFromFen("rnbq1k1r/pp1Pbppp/2p5/8/2B5/8/PPP1NnPP/RNBQK2R w KQ - 1 8");
481:            EXPECT_EQ(perft(0, &board, -1), 1);
482:            EXPECT_EQ(perft(1, &board, -1), 44);
483:            EXPECT_EQ(perft(2, &board, -1), 1486);
484:            EXPECT_EQ(perft(3, &board, -1), 62379);
485:            EXPECT_EQ(perft(4, &board, -1), 2103487);
486:
487:            board = Board();
488:            board.loadFromFen("r4rk1/1pp1qppp/p1np1n2/2b1p1B1/2B1P1b1/P1NP1N2/1PP1QPPP/R4RK1 w - - 0 10");
489:            EXPECT_EQ(perft(0, &board, -1), 1);
490:            EXPECT_EQ(perft(1, &board, -1), 46);
491:            EXPECT_EQ(perft(2, &board, -1), 2079);
492:            EXPECT_EQ(perft(3, &board, -1), 89890);
493:            EXPECT_EQ(perft(4, &board, -1), 3894594);
494:    }
```

UnitTesting\MoveOrdering.cpp

```
001:    #pragma once
002:    #include "gtest/gtest.h"
003:    #include "moveOrdering.h"
004:    #include "utils.h"
005:
006:    TEST(MoveOrdering, SEE)
007:    {
008:            Board board = Board("1k1r4/1pp4p/p7/4p3/8/P5P1/1PP4P/2K1R3 w - -");
009:            Move move = moveFromNotation("e1e5", &board);
010:            EXPECT_EQ(SEE(&move, &board), 100);
011:
012:            board = Board("1k1r3q/1ppn3p/p4b2/4p3/8/P2N2P1/1PP1R1BP/2K1Q3 w - - ");
013:            move = moveFromNotation("d3e5", &board);
014:            EXPECT_EQ(SEE(&move, &board), -1000);
015:
016:            board = Board("8/3n4/8/4N3/3P4/8/8/8 b - - 0 1 ");
017:            move = moveFromNotation("d7e5", &board);
018:            EXPECT_EQ(SEE(&move, &board), 0);
019:    }
```

UnitTesting\Scoring.cpp

```
001:  #pragma once
002:  #include "gtest/gtest.h"
003:  #include "board.h"
004:  #include "scoring.h"
005:
006:  TEST(Scoring, pawnStructureScore)
007:  {
008:          Board board;
009:          board.loadFromFen("8/8/8/8/1P6/1P6/8/8 w - - 0 1");
010:          EXPECT_EQ(calculatePawnStructureScore(&board), 50 - 0);
011:
012:          board.loadFromFen("8/5p2/5p2/8/8/8/8/8 b - - 0 1");
013:          EXPECT_EQ(calculatePawnStructureScore(&board), 10 - 0);
014:
015:          board.loadFromFen("8/8/1P6/1p6/8/2P5/8/8 w - - 0 1 ");
016:          EXPECT_EQ(calculatePawnStructureScore(&board), 92 + 20);
017:
018:          board.loadFromFen("8/8/8/8/8/5P2/4Pp2/8 b - - 0 1 ");
019:          EXPECT_EQ(calculatePawnStructureScore(&board), 100 -52);
020:
021:          board.loadFromFen("8/8/8/8/8/5Pp1/4Pp2/8 w - - 0 1");
022:          EXPECT_EQ(calculatePawnStructureScore(&board), 52 - 212);
023:  }
024:
025:  //Need to be updated for midgame/lategame transition
026:  /*
027:  TEST(Scoring, kingStructureScore)
028:  {
029:          Board board;
030:          board.loadFromFen("k7/8/8/8/8/4P3/2PP4/3K4 b - - 0 1 ");
031:          EXPECT_EQ(calculateKingSafetyScore(&board), -92 - 0);
032:
033:          board.loadFromFen("k7/8/8/4p3/8/8/2PP4/3K4 b - - 0 1");
034:          EXPECT_EQ(calculateKingSafetyScore(&board), -92 -0);
035:
036:          board.loadFromFen("k7/8/8/8/5P2/6P1/8/6K1 b - - 0 1 ");
037:          EXPECT_EQ(calculateKingSafetyScore(&board), -100 + 82);
038:
039:          board.loadFromFen("k7/ppp5/1PP5/8/7p/6pP/5PP1/6K1 b - - 0 1 ");
040:          EXPECT_EQ(calculateKingSafetyScore(&board), -30 +32);
041:  }
042:  */
043:  TEST(Scoring, rookPositionalScore)
044:  {
045:          Board board;
046:          board.loadFromFen("k1r1r3/8/8/8/8/8/2P2P2/5R1K w - - 0 1");
047:          EXPECT_EQ(calculateRookPositionScore(&board), 0 - 25);
048:
049:          board.loadFromFen("k3r3/5pR1/8/8/8/8/8/5R1K w - - 0 1");
050:          EXPECT_EQ(calculateRookPositionScore(&board), 45 - 15);
051:  }
052:  /*
053:  TEST(Scoring, calculateScoreDiff)
054:  {
055:          Board board = Board("5rk1/pp2npp1/2p1r2p/2qpP2P/P3P3/2B2PQ1/2P2P2/3RR1K1 w - - 1 28");
056:
057:          EXPECT_EQ(calculatePawnStructureScore(&board), -96);
058:          EXPECT_EQ(calculateKingSafetyScore(&board), -24);
059:          EXPECT_EQ(calculateMaterialScore(&board), 21);
060:
061:  }*/
```

UnitTesting\Utils.cpp

```
001:  #pragma once
002:  #include "gtest/gtest.h"
003:  #include "utils.h"
004:  #include "move.h"
005:
006:  TEST(Utils, notationFromMove)
007:  {
008:  	Board board;
009:  	EXPECT_EQ(notationFromMove(Move(0, 8, quietMove, pawn, &board)), "a1a2");
010:  	EXPECT_EQ(notationFromMove(Move(3, 11, quietMove, pawn, &board)), "d1d2");
011:  	EXPECT_EQ(notationFromMove(Move(32, 40, quietMove, pawn, &board)), "a5a6");
012:  	EXPECT_EQ(notationFromMove(Move(54, 62, quietMove, pawn, &board)), "g7g8");
013:  	EXPECT_EQ(notationFromMove(Move(48, 56, knightPromotion, pawn, &board)), "a7a8n");
014:  	EXPECT_EQ(notationFromMove(Move(8, 0, queenPromotion, pawn, &board)), "a2a1q");
015:
016:  }
017:
018:  TEST(Utils, moveFromNotation)
019:  {
020:  	Board board = Board();
021:  	board.defaults();
022:  	Move move = moveFromNotation("a2a3", &board);
023:  	EXPECT_EQ(move.from, 8);
024:  	EXPECT_EQ(move.to, 16);
025:  	EXPECT_EQ(move.moveType, quietMove);
026:  	EXPECT_EQ(move.piece, pawn);
027:
028:  	move = moveFromNotation("h2h4", &board);
029:  	EXPECT_EQ(move.from, 15);
030:  	EXPECT_EQ(move.to, 31);
031:  	EXPECT_EQ(move.moveType, pawnDoubleMove);
032:  	EXPECT_EQ(move.piece, pawn);
033:
034:  	move = moveFromNotation("a7a6", &board);
035:  	EXPECT_EQ(move.from, 48);
036:  	EXPECT_EQ(move.to, 40);
037:  	EXPECT_EQ(move.moveType, quietMove);
038:  	EXPECT_EQ(move.piece, pawn);
039:
040:  	board = Board();
041:  	board.setBitboard(white, bishop, 2);
042:  	board.setBitboard(black, pawn, 256);
043:  	board.update();
044:  	move = moveFromNotation("b1a2", &board);
045:  	EXPECT_EQ(move.from, 1);
046:  	EXPECT_EQ(move.to, 8);
047:  	EXPECT_EQ(move.moveType, capture);
048:  	EXPECT_EQ(move.piece, bishop);
049:
050:  	board = Board();
051:  	board.setBitboard(black, king, 34359738368);
052:  	board.setBitboard(white, pawn, 134217728);
053:  	board.update();
054:  	move = moveFromNotation("d5d4", &board);
055:  	EXPECT_EQ(move.from, 35);
056:  	EXPECT_EQ(move.to, 27);
057:  	EXPECT_EQ(move.moveType, capture);
058:  	EXPECT_EQ(move.piece, king);
059:
060:  	board = Board();
061:  	board.setBitboard(white, pawn, 1);
062:  	board.setBitboard(black, pawn, 512);
063:  	board.update();
064:  	move = moveFromNotation("a1b2", &board);
065:  	EXPECT_EQ(move.from, 0);
066:  	EXPECT_EQ(move.to, 9);
067:  	EXPECT_EQ(move.moveType, capture);
068:  	EXPECT_EQ(move.piece, pawn);
069:
070:  	board = Board();
071:  	board.setBitboard(white, pawn, 281474976710656);
072:  	board.update();
073:  	move = moveFromNotation("a7a8q", &board);
074:  	EXPECT_EQ(move.from, 48);
075:  	EXPECT_EQ(move.to, 56);
076:  	EXPECT_EQ(move.moveType, queenPromotion);
077:  	EXPECT_EQ(move.piece, pawn);
078:
079:  	board = Board();
080:  	board.setBitboard(black, pawn, 256);
081:  	board.update();
082:  	move = moveFromNotation("a2a1n", &board);
083:  	EXPECT_EQ(move.from, 8);
084:  	EXPECT_EQ(move.to, 0);
085:  	EXPECT_EQ(move.moveType, knightPromotion);
086:  	EXPECT_EQ(move.piece, pawn);
087:
088:  	board = Board();
089:  	board.setBitboard(white, king, 16);
090:  	board.setBitboard(white, rook, 128);
091:  	board.update();
092:  	move = moveFromNotation("e1g1", &board);
093:  	EXPECT_EQ(move.from,4);
094:  	EXPECT_EQ(move.to, 6);
095:  	EXPECT_EQ(move.moveType, kingSideCastling);
096:  	EXPECT_EQ(move.piece, king);
097:
098:  	board = Board();
099:  	board.setBitboard(white, king, 16);
100:  	board.setBitboard(white, rook, 1);
101:  	board.update();
102:  	move = moveFromNotation("e1c1", &board);
103:  	EXPECT_EQ(move.from, 4);
104:  	EXPECT_EQ(move.to, 2);
105:  	EXPECT_EQ(move.moveType, queenSideCastling);
106:  	EXPECT_EQ(move.piece, king);
107:
108:  	board = Board();
109:  	board.setBitboard(black, king, 1152921504606846976);
110:  	board.setBitboard(black, rook, 9223372036854775808);
```

```
111:            board.update();
112:            move = moveFromNotation("e8g8", &board);
113:            EXPECT_EQ(move.from, 60);
114:            EXPECT_EQ(move.to, 62);
115:            EXPECT_EQ(move.moveType, kingSideCastling);
116:            EXPECT_EQ(move.piece, king);
117:
118:            board = Board();
119:            board.setBitboard(black, king, 1152921504606846976);
120:            board.setBitboard(black, rook, 72057594037927936);
121:            board.update();
122:            move = moveFromNotation("e8c8", &board);
123:            EXPECT_EQ(move.from, 60);
124:            EXPECT_EQ(move.to, 58);
125:            EXPECT_EQ(move.moveType, queenSideCastling);
126:            EXPECT_EQ(move.piece, king);
127:    }
128:
```

UnitTesting\test.cpp

```
001:    #pragma once
002:    #include "gtest/gtest.h"
003:    #include "magicBitboards.h"
004:    #include "bitboard.h"
005:    #include "moveGeneration.h"
006:
007:    #include <iostream>
008:    #include <stdint.h>
009:
010:    int main(int argc, char **argv)
011:    {
012:            magicBitboards temp = magicBitboards();
013:            temp.setupMagicBitboards();
014:            setupBitboardUtils();
015:            setupMoveGen();
016:            ZorbistKeys::initialize();
017:            ::testing::InitGoogleTest(&argc, argv);
018:            return RUN_ALL_TESTS();
019:    }
020:
021:
```

ChessUI\AIManager.cpp

```
001:  #include "AIManager.h"
002:
003:
004:
005:  AIManager::AIManager()
006:  {
007:      status = stopped;
008:      aiProcess = new QProcess(this);
009:      connect(aiProcess, SIGNAL(readyReadStandardOutput()), this, SLOT(processInputCallback()));
010:      engineOutputDialog.setupDialogBox();
011:  }
012:
013:  AIManager::~AIManager()
014:  {
015:      aiProcess->terminate();
016:      delete aiProcess;
017:  }
018:
019:  void AIManager::startAI()
020:  {
021:      if (status == stopped)
022:      {
023:          aiProcess->start("ChessEngine.exe");
024:          aiProcess->setReadChannel(QProcess::StandardOutput);
025:          sendCommand("isready");
026:          status = waitingForReady;
027:      }
028:  }
029:
030:  void AIManager::showEngineOutputDialog()
031:  {
032:      engineOutputDialog.show();
033:  }
034:
035:  void AIManager::findMove(Board board)
036:  {
037:      QString positionCommand = "position fen ";
038:      positionCommand += QString::fromStdString(board.exportAsFen());
039:      sendCommand(positionCommand);
040:
041:      sendCommand("go");
042:
043:      status = waitingForBestmove;
044:      lastBoardState = board;
045:
046:  }
047:
048:  void AIManager::sendCommand(QString command)
049:  {
050:      std::string commandWithNewline = command.toStdString() + "\n";
051:      aiProcess->write(commandWithNewline.c_str());
052:      engineOutputDialog.addNewLine(command);
053:  }
054:
055:  void AIManager::processInputCallback()
056:  {
057:      while (aiProcess->canReadLine())
058:      {
059:          QString input = aiProcess->readLine().simplified();
060:          engineOutputDialog.addNewLine(input);
061:          if (status == waitingForReady)
062:          {
063:              if (input.startsWith("readyok"))
064:              {
065:                  status = ready;
066:              }
067:          }
068:          else if (status == waitingForBestmove)
069:          {
070:              if (input.startsWith("bestmove"))
071:              {
072:                  emit newMove(moveFromNotation(input.remove(0,9).toStdString(), &lastBoardState));
073:                  status = ready;
074:              }
075:          }
076:      }
077:  }
```

ChessUI\AIManager.h

```
001:  #pragma once
002:  #include <QProcess>
003:  #include "move.h"
004:  #include "utils.h"
005:  #include "EngineOutputDialog.h"
006:
007:  enum AIStatus{stopped, ready, waitingForBestmove, waitingForReady};
008:
009:  class AIManager : public QObject
010:  {
011:      Q_OBJECT
012:
013:  public:
014:      AIManager();
015:      ~AIManager();
016:      void startAI();
017:      void showEngineOutputDialog();
018:      void findMove(Board board);
019:
020:  signals:
021:      void newMove(Move newMove);
022:
023:  private:
024:      AIStatus status;
025:      QProcess* aiProcess;
026:      EngineOutputDialog engineOutputDialog;
027:      void sendCommand(QString command);
028:      Board lastBoardState;
029:
030:  private slots:
031:      void processInputCallback();
032:  };
033:
034:
```

ChessUI\BoardDisplay.cpp

```
001:  #include "BoardDisplay.h"
002:
003:
004:
005:  BoardDisplay::BoardDisplay(QWidget *parent) : QGraphicsView(parent)
006:  {
007:  	movingPiece = nullptr;
008:  	isPlayersTurn = true;
009:
010:  	setHorizontalScrollBarPolicy(Qt::ScrollBarAlwaysOff);
011:  	setVerticalScrollBarPolicy(Qt::ScrollBarAlwaysOff);
012:
013:
014:  	setScene(&graphicsScene);
015:  	graphicsScene.setBackgroundBrush(Qt::green);
016:
017:  	positionLabelSize = 50;
018:  	squareSize = 100;
019:  	pieceSize = 75;
020:
021:  	setMinimumSize(squareSize * 8 + positionLabelSize * 2, squareSize * 8 + positionLabelSize * 2);
022:
023:  	resize(100 * 8, 100 * 8);
024:
025:  	//Setup move generation
026:  	magicBitboards temp = magicBitboards();
027:  	temp.setupMagicBitboards();
028:  	setupBitboardUtils();
029:  	setupMoveGen();
030:
031:  	blackSquarePixmap.load("blackSquare.png");
032:  	whiteSquarePixmap.load("whiteSquare.png");
033:
034:  	//Adds the chess squares.
035:  	for (int x = 0; x < 8; x++)
036:  	{
037:  		for (int y = 0; y < 8; y++)
038:  		{
039:  			//If it is a black square on the chess board
040:  			if (y % 2 == 0 & x % 2 == 0 || y % 2 == 1 & x % 2 == 1)
041:  				boardSquares[x][y].setPixmap(blackSquarePixmap);
042:  			else
043:  				boardSquares[x][y].setPixmap(whiteSquarePixmap);
044:
045:  			//Sets the offsets for the squares , flipping vertically as all internal chess representations start from
046:  			//bottom left hand corner, and qt starts from the top left.
047:  			boardSquares[x][y].setOffset(positionLabelSize + squareSize * x, positionLabelSize + squareSize * (7 - y));
048:
049:  			boardSquares[x][y].setScale(squareSize / boardSquares[x][y].boundingRect().width());
050:
051:  			boardSquares[x][y].setZValue(-1);
052:
053:  			graphicsScene.addItem(&boardSquares[x][y]);
054:  		}
055:  	}
056:
057:  	addPositionLabels();
058:  	loadChessPiecePixmaps();
059:  	newGame();
060:  }
061:
062:  void BoardDisplay::addPositionLabels()
063:  {
064:  	QFont font = QFont("Times", 15, QFont::Bold);
065:
066:  	for (int x = 0; x < 8; x++)
067:  	{
068:  		char equivalentLetter = 'a' + x;
069:  		char numericalLetter = '8' - x;
070:
071:  		//Top
072:  		QGraphicsTextItem*  newLabel = new QGraphicsTextItem;
073:  		newLabel->setFont(font);
074:  		newLabel->setPos(1.5 * positionLabelSize + squareSize * x, 0.5 * positionLabelSize);
075:  		newLabel->setPlainText(QString(equivalentLetter));
076:  		graphicsScene.addItem(newLabel);
077:  		positionLabelsTop[x] = newLabel;
078:
079:  		//Bottom
080:  		newLabel = new QGraphicsTextItem;
081:  		newLabel->setFont(font);
082:  		newLabel->setPos(1.5 * positionLabelSize + squareSize * x, 1.5 * positionLabelSize + 8 * squareSize);
083:  		newLabel->setPlainText(QString(equivalentLetter));
084:  		graphicsScene.addItem(newLabel);
085:  		positionLabelsBottom[x] = newLabel;
086:
087:  		//Left
088:  		newLabel = new QGraphicsTextItem;
089:  		newLabel->setFont(font);
090:  		newLabel->setPos(0.5 * positionLabelSize, 1.5 * positionLabelSize + x * squareSize);
091:  		newLabel->setPlainText(QString(numericalLetter));
092:  		graphicsScene.addItem(newLabel);
093:  		positionLabelsLeft[x] = newLabel;
094:
095:  		//Right
096:  		newLabel = new QGraphicsTextItem;
097:  		newLabel->setFont(font);
098:  		newLabel->setPos(1.5 * positionLabelSize + squareSize * 8, 1.5 * positionLabelSize + x * squareSize);
099:  		newLabel->setPlainText(QString(numericalLetter));
100:  		graphicsScene.addItem(newLabel);
101:  		positionLabelsRight[x] = newLabel;
102:  	}
103:  }
104:
105:  void BoardDisplay::mousePressEvent(QMouseEvent * event)
106:  {
107:  	if (!isPlayersTurn) return;
108:  	for (int x = 0; x < chessPieces.size(); x++)
109:  	{
110:  		if (chessPieces[x]->isUnderMouse() && chessPieces[x]->getColour() == chessBoard.nextColour)
```

```
111:                    {
112:                            isPieceBeingDragged = true;
113:                            movingPiece = chessPieces[x];
114:                            originalOffset = movingPiece->offset();
115:                            movingPiece->setZValue(1);
116:                    }
117:            }
118:    }
119:
120:    void BoardDisplay::mouseMoveEvent(QMouseEvent * event)
121:    {
122:            if (isPieceBeingDragged && movingPiece != nullptr)
123:            {
124:                    movingPiece->setOffset(positionLabelSize + event->x() - pieceSize, positionLabelSize + event->y() - pieceSize);
125:            }
126:    }
127:
128:    void BoardDisplay::mouseReleaseEvent(QMouseEvent * event)
129:    {
130:            Move move;
131:            if (isPieceBeingDragged && movingPiece != nullptr)
132:            {
133:                    int newPos = -1;
134:                    for (int x = 0; x < 8; x++)
135:                    {
136:                            for (int y = 0; y < 8; y++)
137:                            {
138:                                    if (boardSquares[x][y].isUnderMouse())
139:                                    {
140:                                            newPos = y * 8 + x;
141:                                            if (isBoardFlipped)
142:                                                    newPos = 63 - newPos;
143:                                    }
144:                            }
145:                    }
146:
147:                    uint64_t moveToBitboard = (uint64_t)1 << newPos;
148:                    if (newPos != -1)
149:                    {
150:                            if ((chessBoard.allPieces & moveToBitboard) == 0 && std::abs(newPos - movingPiece->getPiecePosition()) == 16 && movingPiece->getPieceType() == pawn)
151:                                    move = Move(movingPiece->getPiecePosition(), newPos, pawnDoubleMove, pawn, &chessBoard);
152:                            else if((chessBoard.allPieces & moveToBitboard) == 0 && newPos - movingPiece->getPiecePosition() == 2 && movingPiece->getPieceType() == king)
153:                                    move = Move(movingPiece->getPiecePosition(), newPos, kingSideCastling, king, &chessBoard);
154:                            else if ((chessBoard.allPieces & moveToBitboard) == 0 && newPos - movingPiece->getPiecePosition() == -2 && movingPiece->getPieceType() == king)
155:                                    move = Move(movingPiece->getPiecePosition(), newPos, queenSideCastling, king, &chessBoard);
156:
157:                            //Pawn Promotion Moves
158:                            else if (movingPiece->getPieceType() == pawn && (moveToBitboard & (rank1 | rank8)) > 0)
159:                            {
160:                                    MoveType type = quietMove;
161:                                    while (type == quietMove)
162:                                    {
163:                                            PawnPromotionDialog pawnDialog;
164:                                            pawnDialog.setupDialogBox(&piecePixmaps, chessBoard.nextColour);
165:                                            pawnDialog.exec();
166:                                            type = pawnDialog.getCurrentType();
167:                                    }
168:                                    move = Move(movingPiece->getPiecePosition(), newPos, type, movingPiece->getPieceType(), &chessBoard);
169:                            }
170:
171:                            else if ((chessBoard.allPieces & moveToBitboard) == 0)
172:                                    move = Move(movingPiece->getPiecePosition(), newPos, quietMove, movingPiece->getPieceType(), &chessBoard);
173:                            else if ((chessBoard.allPieces & moveToBitboard) != 0)
174:                                    move = Move(movingPiece->getPiecePosition(), newPos, capture, movingPiece->getPieceType(), &chessBoard);
175:                    }
176:            }
177:
178:            bool isValidMove = false;
179:
180:            for (int x = 0; x < moveListSize; x++)
181:            {
182:                    if (moveList[x] == move)
183:                            isValidMove = true;
184:            }
185:
186:
187:            //If invalid move, move back to original position
188:            if (isPieceBeingDragged && movingPiece != nullptr && !isValidMove)
189:            {
190:                    movingPiece->setOffset(originalOffset);
191:                    movingPiece->setZValue(0);
192:            }
193:            else if(isPieceBeingDragged && movingPiece != nullptr)
194:            {
195:                    applyMove(move);
196:            }
197:
198:            isPieceBeingDragged = false;
199:            movingPiece = nullptr;
200:
201:    }
202:
203:    void BoardDisplay::setBoard(Board newBoard)
204:    {
205:            chessBoard = newBoard;
206:            updateChessPieces();
207:    }
208:
209:    Board BoardDisplay::getBoard()
210:    {
211:            return chessBoard;
212:    }
213:
214:    void BoardDisplay::applyMove(Move newMove)
215:    {
216:            playedMoves.push_back(newMove);
217:            newMove.applyMove(&chessBoard);
218:            updateChessPieces();
219:            emit newTurn();
220:    }
221:
222:    void BoardDisplay::flipBoard()
223:    {
224:            isBoardFlipped = !isBoardFlipped;
```

```
225:        updateChessPieces();
226:    }
227:
228:    void BoardDisplay::newGame()
229:    {
230:        chessBoard.defaults();
231:        updateChessPieces();
232:    }
233:
234:    void BoardDisplay::updateChessPieces()
235:    {
236:        //Deletes pieces
237:        for (int x = 0; x < chessPieces.size(); x++)
238:        {
239:            graphicsScene.removeItem(chessPieces[x]);
240:            delete chessPieces[x];
241:        }
242:        chessPieces.clear();
243:
244:        for (int counter = 0; counter < 64; counter++)
245:        {
246:            uint64_t currentPosBitboard;
247:            if (!isBoardFlipped) currentPosBitboard = (uint64_t)1 << counter;
248:            else currentPosBitboard = (uint64_t)1 << (63-counter);
249:
250:            pieceType currentType = chessBoard.getPieceTypeInSquare(currentPosBitboard);
251:
252:            if (currentType != blank)
253:            {
254:                colours currentColour;
255:                if (chessBoard.getPieceBitboard(white, currentType) & currentPosBitboard)
256:                    currentColour = white;
257:                else
258:                    currentColour = black;
259:
260:                ChessPiece* currentChessPiece = new ChessPiece();
261:
262:                currentChessPiece->setPixmap(piecePixmaps[currentType][currentColour]);
263:
264:                qreal widthOffset = positionLabelSize + squareSize * (counter % 8) + squareSize / 2 - currentChessPiece->boundingRect().width() / 2;
265:                qreal heightOffset = heightOffset = positionLabelSize + squareSize * (7 - (counter / 8)) + squareSize / 2 - currentChessPiece->boundingRect().height() / 2;
266:
267:                currentChessPiece->setOffset(widthOffset, heightOffset);
268:
269:                if (!isBoardFlipped)
270:                    currentChessPiece->setPiecePosition(counter);
271:                else currentChessPiece->setPiecePosition(63 - counter);
272:
273:                currentChessPiece->setColour(currentColour);
274:                currentChessPiece->setPieceType(currentType);
275:
276:                graphicsScene.addItem(currentChessPiece);
277:                chessPieces.push_back(currentChessPiece);
278:            }
279:        }
280:
281:        moveListSize = searchForMoves(&chessBoard, &moveList);
282:    }
283:
284:    void BoardDisplay::loadChessPiecePixmaps()
285:    {
286:        piecePixmaps[pawn][white].load("whitePawn.png");
287:        piecePixmaps[knight][white].load("whiteKnight.png");
288:        piecePixmaps[bishop][white].load("whiteBishop.png");
289:        piecePixmaps[rook][white].load("whiteRook.png");
290:        piecePixmaps[queen][white].load("whiteQueen.png");
291:        piecePixmaps[king][white].load("whiteKing.png");
292:
293:        piecePixmaps[pawn][black].load("blackPawn.png");
294:        piecePixmaps[knight][black].load("blackKnight.png");
295:        piecePixmaps[bishop][black].load("blackBishop.png");
296:        piecePixmaps[rook][black].load("blackRook.png");
297:        piecePixmaps[queen][black].load("blackQueen.png");
298:        piecePixmaps[king][black].load("blackKing.png");
299:
300:        //Scales all pieces to pieceSize (assuming they have the same height and width
301:        for (int x = 0; x < 6; x++)
302:        {
303:            for (int y = 0; y < 2; y++)
304:            {
305:                piecePixmaps[x][y] = piecePixmaps[x][y].scaledToHeight(pieceSize);
306:            }
307:        }
308:    }
309:
```

ChessUI\BoardDisplay.h

```
001:  #pragma once
002:  #include <QGraphicsScene>
003:  #include <QGraphicsView>
004:  #include <QGraphicsPixmapItem>
005:  #include <QPixmap>
006:  #include <QMouseEvent>
007:  #include <QFileDialog>
008:
009:  #include <algorithm>
010:  #include <vector>
011:  #include <array>
012:  #include <stdint.h>
013:  #include <fstream>
014:  #include <sstream>
015:
016:  #include "magicBitboards.h"
017:  #include "ChessPiece.h"
018:  #include "board.h"
019:  #include "piece.h"
020:  #include "move.h"
021:  #include "moveGeneration.h"
022:  #include "PawnPromotionDialog.h"
023:  #include "OptionsMenuDialog.h"
024:
025:  class BoardDisplay :
026:      public QGraphicsView
027:  {
028:      Q_OBJECT
029:
030:  public:
031:      BoardDisplay(QWidget *parent);
032:      void setBoard(Board newBoard);
033:      Board getBoard();
034:      void setIsPlayersTurn(bool newIsPlayersTurn) { isPlayersTurn = newIsPlayersTurn; };
035:      void applyMove(Move newMove);
036:      void flipBoard();
037:      void newGame();
038:
039:  signals:
040:      void newTurn();
041:
042:  private:
043:      QPixmap blackSquarePixmap;
044:      QPixmap whiteSquarePixmap;
045:      qreal positionLabelSize;
046:      qreal squareSize;
047:      QGraphicsPixmapItem boardSquares[8][8];
048:      QGraphicsScene graphicsScene;
049:
050:      qreal pieceSize;
051:      std::vector<ChessPiece*> chessPieces;
052:      void updateChessPieces();
053:
054:      std::array<std::array<QPixmap,2>, 6> piecePixmaps;
055:      void loadChessPiecePixmaps();
056:
057:      bool isPieceBeingDragged;
058:      ChessPiece* movingPiece;
059:      void addPositionLabels();
060:      void mousePressEvent(QMouseEvent * event);
061:      void mouseMoveEvent(QMouseEvent * event);
062:      void mouseReleaseEvent(QMouseEvent * event);
063:      QPointF originalOffset;
064:
065:      Board chessBoard;
066:      std::vector<Move> playedMoves;
067:
068:      int moveListSize;
069:      std::array<Move, 150> moveList;
070:
071:      std::array<QGraphicsTextItem*, 8> positionLabelsTop;
072:      std::array<QGraphicsTextItem*, 8> positionLabelsBottom;
073:      std::array<QGraphicsTextItem*, 8> positionLabelsLeft;
074:      std::array<QGraphicsTextItem*, 8> positionLabelsRight;
075:
076:      bool isPlayersTurn;
077:      bool isBoardFlipped;
078:
079:
080:  protected:
081:      virtual void wheelEvent(QWheelEvent * event) {}; //Overrides the scroll event to disable zooming the graphicsView.
082:  };
083:
084:
```

ChessUI\ChessPiece.cpp

```
001:    #include "ChessPiece.h"
002:
003:
004:
005:    ChessPiece::ChessPiece(QGraphicsItem * parent) : QGraphicsPixmapItem(parent)
006:    {
007:    }
008:
009:    ChessPiece::ChessPiece()
010:    {
011:    }
012:
013:
014:    ChessPiece::~ChessPiece()
015:    {
016:    }
017:
```

ChessUI\ChessPiece.h

```
001:    #pragma once
002:    #include <QGraphicsPixmapItem>
003:
004:    #include "piece.h"
005:
006:    class ChessPiece :
007:            public QGraphicsPixmapItem
008:    {
009:    public:
010:            ChessPiece(QGraphicsItem * parent);
011:            ChessPiece();
012:            ~ChessPiece();
013:
014:            inline void setPiecePosition(int newPos) { pos = newPos; };
015:            inline int getPiecePosition() { return pos; };
016:            inline void setPieceType(pieceType newPieceType) { piece = newPieceType; };
017:            inline pieceType getPieceType() { return piece; };
018:            inline void setColour(colours newColour) { colour = newColour; };
019:            inline colours getColour() { return colour; };
020:
021:    private:
022:            int pos;
023:            pieceType piece;
024:            colours colour;
025:    };
026:
027:
```

ChessUI\EngineOutputDialog.cpp

```
001:  #include "EngineOutputDialog.h"
002:
003:  EngineOutputDialog::EngineOutputDialog()
004:  {
005:          setModal(false);
006:  }
007:
008:
009:  EngineOutputDialog::~EngineOutputDialog()
010:  {
011:          delete okButton;
012:          delete engineDisplay;
013:          delete verticalLayout;
014:          delete lowerBar;
015:          delete clearButton;
016:  }
017:
018:  void EngineOutputDialog::setupDialogBox()
019:  {
020:          verticalLayout = new QVBoxLayout(this);
021:          setLayout(verticalLayout);
022:
023:          engineDisplay = new QTextEdit(this);
024:          engineDisplay->setReadOnly(true);
025:          engineDisplay->setMinimumSize(300, 400);
026:          verticalLayout->addWidget(engineDisplay);
027:
028:          lowerBar = new QHBoxLayout(this);
029:          verticalLayout->addLayout(lowerBar);
030:
031:          clearButton = new QPushButton("clear", this);
032:          lowerBar->addWidget(clearButton);
033:
034:          okButton = new QPushButton("ok", this);
035:          lowerBar->addWidget(okButton);
036:
037:          connect(okButton, SIGNAL(clicked()), this, SLOT(okButtonPressedCallback()));
038:          connect(clearButton, SIGNAL(clicked()), engineDisplay, SLOT(clear()));
039:  }
040:
041:  void EngineOutputDialog::okButtonPressedCallback()
042:  {
043:          emit accept();
044:  }
045:
046:  void EngineOutputDialog::addNewLine(QString line)
047:  {
048:          engineDisplay->append(line);
049:  }
050:
```

ChessUI\EngineOutputDialog.h

```
001:  #pragma once
002:  #include "qdialog.h"
003:  #include <QTextBrowser>
004:  #include <QPushButton>
005:  #include <qlayout.h>
006:
007:  class EngineOutputDialog :
008:  	public QDialog
009:  {
010:  	Q_OBJECT
011:  public:
012:  	EngineOutputDialog();
013:  	~EngineOutputDialog();
014:  	void setupDialogBox();
015:  	void addNewLine(QString line);
016:
017:  private:
018:  	QPushButton* okButton;
019:  	QTextEdit* engineDisplay;
020:  	QVBoxLayout* verticalLayout;
021:  	QHBoxLayout* lowerBar;
022:  	QPushButton* clearButton;
023:
024:  private slots:
025:  	void okButtonPressedCallback();
026:  };
027:
028:
```

ChessUI\GameManager.cpp

```
001:  #include "GameManager.h"
002:
003:
004:
005:  GameManager::GameManager(QWidget *parent) : QWidget(parent)
006:  {
007:      boardDisplay = new BoardDisplay(this);
008:
009:      connect(boardDisplay, SIGNAL(newTurn()), this, SLOT(newTurn()));
010:      connect(&aiManager, SIGNAL(newMove(Move)), this, SLOT(aiNewMove(Move)));
011:  }
012:
013:  GameManager::~GameManager()
014:  {
015:      delete boardDisplay;
016:  }
017:
018:  void GameManager::loadFromFile()
019:  {
020:      QString filename = QFileDialog::getOpenFileName(this,
021:          tr("Open file"), "", tr("Chess Files (*.pgn *.fen)"));
022:
023:      if (filename.count() > 0)
024:      {
025:          std::ifstream file;
026:          file.open(filename.toStdString());
027:
028:          std::stringstream strStream;
029:          strStream << file.rdbuf();
030:
031:          std::string fileContents = strStream.str();
032:
033:          //PGN file type
034:          if (fileContents[0] == '[')
035:          {
036:              //TODO
037:          }
038:          else
039:          {
040:              Board newBoard;
041:              newBoard.loadFromFen(fileContents);
042:              boardDisplay->setBoard(newBoard);
043:          }
044:      }
045:
046:  }
047:
048:  void GameManager::saveToFile()
049:  {
050:      QString filename = QFileDialog::getSaveFileName(this,
051:          tr("Open file"), "", tr("Chess Files (*.pgn *.fen)"));
052:
053:      if (filename.count() < 0)
054:      {
055:          std::string fenData = boardDisplay->getBoard().exportAsFen();
056:          std::ofstream file(filename.toStdString());
057:          file << fenData;
058:          file.close();
059:      }
060:  }
061:
062:  void GameManager::displayOptionsMenu()
063:  {
064:      OptionsMenuDialog dialog;
065:      dialog.setupDialogBox(&currentOptions);
066:      dialog.exec();
067:      currentOptions = dialog.getOptions();
068:      newTurn();
069:  }
070:
071:  void GameManager::newTurn()
072:  {
073:      if (!currentOptions.isAi || (currentOptions.isAi & currentOptions.aiColour != boardDisplay->getBoard().nextColour))
074:      {
075:          boardDisplay->setIsPlayersTurn(true);
076:      }
077:      else
078:      {
079:          boardDisplay->setIsPlayersTurn(false);
080:      }
081:
082:      if (currentOptions.isAi && currentOptions.aiColour == boardDisplay->getBoard().nextColour)
083:      {
084:          aiManager.startAI();
085:          aiManager.findMove(boardDisplay->getBoard());
086:      }
087:  }
088:
089:  void GameManager::displayEngineOutputMenu()
090:  {
091:      aiManager.showEngineOutputDialog();
092:  }
093:
094:  void GameManager::flipBoard()
095:  {
096:      boardDisplay->flipBoard();
097:  }
098:
099:  void GameManager::newGame()
100:  {
101:      boardDisplay->newGame();
102:  }
103:
104:  void GameManager::aiNewMove(Move newMove)
105:  {
106:      boardDisplay->applyMove(newMove);
107:  }
108:
```

ChessUI\GameManager.h

```
001:  #pragma once
002:  #include "qwidget.h"
003:
004:  #include "BoardDisplay.h"
005:  #include "board.h"
006:  #include "OptionsMenuDialog.h"
007:  #include "AIManager.h"
008:
009:
010:  class GameManager :
011:          public QWidget
012:  {
013:          Q_OBJECT
014:
015:  public:
016:          GameManager(QWidget *parent);
017:          ~GameManager();
018:
019:  public slots:
020:          void loadFromFile();
021:          void saveToFile();
022:          void displayOptionsMenu();
023:          void displayEngineOutputMenu();
024:          void flipBoard();
025:          void newGame();
026:
027:  private slots:
028:          void newTurn();
029:          void aiNewMove(Move newMove);
030:
031:  private:
032:          BoardDisplay* boardDisplay;
033:          Options currentOptions;
034:          AIManager aiManager;
035:  };
036:
037:
```

ChessUI\OptionsMenuDialog.cpp

```
001:  #include "OptionsMenuDialog.h"
002:
003:
004:
005:  OptionsMenuDialog::OptionsMenuDialog()
006:  {
007:  }
008:
009:
010:  OptionsMenuDialog::~OptionsMenuDialog()
011:  {
012:          delete isAiCheckbox;
013:          delete aiColourHorizontalLayout;
014:          delete aiColourLabel;
015:          delete aiColourComboBox;
016:          delete verticalLayout;
017:          delete okButton;
018:  }
019:
020:  void OptionsMenuDialog::setupDialogBox(Options * currentOptions)
021:  {
022:          verticalLayout = new QVBoxLayout(this);
023:
024:          isAiCheckbox = new QCheckBox("Play against AI?", this);
025:          isAiCheckbox->setChecked(currentOptions->isAi);
026:          isAiCheckbox->setLayoutDirection(Qt::RightToLeft);
027:          verticalLayout->addWidget(isAiCheckbox);
028:
029:          aiColourHorizontalLayout = new QHBoxLayout(this);
030:          verticalLayout->addLayout(aiColourHorizontalLayout);
031:
032:          aiColourLabel = new QLabel("Colour of AI?", this);
033:          aiColourHorizontalLayout->addWidget(aiColourLabel);
034:
035:          aiColourComboBox = new QComboBox(this);
036:          aiColourComboBox->addItem("White");
037:          aiColourComboBox->addItem("Black");
038:          if (currentOptions->aiColour == white) aiColourComboBox->setCurrentIndex(0);
039:          else aiColourComboBox->setCurrentIndex(1);
040:          aiColourHorizontalLayout->addWidget(aiColourComboBox);
041:
042:          okButton = new QPushButton("Ok", this);
043:          verticalLayout->addWidget(okButton);
044:
045:          connect(okButton, SIGNAL(clicked()), this, SLOT(doneButtonCallback()));
046:  }
047:
048:  Options OptionsMenuDialog::getOptions()
049:  {
050:          Options newOptions;
051:          newOptions.isAi = isAiCheckbox->isChecked();
052:          if (aiColourComboBox->currentText() == "White") newOptions.aiColour = white;
053:          else newOptions.aiColour = black;
054:
055:          return newOptions;
056:  }
057:
058:  void OptionsMenuDialog::doneButtonCallback()
059:  {
060:          emit accept();
061:  }
062:
063:  Options::Options()
064:  {
065:          isAi = false;
066:          aiColour = white;
067:  }
068:
```

ChessUI\OptionsMenuDialog.h

```
001:  #pragma once
002:  #include "qdialog.h"
003:  #include <qlayout.h>
004:  #include <QCheckBox>
005:  #include <QComboBox>
006:  #include <QLabel>
007:  #include <qpushbutton.h>
008:
009:  #include "piece.h"
010:
011:  struct Options
012:  {
013:  	Options();
014:
015:  	bool isAi;
016:  	colours aiColour;
017:  };
018:
019:  class OptionsMenuDialog :
020:  	public QDialog
021:  {
022:  	Q_OBJECT
023:
024:  public:
025:  	OptionsMenuDialog();
026:  	~OptionsMenuDialog();
027:
028:  	void setupDialogBox(Options* currentOptions);
029:  	Options getOptions();
030:
031:  private:
032:  	QCheckBox* isAiCheckbox;
033:  	QHBoxLayout* aiColourHorizontalLayout;
034:  	QLabel* aiColourLabel;
035:  	QComboBox* aiColourComboBox;
036:  	QVBoxLayout* verticalLayout;
037:  	QPushButton* okButton;
038:
039:  private slots:
040:  	void doneButtonCallback();
041:
042:  };
043:
044:
```

ChessUI\PawnPromotionDialog.cpp

```
001:  #include "PawnPromotionDialog.h"
002:
003:
004:
005:  PawnPromotionDialog::PawnPromotionDialog(QWidget * parent, Qt::WindowFlags f) : QDialog(parent,f)
006:  {
007:          currentType = quietMove;
008:  }
009:
010:  PawnPromotionDialog::~PawnPromotionDialog()
011:  {
012:          delete rookPromotionButton;
013:          delete bishopPromotionButton;
014:          delete queenPromotionButton;
015:          delete knightPromotionButton;
016:          delete horizontalLayout;
017:  }
018:
019:  void PawnPromotionDialog::setupDialogBox(std::array<std::array<QPixmap, 2>, 6>* piecePixmaps, colours currentColour)
020:  {
021:          horizontalLayout = new QHBoxLayout(this);
022:
023:          if (currentColour == black)
024:          {
025:                  rookPromotionButton = new QPushButton(QIcon((*piecePixmaps)[rook][black]), "Rook", this);
026:                  bishopPromotionButton = new QPushButton(QIcon((*piecePixmaps)[bishop][black]), "Bishop", this);
027:                  queenPromotionButton = new QPushButton(QIcon((*piecePixmaps)[queen][black]), "Queen", this);
028:                  knightPromotionButton = new QPushButton(QIcon((*piecePixmaps)[knight][black]), "Knight", this);
029:          }
030:          else
031:          {
032:                  rookPromotionButton = new QPushButton(QIcon((*piecePixmaps)[rook][white]), "Rook", this);
033:                  bishopPromotionButton = new QPushButton(QIcon((*piecePixmaps)[bishop][white]), "Bishop", this);
034:                  queenPromotionButton = new QPushButton(QIcon((*piecePixmaps)[queen][white]), "Queen", this);
035:                  knightPromotionButton = new QPushButton(QIcon((*piecePixmaps)[knight][white]), "Knight", this);
036:          }
037:
038:          connect(rookPromotionButton, SIGNAL(clicked()), this, SLOT(rookPromotionButtonCallback()));
039:          connect(bishopPromotionButton, SIGNAL(clicked()), this, SLOT(bishopPromotionButtonCallback()));
040:          connect(queenPromotionButton, SIGNAL(clicked()), this, SLOT(queenPromotionButtonCallback()));
041:          connect(knightPromotionButton, SIGNAL(clicked()), this, SLOT(knightPromotionButtonCallback()));
042:
043:          horizontalLayout->addWidget(queenPromotionButton);
044:          horizontalLayout->addWidget(rookPromotionButton);
045:          horizontalLayout->addWidget(bishopPromotionButton);
046:          horizontalLayout->addWidget(knightPromotionButton);
047:  }
048:
049:  void PawnPromotionDialog::rookPromotionButtonCallback()
050:  {
051:          currentType = rookPromotion;
052:          emit accept();
053:  }
054:  void PawnPromotionDialog::queenPromotionButtonCallback()
055:  {
056:          currentType = queenPromotion;
057:          emit accept();
058:  }
059:  void PawnPromotionDialog::knightPromotionButtonCallback()
060:  {
061:          currentType = knightPromotion;
062:          emit accept();
063:  }
064:  void PawnPromotionDialog::bishopPromotionButtonCallback()
065:  {
066:          currentType = bishopPromotion;
067:          emit accept();
068:  }
069:
```

ChessUI\PawnPromotionDialog.h

```
001:  #pragma once
002:  #include <qdialog.h>
003:  #include <qpushbutton.h>
004:  #include <qlayout.h>
005:
006:  #include <array>
007:  #include "piece.h"
008:  #include "move.h"
009:
010:  class PawnPromotionDialog :
011:          public QDialog
012:  {
013:          Q_OBJECT
014:
015:  public:
016:          PawnPromotionDialog(QWidget * parent = 0, Qt::WindowFlags f = 0);
017:          ~PawnPromotionDialog();
018:
019:          void setupDialogBox(std::array<std::array<QPixmap, 2>, 6>* piecePixmaps, colours currentColour);
020:
021:          inline MoveType getCurrentType() { return currentType; };
022:
023:  private:
024:          QPushButton* rookPromotionButton;
025:          QPushButton* bishopPromotionButton;
026:          QPushButton* queenPromotionButton;
027:          QPushButton* knightPromotionButton;
028:          QHBoxLayout* horizontalLayout;
029:
030:          MoveType currentType;
031:
032:  private slots:
033:          void rookPromotionButtonCallback();
034:          void bishopPromotionButtonCallback();
035:          void queenPromotionButtonCallback();
036:          void knightPromotionButtonCallback();
037:
038:  };
039:
040:
```

ChessUI\chessui.cpp

```
001:    #include "chessui.h"
002:
003:    ChessUI::ChessUI(QWidget *parent)
004:            : QMainWindow(parent)
005:    {
006:            ui.setupUi(this);
007:    }
008:
```

ChessUI\chessui.h

```
001:    #pragma once
002:
003:    #include <QtWidgets/QMainWindow>
004:    #include "ui_chessui.h"
005:
006:    class ChessUI : public QMainWindow
007:    {
008:            Q_OBJECT
009:
010:    public:
011:            ChessUI(QWidget *parent = Q_NULLPTR);
012:
013:    private:
014:            Ui::ChessUIClass ui;
015:    };
016:
```

ChessUI\main.cpp

```
001:    #include "chessui.h"
002:    #include <QtWidgets/QApplication>
003:
004:    int main(int argc, char *argv[])
005:    {
006:            QApplication a(argc, argv);
007:            ChessUI w;
008:            w.show();
009:            return a.exec();
010:    }
011:
```