Szachy

Szymon Makulec Jan Moskal

10 lutego 2024

Cel i efekty pracy

Opis

Projekt zakładał stworzenie gry w szachy dla dwóch graczy. Kod oparty jest na bibliotece wxWidgets, a interfejs użytkownika wykorzystuje siatkę wxBitmapButton z obrazkami figur połączonymi z kolorem tła, do reprezentacji szachownicy i figur na niej położonych. Projekt udało nam się doprowadzić do stanu gdzie są przestrzegane zasady gry w szachy takie jak: ruszanie się na zmianę graczy, figury mają tylko takie możliwości ruchy jakby była to normalna gra, mamy dostęp do roszady, pierwszego ruchu piona o 2, promocji piona. Przestrzegane są również zasady związane z szachowaniem tj. nie możemy się ruszyć w taki sposób by po ruchu król był dalej pod biciem.

Tablice obiektów

Użycie obrazków (images[2][13])

```
// Black Pieces Images
images[0][0] = wxBitmap(wxImage("images/D.jpg"));
images[0][1] = wxBitmap(wxImage(_T("images/Pieces/bpB.png")));
images[0][2] = wxBitmap(wxImage(_T("images/Pieces/bpD.png")));
images[0][3] = wxBitmap(wxImage(_T("images/Pieces/brB.png")));
images[0][4] = wxBitmap(wxImage(_T("images/Pieces/brD.png")));
images[0][5] = wxBitmap(wxImage(_T("images/Pieces/bnB.png")));
images[0][6] = wxBitmap(wxImage(_T("images/Pieces/bnD.png")));
images[0][7] = wxBitmap(wxImage(_T("images/Pieces/bbB.png")));
images[0][8] = wxBitmap(wxImage(_T("images/Pieces/bbD.png")));
images[0][9] = wxBitmap(wxImage(_T("images/Pieces/bqB.png")));
images[0][10] = wxBitmap(wxImage(_T("images/Pieces/bqD.png")));
images[0][11] = wxBitmap(wxImage(_T("images/Pieces/bkB.png")));
images[0][12] = wxBitmap(wxImage(_T("images/Pieces/bkD.png")));

// White Pieces Images
images[1][0] = wxBitmap(wxImage("images/B.jpg"));
images[1][1] = wxBitmap(wxImage(_T("images/Pieces/wpB.png")));
images[1][2] = wxBitmap(wxImage(_T("images/Pieces/wpD.png")));
images[1][3] = wxBitmap(wxImage(_T("images/Pieces/wrB.png")));
images[1][4] = wxBitmap(wxImage(_T("images/Pieces/wrD.png")));
images[1][5] = wxBitmap(wxImage(_T("images/Pieces/wnB.png")));
images[1][6] = wxBitmap(wxImage(_T("images/Pieces/wnD.png")));
images[1][7] = wxBitmap(wxImage(_T("images/Pieces/wbB.png")));
images[1][8] = wxBitmap(wxImage(_T("images/Pieces/wbD.png")));
images[1][9] = wxBitmap(wxImage(_T("images/Pieces/wqB.png")));
images[1][10] = wxBitmap(wxImage(_T("images/Pieces/wqD.png")));
images[1][11] = wxBitmap(wxImage(_T("images/Pieces/wkB.png")));
images[1][12] = wxBitmap(wxImage(_T("images/Pieces/wkD.png")));
```

Do wyświetlania figur na przyciskach wxBitmapButton, używamy zestawu dwudziestu sześciu obrazków, reprezentujących figury szachowe. Dwa puste pola (czarne i białe) oraz sześć różnych typów figur (pion, wieża, skoczek, goniec,

hetman, król) w dwóch kolorach (biały i czarny) na dwóch tłach (jasnym, ciemnym). images, jest to tablica, w której przechowywane są obrazy figur szachowych używane do wyświetlania na przyciskach wxBitmapButton na szachownicy. Umożliwia łatwe zarządzanie obrazkami figur szachowych w grze.

Użycie przycisków wxBitmapButton (board[8][8])

```
board[0][0] = BitmapButton1;
board[0][1] = new wxBitmapButton(this, wxNewId(), images[0][6],
    wxDefaultPosition, wxDefaultSize, wxBU_AUTODRAW,
    wxDefaultValidator);
board[0][2] = new wxBitmapButton(this, wxNewId(), images[0][7],
    wxDefaultPosition, wxDefaultSize, wxBU_AUTODRAW,
    wxDefaultValidator);
board[0][3] = new wxBitmapButton(this, wxNewId(), images[0][10],
    wxDefaultPosition, wxDefaultSize, wxBU_AUTODRAW,
    wxDefaultValidator);
board[0][4] = new wxBitmapButton(this, wxNewId(), images[0][11],
    wxDefaultPosition, wxDefaultSize, wxBU_AUTODRAW,
    wxDefaultValidator);
board[0][5] = new wxBitmapButton(this, wxNewId(), images[0][8],
    wxDefaultPosition, wxDefaultSize, wxBU_AUTODRAW,
    wxDefaultValidator);
board[0][6] = new wxBitmapButton(this, wxNewId(), images[0][5],
    wxDefaultPosition, wxDefaultSize, wxBU_AUTODRAW,
    wxDefaultValidator);
board[0][7] = new wxBitmapButton(this, wxNewId(), images[0][4],
    wxDefaultPosition, wxDefaultSize, wxBU_AUTODRAW,
    wxDefaultValidator);

for(int j = 0; j < 8; j++){
  if(j%2 == 0){
    board[1][j] = new wxBitmapButton(this, wxNewId(),images[0][2] ,
         wxDefaultPosition, wxDefaultSize, wxBU_AUTODRAW,
        wxDefaultValidator);
  }else{
    board[1][j] = new wxBitmapButton(this, wxNewId(),images[0][1] ,
         wxDefaultPosition, wxDefaultSize, wxBU_AUTODRAW,
        wxDefaultValidator);

  }
}

for(int i = 2; i < 6; i++){
  for(int j = 0; j < 8; j++){
    if((i+j)%2 == 0){
      board[i][j] = new wxBitmapButton(this, wxNewId(), images
          [1][0], wxDefaultPosition, wxDefaultSize, wxBU_AUTODRAW,
          wxDefaultValidator);
    }else{
      board[i][j] = new wxBitmapButton(this, wxNewId(), images
          [0][0], wxDefaultPosition, wxDefaultSize, wxBU_AUTODRAW,
          wxDefaultValidator);
    }
  }
}
```

```
28
29  for(int j = 0; j < 8; j++){
30    if((j)%2 != 0){
31      board[6][j] = new wxBitmapButton(this, wxNewId(), images[1][2],
            wxDefaultPosition, wxDefaultSize, wxBU_AUTODRAW,
          wxDefaultValidator);
32    }else{
33      board[6][j] = new wxBitmapButton(this, wxNewId(), images[1][1],
            wxDefaultPosition, wxDefaultSize, wxBU_AUTODRAW,
          wxDefaultValidator);
34
35    }
36  }
37
38  board[7][0] = new wxBitmapButton(this, wxNewId(), images[1][4],
      wxDefaultPosition, wxDefaultSize, wxBU_AUTODRAW,
      wxDefaultValidator);
39  board[7][1] = new wxBitmapButton(this, wxNewId(), images[1][5],
      wxDefaultPosition, wxDefaultSize, wxBU_AUTODRAW,
      wxDefaultValidator);
40  board[7][2] = new wxBitmapButton(this, wxNewId(), images[1][8],
      wxDefaultPosition, wxDefaultSize, wxBU_AUTODRAW,
      wxDefaultValidator);
41  board[7][3] = new wxBitmapButton(this, wxNewId(), images[1][9],
      wxDefaultPosition, wxDefaultSize, wxBU_AUTODRAW,
      wxDefaultValidator);
42  board[7][4] = new wxBitmapButton(this, wxNewId(), images[1][12],
      wxDefaultPosition, wxDefaultSize, wxBU_AUTODRAW,
      wxDefaultValidator);
43  board[7][5] = new wxBitmapButton(this, wxNewId(), images[1][7],
      wxDefaultPosition, wxDefaultSize, wxBU_AUTODRAW,
      wxDefaultValidator);
44  board[7][6] = new wxBitmapButton(this, wxNewId(), images[1][6],
      wxDefaultPosition, wxDefaultSize, wxBU_AUTODRAW,
      wxDefaultValidator);
45  board[7][7] = new wxBitmapButton(this, wxNewId(), images[1][3],
      wxDefaultPosition, wxDefaultSize, wxBU_AUTODRAW,
      wxDefaultValidator);
46
47  for(int i = 0;i < 8; i++){
48    for(int j = 0; j < 8; j++){
49      if(i + j == 0){
50        j++;
51      }
52      GridSizer1->Add(board[i][j], 1, wxALIGN_CENTER_HORIZONTAL|
            wxALIGN_CENTER_VERTICAL, 5);
53      Connect(board[i][j]->GetId(),wxEVT_COMMAND_BUTTON_CLICKED,(
            wxObjectEventFunction)&Chess2DDialog::OnBitmapButton1Click)
            ;
54
55
56    }
57  }
```

Każdy element board[i][j] (8x8) reprezentuje jedno pole z wyświetloną figurą danego koloru i tłem na szachownicy, zawiera przycisk do interakcji z użytkow-

nikiem. Każdy przycisk jest obsługiwany przez tą samą funkcję OnBitmapButton1Click.

Użycie figur (pieces[2][7])

```
pieces[0][0] = new Pawn(false);
pieces[0][1] = new Rook(false);
pieces[0][2] = new Knight(false);
pieces[0][3] = new Bishop(false);
pieces[0][4] = new Queen(false);
pieces[0][5] = new King(false);
pieces[0][6] = new Rook(false);
pieces[1][0] = new Pawn(true);
pieces[1][1] = new Rook(true);
pieces[1][2] = new Knight(true);
pieces[1][3] = new Bishop(true);
pieces[1][4] = new Queen(true);
pieces[1][5] = new King(true);
pieces[1][6] = new Rook(true);
```

pieces[i][j] jest to dwuwymiarowa tablica wskaźników do obiektów klasy Piece. Jest to tablica, w której przechowywane są obiekty reprezentujące różne typy figur szachowych w dwóch kolorach: białym i czarnym. Każdy obiekt klasy Piece przechowuje informacje o tym czy figura już się ruszała. Dlatego tworzymy dwie wieże, żeby każdej z osobna można było sprawdzać czy się ruszała. W przypadku pionów uznaliśmy tworze oddzielnych Pawnów za zbyteczne, ponieważ każdy może poruszać się tylko w 1 kierunku.

Przechowywanie przycisków i figur w jednym obiekcie (squares[8][8])

```
for(int i = 0; i < 8; i++){
  for(int j = 0; j < 8; j++){
    if((i+j) % 2 == 0){
      squares[i][j] = new Square(i, j , true, board[i][j]);
    }else{
      squares[i][j] = new Square(i, j , false, board[i][j]);
    }
  }
}

for(int i = 0; i < 8; i++){
  squares[1][i]->setPiece(pieces[0][0]);
  squares[6][i]->setPiece(pieces[1][0]);
}

// Assigning other Pieces to Squares
squares[0][0]->setPiece(pieces[0][1]);
squares[0][1]->setPiece(pieces[0][2]);
squares[0][2]->setPiece(pieces[0][3]);
squares[0][3]->setPiece(pieces[0][4]);
squares[0][4]->setPiece(pieces[0][5]);
squares[0][5]->setPiece(pieces[0][3]);
squares[0][6]->setPiece(pieces[0][2]);
```

```
24 squares[0][7]->setPiece(pieces[0][6]);
25 squares[7][0]->setPiece(pieces[1][1]);
26 squares[7][1]->setPiece(pieces[1][2]);
27 squares[7][2]->setPiece(pieces[1][3]);
28 squares[7][3]->setPiece(pieces[1][4]);
29 squares[7][4]->setPiece(pieces[1][5]);
30 squares[7][5]->setPiece(pieces[1][3]);
31 squares[7][6]->setPiece(pieces[1][2]);
32 squares[7][7]->setPiece(pieces[1][6]);
```

squares[i][j] to dwuwymiarowa tablica wskaźników do obiektów typu Square, reprezentująca pola szachownicy. Każdy element tej tablicy to obiekt Square, który przechowuje informacje o polu (przycisk, figura oraz wiersz, kolumna i kolor pola).

Struktura klas

Projekt składa się z kilku klas, które są ze sobą powiązane w celu zorganizowania logiki gry w szachy. Poniżej przedstawiona jest struktura klas, ich relacje oraz krótkie opisy.

Chess2DDialog: Główna klasa projektu, dziedzicząca po wxDialog, reprezentująca okno gry w szachy. Odpowiada za obsługę interfejsu użytkownika i inicjalizację gry.

Board: Klasa reprezentująca szachownicę. Odpowiada za logikę gry, przechowuje obiekty klasy Square reprezentujące pola na szachownicy.

Square: Klasa reprezentująca pole na szachownicy. Zawiera informacje o kolorze pola, wierszu i kolumnie pola oraz referencje do przycisku wxBitmapButton i do figury Piece.

Piece: Klasa abstrakcyjna, po niej dziedziczą konkretne rodzaje figur szachowych. Zawiera podstawowe metody i pola, takie jak: kolor, czy to czy figura się ruszała. Wprowadza metode wirtualną dotyczącą zwracania typu figury.

Pawn, Rook, Knight, Bishop, Queen, King: Klasy dziedziczące po klasie Piece, reprezentujące poszczególne rodzaje figur szachowych. Każda z tych klas implementuje pola z informacjami na temat danej figury.

Relacje między klasami:

Chess2DDialog zawiera obiekt klasy **Board**, który odpowiada za logikę gry i przechowuje informacje o szachownicy.

Klasa **Board** zarządza tablicą obiektów **Square** reprezentujących pola na szachownicy.

Każdy obiekt **Square** zawiera referencję do obiektu wxBitmapButton oraz informacje o obecności figury (**Piece**).

Klasy **Pawn, Rook, Knight, Bishop, Queen, King** dziedziczą po klasie **Piece** i zawierają specyficzne implementacje metod związanych z ich ruchami.

Ta struktura klas pozwala na uporządkowanie kodu i łatwość rozbudowy gry.

Szczegóły konkretnych klas:

Chess2DDialog

```
wxBitmapButton* board[8][8];
wxBitmap images[2][13];

int counter = 1;
int whiteOrBlack = 1;

_B = new Board(1);
```

W konstruktorze klasy tworzony jest obiekt klasy Board, który będzie reprezentować szachownicę w grze. Tutaj są wczytywane obrazki do images[2][13] oraz tworzone przyciski umieszczane w board[8][8], a także liczniki odpowiadające za liczenie kliknięć i pilnowanie czyj mamy ruch. Zawiera również obsługę zdarzeń związanych z interakcją użytkownika (OnBitmapButtonClick()). Także w tym miejscu jest kod odpowiedzialny za wyświetlanie szachownicy.

Board

```
class Board
{
  public:
  Board(int i);
  ~Board();
  void cleaning();
  void creatingSquares();
  void creatingPieces();
  void assigningPieces();
  void restart();
  void setClicked(int _nrBB);
  void setDestination(int _nrBB);
  bool isClickedPiece();
  bool isDestinationPiece();
  bool arePiecesSameColor();
  bool isGoodColorMoving();
  void updateSquares(Square* _clicked, Square* _destination);
  bool isKingInside();
  void wasKingMoving();
  int whereIsKing();
```

```
  bool castling();
  bool pawnBlockedByPieceInFront();
  void pawnMovesButNothingIsInFront();
  void pawnTakes();
  void pawnPromotion();
  void whereICanMove(Square* clicked);
  bool isInSetOfMoves();
  bool isSomethingBetween(Square* _squareOne, Square* _squareTwo,
      int _typeInt);
  bool isBeatable(Square* _square);
  bool moveSimulation(Square* clicked, Square* destination);
  bool isMate();
  Piece* pieces[2][7];
  Square* squares[8][8];
  Square* clicked;
  Square* destination;
  Square* whiteKing;
  Square* blackKing;
  Square* squareBetween;
  Square* target;
  set<int> setOfMoves;
  list<int> listOfThreats;
  list<int> listOfDefenders;
  list<int> listOfHope;
  list<int> listOfInsanity;
};
```

Klasa Board jest centralnym punktem logiki gry, zarządzającym szachownicą. Zawiera tablicę obiektów Square oraz tablicę obiektów Piece. Jest w niej kod odpowiedzialny za ustawienie początkowego rozmieszczenia figur na szachownicy. Klasa Board implementuje logikę ruchu figur, sprawdzając, czy dany ruch jest zgodny z zasadami gry w szachy. Odpowiada za obsługę zdarzeń związanych z interakcją użytkownika na poziomie szachownicy, takich jak kliknięcia na pola. Po wykonaniu ruchu odpowiednio aktualizuje widok.

Square

```
class Square
{
  public:
  friend class Board;
  Square(int _row, int _col, bool _backgroundColor, wxBitmapButton*
      _button, Piece* _piece = nullptr);
  ~Square();
  int getRow();
  int getCol();
  bool getBackgroundColor();
  wxBitmapButton* getButton();
  Piece* getPiece();
  void setPiece(Piece* _piece);

  protected:
  int row;
```

```
  int col;
  bool backgroundColor;
  wxBitmapButton* button;
  Piece* piece;
};
```

Klasa Square reprezentuje pojedyncze pole na szachownicy. Przechowuje informację o kolorze pola na szachownicy (jasne/ciemne), wierszu i kolumnie pola. Zawiera referencję do obiektu klasy Piece reprezentującego figurę. Jeśli pole jest puste, wartość ta to nullptr. Przechowuje referencję do przycisku, który jest powiązany z danym polem na szachownicy.

Piece

```
class Piece
{
  public:
  Piece(bool _color, bool _moved);
  virtual ~Piece();
  bool getColor();
  bool getMoved();
  void setMoved();
  virtual int getTypeInt()=0;

  protected:
  bool color;
  bool moved;
};
```

Klasa Piece stanowi abstrakcję dla poszczególnych rodzajów figur (piona, wieży, skoczka, gońca, hetmana, króla). Klasa posiada konstruktor, który inicjuje kolor figury oraz informację o tym, czy figura wykonała ruch. Posiada metode virtual int getTypeInt()=0: Jest to metoda czysto wirtualna, zwracająca typ figury jako wartość liczbową (przydatna do indeksach naszych tablic).

Pawn, Rook, Knight, Bishop, Queen, King

```
class Pawn:public Piece
{
  public:
  Pawn(bool _color, bool _moved = false);
  ~Pawn();
  int getTypeInt();

  private:
  int TypeInt = 1;
};
```

Klasy te dziedziczą po klasie Piece i reprezentują konkretne figury w grze. Poniżej przedstawiono kluczowe elementy tech klas: Posiada informację o tym, czy figura wykonała swój pierwszy ruch w grze. To istotne np. dla reguły pierwszego ruchu piona lub dla roszady Posiada informację (int) o typie konkretnej figury. Wszytskie te klasy są identyczne.

Metody w klasie Board:

Odpowiedzialne za restartowanie gry oraz usuwanie i tworzenie obiektów:

```
void cleaning();
void creatingSquares();
void creatingPieces();
void assigningPieces();
void restart();
```

• **cleaning()**: Usuwa zmienne tworzone dynamicznie (korzysta z niej destruktor i restart).
• **creatingSquares()**: Tworzy obiekty reprezentujące pola na szachownicy.
• **creatingPieces()**: Tworzy obiekty reprezentujące figury szachowe.
• **assigningPieces()**: Przypisuje figury do odpowiednich pól na szachownicy (pozycje startowe).
• **restart()**: Przywraca początkowe ustawienie figur na szachownicy.
Odpowiedzialne za obsługę interakcji użytkownika z szachownicą, kontrolujące kliknięcia, aby zapobiec nieoczekiwanym i niechcianym zachowaniom:

```
void setClicked(int _nrBB);
void setDestination(int _nrBB);
bool isClickedPiece();
bool isDestinationPiece();
bool arePiecesSameColor();
bool isGoodColorMoving();
```

• **setClicked(int _nrBB)**: ustawia kliknięte pole
• **setDestination(int _nrBB)**: ustawia docelowe pole
• **isClickedPiece()**: sprawdza czy w kliknięty polu jest jakaś figura (używana po to żeby sprawdzić czy w wybranym polu jest jakaś figura, którą można się ruszyć)
• **isDestinationPiece()**: sprawdza czy w docelowym polu jest jakaś figura
• **arePiecesSameColor()**: sprawdza czy w kliknięty i docelowym polu są figury tego samego koloru (używana żeby zapobiec ruszaniu się na pola gdzie mamy własne figury)
• **isGoodColorMoving()**: sprawdza czy ruch wykonuje kolor, którego jest teraz kolej
Odpowiedzialna za aktualizacje szachownicy po ruchu:

```
void updateSquares(Square* _clicked, Square* _destination);
```

• **updateSquares(Square* _clicked, Square* _destination)**: ustawia obrazki i figury na polu kliknięty m i docelowym

Odpowiedzialne za lokalizację oraz zmianę pozycji króli królów:

```
bool isKingInside();
void wasKingMoving();
int whereIsKing();
```

• **isKingInside()**: sprawdza czy król jest w kliknięty m polu
• **wasKingMoving()**: sprawdza na koniec ruchu czy król się ruszył i aktualizuje jego lokalizacje
• **whereIsKing()**: zwraca pozycje króla strony, która jest teraz na ruchu

Odpowiedzialna za roszadę

```
bool castling();
```

• **castling()**: sprawdza czy w danym ruchu roszada miała miejsce, jeśli nie miała lub się udała to zwraca false, jeśli nie można było jej zrobić zwraca true

Odpowiedzialne za ruch i bicie (przez niego) piona oraz jego promocje:

```
bool pawnBlockedByPieceInFront();
void pawnMovesButNothingIsInFront();
void pawnTakes();
void pawnPromotion();
```

• **pawnBlockedByPieceInFront()**: sprawdza czy ruch pion jest blokowany przez figury przed nim
• **pawnMovesButNothingIsInFront()**: wykonuje ruch piona, gdy nic przed nim nie stoi
• **pawnTakes()**: wykonuje bicie po przekątnej gdy są tam jakieś figury przeciwnika
• **pawnPromotion()**: sprawdza czy pion znajduje się w pierwszym rzędzie u przeciwnika wtedy zamienia go w hetmana

Odpowiedzialne za sprawdzanie legalnych ruchów dla każdej figury:

```
void whereICanMove();
bool isInSetOfMoves();
bool isSomethingBetween(Square* _squareOne, Square* _squareTwo, int
     _typeInt);
```

• **whereICanMove()**: tworzy zbiór legalnych ruchów dla figury wybranej przez gracza
• **isInSetOfMoves()**: sprawdza czy ruch wybrany przez gracza jest w zbiorze legalnych ruchów dla danej figury
• **isSomethingBetween(Square* _squareOne, Square* _squareTwo, int _typeInt)**: sprawdza czy między dwoma polami jest jakaś figura (używane do sprawdzenia czy nic nie stoi na drodze na dane pole)

Odpowiedzialne za szachowanie i symulacje ruchów

```
bool isBeatable(Square* _square);
bool moveSimulation();
```

• **isBeatable(Square* _square)**: sprawdza czy wybrana figura jest podbiciem przez jakąś figure (używane głównie do sprawdzania szachowania króla)
• **moveSimulation()**: przeprowadza symulacje ruchu (potem cofa wprowadzone zmiany) w sytuacji gdy król jest pod szachem, jeśli ruch gracza sprawia, że król nie jest już pod szachem zwraca true, w przeciwnym wypadku false

Przeprowadzenie przez rozgrywkę

Obsługa przycisków (OnBitmapButton1Click)

```
void Chess2DDialog::OnBitmapButton1Click(wxCommandEvent& event){
  int nrBB = event.GetId() - 100;

  if(counter%2 != 0){
    _B->setClicked(nrBB);
    if(!(_B->isClickedPiece()) || !(_B->isGoodColorMoving())){
      return;
    }
    counter++;
    return;
  }

  if(counter%2 == 0){
    _B->setDestination(nrBB);
    if(_B->isDestinationPiece() && _B->arePiecesSameColor()){
      counter--;
      return;
    }
    _B->whereICanMove();

    if(!_B->isInSetOfMoves()){
      counter--;
      return;
    }
    if(_B->isSomethingBetween(_B->clicked, _B->destination, _B->
        clicked->getPiece()->getTypeInt())){
      counter--;
      return;
    }

    if(_B->moveSimulation()){
      wxLogMessage("Tutaj nie wolno");
      counter--;
      return;
    }

    if(_B->castling()){
      counter--;
      return;
```

```
39      }
40      _B->updateSquares(_B->clicked, _B->destination);
41
42      _B->pawnPromotion();
43      _B->wasKingMoving();
44
45      whiteOrBlack = (whiteOrBlack + 1)%2;
46
47      if( _B->isMate() == true ){
48        wxLogMessage("Mat");
49      }
50      counter++;
51      return;
52    }
53  }
```

Funkcja OnBitmapButton1Click w każdym kliknięciu zapisuje nrBB (czyli numer bitmapy indywidualny dla każdego przycisku) a następnie sprawdza po kolei warunki z taką zasadą, że jeśli coś jest nie tak to wracamy do kliknięcia 1. W kliknięciu 1, poprzez wskaźnik do clicked zapisuje informacje o kliknięty Squarze. Następnie sprawdza po kolei czy na klikniętym polu znajdowała się figura a potem czy ma ona odpowiedni kolor (czyli czy dobra osoba się rusza). Kolejność jest ważna, ponieważ bez sprawdzenia czy na klikniętym polu jest figura mogłoby dojść do sytuacji, że chcemy użyć metody z klasy Piece na polu, w którym Piece-a nie ma i program zakończyłby działanie. Jeśli wszystko się zgadza zwiększamy counter o 1 i wychodzimy z metody. Przy kliknięciu 2 tak jak w 1 przekazujemy wskaźnik tylko tym razem do destination. Sprawdzamy wpierw czy kliknięte miejsce ma figurę a jeśli ma to sprawdzamy czy jest tego samego koloru. Jeśli jednak nie było z tym problemu to do zbioru poprzez funkcje whereICanMove() zapisujemy możliwe ruchy nie uwzględniając tego, że figura może zostać zablokowana. Następnie sprawdzamy czy destination w ogóle zawiera się w naszym zbiorze ruchów. Jeśli tak to następnie sprawdzamy czy jest coś między polem startowym a docelowym a jeśli i to przejdzie bez problemów to wykonywana jest symulacja ruchu, która ma sprawdzić czy po ruchu nasz król nie będzie pod szachem. Pod koniec sprawdzamy czy może ktoś niechciałby zrobić roszady. Uaktualniamy pozycje, sprawdzamy czy na ostatnich liniach nie stoją piony, które chciałyby dostać awans. Jeśli ruszał się król to poprzez metodę wasKingMoving() sprawdzamy jego kolor i w zależności od tego uaktualniamy blackKing lub whiteKing, dzięki którym możemy na bieżąco sprawdzać gdzie znajdują się króle na planszy. Na koniec naszego ruchu sprawdzamy czy nie zmatowaliśmy przeciwnika.

Wybrane metody z Board.cpp

updateSquares(Square* _clicked, Square* _destination)

```
1  void Board::updateSquares(Square* _clicked, Square* _destination){
2    this->destination->setPiece(this->clicked->getPiece());
3    this->destination->getButton()->SetBitmap(images[this->clicked->
         getPiece()->getColor()][this->clicked->getPiece()->getTypeInt
         ()+1-this->destination->getBackgroundColor()]);
4    this->destination->getPiece()->setMoved();
5    this->clicked->setPiece(nullptr);
6    this->clicked->getButton()->SetBitmap(images[this->clicked->
         getBackgroundColor()][0]);
7    }
```

Metoda działa w taki sposób, że najpierw w miejscu docelowym ustawiamy jaka figura będzie stać na tym polu, następnie obrazek figury, którą chcemy się ruszyć. Wykorzystujemy tutaj fakt, że w tablicy images[a][b] a odpowiada za kolor figury a b za to jaka to figura oraz czy stoi na polu białym czy czarnym gdzie w naszej tablicy obrazki na biały i czarnym polu znajdują się od siebie w odległości 1. Następnie ustawiamy figurze, że się ruszyła a na koniec usuwamy ślady naszej figury z pola startowego.

castling()

```
1  bool Board::castling(){
2    if(this->clicked != this->squares[whereIsKing()/8][whereIsKing()
         %8]){
3      return 0;
4    }
5
6    if(abs((this->destination->getCol() - this->squares[whereIsKing()
         /8][whereIsKing()%8]->getCol())) == 2){
7      if(this->destination->getCol() == 2){
8        if(this->squares[this->destination->getRow()][0]->getPiece()
             != nullptr && this->squares[this->destination->getRow()
             ][0]->getPiece()->getMoved() == false){
9          if(this->isSomethingBetween(this->clicked,this->squares[
               this->destination->getRow()][0],3 ) == false){
10           this->squares[this->destination->getRow()][3]->setPiece(
                 pieces[this->clicked->getPiece()->getColor()][1]);
11           if(!(this->isBeatable(this->clicked)) && !(this->
                 isBeatable(this->squares[this->destination->getRow()
                 ][3]))){
12             this->squares[this->destination->getRow()][3]->setPiece
                   (this->squares[this->destination->getRow()][0]->
                   getPiece());
13             this->squares[this->destination->getRow()][3]->
                   getButton()->SetBitmap(images[this->squares[this->
                   destination->getRow()][0]->getPiece()->getColor()][
                   this->squares[this->destination->getRow()][0]->
                   getPiece()->getTypeInt()+1-this->squares[this->
                   destination->getRow()][3]->getBackgroundColor()]);
14             this->squares[this->destination->getRow()][3]->getPiece
                   ()->setMoved();
```

```
15              this->squares[this->destination->getRow()][0]->setPiece
                    (nullptr);
16              this->squares[this->destination->getRow()][0]->
                    getButton()->SetBitmap(images[this->squares[this->
                    destination->getRow()][0]->getBackgroundColor()
                    ][0]);
17              return 0;
18            }return 1;
19
20          }return 1;
21
22        }return 1;
23
24      }else if(this->destination->getCol() == 6){
25        if(this->squares[this->destination->getRow()][7]->getPiece()
              != nullptr && this->squares[this->destination->getRow()
              ][7]->getPiece()->getMoved() == false){
26          if(this->isSomethingBetween(this->clicked,this->squares[
                this->destination->getRow()][7], 3) == false){
27            this->squares[this->destination->getRow()][5]->setPiece(
                  pieces[this->clicked->getPiece()->getColor()][1]);
28            if(!(this->isBeatable(this->clicked)) && !(this->
                  isBeatable(this->squares[this->destination->getRow()
                  ][5]))){
29              this->squares[this->destination->getRow()][5]->setPiece
                    (this->squares[this->destination->getRow()][7]->
                    getPiece());
30              this->squares[this->destination->getRow()][5]->
                    getButton()->SetBitmap(images[this->squares[this->
                    destination->getRow()][7]->getPiece()->getColor()][
                    this->squares[this->destination->getRow()][7]->
                    getPiece()->getTypeInt()+1-this->squares[this->
                    destination->getRow()][5]->getBackgroundColor()]);
31              this->squares[this->destination->getRow()][5]->getPiece
                    ()->setMoved();
32              this->squares[this->destination->getRow()][7]->setPiece
                    (nullptr);
33              this->squares[this->destination->getRow()][7]->
                    getButton()->SetBitmap(images[this->squares[this->
                    destination->getRow()][7]->getBackgroundColor()
                    ][0]);
34              return 0;
35            }return 1;
36
37          }return 1;
38
39        }return 1;
40
41      }return 1;
42    }
43    return 0;
44  }
```

Na początek sprawdzamy czy w ogóle próbowaliśmy ruszyć się królem.
Następnie sprawdzamy w warunkach kolejno:
• Czy pole docelowe znajduje się w odległości 2 od króla - Jest to jedyna taka

sytuacja gdy król może poruszyć się o 2 pola
• Czy król próbuje zrobić roszadę długą - Czyli czy kolumna docelowa ma numer 2 (numerując od 0)
• Czy po drugiej stronie w ogóle stoi wieża z którą moglibyśmy zrobić roszadę oraz czy dla wieży to będzie pierwszy ruch
• Czy pomiędzy królem a wieżą jest pusta przestrzeń - z pomocą metody isSomethingBetween
Następnie w pole, w którym znajdzie się wieża po roszadzie wpisujemy wieżę by móc dla tego pola wykonać metodę isBeatable i sprawdzamy czy pola, na których był lub przez, które będzie przechodził król są szachowane. Jeśli wszystko jest jak powinno to następuje roszada. To samo wykonuje się dla roszady krótkiej tyle, że sprawdzamy inne pola.

isSomethingBetween(Square* _squareOne, Square* _squareTwo, int _typeInt)

```
1  bool Board::isSomethingBetween(Square* _squareOne, Square*
       _squareTwo, int _typeInt){
2    listOfDefenders.clear();
3    int _col = _squareOne->getCol();
4    int _row = _squareOne->getRow();
5    int wsp_col = - (_col - _squareTwo->getCol())/abs(_col -
       _squareTwo->getCol());
6    this->squareBetween = squares[this->whereIsKing()/8][this->
       whereIsKing()%8];
7    switch(_typeInt){
8      case 1:
9      if(abs(_row - _squareTwo->getRow()) == 2 && this->clicked->
         getPiece()->getColor() == 1){
10       if(this->squares[ 5 ][_col]->getPiece() != nullptr){
11         return true;
12       }
13     }
14     if(abs(_row - _squareTwo->getRow()) == 2 && this->clicked->
         getPiece()->getColor() == 0){
15       if(this->squares[ 2 ][_col]->getPiece() != nullptr){
16         return true;
17       }
18     }
19     return false;
20     break;
21     case 3:
22     if(_col - _squareTwo->getCol() == 0){
23       for(int row = _row - (_row - _squareTwo->getRow())/abs(_row -
           _squareTwo->getRow()); abs(row - _squareTwo->getRow())
         >0; row = row - (row - _squareTwo->getRow())/abs(row -
         _squareTwo->getRow())){
24         this->listOfDefenders.push_back(row * 8 + _col);
25         if(this->squares[row][_col]->getPiece() != nullptr){
26           this->squareBetween = this->squares[row][_col];
27           return true;
28         }
29       }
30       return false;
```

```
31      }
32      if(_row - _squareTwo->getRow() == 0){
33        for(int col = _col - (_col - _squareTwo->getCol())/abs(_col -
               _squareTwo->getCol()); abs(col - _squareTwo->getCol())
              >0; col = col - (col - _squareTwo->getCol())/abs(col -
              _squareTwo->getCol())){
34          this->listOfDefenders.push_back(_row * 8 + col);
35          if(this->squares[_row][col]->getPiece() != nullptr){
36            this->squareBetween = this->squares[_row][col];
37            return true;
38          }
39        }
40        return false;
41      }
42      break;
43      case 7:
44      for(int wsp = _row - (_row - _squareTwo->getRow())/abs(_row -
            _squareTwo->getRow()); abs(wsp - _squareTwo->getRow()) >0;
            wsp = wsp - (wsp - _squareTwo->getRow())/abs(wsp -
            _squareTwo->getRow())){
45        this->listOfDefenders.push_back(wsp * 8 + _col + wsp_col);
46        if(this->squares[wsp][_col + wsp_col]->getPiece() != nullptr)
              {
47          this->squareBetween = this->squares[wsp][_col + wsp_col];
48          return true;
49        }else{
50          _col = wsp_col + _col;
51        }
52      }
53      return false;
54      break;
55      case 9:
56      wsp_col = - (_col - _squareTwo->getCol())/abs(_col - _squareTwo
            ->getCol());
57      if(_col - _squareTwo->getCol() == 0){
58        for(int row = _row - (_row - _squareTwo->getRow())/abs(_row -
               _squareTwo->getRow()); abs(row - _squareTwo->getRow())
              >0; row = row - (row - _squareTwo->getRow())/abs(row -
              _squareTwo->getRow())){
59          this->listOfDefenders.push_back(row * 8 + _col);
60          if(this->squares[row][_col]->getPiece() != nullptr){
61            return true;
62          }
63        }
64        return false;
65      }
66      if(_row - _squareTwo->getRow() == 0){
67        for(int col = _col - (_col - _squareTwo->getCol())/abs(_col -
               _squareTwo->getCol()); abs(col - _squareTwo->getCol())
              >0; col = col - (col - _squareTwo->getCol())/abs(col -
              _squareTwo->getCol())){
68          this->listOfDefenders.push_back(_row * 8 + col);
69          if(this->squares[_row][col]->getPiece() != nullptr){
70            return true;
71          }
72        }
73        return false;
```

```
74      }
75      for(int wsp = _row - (_row - _squareTwo->getRow())/abs(_row -
            _squareTwo->getRow()); abs(wsp - _squareTwo->getRow()) >0;
            wsp = wsp - (wsp - _squareTwo->getRow())/abs(wsp -
            _squareTwo->getRow())){
76        this->listOfDefenders.push_back(wsp * 8 + _col + wsp_col);
77        if(this->squares[wsp][_col + wsp_col]->getPiece() != nullptr)
              {
78          return true;
79        }else{
80          _col = wsp_col + _col;
81        }
82      }
83      return false;
84      break;
85
86      default:
87      return false;
88    }
89    return false;
90  }
```

W tej metodzie robimy wiele rzeczy na raz. Była ona robiona z myślą, żeby sprawdzać czy jak chcemy się gdzieś ruszyć to nie stoi nam coś na drodze jednak potem uznaliśmy, że może nam się przydać żeby sprawdzić czy coś nie atakuje króla. Dlatego w momencie gdy funkcja natknie sie na jakieś pole, to zapisuje je do squareBetween. Zapisuje ona również do listy listOfDefenders wszystkie pola pomiędzy polem startowym a końcowym. Co do działania funkcji to w argumentach przekazujemy jej pole początkowe oraz końcowe oraz jako kto chcemy się poruszyć (jaką figurą). W zależności od naszego _typeInt wchodzimy do innego case-a w switchu i kolejno dla piona sprawdzamy tylko sytuację gdy może ruszyć się o 2, dla wieży sprawdzamy czy ruch odbywa się na kierunku poziomym czy pionowym a następnie odpowiednio czy góra/prawo czy dół/lewo. Dla gońca sprawdzamy po której przekątnej będziemy sprawdzać na zasadzie +/- 1 na kierunku poziomym i +/- 1 pionowym. Hetman to połączona wieża i goniec. Pozostałe figury nie mają takich ograniczeń.

isBeatable(Square* _square)

```
1   bool Board::isBeatable(Square* _square){
2     this->listOfThreats.clear();
3
4     int _Row = _square->getRow();
5     int _Col = _square->getCol();
6
7     //is Rook checking us
8     for(int i = 0; i < 8; i = i+7){
9       if(_Row != i && isSomethingBetween(squares[_Row][_Col], squares
            [i][_Col], 3)){
10        if(this->squareBetween->getPiece()->getTypeInt() == 3 || this
              ->squareBetween->getPiece()->getTypeInt() == 9){
11          if(this->squares[_Row][_Col]->getPiece()->getColor() !=
                this->squareBetween->getPiece()->getColor()){
```

```
12            this->listOfThreats.push_back(this->squareBetween->getRow
                  () * 8 + this->squareBetween->getCol());
13
14          }
15        }
16      }else{
17        if(_Row != i && this->squares[i][_Col]->getPiece() != nullptr
               && this->squares[_Row][_Col]->getPiece()->getColor() !=
              this->squares[i][_Col]->getPiece()->getColor()){
18          if(this->squares[i][_Col]->getPiece()->getTypeInt() == 9 ||
                this->squares[i][_Col]->getPiece()->getTypeInt() == 3)
                {
19            this->listOfThreats.push_back(this->squares[i][_Col]->
                  getRow() * 8 + this->squares[i][_Col]->getCol());
20          }
21        }
22      }
23      if(_Col != i && isSomethingBetween(squares[_Row][_Col], squares
            [_Row][i], 3)){
24        if(this->squareBetween->getPiece()->getTypeInt() == 3 || this
              ->squareBetween->getPiece()->getTypeInt() == 9){
25          if(this->squares[_Row][_Col]->getPiece()->getColor() !=
                this->squareBetween->getPiece()->getColor()){
26            this->listOfThreats.push_back(this->squareBetween->getRow
                  () * 8 + this->squareBetween->getCol());
27          }
28        }
29      }else{
30        if(_Col != i && this->squares[_Row][i]->getPiece() != nullptr
               && this->squares[_Row][_Col]->getPiece()->getColor() !=
              this->squares[_Row][i]->getPiece()->getColor()){
31          if(this->squares[_Row][i]->getPiece()->getTypeInt() == 9 ||
                this->squares[_Row][i]->getPiece()->getTypeInt() == 3)
                {
32            this->listOfThreats.push_back(this->squares[_Row][i]->
                  getRow() * 8 + this->squares[_Row][i]->getCol());
33          }
34        }
35      }
36
37    }
38
39    //is Bishop checking us
40    for(int i = 1; i < 8; i++){
41      if((_Row)- i >=0 && _Col-i >=0){
42        if(squares[_Row-i][_Col-i]->getPiece() != nullptr){
43          if(this->squares[_Row][_Col]->getPiece()->getColor()!= this
                ->squares[_Row-i][_Col-i]->getPiece()->getColor()){
44            if(this->squares[_Row-i][_Col-i]->getPiece()->getTypeInt
                  () == 7 || this->squares[_Row-i][_Col-i]->getPiece()
                  ->getTypeInt() == 9){
45              this->listOfThreats.push_back(this->squares[_Row - i][
                    _Col - i]->getRow() * 8 + this->squares[_Row - i][
                    _Col - i]->getCol());
46              continue;
47            }
48          }
```

```
49          i = i+10;
50        }
51      }
52    }
53
54    for(int i = 1; i < 8; i++){
55      if((_Row)- i >=0 && _Col+i <=7){
56        if(squares[_Row-i][_Col+i]->getPiece() != nullptr){
57          if(this->squares[_Row][_Col]->getPiece()->getColor()!= this
                ->squares[_Row-i][_Col+i]->getPiece()->getColor()){
58            if(this->squares[_Row-i][_Col+i]->getPiece()->getTypeInt
                  () == 7 || this->squares[_Row-i][_Col+i]->getPiece()
                  ->getTypeInt() == 9){
59              this->listOfThreats.push_back(this->squares[_Row - i][
                    _Col + i]->getRow() * 8 + this->squares[_Row - i][
                    _Col + i]->getCol());
60              continue;
61            }
62          }
63          i = i+10;
64        }
65      }
66    }
67
68    for(int i = 1; i < 8; i++){
69      if((_Row)+ i <=7 && _Col+i <=7){
70        if(squares[_Row+i][_Col+i]->getPiece() != nullptr){
71          if(this->squares[_Row][_Col]->getPiece()->getColor()!= this
                ->squares[_Row+i][_Col+i]->getPiece()->getColor()){
72            if(this->squares[_Row+i][_Col+i]->getPiece()->getTypeInt
                  () == 7 || this->squares[_Row+i][_Col+i]->getPiece()
                  ->getTypeInt() == 9){
73              this->listOfThreats.push_back(this->squares[_Row + i][
                    _Col + i]->getRow() * 8 + this->squares[_Row + i][
                    _Col + i]->getCol());
74              continue;
75            }
76          }
77          i = i+10;
78        }
79      }
80    }
81
82    for(int i = 1; i < 8; i++){
83      if((_Row)+ i <=7 && _Col-i >=0){
84        if(squares[_Row+i][_Col-i]->getPiece() != nullptr){
85          if(this->squares[_Row][_Col]->getPiece()->getColor()!= this
                ->squares[_Row+i][_Col-i]->getPiece()->getColor()){
86            if(this->squares[_Row+i][_Col-i]->getPiece()->getTypeInt
                  () == 7 || this->squares[_Row+i][_Col-i]->getPiece()
                  ->getTypeInt() == 9){
87              this->listOfThreats.push_back(this->squares[_Row + i][
                    _Col - i]->getRow() * 8 + this->squares[_Row + i][
                    _Col - i]->getCol());
88              continue;
89            }
90          }
```

```
91          i = i+10;
92        }
93      }
94    }
95
96    //is Pawn checking us
97    for(int i = -1; i < 2; i = i + 2){
98      if(_Row == 0){
99      }else{
100       if(_Col - i >= 0 && _Col - i <= 7 && squares[_Row - 1][_Col -
              i]->getPiece() != nullptr){
101         if(this->squares[_Row][_Col]->getPiece()->getColor() !=
              this->squares[_Row - 1][_Col - i]->getPiece()->getColor
              ()){
102           if(this->squares[_Row - 1][_Col - i]->getPiece()->
                getTypeInt() == 1 && this->squares[_Row - 1][_Col - i
                ]->getPiece()->getColor() == 0){
103             this->listOfThreats.push_back(this->squares[_Row - 1][
                  _Col - i]->getRow() * 8 + this->squares[_Row - 1][
                  _Col - i]->getCol());
104           }
105         }
106       }
107     }
108   }
109
110   for(int i = -1; i < 2; i = i + 2){
111     if(_Row == 7){
112     }else{
113       if(_Col - i >= 0 && _Col - i <= 7 && squares[_Row + 1][_Col -
              i]->getPiece() != nullptr){
114         if(this->squares[_Row][_Col]->getPiece()->getColor()!= this
              ->squares[_Row + 1][_Col - i]->getPiece()->getColor()){
115           if(this->squares[_Row + 1][_Col - i]->getPiece()->
                getTypeInt() == 1 && this->squares[_Row + 1][_Col - i
                ]->getPiece()->getColor() == 1){
116             this->listOfThreats.push_back(this->squares[_Row + 1][
                  _Col - i]->getRow() * 8 + this->squares[_Row + 1][
                  _Col - i]->getCol());
117           }
118         }
119       }
120     }
121   }
122
123   // is Knight checking us
124   for(int i = 0; i < 5; i = i+4){
125     for(int j = 0; j < 3; j = j+2){
126       if(_Row + i  <= 9 && _Row + i >=2 && _Col + j <= 8 && _Col +
              j >=1 ){
127         if(squares[_Row + i - 2][_Col + j - 1]->getPiece() !=
              nullptr){
128           if(this->squares[_Row][_Col]->getPiece()->getColor()!=
                this->squares[_Row + i - 2][_Col + j - 1]->getPiece()
                ->getColor()){
129             if(this->squares[_Row + i - 2][_Col + j - 1]->getPiece
                  ()->getTypeInt() == 5){
```

```
130                 this->listOfThreats.push_back(this->squares[_Row + i
                        - 2][_Col + j - 1]->getRow() * 8 + this->squares[
                        _Row + i - 2][_Col + j - 1]->getCol());
131               }
132             }
133           }
134         }
135         if(_Row + j <= 8 && _Row + j >=1 && _Col + i <= 9 && _Col + i
                >= 2){
136           if(squares[_Row + j - 1][_Col + i - 2]->getPiece() !=
                nullptr){
137             if(this->squares[_Row][_Col]->getPiece()->getColor()!=
                    this->squares[_Row + j - 1][_Col + i - 2]->getPiece()
                    ->getColor()){
138               if(this->squares[_Row + j - 1][_Col + i - 2]->getPiece
                      ()->getTypeInt() == 5){
139                 this->listOfThreats.push_back(this->squares[_Row + j
                        - 1][_Col + i - 2]->getRow() * 8 + this->squares[
                        _Row + j - 1][_Col + i - 2]->getCol());
140               }
141             }
142           }
143         }
144       }
145     }
146
147     // is King checking us
148     if(_square->getPiece()->getColor() == 0){
149       if(abs(this->whiteKing->getCol() - _square->getCol())<=1 && abs
              (this->whiteKing->getRow() - _square->getRow())<=1){
150         this->listOfThreats.push_back(this->whiteKing->getRow() * 8 +
                this->whiteKing->getCol());
151       }
152     }else{
153       if(abs(this->blackKing->getCol() - _square->getCol())<=1 && abs
              (this->blackKing->getRow() - _square->getRow())<=1){
154         this->listOfThreats.push_back(this->blackKing->getRow() * 8 +
                this->blackKing->getCol());
155       }
156     }
157     if(!this->listOfThreats.empty()){
158       return true;
159     }
160     return false;
161 }
```

Metoda działa w taki sposób, że dla pola, które chcemy sprawdzić czy jest atakowane wykonuje po kolei oraz w każdym kierunku isSomethingBetween z type intem odpowiednim w zależności od tego czego szukamy np. przed nami w oddali będziemy poszukiwać hetmana lub wieży zaś po skosie będziemy szukać hetmana lub gońca. Każde pole, z którego można zbić wybrane przez nas pole zapisywane jest do listOfThreats. Wiele podobnych rzeczy co w isSomethingBetween więc nie będziemy się nad nimi rozwodzić.

moveSimulation(Square* _clicked, Square* _destination)

```
bool Board::moveSimulation(Square* _clicked, Square* _destination){
  bool x = false;
  if((this->blackKing == _clicked) + (this->whiteKing == _clicked)
      == 1){
    if(_clicked->getPiece()->getColor()){
      this->whiteKing = _destination;
    }else{
      this->blackKing = _destination;
    }
  }
  Square* pieceStorage = new Square(-1, -1, 0, board[0][0], nullptr
      );
  if(_destination->getPiece() != nullptr){
    pieceStorage->setPiece(_destination->getPiece());
  }
  _destination->setPiece(_clicked->getPiece());
  _clicked->setPiece(nullptr);
  if(this->isBeatable(this->squares[whereIsKing()/8][whereIsKing()
      %8])){
    x = true;
  }
  _clicked->setPiece(_destination->getPiece());
  if(pieceStorage->getPiece() == nullptr){
    _destination->setPiece(nullptr);
  }else{
    _destination->setPiece(pieceStorage->getPiece());
  }
  if((this->blackKing == _destination) + (this->whiteKing ==
      _destination) == 1){
    if(_clicked->getPiece()->getColor()){
      this->whiteKing = _clicked;
    }else{
      this->blackKing = _clicked;
    }
  }
  delete pieceStorage;
  return x;
}
```

Jest to jedna z najistotniejszych metod naszego programu. Jest to sztuczne wywołanie ruchu, poza wzrokiem gracza. Na początek sprawdzamy czy ruszamy się królem jeśli tak to zmieniamy jego pozycje. Następnie wykonujemy prawie, że zwykły ruch. Prawie, bo nie interesują nas wyświetlane obrazki oraz przechowujemy to pole, do którego ktoś próbuje się ruszyć. Robimy to po to by nie stracić informacji o tym co się w nim znajduje. Po wykonaniu ruchu sprawdzamy czy król jest po nim szachowany i tą informacje zwrócimy na koniec programu. Przed zwróceniem tej informacji wracamy ze wszystkim do pozycji początkowej. Funkcja prosta a jednak jakże przydatna.

isMate()

```
bool Board::isMate(){

  int _col = this->whereIsKing() % 8;
  int _row = this->whereIsKing() / 8;
  this->clicked = this->squares[_row][_col];
  if(!isBeatable(this->squares[_row][_col])){
    return false;
  }
  listOfHope = listOfThreats;
  this->whereICanMove(this->squares[_row][_col]);

  for(auto it = this->setOfMoves.begin(); it != this->setOfMoves.
      end(); ++it){
    this->clicked = this->squares[_row][_col];
    if(abs(_col - *it % 8) == 2){
      continue;
    }

    if(this->squares[*it / 8][*it % 8]->getPiece() == nullptr ||
        this->squares[*it / 8][*it % 8]->getPiece()->getColor() !=
        this->squares[_row][_col]->getPiece()->getColor()){
      this->destination = this->squares[*it / 8][*it % 8];
      if(!this->moveSimulation(this->squares[_row][_col], this->
          squares[*it / 8][*it % 8])){
        return false;
      }
    }
  }



  this->target = squares[*(listOfHope.begin()) / 8][*(listOfHope.
      begin()) % 8];


  if(this->isBeatable(this->target)){
    listOfHope = listOfThreats;
    for(auto it = listOfHope.begin(); it != listOfHope.end(); ++it)
        {
      if(!this->moveSimulation(this->squares[*it / 8][*it % 8],
          this->target)){
        return false;
      }
    }
  }

  if(abs(_row - this->target->getRow()) > 1 ||  abs(_col - this->
      target->getCol()) > 1 ){
    this->isSomethingBetween(this->squares[_row][_col], this->
        target, this->target->getPiece()->getTypeInt());
    listOfHope = listOfDefenders;

    for(auto it = listOfHope.begin(); it != listOfHope.end(); ++it)
        {
      this->squares[*it / 8][*it % 8]->setPiece(pieces[(this->
          squares[_row][_col]->getPiece()->getColor()+1)%2][1]);
```

```
46        this->destination = this->squares[*it / 8][*it % 8];
47        if(isBeatable(squares[*it / 8][*it % 8])){
48          this->squares[*it / 8][*it % 8]->setPiece(nullptr);
49          listOfInsanity = listOfThreats;
50          this->squares[*it / 8][*it % 8]->setPiece(nullptr);
51
52
53
54          if(this->squares[*it / 8][*it % 8]->getRow() == 4 &&
                squares[_row][_col]->getPiece()->getColor() == 1){
55            if(this->squares[6][*it % 8]->getPiece() != nullptr &&
                  this->squares[6][*it % 8]->getPiece()->getTypeInt()
                  == 1){
56              this->clicked = this->squares[6][*it % 8];
57              if(!this->moveSimulation(this->squares[6][*it % 8],
                    this->squares[4][*it % 8])){
58                return false;
59              }
60            }
61          }
62          if(this->squares[*it / 8][*it % 8]->getRow() == 3 &&
                squares[_row][_col]->getPiece()->getColor() == 0){
63            if(this->squares[1][*it % 8]->getPiece() != nullptr &&
                  this->squares[1][*it % 8]->getPiece()->getTypeInt()
                  == 1){
64              this->clicked = this->squares[1][*it % 8];
65              if(!this->moveSimulation(this->squares[1][*it % 8],
                    this->squares[3][*it % 8])){
66                return false;
67              }
68            }
69          }
70
71          if(*it / 8 < 6 && this->squares[*it / 8 + 1][*it % 8]->
                getPiece() != nullptr && this->squares[*it / 8 + 1][*it
                 % 8]->getPiece()->getTypeInt() == 1 && this->squares[*
                it / 8 + 1][*it % 8]->getPiece()->getColor() == 1){
72            this->clicked = this->squares[*it / 8 + 1][*it % 8];
73            if(!this->moveSimulation(this->squares[*it / 8 + 1][*it %
                  8], this->squares[*it / 8][*it % 8])){
74              return false;
75            }
76          }
77          if(*it / 8 > 1 && this->squares[*it / 8 - 1][*it % 8]->
                getPiece() != nullptr && this->squares[*it / 8 - 1][*it
                 % 8]->getPiece()->getTypeInt() == 1 && this->squares[*
                it / 8 - 1][*it % 8]->getPiece()->getColor() == 0){
78            this->clicked = this->squares[*it / 8 - 1][*it % 8];
79            if(!this->moveSimulation(this->squares[*it / 8 - 1][*it %
                  8], this->squares[*it / 8][*it % 8])){
80              return false;
81            }
82          }
83
84
85          for(auto itt = listOfInsanity.begin(); itt !=
                listOfInsanity.end(); ++itt){
```

```
86            this->clicked = this->squares[*itt / 8][*itt % 8];
87            if(isKingInside()){
88              continue;
89            }
90            if(!this->moveSimulation(this->squares[*itt / 8][*itt %
                  8], this->squares[*it / 8][*it % 8]) && clicked->
                  getPiece()->getTypeInt() != 1){
91              return false;
92            }
93          }
94        }
95
96      }
97    }
98    return true;
99  }
```

Na sam koniec zrodzony w bólach i mękach mat. Funkcja pobiera od nas lokalizacje króla (przeciwnego, ponieważ w OnBitmapButtonClicku używamy jej po zwiększeniu licznika whiteOrBlack). Sprawdzamy czy król jest w ogóle szachowany. Następnie sprawdzamy czy może się gdzieś poruszyć bez bycia dalej szachowanym. Jeśli to nie wyjdzie to ustawiamy nasz cel do zbicia i tworzymy listę figur, które mogłyby ten cel zbić. Gdy taka lista zostanie utworzona to robimy symulację ruchów co by było gdybyśmy tą figurą zbili nasz cel. Robimy to dlatego, że ruch naszej figury może odsłonić króla od innej strony. Jeśli nasz cel jest w odległości jednej kratki od nas, nie możemy się nigdzie ruszyć bez bycia szachowanym oraz nie możemy go zbić to jest to już koniec - mamy mata. W innym przypadku tworzymy listę pól spomiędzy naszego celu i króla a następnie po pętli ustawiamy w niej figurę żeby móc wykonać metodę isBeatable i figury, które mogą wejść w to pole by je zasłonić zapisujemy do listy by i dla nich zrobić symulację ruchu. Jako, że pion jako jedyny ma inne ruszanie się od bicia to dla niego musimy oddzielnie sprawdzić czy może zasłonić. Nie dość, że oddzielnie to jeszcze dla samego piona, oddzielnie dla koloru i dlatego czy możemy się nim ruszyć o 2. Jeśli nic z tych rzeczy nie pomogło to już na pewno mamy mata.

Wkład pracy:

Robienie projektu odbywało się w większości w taki sposób, że naprzemiennie jedna osoba pisała i cała dwójka myślała. Czasem zdarzało się, że ktoś coś robił "po godzinach".

- Tworzenie koncepcji projektu : Jan Moskal - 70% , Szymon Makulec - 30%
- Obrabianie obrazków : Jan Moskal - 10%, Szymon Makulec - 90%
- Tworzenie szkieletu klas : Jan Moskal - 90%, Szymon Makulec - 10%

• Board.cpp oraz Chess2DMain.cpp : Jan Moskal - 25%, Szymon Makulec - 75%
• Dokumentacja : Jan Moskal - 45%, Szymon Makulec - 55%
• Debugging : Jan Moskal - 50%, Szymon Makulec - 50%

Podsumowanie

Projekt uważamy za udany. Baliśmy się go, często nie wierzyliśmy, że jest szansa na zrobienie go do końca. Nauczył nas wielu przydatnych umiejętności, sprawił, że nasza psychika jest w stanie wytrzymać więcej niż wcześniej. Pomógł zaznajomić się z biblioteką wxWidgets oraz programu zewnętrznego jak github. Jesteśmy świadomi, na ile nasz kod jest niewydajny dlatego w przyszłości planujemy gruntowne zmiany optymalizacyjne.