תיק פרויקט

שם המגיש: ירדן בצרי
ת.ז: 213217268
תאריך הגשה: 15.5.2022
שם הפרויקט: Checkmate
שם המנחה: יודה אור

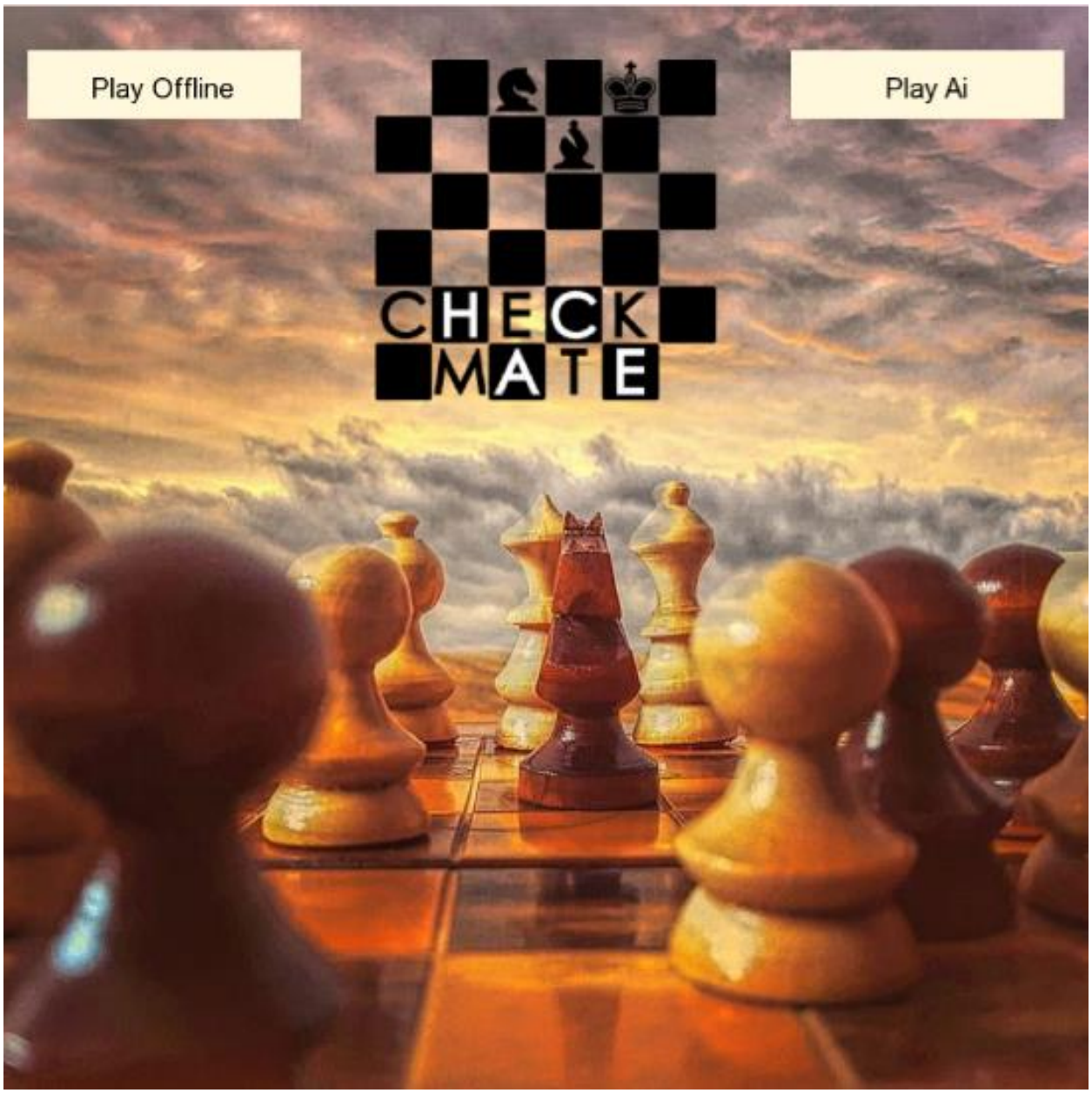
תוכן עניינים

מאת ירדן בצרי

מאת ירדן בצרי

תמונות מהמשחק

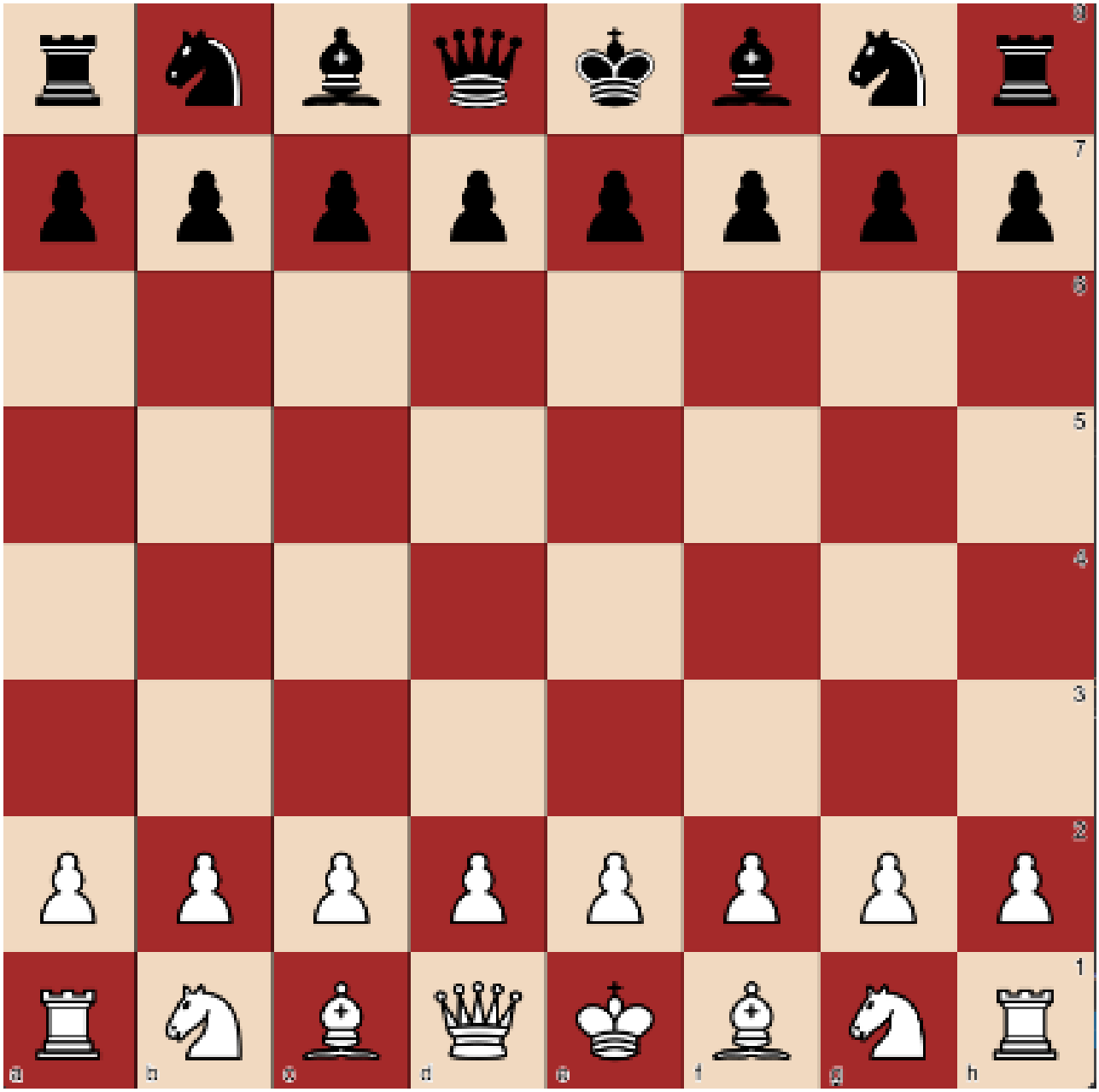
תפריט:



- בפרויקט יש אפשרות לבחור לשחק משחק offline או לבחור לשחק נגד ה Ai.

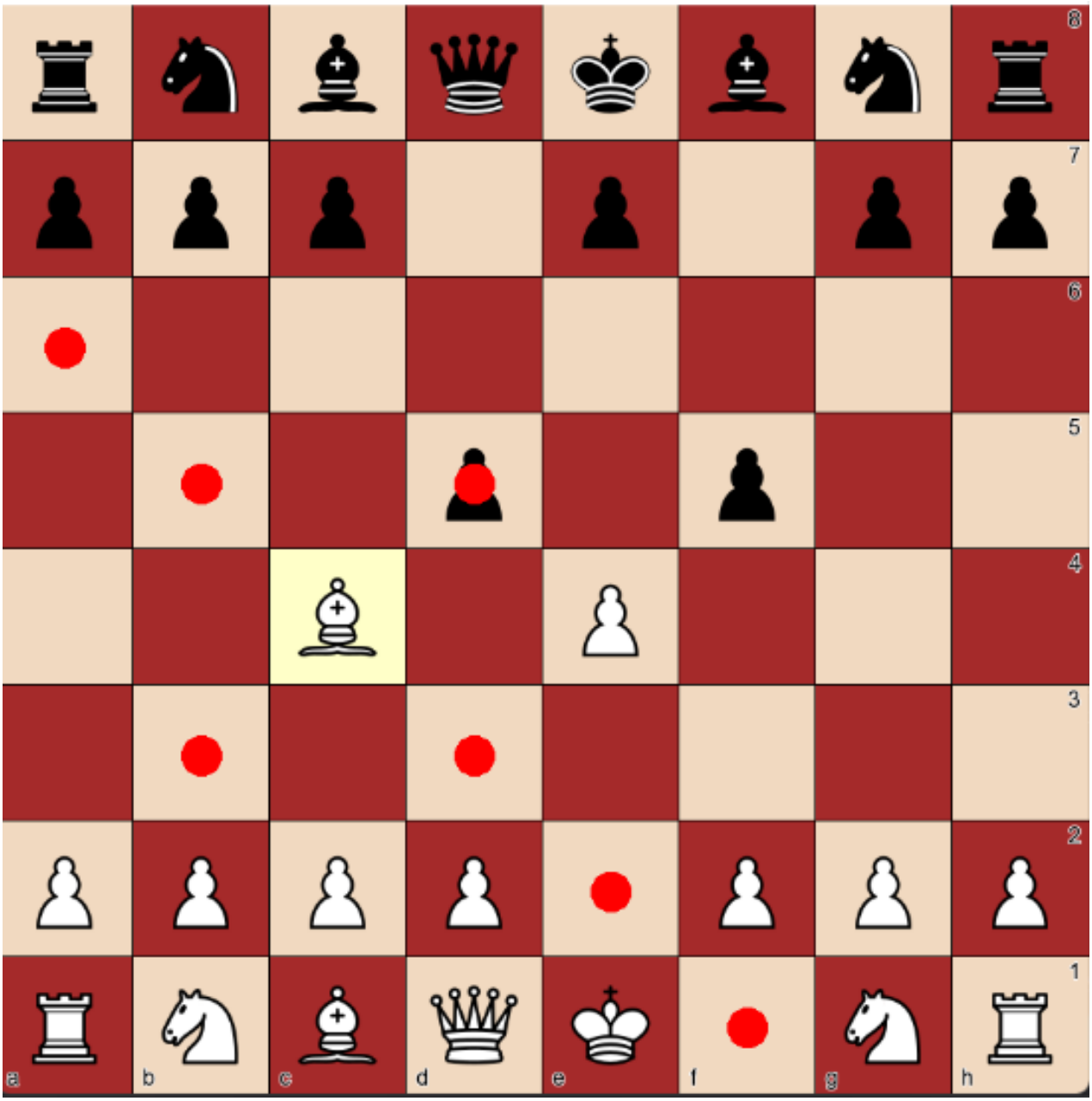
מאת ירדן בצרי

מצב תחילת משחק:



מאת ירדן בצרי

מצב לחיצת כלי:



- כאשר כלי יילחץ יופיעו כל המהלכים האפשריים של אותו הכלי.

מאת ירדן בצרי

מצב עצירת משחק (על ידי מקש ה ESC):

❖ במצב עצירת משחק תהיינה האפשרות לאתחל את המשחק, להמשיך את המשחק ולצאת ממנו.

מאת ירדן בצרי

מצב ניצחון לבן:

- ניתן לראות שהלבן ניצח כי השחור ב"שח" ואין לו לאן לזוז ואין שום כלי אחר שיכול למנוע את ה"שח".

מאת ירדן בצרי

מצב ניצחון שחור:

❖ ניתן לראות שהשחור ניצח כי הלבן ב"שח" ואין לו לאן לזוז ואין שום כלי אחר שיכול למנוע את ה"שח".

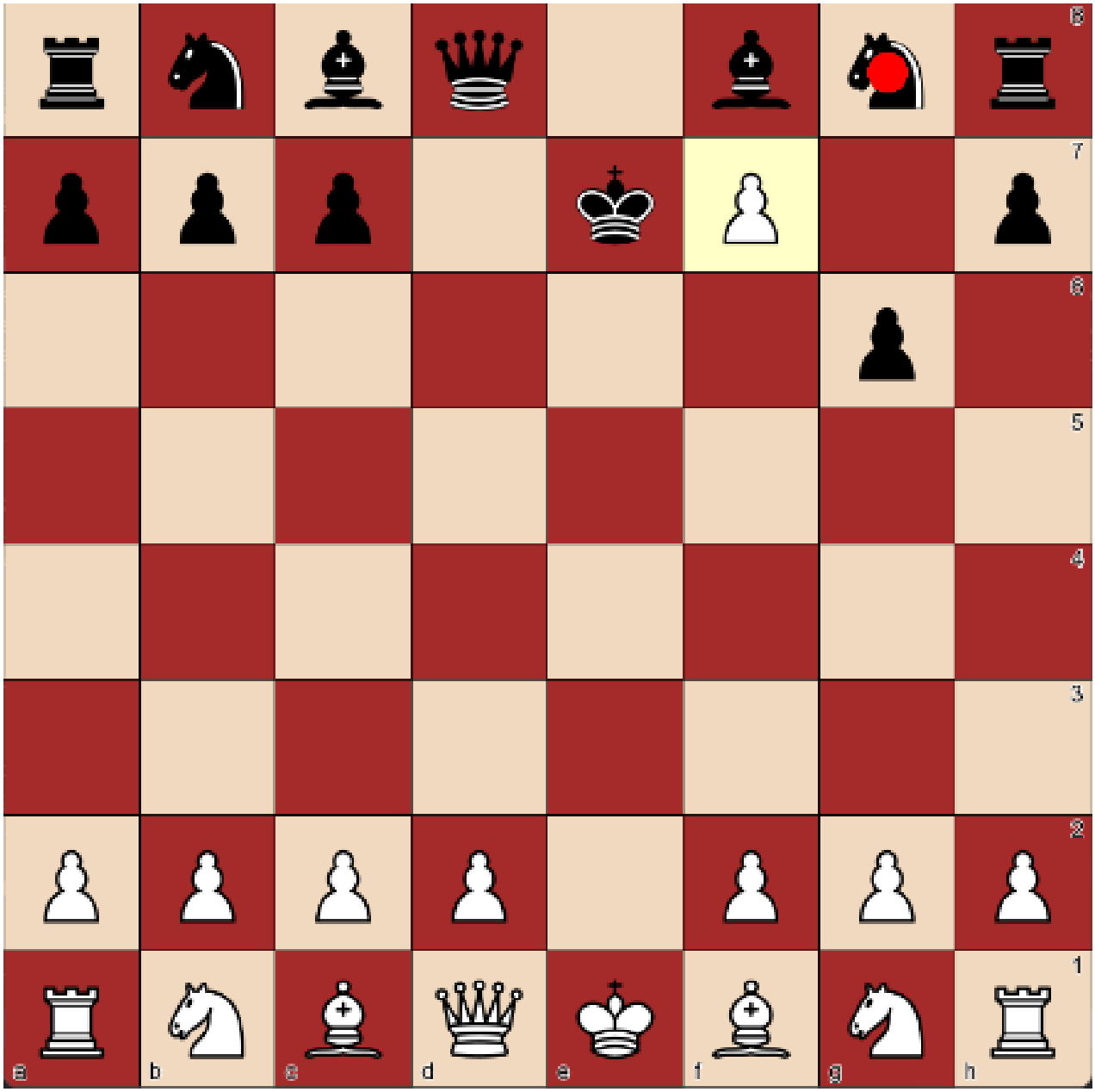
מאת ירדן בצרי

מצב תיקו:

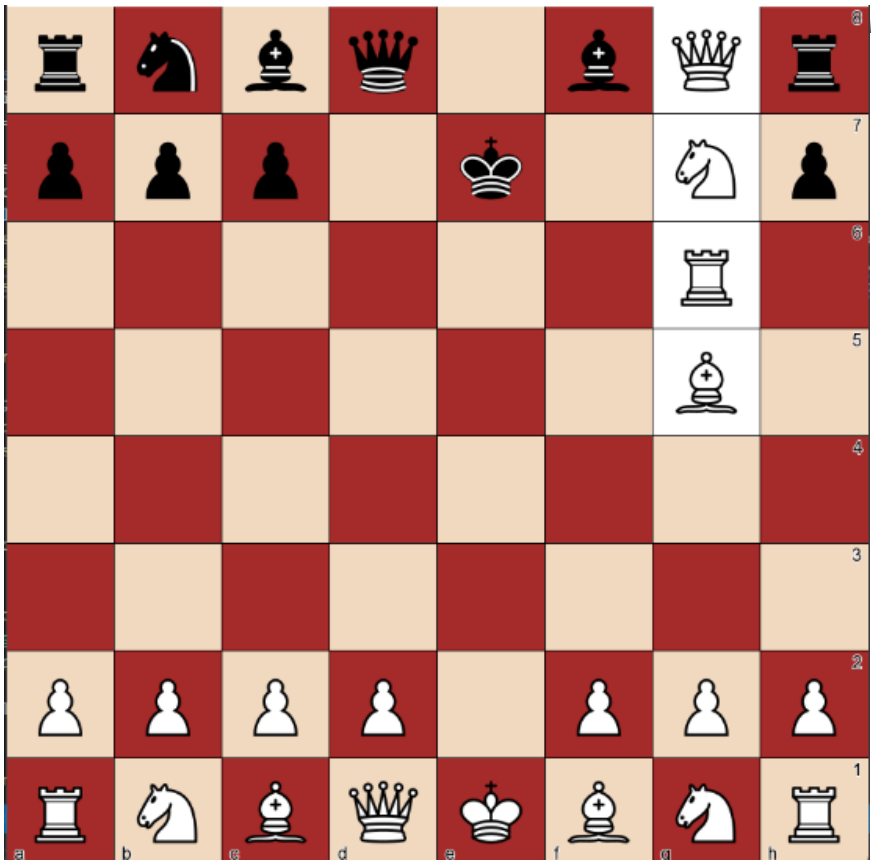
❖ ניתן לראות שיש תיקו בגלל שהמהלך הבא הוא של השחור והשחור לא יכול לעשות אף מהלך.

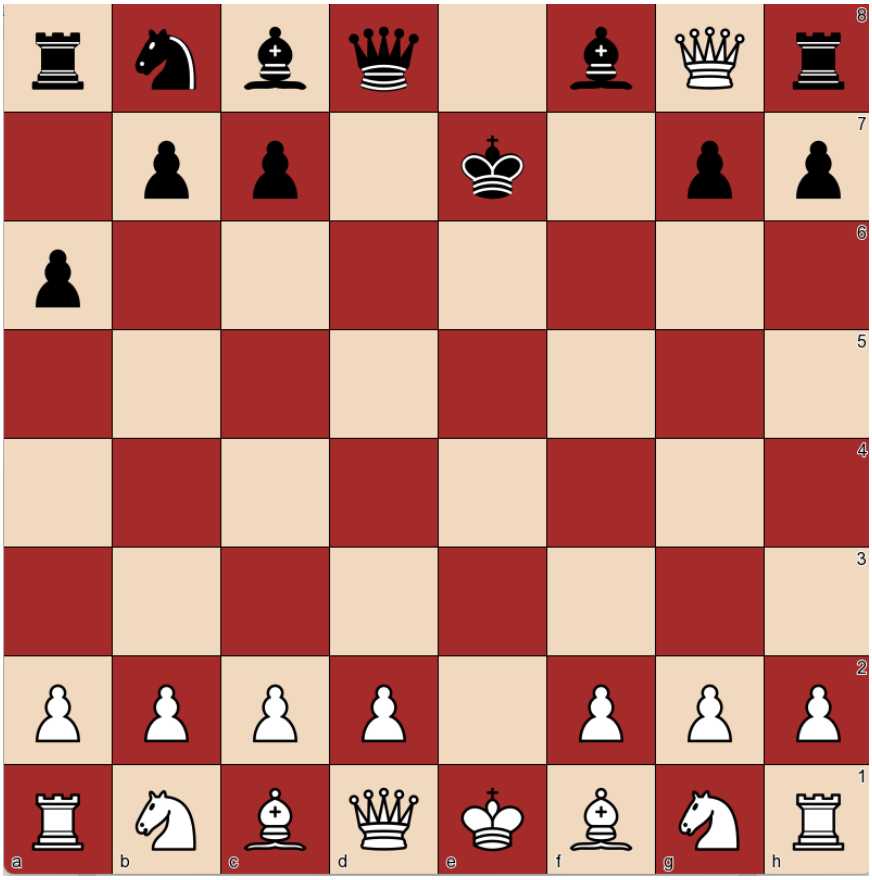
מאת ירדן בצרי

מצב קידום:



❖ ניתן לראות שבמצב זה הלבן עומד לעשות מלכה בעזרת הפיון.





❖ ניתן לראות שלאחר הקידום יש אפשרות לבחור בקידום הרצוי ושהמשתמש בחר לקדם למלכה.

רפלקציה:

התהליך של יצירת הפרויקט היה מאתגר ומעניין מאוד עבורי.

אני פיתחתי במהלך החודשים האחרונים את הידע שלי ואת היכולות האוטודידקטיות שלי לחקור הרבה יותר וללמוד לבד.

אני מאוד נהניתי לכתוב את הפרויקט.

בחרתי "ללכת" על שפה שפחות התנסיתי בה עד היום ונאלצתי ללמוד ולחקור הרבה על תכנות בשפה והכירות עם ספריית std ו sfml.

הייתי צריך לממש אלגוריתמים לא פשוטים כמו מציאת המהלכים החוקיים, התייחסות לכל המהלכים המיוחדים וכמובן שמימוש רקורסיבי לבינה המלאכותית שממומשת על ידי minimax עם גיזום alpha-beta.

אני מאוד מרוצה וגאה בעצמי על התוצאה הסופית, אני מרגיש שהשקעתי רבות בפרויקט במהלך שנת הלימודים במטרה שיעזור לי לפתח את הידע שלי ולצבור עוד ניסיון.

בהמשך אני מקווה שאקבל את ההזדמנות להציג את הפרויקט בראיונות בצבא.

בחירת שפה וסביבת עבודה

בחרתי לעבוד בשפה C++.

- ❖ לפני פרויקט זה לא כתבתי שום פרויקט בשפת התכנות C++ והחלטתי שזו הזדמנות בשבילי להתנסות בשפה נוספת.
- ❖ הייתרון היה בכך שבלימודינו במכללה כתבנו בשפת C ולכן את הבסיס כבר רכשתי מכיוון ששפת C++ היא בעצם שדרוג של שפת C כך שהחידוש העיקרי של C++ הוא תכנון מונחה עצמים (OOP).

השתמשתי בספריית SFML.

- ❖ ספרייה זו היא המובילה בעולם למתכנתים פרטיים בC++.
- ❖ הספרייה מאפשרת להשתמש ב5 קטגוריות שונות:
  - Window
  - System
  - Network
  - Graphics
  - Audio

בחרתי להשתמש בסביבת העבודה VISUAL STUDIO.

- ❖ סביבה זו היא מאוד נוחה וידידותית למתכנת.
  היא מאפשרת "לדבג" את הפרויקט בצורה נוחה ותומכת בהעלאת הפרויקט לGitHub.

תהליך העבודה ואלגוריתמים מרכזיים

הקדמה

- בפרק זה אסביר על תהליך העבודה שלי בפרויקט ואפרט על פעולות עיקריות בפרויקט באופן תמציתי.
- בתור התחלה על מנת להתנסות עם שפת התכנות C++ וספריית SFML החלטתי לבנות פרויקט קטן של איקס עיגול עם אלגוריתם של Minimax שאותו התכוונתי לממש על השחמט.
- ניתן לראות קישור של הפרויקט בGithub: yardenbasri · GitHub.
- בתהליך למדתי את העקרונות של הספריות SFML וSTD וצברתי מספיק ידע בשביל להתחיל לעבוד על הפרויקט של השחמט.

יצירת הלוח

מחלקת Game היא המחלקה הראשונה שפיתחתי.

- ראשית כתבתי את הפונקציות הבסיסיות של המחלקה:
  Game()
  void Start()
  void LoadData()
  void LoadSprites()
  Void UpdateEvents()
  void CreateBoard()

- לאחר מכן בניתי את הפונקציות שימומשו בצורה שונה עבור כל סוג של משחק כמו משחק "אופליין", משחק נגד Ai, משחק אונליין ועוד. פונקציות כאלה נקראות פונקציות וירטואליות ולהלן ההכרזה שלהם:

  void virtual Print() = 0
  void virtual Update() = 0
  void virtual Init() = 0

- האתחול לאפס אומר שהפונקציות ימומשו במחלקה / מחלקות שיירשו ממחלקת Game.

- יצירת לוח המשחק הייתה על ידי אובייקט sf::rectangleShape של מחלקת SFML:

```
sf::RectangleShape rectangle(sf::Vector2f(DELTA_POS, DELTA_POS));
int mod = 0;
for (int i = 0; i < DIMENSIONS; i++)
{
        for (int j = 0; j < DIMENSIONS; j++)
        {
                rectangle.setPosition((DELTA_POS)*j, (DELTA_POS)*i);
                if (j % 2 == mod)
                {
                        rectangle.setFillColor(lightSqure);
                        rectangle.setOutlineThickness(1);
rectangle.setOutlineColor(sf::Color::Black);
                }
                else
                {
                        rectangle.setFillColor(darkSqure);
                        rectangle.setOutlineThickness(1);
rectangle.setOutlineColor(sf::Color::Black);
                }
                squres.push_back(rectangle);
        }
        mod = !mod;
}
```

- ראשית יצרתי וקטור של ריבועים שאליו יתווספו ריבועי המשחק.
- ניתן לראות שעברתי על גבולות הלוח (8X8) ובהתאם למיקום הריבוע צבעתי את הריבוע בצבע אחר והוספתי אותו לוקטור.

- על מנת לטעון את הכלים ללוח המשחק בצורה נוחה הגדרתי מטריצה 8X8 שמייצגת את המצב ההתחלתי בלוח:

```
std::string boardStatus[DIMENSIONS][DIMENSIONS] =
        {
                {"bR", "bN", "bB", "bQ", "bK", "bB", "bN", "bR"},
                {"bp", "bp", "bp", "bp", "bp", "bp", "bp", "bp"},
                {"--", "--", "--", "--", "--", "--", "--", "--"},
                {"--", "--", "--", "--", "--", "--", "--", "--"},
                {"--", "--", "--", "--", "--", "--", "--", "--"},
                {"--", "--", "--", "--", "--", "--", "--", "--"},
                {"wp", "wp", "wp", "wp", "wp", "wp", "wp", "wp"},
                {"wR", "wN", "wB", "wQ", "wK", "wB", "wN", "wR"}
         };
```

- בנוסף הגדרתי את המחלקה כלי בשביל לטעון את הכלים ללוח, להלן הבנאי של המחלקה כלי:

```
Piece(sf::Texture& pieceTex, sf::Vector2f position,
    int isWhite, sf::Vector2i indexes, std::string type);
```

- על מנת לייחד כל כלי בצורה נפרדת יצרתי 6 מחלקות נוספות המייחדות את הכלים Rook, Bishop, Knight, King, Queen, Pown מחלקות אלה יורשות ממחלקת Piece.

- הגדרתי את מטריצת הכלים במחלקת Game:

```
Piece* pieces[DIMENSIONS][DIMENSIONS];
```

- איתחול המטריצה בפעולה LoadSprites():

```
for (int i = 0; i < DIMENSIONS; i++)
	{
		for (int j = 0; j < DIMENSIONS; j++)
		{
			std::string p = boardStatus[i][j];
			int isWhite = (p[0] == 'w');
			switch (p[1])
			{
			case 'p':
				pieces[i][j] = new Pown(piecesTex[p],
Indexes2Position(sf::Vector2i(i, j)), isWhite, sf::Vector2i(i, j), p);
				break;
			case 'K':
				pieces[i][j] = new King(piecesTex[p],
Indexes2Position(sf::Vector2i(i, j)), isWhite, sf::Vector2i(i, j), p);

				break;
			case 'Q':
				pieces[i][j] = new Queen(piecesTex[p],
Indexes2Position(sf::Vector2i(i, j)), isWhite, sf::Vector2i(i, j), p);
				break;
			case 'R':
				pieces[i][j] = new Rock(piecesTex[p],
Indexes2Position(sf::Vector2i(i, j)), isWhite, sf::Vector2i(i,j), p);
				break;
			case 'B':
				pieces[i][j] = new Bishop(piecesTex[p],
Indexes2Position(sf::Vector2i(i, j)), isWhite, sf::Vector2i(i, j), p);
				break;
			case 'N':
				pieces[i][j] = new Knight(piecesTex[p],
Indexes2Position(sf::Vector2i(i, j)), isWhite, sf::Vector2i(i, j), p);
				break;

			default:
				pieces[i][j] = NULL;
				break;
			}

		}
	}
```

- ניתן לראות שבאמצעות boardStatus איתחלתי את כל הלוח במצב התחלתי.
- בעזרת דרך זו יהיה ניתן לאתחל את הלוח בכל פוזיציה שנרצה.

מציאת מהלכים חוקיים

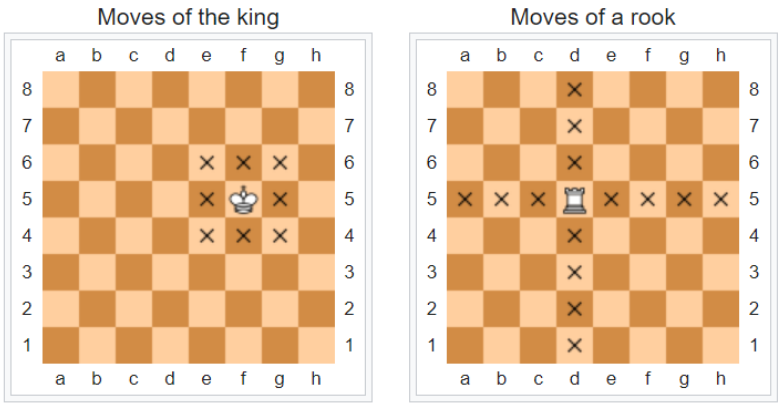
- בשביל לדאוג שהמשחק יתקיים על פי כל חוקי השחמט יש לממש בכל מחלקה של כלי ספציפי פעולה שבודקת מהלכים אפשריים של כלי.
- נגדיר במחלקה Piece את הפעולה הוירטואלית:

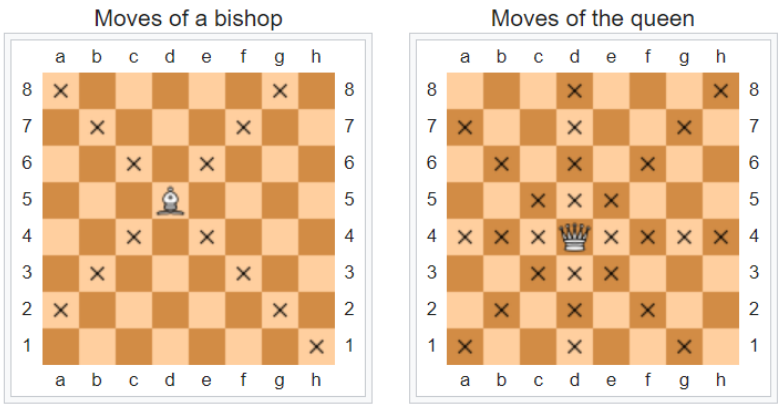
```
virtual std::vector<sf::Vector2i> GetPossibleMoves(Piece*
                                pieces[DIMENSIONS][DIMENSIONS]) ;
```

- עבור כל כלי שונה הפעולה תמומש באופן שונה.
- בשביל לשמור על כתיבה גנרית נגדיר פעולה נוספת במחלקה Piece:

```
void UpdatePossibleMoves(std::vector<sf::Vector2i> *possibleMoves, int
    xDir, int yDir, Piece* pieces[DIMENSIONS][DIMENSIONS], int limit);
```

- באמצעות פעולה זו נוכל למצוא את המהלכים האפשריים של המלכה, רץ, צריח ומלך.
- להלן תזכורת כיצד חיילים אלה זזים:
  - המלך – יכול לזוז צעד אחד לכל כיוון.
  - המלכה – יכולה לזוז לכל כיוון כמה צעדים שרוצה.
  - רץ – יכול לזוז באלכסון כמה צעדים שרוצה.
  - צריח – יכול לזוז מאונך ומאוזן כמה צעדים שרוצה.





- הפעולה תקבל וקטור שאליו תעדכן את המהלכים האפשריים, כיוון X,Y שממנו תבדוק כמות צעדים חוקית קדימה, את הכלים, ומגבלה במקרה ויש לבדוק מספר צעדים מוגבל קדימה, כמו במלך שיש לבדוק צעד אחד קדימה.

```
void Piece::UpdatePossibleMoves(std::vector<sf::Vector2i> *possibleMoves, int
xDir, int yDir, Piece* pieces[DIMENSIONS][DIMENSIONS], int limit)
{
	sf::Vector2i temp = indexes;
	temp.x += xDir;
	temp.y += yDir;
	while (InBoard(temp) && limit)
	{
		if (!pieces[temp.x][temp.y])
			possibleMoves->push_back(temp);
		else
		{
			if (OppositeColors(temp, pieces))
				possibleMoves->push_back(temp);
			break;
		}
		temp.x += xDir;
		temp.y += yDir;
		limit--;
	}
}
```

- הפונקציה תלך בכיוון שנתנו לה limit צעדים קדימה עד שהיא נתקלת בכלי מסוים. אם הכלי הוא של אותו השחקן אז היא תעצור לפני הכלי ואם הכלי הוא של השחקן היריב היא תעצור במיקום של הכלי.

להלן דוגמא לקריאה לפונקציה על ידי הפוקציה GetPossibleMove מהמחלקה של המלכה:

```
UpdatePossibleMoves(&possibleMoves, 1, 0, pieces, DIMENSIONS);
UpdatePossibleMoves(&possibleMoves, -1, 0, pieces, DIMENSIONS);
UpdatePossibleMoves(&possibleMoves, 0, 1, pieces, DIMENSIONS);
UpdatePossibleMoves(&possibleMoves, 0, -1, pieces, DIMENSIONS);
UpdatePossibleMoves(&possibleMoves, 1, 1, pieces, DIMENSIONS);
UpdatePossibleMoves(&possibleMoves, 1, -1, pieces, DIMENSIONS);
UpdatePossibleMoves(&possibleMoves, -1, 1, pieces, DIMENSIONS);
UpdatePossibleMoves(&possibleMoves, -1, -1, pieces, DIMENSIONS);
```

להלן קריאה נוספת מהמחלקה של המלך:

```
UpdatePossibleMoves(&possibleMoves, 1, 0, pieces, 1);
UpdatePossibleMoves(&possibleMoves, -1, 0, pieces, 1);
UpdatePossibleMoves(&possibleMoves, 0, 1, pieces, 1);
UpdatePossibleMoves(&possibleMoves, 0, -1, pieces, 1);
UpdatePossibleMoves(&possibleMoves, 1, -1, pieces, 1);
UpdatePossibleMoves(&possibleMoves, 1, 1, pieces, 1);
UpdatePossibleMoves(&possibleMoves, -1, -1, pieces, 1);
UpdatePossibleMoves(&possibleMoves, -1, 1, pieces, 1);
```

- על מימוש המהלכים האפשריים של הפיון והפרש אסביר בחלק מתקדם יותר של הספר, כאמור פרק זה עוסק בתהליך העבודה באופן תמציתי.
- לאחר מימוש המהלכים האפשריים של כל כלי יש צורך לממש את המהלכים החוקיים של כל כלי.
- אז מה ההבדל בעצם ?
  - מהלך אפשרי יכול להיות לא חוקי בשלושה מקרים:
  - הכלי מאוים על ידי כלי אחר כך שאם הוא יזוז המלך יחשף (Pin). במצב זה הכלי לא יכול לזוז לשום מקום.
  - מצב שח על ידי כלי אחד. במצב זה הכלי יכול לזוז אך ורק במידה ואינו בPin והוא יכול לחסום את ה "שח" או במידה ויכול לאכול את הכלי שמאיים בשח.
  - מצב שח על ידי שני כלים (double check). במצב זה רק המלך יכול לזוז והוא יכול לזוז רק למקום שבו הוא לא יהיה בשח יותר.

- למימוש המהלכים החוקיים כתבתי שני אלגוריתמים, האלגוריתם הראשון הוא אלגוריתם נאיבי שכתבתי במחלקת Game ואלגוריתם נוסף הוא אלגוריתם שנכתב במחלקת Piece שהוא אלגוריתם מורכב יותר אשר משפר את זמן הריצה.

- להלן מימוש האלגוריתם הראשון:

```
std::vector<sf::Vector2i> moves = piece->GetPossibleMoves(pieces);
	sf::Vector2i indexesBefore = piece->GetIndexes();
	sf::Vector2i indexesAfter;


	for (int i = moves.size() - 1; i >= 0; i--)
	{
		// Do the move.
		indexesAfter = moves[i];
		Move(indexesBefore, indexesAfter);

		if (InCheck())
			moves.erase(moves.begin() + i);
		UndoMove(indexesBefore, indexesAfter);
	}


	return moves;
```

- תיאור האלגוריתם:
  - מציאת כל המהלכים האפשריים של כלי מסוים והשמתם בוקטור המהלכים החוקיים.
  - עבור כל אחד מהמהלכים האפשריים:
    - עשה את המהלך.
    - בדיקה האם יש "שח".
      - אם כן: תמחק את המהלך מוקטור המהלכים החוקיים.
    - תחזיר את המהלך.

אלגוריתם זה הוא מאוד נאיבי וקל לכתיבה אך יגרום לזמן ריצה ארוך, דבר שלא יהיה טוב עבור הבינה המלאכותית, שנרצה שתעבוד בזמן הריצה המהיר ביותר.

- להלן מימוש האלגוריתם המורכב יותר:

```
sf::Vector2i kingIndexes;
	if (isWhite)
		kingIndexes = kings.front();
	else
		kingIndexes = kings.back();
	std::vector<sf::Vector2i> possibleMoves = GetPossibleMoves(pieces);
	if (kingIndexes == this->indexes)
	{
		// if the piece is the king than return from all the possible
squres to move, the squres that not under attack.
		Piece* kingPiece = pieces[kingIndexes.x][kingIndexes.y];
		pieces[kingIndexes.x][kingIndexes.y] = nullptr;
		for (int i = possibleMoves.size() - 1; i >= 0; i--)
		{
			if(squreUnderAttack(pieces, possibleMoves[i]))
				possibleMoves.erase(possibleMoves.begin() + i);

		}
		pieces[kingIndexes.x][kingIndexes.y] = kingPiece;
		return possibleMoves;
	}
	int count = squreUnderAttack(pieces, kingIndexes);
	// from now we check for other pieces than the king:
	if (count == 2)// in a case of double check only the king can move.
	{
		pinSqures.clear();
		attackSqures.clear();
		return std::vector<sf::Vector2i>();
	}
	else
	{
		if (count == 1)
		{
			// if the king is in check and the piece is not the king,
the piece can only move in order to:
			// 1. block the check
			// 2. capture the attacking
			// the piece can only move if it is not a pin.
			sf::Vector2i squre = attackSqures.back();
			attackSqures.pop_back();

			if (pieces[squre.x][squre.y]->GetType()[1] == 'N')
			{
				// if the attacking piece is knight than the piece
can only move to capture the attacking piece (only if the piece is not pinned).
				for (int i = possibleMoves.size() - 1; i >= 0; i--)
				{
					if (possibleMoves[i] != squre ||
IsMovePinned(possibleMoves[i], kingIndexes, pieces)) // check if the possible
move is not a capture.

	possibleMoves.erase(possibleMoves.begin() + i);
				}

			}
			else
			{
```

```
                            std::vector<sf::Vector2i> squresBetween =
GetSquresBetween(kingIndexes, squre);
                            for (int i = possibleMoves.size() - 1; i >= 0; i--)
                            {
                                    if (!IsMovePinned(possibleMoves[i],
kingIndexes, pieces) && possibleMoves[i] == squre) // check if the possible
move is capture.
                                            continue;
                                    if (std::find(squresBetween.begin(),
squresBetween.end(), possibleMoves[i]) == squresBetween.end())

       possibleMoves.erase(possibleMoves.begin() + i);
                            }

                    }

                    pinSqures.clear();
                    return possibleMoves;
            }
            else
            {
                    // if the king is not in check than the move is valid
unless it pinned by an enemy piece..
                    for (int j = possibleMoves.size() - 1; j >= 0; j--)
                    {
                            if(IsMovePinned(possibleMoves[j], kingIndexes,
pieces))
                                    possibleMoves.erase(possibleMoves.begin() +
j);
                    }
                    return possibleMoves;
            }
                                                            }
```

מאת ירדן בצרי

- תיאור האלגוריתם:
- מצא את כל המהלכים האפשריים של כלי.
- האם הכלי מלך ?
- אם כן: עבור כל המהלכים האפשריים של המלך:
  - בדוק האם המהלך האפשרי תחת איום.
  - אם כן, מחק אותו מוקטור המהלכים האפשריים.
  - החזר את וקטור המהלכים החוקיים.
- אחרת:
- בדוק האם יש במשחק מצב double check:
- אם כן: החזר וקטור ריק
- אחרת:
- בדוק האם יש מצב שח על ידי כלי אחד:
- אם כן:
  - עבור כל אחד מהמהלכים האפשריים בדוק:
  - אם המהלך Pinned או שהמהלך לא "יאכל" את התוקף או שהמהלך לא חוסם את ה"שח":
    - מחק את המהלך מוקטור המהלכים החוקיים.
- אחרת:
- עבור כל אחד מהמלכים החוקיים בדוק:
- האם המהלך Pinned
- אם כן:
  - מחק את המהלך מוקטור המהלכים החוקיים.
- החזר את וקטור המהלכים החוקיים.

ניהול לחיצות העכבר ותזוזת הכלים

- על מנת לנהל באופן חכם את לחיצות העכבר בניתי פעולה מרכזית שתדע מתי להזיז את הכלים ומתי לא.
- `void Game::HandleClickes(sf::Vector2i indexes);`
- בלחיצה על העכבר הפונקציה תקרא ותקבל את האינדקסים של משבצת הלוח שנלחצה על ידי המשתמש.
- להלן המימוש:

```
void Game::HandleClickes(sf::Vector2i indexes)
{
	if (indexes.x != -1 && pieces[indexes.x][indexes.y] ||
playerClickes.size()) // check if the first click was not an empty squre
		if (sqSelected == indexes && playerClickes.size() == 1) // check
if the same squre is pressed twice.
		{
			if(pieces[indexes.x][indexes.y]->IsWhite())
				cleanSqure(sqSelected);
			else
				cleanSqure(sqSelected);
			possibleCircles.clear();
			sqSelected = { -1,-1 };
			playerClickes.clear();
		}
		else
		{
			sqSelected = indexes;
			playerClickes.push_back(sqSelected);
			MarkSqure(sqSelected, markedSqure);
			if (playerClickes.size() == 1 &&
PlayerTurn(playerClickes[0])) {
				legalMoves =
GetLegalMoves(pieces[playerClickes[0].x][playerClickes[0].y]);
				possibleCircles = GetPossibleCircles(legalMoves);
			}

			if (playerClickes.size() == 2) // check if two different
squres has been pressed.
			{
				auto it = std::find(legalMoves.begin(),
legalMoves.end(), playerClickes[1]);
				int legalMove = (it != legalMoves.end());
				if (PlayerTurn(playerClickes[0]) && legalMove)
				{
					lastType =
pieces[playerClickes[0].x][playerClickes[0].y]->GetType();
					Move(playerClickes[0], playerClickes[1]);
					UpdatesAfterMove();
				}
				else
				{
					if (whiteTurn) {
						cleanSqure(playerClickes[0]);
```

```
                        cleanSqure(playerClickes[1]);
                    }
                    else
                    {
                        cleanSqure(playerClickes[0]);
                        cleanSqure(playerClickes[1]);
                    }

                    possibleCircles.clear();
                    playerClickes.clear();
                }

            }
        }

    }
```

- הכנות:
- בשביל לממש את האלגוריתם ניצור וקטור שישמור את הערכים של לחיצות העכבר וניצור משתנה שיכיל את הריבוע האחרון שנלחץ.

```
sf::Vector2i sqSelected; // save the indexes of the last selected squre.

std::vector<sf::Vector2i> playerClickes; // a vector that saves the player
indexes of the squres he has clicked.
```

- בפונקציה זו נקרא לפונקציה Move רק כאשר שני ריבועים שונים נלחצו כאשר הריבוע הראשון הוא של הכלי של השחקן עם אותו התור והריבוע השני הוא ריבוע ריק או ריבוע של היריב.
- שתי פונקציות חשובות מאוד בפרויקט שלי הם הפעולות Move ו UndoMove.
- על מנת שפונקציות אלה יוכלו להיקרא גם באופן איטרטיבי וגם באופן רקורסיבי (על ידי האלגוריתם של הבינה המלאכותית), שמרתי בוקטורים אשר התחייסתי אליהם כמחסניות ערכים שונים כמו המהלך שבוצע, סוג המהלך (מהלך רגיל, הצרחה, קידום פיון..) והכלי ש"נאכל" על מנת שתוכל להיות גישה אליהם כאשר מחזירים מהלך.
- הצורך במחסנית נובע מכך הפונקציות Move וUndoMove יזומנו בפונקציה רקורסיבית כך שהפונקציה Move תקרא בהתחלה, לאחר מכן תתבצע הקריאה הרקורסיבית ולאחר מכן תזומן הפונקציה Undo.

- מימוש הפעולה Move:

```
void Game::Move(sf::Vector2i indexesBefore, sf::Vector2i indexesAfter)
{
	lastTypeVec.push_back(pieces[indexesBefore.x][indexesBefore.y]-
>GetType());
	lastMoveVec.push_back(std::pair<sf::Vector2i,
sf::Vector2i>(indexesBefore, indexesAfter));
	for (int i = 0; i < DIMENSIONS; i++)
	{
		for (int j = 0; j < DIMENSIONS; j++)
		{
			if (pieces[i][j]) pieces[i][j]-
>SetLastMove(lastMoveVec.back());
		}
	}
	pieceMovedVec.push_back(pieces[indexesBefore.x][indexesBefore.y]-
>PieceMoved());
	Piece* pieceCaptured = pieces[indexesAfter.x][indexesAfter.y];
	capturedSqure = pieces[indexesAfter.x][indexesAfter.y];
	// check for special moves :
	if (pieces[indexesBefore.x][indexesBefore.y]->GetType() == "bp" &&
indexesAfter.x == 7)  // check for bp promotion.
	{
		Promotion(indexesBefore, indexesAfter, false);
		moveTypeVec.push_back(PROMOTION);
	}
	else if (pieces[indexesBefore.x][indexesBefore.y]->GetType() == "wp" &&
indexesAfter.x == 0)  // check for wp promotion.
	{
		Promotion(indexesBefore, indexesAfter, true);
		moveTypeVec.push_back(PROMOTION);
	}
	else if (pieces[indexesBefore.x][indexesBefore.y]->GetType() == "wp" &&
		indexesBefore.x == 3 && indexesAfter.y != indexesBefore.y &&
!pieces[indexesAfter.x][indexesAfter.y]) // check white en passent
	{
		pieceCaptured = pieces[indexesAfter.x + 1][indexesAfter.y];
		EnPassent(true, indexesBefore, indexesAfter);
		moveTypeVec.push_back(WHITE_ENPASSENT);
	}
	else if (pieces[indexesBefore.x][indexesBefore.y]->GetType() == "bp" &&
indexesBefore.x == 4 &&
		indexesAfter.y != indexesBefore.y &&
!pieces[indexesAfter.x][indexesAfter.y])  // check black en passent
	{
		pieceCaptured = pieces[indexesAfter.x - 1][indexesAfter.y];
		EnPassent(false, indexesBefore, indexesAfter);
		moveTypeVec.push_back(BLACK_ENPASSENT);
	}
	else if (pieces[indexesBefore.x][indexesBefore.y]->GetType()[1] == 'K'
&& abs(indexesAfter.y-indexesBefore.y) == 2 ) // check castling
	{
		Castling(indexesBefore, indexesAfter);

	}
	else // A regular move :
	{
		RegularMove(indexesBefore, indexesAfter);
		moveTypeVec.push_back(REGULARMOVE);
	}
	capturedPieces.push_back(pieceCaptured);}
```

- כפי שניתן לראות הפונקציה תטפל בכל המצבים האפשריים ותקרא לפונקציות הזזה שונות בהתאם למהלך שבוצע על ידי המשתמש:

להלן המצבים:

- קידום פיון לבן
- קידום פיון שחור
- UnPassent לבן
- UnPassent שחור
- הצרחה
- מהלך רגיל

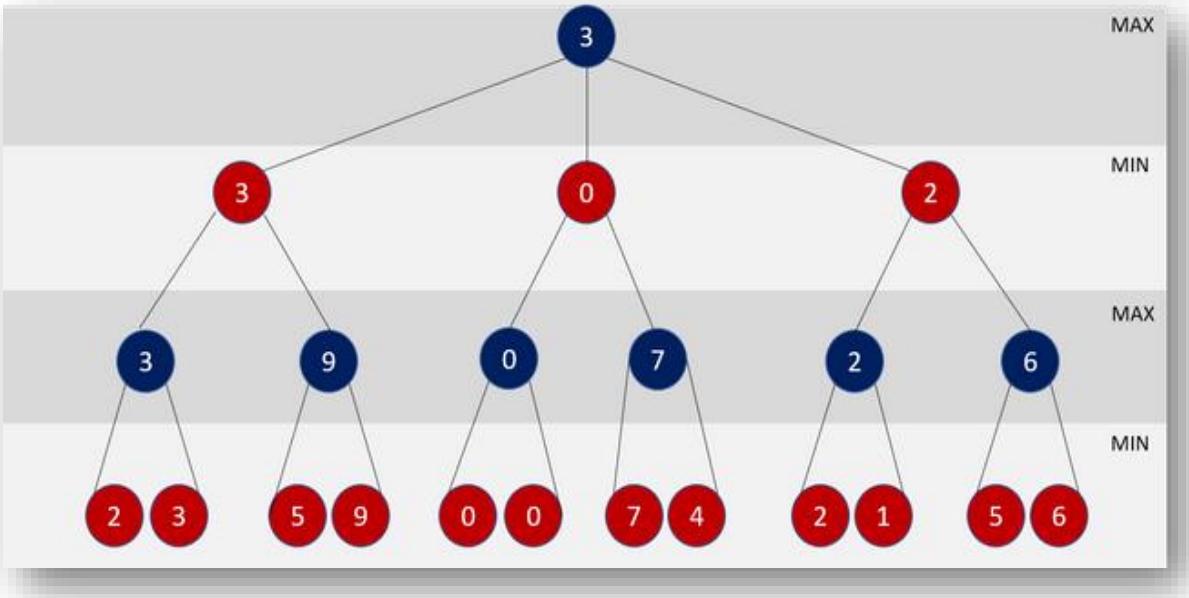
- מימוש UndoMove:

```
lastMoveVec.pop_back();
	for (int i = 0; i < DIMENSIONS; i++)
	{
		for (int j = 0; j < DIMENSIONS; j++)
		{
			if (pieces[i][j]) pieces[i][j]-
>SetLastMove(lastMoveVec.back());
		}
	}
	RegularMove(indexesAfter, indexesBefore);
	pieces[indexesBefore.x][indexesBefore.y]-
>setPieceMoved(pieceMovedVec.back());
	pieceMovedVec.pop_back();
	auto t = moveTypeVec.back();
	switch (t)
	{
	case(moveTypes::PROMOTION):
	{
		std::string type = lastTypeVec[lastTypeVec.size() - 1];
		pieces[indexesBefore.x][indexesBefore.y] = new
Pown(piecesTex[type], // create a new pown
			Indexes2Position(sf::Vector2i(indexesBefore.x,
indexesBefore.y)),
			type[0] == 'w', sf::Vector2i(indexesBefore.x,
indexesBefore.y), type);
		break;
	}
	case(moveTypes::WHITE_ENPASSENT):
	{
		pieces[indexesAfter.x + 1][indexesAfter.y] =
capturedPieces[capturedPieces.size() - 1];
		break;
	}
	case(moveTypes::BLACK_ENPASSENT):
	{
```

```
                pieces[indexesAfter.x - 1][indexesAfter.y] =
capturedPieces[capturedPieces.size() - 1];
                break;
        }
        case(moveTypes::KING_CASTLING):
        {
                auto before = sf::Vector2i(indexesAfter.x, indexesAfter.y - 1);
                auto after = sf::Vector2i(indexesAfter.x, indexesAfter.y + 1);
                RegularMove(before, after);
                pieces[after.x][after.y]->setPieceMoved(false);
                break;
        }
        case(moveTypes::QUEEN_CASTLING):
        {
                auto before = sf::Vector2i(indexesAfter.x, indexesAfter.y + 1);
                auto after = sf::Vector2i(indexesAfter.x, indexesAfter.y - 2);
                RegularMove(before, after);
                pieces[after.x][after.y]->setPieceMoved(false);
                break;
        }

        }

        if(t != moveTypes::WHITE_ENPASSENT && t != moveTypes::BLACK_ENPASSENT)
                pieces[indexesAfter.x][indexesAfter.y] = capturedPieces.back();
        capturedPieces.pop_back();
        moveTypeVec.pop_back();
                                        lastTypeVec.pop_back();
```

מימוש הבינה המלאכותית

- לצורך מימוש הבינה המלאכותית השתמשתי באלגוריתם "מיני-מקס".
- מינמקס הוא אלגוריתם שמתאים למשחק בין שני אנשים לסירוגין עם סט קבוע של חוקים.
- האלגוריתם הוא רקורסיבי ואפשר להסתכל עליו כמעין עץ דמיוני הפורש את האפשרויות למשחק של שחקן א', את התגובות של שחקן ב' לכל פעולה של שחקן א', את התגובות של א' כתוצאה מכל אחת מהתגובות של שחקן ב' וכך הלאה.
- לכל מהלך נגדיר ניקוד על ידי חישוב של כמות הכלים של השחור לעומת כמות הכלים של הלבן.
- העלים בעץ שנוצר הם מצבים סטטיים שנגיע אליהם לאחר רצף של מהלכים (הנתיבים בעץ הם למעשה תרחישים אפשריים). ניתן ציון לכל מצב סטטי שכזה (מצב סטטי בלוח שחמט לדוגמה), שישקף כמה המצב טוב מבחינתנו.
- נסתכל לצורך העניין על העץ מיני-מקס הבא:



- נגדיר שככל שהניקוד בעץ גבוה יותר כך התוצאה טובה יותר עבורי וככל שהתוצאה נמוכה יותר היא פחות טובה עבורי, מה שהופך אותה להיות טובה עבור היריב שלי. לכן אני ארצה "למקסם" את התוצאה ויריבי ירצה "למזער" את התוצאה (הסיבה שאלגוריתם זה נקרא "מיני-מקס").
- ניתן לראות על פי העץ הרקורסיבי בתמונה שראש העץ שהוא כמובן המהלך שנבחר הוא לא המצב הסטטי הטוב ביותר בלוח עבורו.

- הסיבה לכך היא שאנחנו מניחים שהיריב מבצע את המהלך הטוב ביותר עבורו, לאחר קבלת העלים בעץ ניקח את המהלך עם הניקוד המקסימלי מבין כל המהלכים עם הניקוד המינימלי.

- המימוש מתחיל מפונקציה של ניתוח המצב הסטטי, לכן נגדיר את הפונקציה:

```
int AiGame::evaluate();
```

- על מנת לנתח את המצב הסטטי כמו שצריך נקבע לכל כלי ערך כלשהו על פי חשיבות הכלי. לכן נגדיר את המילון:

```
std::map<char, int> scores;
```

- אתחול המילון:

```
scores['p'] = 100;
scores['B'] = 300;
scores['N'] = 320;
scores['R'] = 500;
scores['Q'] = 900;
```

להלן המימוש:

```
int metirial = 0, mul;

	for (int i = 0; i < DIMENSIONS; i++)
	{
		for (int j = 0; j < DIMENSIONS; j++)
		{
			if (!pieces[i][j]) continue;
			mul = pieces[i][j]->IsWhite() ? -1 : 1;
			char type = pieces[i][j]->GetType()[1]; // the letter of the
piece.
			metirial += mul * scores[type];
		}
}
return metirial;
```

- הפונקציה תעבור על כל הכלים בלוח ותחזיר את כמות הערכים של כליי השחור כחיוביים ועוד כמות הערכים של כליי הלבן כשליליים.

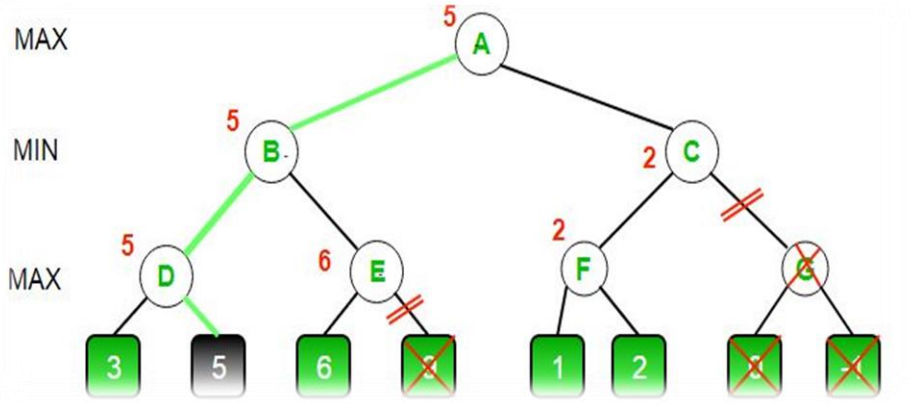
- לאחר מכן נממש את אלגוריתם המיני-מקס:

```
int AiGame::minimax(int depth, int isMax)
{
	if (depth == 0)
		return evaluate();
	std::vector<std::pair<sf::Vector2i, sf::Vector2i>> moves =
AllLegalMoves(!isMax);
	if (isMax)
	{
		int bestEval = -INFINITY;
		if (moves.size() == 0) {
			if (InCheck())
				return bestEval;
			return 0;
		}
		for (auto move : moves)
		{
			// do the move :
			Move(move.first, move.second);
			int eval = minimax(depth - 1, !isMax);
			if (eval > bestEval)
				bestEval = eval;
			// undo the move :
			UndoMove(move.first, move.second);
		}
		return bestEval;
	}
	else
	{
		int bestEval = +INFINITY;
		if (moves.size() == 0) {
			if (InCheck())
				return bestEval;
			return 0;
		}
		for (auto move : moves)
		{
			// do the move :
			Move(move.first, move.second);
			int eval = minimax(depth - 1, !isMax);
			if (eval < bestEval)
				bestEval = eval;
			// undo the move :
			UndoMove(move.first, move.second);
		}
		return bestEval;
	}
	return 0;
}
```

- להלן שלבי האלגוריתם:
- בדיקה האם העומק הגיע לאפס, אם כן החזר את הניתוח הסטטי.
- מציאת כל המהלכים החוקיים של השחקן בעל התור.
- בדיקה האם שחקן ממקסם:
- אם כן:
  - הגדרת משתנה bestEval המייצג את הניתוח הסטטי הטוב ביותר והשמתו כמינוס אינסוף.
  - בדיקה האם אין מהלכים חוקיים
  - אם כן:
    - האם השחקן ב"שח" (מצב שחמט), אם כן החזר מינוס אינסוף.
    - אחרת (מצב תיקו) , החזר אפס.
  - עבור כל אחד המהלכים החוקיים:
    - עשה את המהלך
    - טען לתור משתנה המייצג מצב סטטי את הערך שהפונקציה תחזיר באופן רקורסיבי כאשר התור עובר לשחקן השני והשכבה יורדת באחד.
    - בדיקה האם המצב הסטטי החדש גדול יותר מbestEval אם כן מעדכנים את המשתנה למצב הסטטי החדש.
    - החזר את המהלך.
  - החזר את bestEval.
- אחרת:
  - הגדרת משתנה bestEval המייצג את הניתוח הסטטי הטוב ביותר והשמתו כפלוס אינסוף.
  - בדיקה האם אין מהלכים חוקיים
  - אם כן:
    - האם השחקן ב"שח" (מצב שחמט), אם כן החזר פלוס אינסוף.
    - אחרת (מצב תיקו) , החזר אפס.
  - עבור כל אחד המהלכים החוקיים:
    - עשה את המהלך
    - טען לתור משתנה המייצג מצב סטטי את הערך שהפונקציה תחזיר באופן רקורסיבי כאשר התור עובר לשחקן השני והשכבה יורדת באחד.

- בדיקה האם המצב הסטטי החדש קטן יותר מbestEval אם כן מעדכנים את המשתנה למצב הסטטי החדש.
- החזר את המהלך.
  - החזר את bestEval.

- בעיה קשה בבניית עץ מינימקס היא הזיכרון הרב שהוא צורך. מספר הקודקודים שיש לפתח עולה בטור הנדסי ככל שנעמיק את החיפוש. קיימת שיטת "גיזום" המבטלת בנייה של ענפים שברור לנו עוד בשלב מוקדם כי הם לא מועילים לחיפוש שלנו. ניתן כך להקטין את מספר הצמתים בעץ לשורש המספר שהיה מתקבל ללא הגיזום (בממוצע). שיטה זו נקראת גיזום אלפא-ביתא.
- ניקח לדוגמא את עץ המינימקס הבא:



- נתחיל מהרכבת העץ מהשכבה התחתונה:
בקודקוד D נבחר את הערך המקסימלי לכן נבחר בחמש, מכאן אנו יודעים שקודקוד B יהיה עם ערך של 5 ומטה בגלל שהיריב רוצה למזער את התוצאה, בקוקוד E אנו רואים כי האיבר הראשון הוא 6 בגלל שבשכבה זו נרצה למקסם את התוצאה אז קודקוד E יהיה גדול או שווה ל6, מכאן ניתן לטעון שאין צורך לבדוק ענפים נוספים בקודקוד זה בגלל שקודקוד B יהיה קטן או שווה ל5 לכן יהיה ניתן לבצע "גיזום".

- להלן מימוש אלגוריתם המיני-מקס עם גיזום האלפא בטא:

```
if (depth == 0) {
                int a = evaluate();
                //std::cout << a << std::endl;
                return a;
        }
        //std::vector<std::pair<sf::Vector2i, sf::Vector2i>> moves =
AllLegalMoves(!isMax);
        std::vector<std::pair<sf::Vector2i, sf::Vector2i>> moves =
OrderMoves(AllLegalMoves(!isMax));


        if (isMax)
        {
                int bestEval = -INFINITY;
                if (moves.size() == 0) {
                        if (InCheck())
                                return bestEval;
                        return 0;
                }
                for (auto move : moves)
                {
                        // do the move :
                        Move(move.first, move.second);
                        whiteTurn = !whiteTurn;
                        int eval = minimax(depth - 1, !isMax, alpha, beta);
                        whiteTurn = !whiteTurn;
                        bestEval = std::max(bestEval, eval);
                        alpha = std::max(alpha, eval);
                        if (beta <= alpha) {
                                // undo the move :
                                UndoMove(move.first, move.second);
                                /*printf("%d\n", a++);*/
                                break;
                        }

                        // undo the move :
                        UndoMove(move.first, move.second);
                }
                return bestEval;
        }
        else
        {
                int bestEval = +INFINITY;
                if (moves.size() == 0) {
                        if (InCheck())
                                return bestEval;
                        return 0;
                }
                for (auto move : moves)
                {
                        // do the move :
                        Move(move.first, move.second);
                        whiteTurn = !whiteTurn;
                        int eval = minimax(depth - 1, !isMax, alpha, beta);
                        whiteTurn = !whiteTurn;
                        bestEval = std::min(bestEval, eval);
                        beta = std::min(beta, eval);
                        if (beta <= alpha) {
                                // undo the move :
                                UndoMove(move.first, move.second);
```

```
                        /*printf("%d\n", a++);*/
                        break;
                }
                // undo the move :
                UndoMove(move.first, move.second);
        }
        return bestEval;
}
                                                return 0;
```

גיזום העץ מבוצע באופן הבא:

- ✓ גיזום אלפא (Alpha cut)

אם הקודקוד הוא קודקוד של שחקן ה'מקס' - זאת אומרת, זהו תורו של 'מקס' לשחק - השחקן לא יבחר תתי-עצים בעלי תוצאה נמוכה יותר מזו שהושגה בתת-עץ קודם. הרציונל לכך הוא פשוט. זהו תורו של 'מקס', 'מקס' תמיד בוחר למקסם. בשל כך, ברור כי אם 'מקס' כבר יודע על מהלך בעל ניקוד מסוים, אם באחד התתי עצים שלו שחקן ה'מין' הצליח להשיג ניקוד נמוך יותר על ידי אחד התתי עצים שלו עצמו, שחקן ה'מין' לא צריך לבחון את שאר תתי העצים שלו משום שהוא לא יבחר ניקוד גבוה יותר ממה שהשיג כעת ולכן שחקן ה'מקס' (שהוא אב קדמון שלו) לא יבחר בניקוד המגיע ממנו כי יש לו כבר ניקוד גבוה יותר.

- ✓ גיזום ביתא (Beta cut)

אם הקודקוד הוא קודקוד של שחקן ה'מין' - ננקוט בפעולה ההפוכה.
האלגוריתם מחזיק שני משתני עזר, אלפא (α) וביתא (β), המייצגים את התוצאה המינימלית המובטחת לשחקן ה'מקס', ואת התוצאה המקסימלית המובטחת לשחקן ה'מין', בהתאמה.
בתחילה, נקבעים משתנים אלו להיות בעלי ערכי קיצון שרירותיים - אלפא למינוס אינסוף, וביתא לפלוס אינסוף. ערכים תחיליים אלו, מייצגים את המצב הגרוע ביותר מבחינתו של כל שחקן - לשחקן הממקסם מובטחת תוצאה גרועה ביותר (מינוס אינסוף), ולשחקן הממזער גם כן תוצאה גרועה ביותר לפי השקפתו (פלוס אינסוף). עם התקדמות החיפוש, הולך ומצטמצם המרחק בין אלפא לביתא. כאשר ביתא מקבלת ערך נמוך מאלפא, פירושו של דבר כי העמדה הנוכחית אינה תוצאה של משחק מושלם על ידי שני השחקנים (ועל-כן, לא תשוחק על ידי אחד מהם, שיכול לשפר תוך שהוא משחק באופן מושלם), ואין צורך להעמיק חקר בתת-העץ הנוכחי.

אופטימיזציה

- בשביל למנוע שגיאות וקריסות של הבינה המלאכותית מימשתי פונקציה שתקבל מספר שכבות קדימה ותחשב עבור הפוזיציה בלוח n שכבות קדימה עבור כל מהלך אפשרי בדומה למנוע stockfish עם הרצת הפקודה perft ומספר n.

להלן המימוש:

```
if (depth == 0)
return 1;
auto moves = AllLegalMoves(isWhite);
//moves = OrderMoves(moves);
int numPositions = 0;
int numPositionsEachMove = 0;
std::string str;
for (auto move : moves)
{
/*if (depth == startDepth) {
if(TranslateMove(move) == )
}*/
Move(move.first, move.second);
whiteTurn = !whiteTurn;
numPositionsEachMove = MoveGenerarionTest(depth - 1, !isWhite,
startDepth);
whiteTurn = !whiteTurn;
numPositions += numPositionsEachMove;
str = TranslateMove(move);
if (depth == startDepth) {
// check if the move was promotion and calc for each option.
if (moveTypeVec.back() == PROMOTION) {
char t = pieces[move.second.x][move.second.y]->GetType()[1];
str.push_back(t);
}
moveCounter[str] = numPositionsEachMove;
std::cout << str << " : " << numPositionsEachMove << std::endl;
}
if (moveTypeVec.back() == PROMOTION)
{
UndoMove(move.first, move.second);
for (int i = 0; i < 3; i++)
{
promotionTypeVec.push_back(i + 1);
Move(move.first, move.second);
whiteTurn = !whiteTurn;
numPositionsEachMove = MoveGenerarionTest(depth - 1, !isWhite,
startDepth);
whiteTurn = !whiteTurn;
numPositions += numPositionsEachMove;
if (depth == startDepth) {
// check if the move was promotion and calc for each option.
str = TranslateMove(move);
if (moveTypeVec.back() == PROMOTION) {
char t = pieces[move.second.x][move.second.y]->GetType()[1];
str.push_back(t);
}
moveCounter[str] = numPositionsEachMove;
std::cout << str << " : " << numPositionsEachMove << std::endl;
```

```
}
UndoMove(move.first, move.second);
promotionTypeVec.pop_back();
}

}
else
UndoMove(move.first, move.second);


}
return numPositions;
```

- כעת אציג השוואה בין התוצאות של המנוע שלי למנוע של stockfish 4 שכבות קדימה:

- עבור פוזיציית תחילת משחק:

Me:

C:\Users\yarde\source\repos\Chess\x64\Debug\Chess.exe

```
a2a3 : 8457
a2a4 : 9329
b2b3 : 9345
b2b4 : 9332
c2c3 : 9272
c2c4 : 9744
d2d3 : 11959
d2d4 : 12435
e2e3 : 13134
e2e4 : 13160
f2f3 : 8457
f2f4 : 8929
g2g3 : 9345
g2g4 : 9328
h2h3 : 8457
h2h4 : 9329
b1c3 : 9755
b1a3 : 8885
g1h3 : 8881
g1f3 : 9748
-------------------197281---------------
```

Stockfish:

C:\Users\yarde\Downloads\stockfish_14.1_win_x64

```
go perft 4
a2a3: 8457
b2b3: 9345
c2c3: 9272
d2d3: 11959
e2e3: 13134
f2f3: 8457
g2g3: 9345
h2h3: 8457
a2a4: 9329
b2b4: 9332
c2c4: 9744
d2d4: 12435
e2e4: 13160
f2f4: 8929
g2g4: 9328
h2h4: 9329
b1a3: 8885
b1c3: 9755
g1f3: 9748
g1h3: 8881

Nodes searched: 197281
```

- עבור הפוזיציה הבאה שהפילה מנועי שחמט רבים בשכבה 3:

מאת ירדן בצרי

Me:

C:\Users\yarde\source\repos\Chess\x64\Debug\Chess.exe

```
d7c8R : 38077
d7c8B : 65053
c4d3 : 43565
c4d5 : 48002
c4e6 : 49872
c4f7 : 43289
c4b3 : 43453
c4b5 : 45559
c4a6 : 41884
a2a3 : 46833
a2a4 : 48882
b2b3 : 46497
b2b4 : 46696
c2c3 : 49406
e2f4 : 51127
e2d4 : 52109
e2g1 : 48844
e2g3 : 51892
e2c3 : 54792
g2g3 : 44509
g2g4 : 45506
h2h3 : 46762
h2h4 : 47811
b1c3 : 50303
b1a3 : 44378
b1d2 : 40560
c1d2 : 46881
c1e3 : 53637
c1f4 : 52350
c1g5 : 45601
c1h6 : 40913
d1d2 : 48843
d1d3 : 57153
d1d4 : 57744
d1d5 : 56899
d1d6 : 43766
e1f1 : 49775
e1d2 : 33423
e1f2 : 36783
e1g1 : 47054
h1g1 : 44668
h1f1 : 46101
-------------------2103487----------------
```

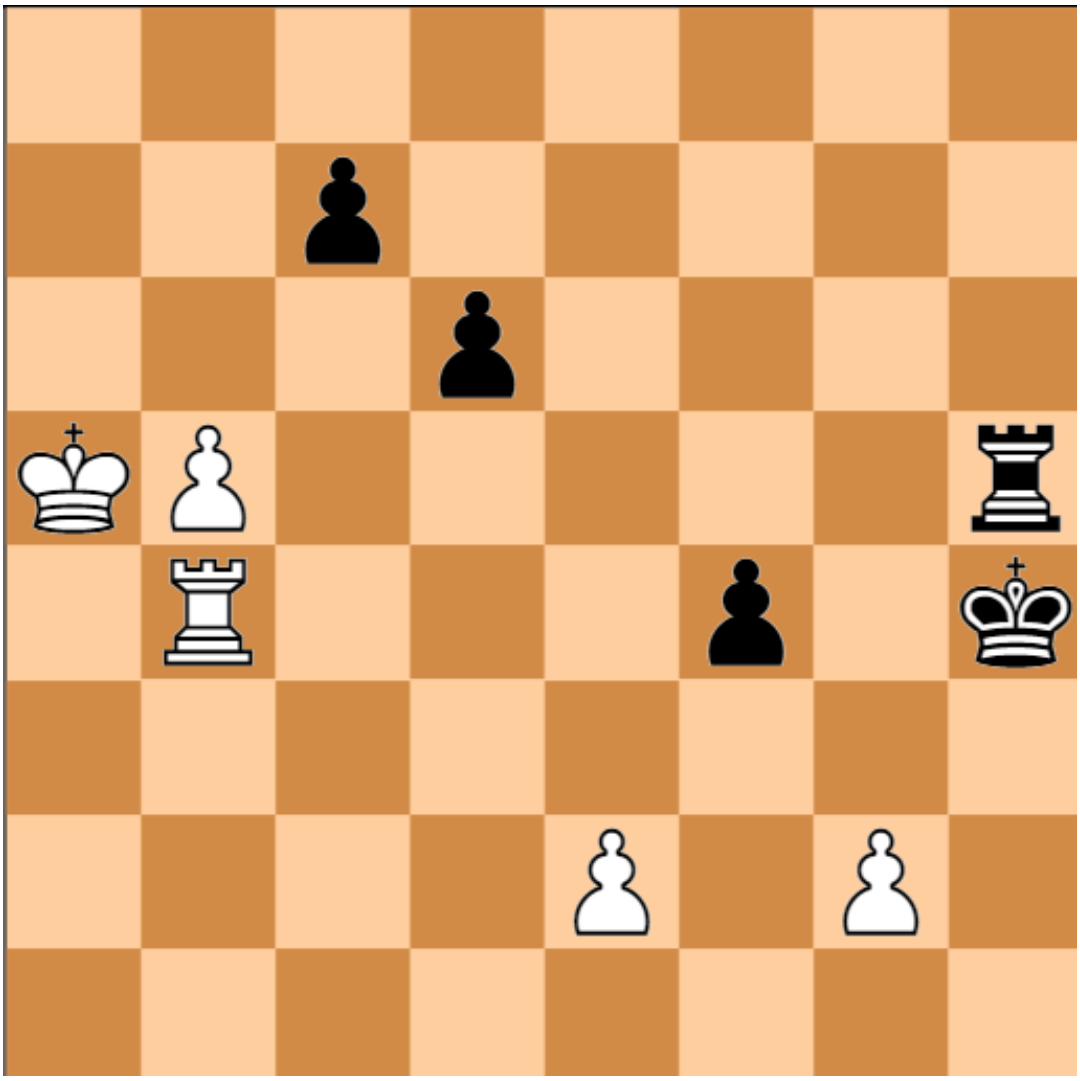
STOCKFISH:

C:\Users\yarde\Downloads\stockfish_14.1_win_x64_avx2\stockfish_1

```
go perft 4
a2a3: 46833
b2b3: 46497
c2c3: 49406
g2g3: 44509
h2h3: 46762
a2a4: 48882
b2b4: 46696
g2g4: 45506
h2h4: 47811
d7c8q: 44226
d7c8r: 38077
d7c8b: 65053
d7c8n: 62009
b1d2: 40560
b1a3: 44378
b1c3: 50303
e2g1: 48844
e2c3: 54792
e2g3: 51892
e2d4: 52109
e2f4: 51127
c1d2: 46881
c1e3: 53637
c1f4: 52350
c1g5: 45601
c1h6: 40913
c4b3: 43453
c4d3: 43565
c4b5: 45559
c4d5: 48002
c4a6: 41884
c4e6: 49872
c4f7: 43289
h1f1: 46101
h1g1: 44668
d1d2: 48843
d1d3: 57153
d1d4: 57744
d1d5: 56899
d1d6: 43766
e1f1: 49775
e1d2: 33423
e1f2: 36783
e1g1: 47054

Nodes searched: 2103487
```

מאת ירדן בצרי

- עבור הפוזיציה הבאה:



מאת ירדן בצרי

Stockfish:

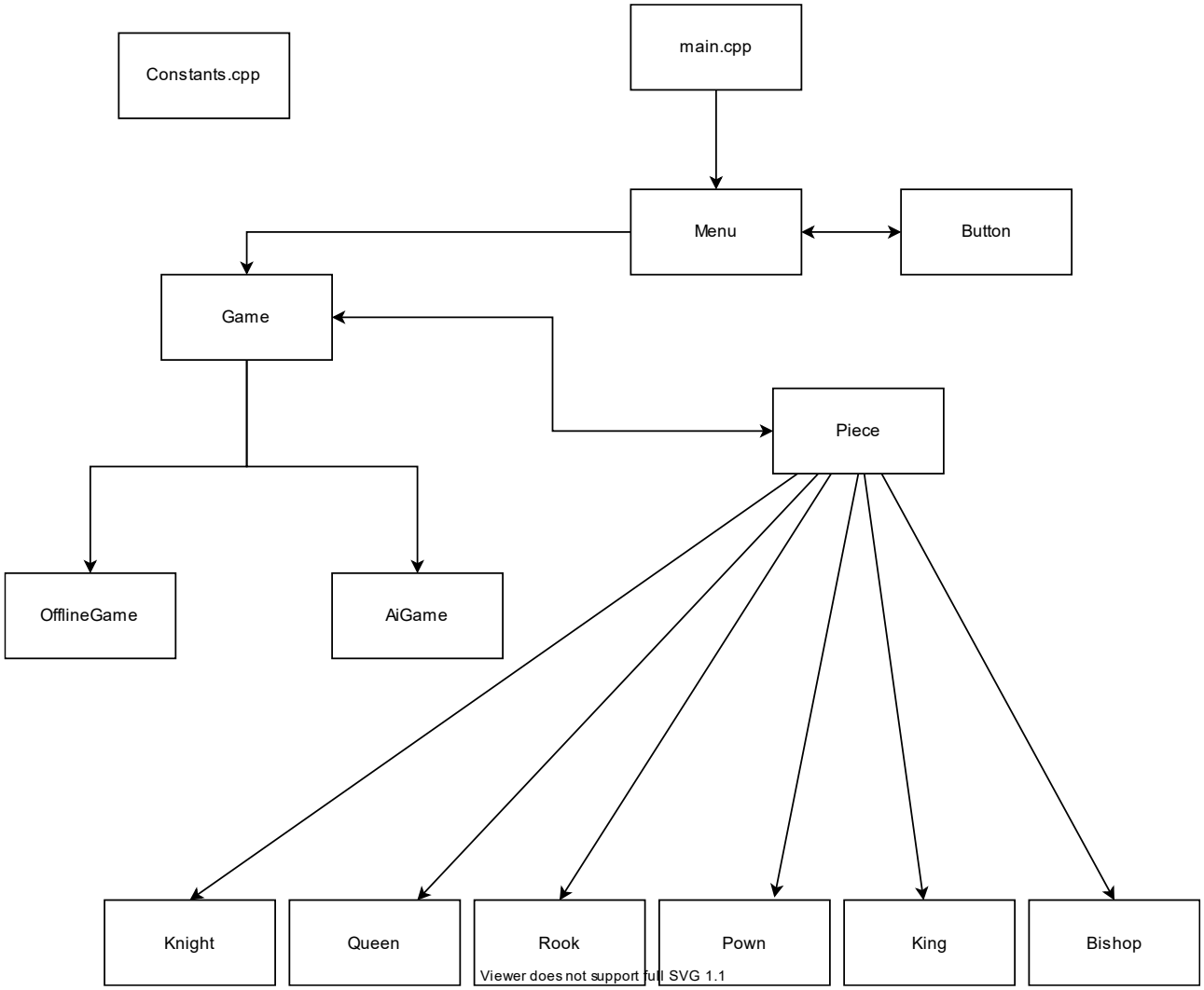
C:\Users\yarde\Downloads\stockfish_14.1_win_x64_avx2\stockfish_14.1

```
Nodes searched: 2103487

position fen 8/2p5/3p4/KP5r/1R3p1k/8/4P1P1/8 w - -
d

 +---+---+---+---+---+---+---+---+
 |   |   |   |   |   |   |   |   | 8
 +---+---+---+---+---+---+---+---+
 |   |   | p |   |   |   |   |   | 7
 +---+---+---+---+---+---+---+---+
 |   |   |   | p |   |   |   |   | 6
 +---+---+---+---+---+---+---+---+
 | K | P |   |   |   |   |   | r | 5
 +---+---+---+---+---+---+---+---+
 |   | R |   |   |   | p |   | k | 4
 +---+---+---+---+---+---+---+---+
 |   |   |   |   |   |   |   |   | 3
 +---+---+---+---+---+---+---+---+
 |   |   |   |   | P |   | P |   | 2
 +---+---+---+---+---+---+---+---+
 |   |   |   |   |   |   |   |   | 1
 +---+---+---+---+---+---+---+---+
   a   b   c   d   e   f   g   h

Fen: 8/2p5/3p4/KP5r/1R3p1k/8/4P1P1/8 w - - 0 1
Key: C5F0C6634A313C2C
Checkers:
go perft 4
e2e3: 3107
g2g3: 1014
a5a6: 3653
e2e4: 2748
g2g4: 3702
b4b1: 4199
b4b2: 3328
b4b3: 3658
b4a4: 3019
b4c4: 3797
b4d4: 3622
b4e4: 3391
b4f4: 606
a5a4: 3394

Nodes searched: 43238
```

Me:

```
a5a4 : 3394
a5a6 : 3653
b4b3 : 3658
b4b2 : 3328
b4b1 : 4199
b4c4 : 3797
b4d4 : 3622
b4e4 : 3391
b4f4 : 606
b4a4 : 3019
e2e3 : 3107
e2e4 : 2748
g2g3 : 1014
g2g4 : 3702
-------------------43238----------------
```

תרשים מחלקות וקבצים:



יצירת המחלקות וקוד הפרויקט

יצירת המחלקה BUTTON:

DECLARATION

```
#include <SFML/Graphics.hpp>
#include <iostream>
#include <functional>

class Button
{
public:
	Button();
	Button(std::string text, int charSize, sf::Vector2f size,
sf::Color color, sf::Color bgColor, sf::Vector2f position );

	void SetBackColor(sf::Color color);
	void SetTextColor(sf::Color color);

	void DrawTo(sf::RenderWindow& window);
	bool IsMouseHover(sf::RenderWindow& window);
	void HandleMouseHover(sf::RenderWindow& window);


private:
	sf::Text text;
	sf::RectangleShape button;
	sf::Font font;
	sf::Color btnColor;


};
```

IMPLEMENTATION

```
#include "Button.h"

Button::Button()
{
}

Button::Button(std::string text, int charSize, sf::Vector2f size,
sf::Color color, sf::Color bgColor, sf::Vector2f position)
{
	this->text.setString(text);
	this->text.setFillColor(sf::Color::Black);
	this->text.setCharacterSize(charSize);
	this->button.setSize(size);
	this->button.setFillColor(bgColor);
	position.x =  (position.x  - this-
>button.getLocalBounds().width/2);
	position.y = (position.y + this-
>button.getLocalBounds().height / 2);
	this->button.setPosition(position);
	float xPos = (position.x + this-
>button.getGlobalBounds().width + position.x) / 2 - this-
>text.getString().getSize() * 4.5;
	float yPos = (position.y + this-
>button.getGlobalBounds().height + position.y) / 2 - 10;
	this->text.setPosition({ xPos, yPos });
	font.loadFromFile("Assets/arial.ttf");
	this->text.setFont(font);
	this->btnColor = bgColor;
}



void Button::SetBackColor(sf::Color color)
{
	this->button.setFillColor(color);
}

void Button::SetTextColor(sf::Color color)
{
	this->text.setFillColor(color);
}



void Button::DrawTo(sf::RenderWindow& window)
{
	window.draw(button);
	window.draw(text);
}

bool Button::IsMouseHover(sf::RenderWindow& window)
```

```
{
	sf::Vector2i mousePos = sf::Mouse::getPosition(window);
	if(this-
>button.getGlobalBounds().contains(sf::Vector2f(mousePos)))
		return true;
	return false;
}

void Button::HandleMouseHover(sf::RenderWindow& window)
{
	if (IsMouseHover(window))
	{
		this->button.setFillColor(sf::Color::White);
	}
	else
		this->button.setFillColor(btnColor);


}
```

יצירת CONSTANTS.H:

```
#pragma once
#define SCREEN_SIZE 800
#define DIMENSIONS 8
#define BOARD_IMG "Assets/board.png"
#define START_POS 12 // 12 WITHOUT SCALING
#define DELTA_POS SCREEN_SIZE / DIMENSIONS
#define INFINITY 10000000


// static variables
```

יצירת המחלקה GAME:

DECLARATION

```
#include <SFML/Graphics.hpp>
#include <map>
#include <iostream>
#include "Constants.h"
#include "Piece.h"
#include "Button.h"
#include <vector>
// include all pieces:
#include "Pown.h"
#include "King.h"
#include "Queen.h"
#include "Bishop.h"
#include "Rock.h"
#include "Knight.h"

class Game
{
public:
Game();
// base functions
void Start(); // start calls the functions that need to be done only
once and than call to Init.
void LoadData(); // load all the files to there textures.
void LoadSprites(); // load the textures to there sprites.
void CreateBoard(); // init the recatngle vector correctly with 8*8
squres.
void DrawNumbersLetters(sf::RenderWindow* window);
// game functions
sf::Vector2f Indexes2Position(sf::Vector2i indexes); // gets two
indexes and return the squre position.
sf::Vector2i Position2Indexes(sf::Vector2f mPos); // gets the
position and return the indexes.
void Move(sf::Vector2i indexesBefore, sf::Vector2i indexesAfter); //
change the pointer of the piece that need to move and the sprite`s
position.
void UndoMove(sf::Vector2i indexesBefore, sf::Vector2i
indexesAfter); // The function undo a move.
void UpdatesAfterMove();
```

```
void MarkSqure(sf::Vector2i indexes, sf::Color color); // gets the
indexes of a certian squre and a color and mark it.
void cleanSqure(sf::Vector2i indexes); // gets the indexes of a
marked squre and cleans it.
void CleanBoard(); // restart the board
/* the function gets a vector of all possible moves indexes and
return a vector of circleShapes in the current indexes.*/
std::vector<sf::CircleShape>
GetPossibleCircles(std::vector<sf::Vector2i> possibleMoves);
/* the function gets the type of castling and the indexes of the
king and rock and does the castle.*/
// move functions :
void HandleClickes(sf::Vector2i indexes); // gets the indexes of the
mouse position when it was clicked and handles all the clickes.
void RegularMove(sf::Vector2i indexesBefore, sf::Vector2i
indexesAfter);
void Castling(sf::Vector2i indexesBefore, sf::Vector2i
indexesAfter);
void Promotion(sf::Vector2i indexesBefore, sf::Vector2i
indexesAfter, bool isWhite);
void EnPassent(bool isWhite, sf::Vector2i indexesBefore,
sf::Vector2i indexesAfter);
/*the function checks if the move was castling and if is was than it
checks if the castling is a legal move
if it is than return 1 else 0. if the move is not castling than
return 1.*/
int IsCastlingLegal(sf::Vector2i indexesBefore, sf::Vector2i
indexesAfter);
std::vector<sf::Vector2i> GetLegalMoves(Piece *piece);
int InBoard(sf::Vector2i indexes);
/* A function that return a vector of the indexes of all the legal
moves. the steps are:
1. generate all possible moves.
2. for each move, make the move.
3. generate all opponent possible moves.
4. for each of the opponent`s moves, see if they attack your king.
5. if they do attack your king, it is`nt a valid move.*/

int InCheck();
int squreUnderAttack(sf::Vector2i location);
std::vector<sf::Vector2i> GetKingsLocation(); // the function
returns two vector2i variables first for wk pos and second for bk
pos.
void HandleChecks(); // the function check if the kings in check and
mark them in red if they are.
void checkmateOrStalemate(); // the function check if a checkmate or
stalemate happend.
void RotatePieces(); // rotates all the pieces in the board.
std::vector<std::pair<sf::Vector2i, sf::Vector2i>>
AllLegalMoves(bool isWhite);
void UpdateEvents(); // this function checks for current events and
act accordingly.
void UpdateBoard();
```

```
std::string TranslateMove(std::pair<sf::Vector2i, sf::Vector2i>
move);
// virtual functions
void virtual Print() = 0; // this function draws to the screen all
the sprites.
void virtual Update() = 0; // this function updates the game
situation.
void virtual Init() = 0; // game loop.
bool virtual PlayerTurn(sf::Vector2i indexes) = 0;
protected:
// window
sf::RenderWindow* window;
// textures
sf::Texture boardTex;
std::map<std::string, sf::Texture> piecesTex;
// fonts
sf::Font font;
// texts
sf::Text text_board;
sf::Text text_msg;
// sprites
sf::Sprite board;
std::vector<sf::RectangleShape> squres;
std::vector<sf::Vector2i> legalMoves;
// squres
Piece* pieces[DIMENSIONS][DIMENSIONS]; // A pointer's matrix that
represent the squres.
// colors
sf::Color darkSqure;
sf::Color lightSqure;
sf::Color markedSqure;
// bool variables
bool mReleased; // A bool that represent if mouse released.
int whiteTurn; // 1 if it is white move , 0 if it is black move.
int checkmate;
int stalemate;
int gameOver;
bool inCheck;
bool isUndoPossible;
bool gamePaused;
// game vars
sf::Vector2i sqSelected; // save the indexes of the last selected
squre.
std::vector<sf::Vector2i> markedSqures; // A vector that represents
all the squres that need to be marked.
std::vector<Button*> buttons;
//std::vector<sf::Vector2i> lastMove; // the last move that has been
played.
std::vector<std::pair<sf::Vector2i, sf::Vector2i>> lastMoveVec;
std::vector<bool> pieceMovedVec;
enum moveTypes { REGULARMOVE, PROMOTION, WHITE_ENPASSENT,
BLACK_ENPASSENT, KING_CASTLING, QUEEN_CASTLING };
enum promotionTypes {QUEEN, KNIGHT, ROCK, BISHOP};
// promotion :
```

```
std::vector<int> promotionTypeVec;
int promtionChoice;
std::vector<sf::Vector2i> promotionIndexes;
std::vector<Piece*> promotionPieces;
std::vector<Piece*> promotionPiecesReplace;
int doOnce = 1;
//
std::vector<int> moveTypeVec;
std::string lastType = "--";
std::vector<std::string> lastTypeVec;
std::vector<Piece*> capturedPieces;
Piece* capturedSqure = nullptr;
int flag;
std::vector<sf::Vector2i> playerClickes; // a vector that saves the
player indexes of the squres he has clicked.
std::vector<sf::CircleShape> possibleCircles; // A vector of all the
circles shapes that need to be draw.
//position 1
std::string boardStatus[DIMENSIONS][DIMENSIONS] =
{
{"bR", "bN", "bB", "bQ", "bK", "bB", "bN", "bR"},
{"bp", "bp", "bp", "bp", "bp", "bp", "bp", "bp"},
{"--", "--", "--", "--", "--", "--", "--", "--"},
{"--", "--", "--", "--", "--", "--", "--", "--"},
{"--", "--", "--", "--", "--", "--", "--", "--"},
{"--", "--", "--", "--", "--", "--", "--", "--"},
{"wp", "wp", "wp", "wp", "wp", "wp", "wp", "wp"},
{"wR", "wN", "wB", "wQ", "wK", "wB", "wN", "wR"}
};

};
```

```
Game::Game()
{
	window = new sf::RenderWindow(sf::VideoMode(SCREEN_SIZE,
SCREEN_SIZE), "Yarden`s Chess");
	isUndoPossible = false;
	darkSqure = sf::Color(165, 42, 42); //(160,82,45);
	lightSqure = sf::Color(241, 217, 192);
	markedSqure = sf::Color(255, 255, 200);
	lastMoveVec.push_back(std::pair<sf::Vector2i, sf::Vector2i>({
-1,-1 }, { -1, -1 }));
	auto btnColor = sf::Color(200, 157, 124);
	buttons.push_back(new Button("Restart", 20, { SCREEN_SIZE / 2,
SCREEN_SIZE / 8 }, sf::Color::Black, btnColor, { SCREEN_SIZE * 0.5,
SCREEN_SIZE * 0.1 }));
	buttons.push_back(new Button("Resume", 20, { SCREEN_SIZE / 2,
SCREEN_SIZE / 8 }, sf::Color::Black, btnColor, { SCREEN_SIZE * 0.5,
SCREEN_SIZE * 0.4 }));
	buttons.push_back(new Button("Exit", 20, { SCREEN_SIZE / 2,
SCREEN_SIZE / 8 }, sf::Color::Black, btnColor, { SCREEN_SIZE * 0.5,
SCREEN_SIZE * 0.7 }));
}

void Game::Start()
{
	LoadData();
	CreateBoard();
	LoadSprites();
	Init();
}

void Game::LoadData()
{

	boardTex.loadFromFile(BOARD_IMG);
	std::string pieces[] = { "bR", "bN", "bB", "bK",
"bQ","bp","wR", "wN", "wB", "wK", "wQ","wp" };
	for (auto piece : pieces)
	{
		piecesTex[piece].loadFromFile("Assets/pieces/" + piece +
".png");
	}

	font.loadFromFile("Assets/arial.ttf");
	// letters and numbers text
	text_board.setFont(font);
	text_board.setCharacterSize(16);
	text_board.setFillColor(sf::Color::Black);
	text_board.setOutlineThickness(1);
	text_board.setOutlineColor(sf::Color::White);
	// message text
```

```
	text_msg.setFont(font);
	text_msg.setOutlineThickness(2);
	text_msg.setOutlineColor(sf::Color::Black);
	text_msg.setCharacterSize(90);
	text_msg.setFillColor(sf::Color(200, 157, 124));
	text_msg.setPosition(100, 350);
	// clear vectors
}


void Game::CreateBoard()
{
	sf::RectangleShape rectangle(sf::Vector2f(DELTA_POS,
DELTA_POS));



	int mod = 0;
	for (int i = 0; i < DIMENSIONS; i++)
	{
		for (int j = 0; j < DIMENSIONS; j++)
		{
			rectangle.setPosition((DELTA_POS)*j,
(DELTA_POS)*i);
			if (j % 2 == mod)
			{
				rectangle.setFillColor(lightSqure);
				rectangle.setOutlineThickness(1);
				rectangle.setOutlineColor(sf::Color::Black);
			}
			else
			{
				rectangle.setFillColor(darkSqure);
				rectangle.setOutlineThickness(1);
				rectangle.setOutlineColor(sf::Color::Black);
			}
			squres.push_back(rectangle);

		}
		mod = !mod;
	}
}
```

```
void Game::LoadSprites()
{
	board.setTexture(boardTex);
	for (int i = 0; i < DIMENSIONS; i++)
	{
		for (int j = 0; j < DIMENSIONS; j++)
		{
			std::string p = boardStatus[i][j];
			int isWhite = (p[0] == 'w');
			switch (p[1])
			{
			case 'p':
				pieces[i][j] = new Pown(piecesTex[p],
Indexes2Position(sf::Vector2i(i, j)), isWhite, sf::Vector2i(i, j),
p);
				break;
			case 'K':
				pieces[i][j] = new King(piecesTex[p],
Indexes2Position(sf::Vector2i(i, j)), isWhite, sf::Vector2i(i, j),
p);
				break;
			case 'Q':
				pieces[i][j] = new Queen(piecesTex[p],
Indexes2Position(sf::Vector2i(i, j)), isWhite, sf::Vector2i(i, j),
p);
				break;
			case 'R':
				pieces[i][j] = new Rock(piecesTex[p],
Indexes2Position(sf::Vector2i(i, j)), isWhite, sf::Vector2i(i,j),
p);
				break;
			case 'B':
				pieces[i][j] = new Bishop(piecesTex[p],
Indexes2Position(sf::Vector2i(i, j)), isWhite, sf::Vector2i(i, j),
p);
				break;
			case 'N':
				pieces[i][j] = new Knight(piecesTex[p],
Indexes2Position(sf::Vector2i(i, j)), isWhite, sf::Vector2i(i, j),
p);
				break;

			default:
				pieces[i][j] = NULL;
				break;
			}

		}
	}

}
```

```
void Game::UpdateEvents()
{
	sf::Event event;

	while (window->pollEvent(event))
	{
		if (event.type == sf::Event::Closed)
		{
			window->close();
		}
		if (event.type == sf::Event::MouseButtonReleased)
			mReleased = true;
		if (event.type == sf::Event::Resized)
		{
			// update the view to the new size of the window
			sf::View view;
			if (event.size.width < event.size.height)
			{
				sf::FloatRect visibleArea(0.f, 0.f,
SCREEN_SIZE, (SCREEN_SIZE * event.size.height) / event.size.width);
				view = sf::View(visibleArea);
				window->setView(view);

			}
			else
			{
				sf::FloatRect visibleArea(0.f, 0.f,
(SCREEN_SIZE * event.size.width) / event.size.height, SCREEN_SIZE);
				view = sf::View(visibleArea);
				window->setView(view);
			}
			view.setCenter(sf::Vector2f(SCREEN_SIZE / 2,
SCREEN_SIZE / 2));
			window->setView(view);


		}
		if (event.type == sf::Event::KeyPressed &&
sf::Keyboard::isKeyPressed(sf::Keyboard::Escape))
		{
			gamePaused = true;
		}
	}
}
```

```
sf::Vector2f Game::Indexes2Position(sf::Vector2i indexes)
{
	return sf::Vector2f(indexes.y * DELTA_POS + START_POS,
indexes.x * DELTA_POS + START_POS);
}

sf::Vector2i Game::Position2Indexes(sf::Vector2f mPos)
{
	for (int i = 0; i < DIMENSIONS; i++)
	{
		for (int j = 0; j < DIMENSIONS; j++)
		{
			if (squres[i * DIMENSIONS +
j].getGlobalBounds().contains(mPos))
			{
				return sf::Vector2i(i, j);
			}
		}
	}
	return sf::Vector2i(-1, -1);
}


void Game::UpdateBoard()
{
	auto mousePosition = sf::Mouse::getPosition(*window);
	auto mPos = window->mapPixelToCoords(mousePosition);
	sf::Vector2i indexes = Position2Indexes(mPos);
	if (sf::Mouse::isButtonPressed(sf::Mouse::Left) && mReleased )
	{
		if (promtionChoice && indexes.x != -1)
		{
			int found = 0;
			for (int i = 0; i < promotionIndexes.size(); i++)
			{
				if (promotionIndexes[i] == indexes )
				{
					if (i > 0)
						RegularMove(indexes,
promotionIndexes[0]);
					found = 1;
					break;
				}

			}
			if (found)
			{
				for (int i = 0; i < 4; i++)
				{
					cleanSqure(promotionIndexes[i]);
					if (!i) continue;

	pieces[promotionIndexes[i].x][promotionIndexes[i].y] =
promotionPiecesReplace[i];
```

```
                        }
                        HandleChecks();
                        promtionChoice = 0;
                        doOnce = 1;
                  }


            }
            else
                  HandleClickes(indexes);
            mReleased = false;
      }
      if (sf::Mouse::isButtonPressed(sf::Mouse::Right) && mReleased)
      {
            sf::View view = window->getView();
            view.rotate(180.f);
            window->setView(view);
            RotatePieces();
            mReleased = false;
      }
      if (sf::Keyboard::isKeyPressed(sf::Keyboard::Left) &&
isUndoPossible)
      {
            UndoMove(lastMoveVec.back().first,
lastMoveVec.back().second);
            whiteTurn = !whiteTurn;
            HandleChecks();
            isUndoPossible = false;
      }
      if (promtionChoice)
      {
            if (doOnce)
            {
                  promotionIndexes.clear();
                  promotionPieces.clear();
                  promotionPiecesReplace.clear();
                  int xDir = whiteTurn ? -1 : 1;
                  auto temp = lastMoveVec.back().second;
                  int isWhite = pieces[temp.x][temp.y]->IsWhite();
                  for (int i = 0; i < 4; i++)
                  {
                        switch (i)
                        {
                        case 1:
                              if (pieces[temp.x][temp.y])

      promotionPiecesReplace.push_back(pieces[temp.x][temp.y]);
                              else

      promotionPiecesReplace.push_back(nullptr);
                              if (isWhite)
                                    pieces[temp.x][temp.y] = new
Knight(piecesTex["wN"], Indexes2Position(temp), 1, temp, "wN");
                              else
```

```
                                    pieces[temp.x][temp.y] = new
Knight(piecesTex["bN"], Indexes2Position(temp), 0, temp, "bN");
                              break;
                        case 2:
                              if (pieces[temp.x][temp.y])

      promotionPiecesReplace.push_back(pieces[temp.x][temp.y]);
                              else

      promotionPiecesReplace.push_back(nullptr);
                              if (isWhite)
                                    pieces[temp.x][temp.y] = new
Rock(piecesTex["wR"], Indexes2Position(temp), 1, temp, "wR");
                              else
                                    pieces[temp.x][temp.y] = new
Rock(piecesTex["bR"], Indexes2Position(temp), 0, temp, "bR");
                              break;
                        case 3:
                              if (pieces[temp.x][temp.y])

      promotionPiecesReplace.push_back(pieces[temp.x][temp.y]);
                              else

      promotionPiecesReplace.push_back(nullptr);
                              if (isWhite)
                                    pieces[temp.x][temp.y] = new
Bishop(piecesTex["wB"], Indexes2Position(temp), 1, temp, "wB");
                              else
                                    pieces[temp.x][temp.y] = new
Bishop(piecesTex["bB"], Indexes2Position(temp), 0, temp, "bB");
                              break;
                        default:
                              break;
                        }
                        promotionIndexes.push_back(temp);
                        if (!i) {

      promotionPieces.push_back(pieces[temp.x][temp.y]);

      promotionPiecesReplace.push_back(nullptr);
                        }
                        else

      promotionPieces.push_back(pieces[temp.x][temp.y]);
                        MarkSqure(temp, sf::Color::White);
                        temp.x += xDir;
                  }
                  doOnce = 0;
            }

            if (indexes.x != -1)
                  for (auto i : promotionIndexes)
                  {
                        if (i == indexes)
```

```
                                MarkSqure(i, markedSqure);
                        else
                                MarkSqure(i, sf::Color::White);
                }
        }


}


std::string Game::TranslateMove(std::pair<sf::Vector2i,
sf::Vector2i> move)
{
        char letter = 'a', number = '8';
        letter += move.first.y;
        number -= move.first.x;
        std::string str;
        str.push_back(letter);
        str.push_back(number);
        letter = 'a';
        number = '8';
        letter += move.second.y;
        number -= move.second.x;
        str.push_back(letter);
        str.push_back(number);
        return str;
}
void Game::HandleClickes(sf::Vector2i indexes)
{
        if (indexes.x != -1 && pieces[indexes.x][indexes.y] ||
playerClickes.size()) // check if the first click was not an empty
squre
                if (sqSelected == indexes && playerClickes.size() == 1)
// check if the same squre is pressed twice.
                {
                        if(pieces[indexes.x][indexes.y]->IsWhite())
                                cleanSqure(sqSelected);
                        else
                                cleanSqure(sqSelected);
                        possibleCircles.clear();
                        sqSelected = { -1,-1 };
                        playerClickes.clear();
                }
                else
                {
                        sqSelected = indexes;
                        playerClickes.push_back(sqSelected);
                        MarkSqure(sqSelected, markedSqure);
                        if (playerClickes.size() == 1 &&
PlayerTurn(playerClickes[0])) {
                                legalMoves =
GetLegalMoves(pieces[playerClickes[0].x][playerClickes[0].y]);
                                possibleCircles =
GetPossibleCircles(legalMoves);
```

```
                }

                if (playerClickes.size() == 2) // check if two
different squres has been pressed.
                {
                        auto it = std::find(legalMoves.begin(),
legalMoves.end(), playerClickes[1]);
                        int legalMove = (it != legalMoves.end());
                        if (PlayerTurn(playerClickes[0]) &&
legalMove)
                        {
                                lastType =
pieces[playerClickes[0].x][playerClickes[0].y]->GetType();
                                Move(playerClickes[0],
playerClickes[1]);
                                UpdatesAfterMove();
                        }
                        else
                        {
                                if (whiteTurn) {
                                        cleanSqure(playerClickes[0]);
                                        cleanSqure(playerClickes[1]);
                                }
                                else
                                {
                                        cleanSqure(playerClickes[0]);
                                        cleanSqure(playerClickes[1]);
                                }

                                possibleCircles.clear();
                                playerClickes.clear();
                        }

                }
            }
}
```

```
void Game::RegularMove(sf::Vector2i indexesBefore, sf::Vector2i
indexesAfter)
{
	pieces[indexesBefore.x][indexesBefore.y]-
>SetIndexes(indexesAfter);
	pieces[indexesBefore.x][indexesBefore.y]-
>SetPosition(Indexes2Position(indexesAfter));
	pieces[indexesAfter.x][indexesAfter.y] =
pieces[indexesBefore.x][indexesBefore.y];
	pieces[indexesBefore.x][indexesBefore.y] = NULL;
}

void Game::MarkSqure(sf::Vector2i indexes, sf::Color color)
{
	squres[indexes.x * DIMENSIONS +
indexes.y].setFillColor(color);
}

void Game::cleanSqure(sf::Vector2i indexes)
{
	if (indexes.x % 2 == 0 && indexes.y % 2 == 0 || indexes.x % 2
== 1 && indexes.y % 2 == 1)
	{
		squres[indexes.x * DIMENSIONS +
indexes.y].setFillColor(lightSqure);
	}
	else
	{
		squres[indexes.x * DIMENSIONS +
indexes.y].setFillColor(darkSqure);
	}
}

void Game::CleanBoard()
{
	for (int i = 0; i < DIMENSIONS; i++)
	{
		for (int j = 0; j < DIMENSIONS; j++)
		{
			free(pieces[i][j]);
			std::string p = boardStatus[i][j];
			switch (p[1])
			{
			case 'p':
				pieces[i][j] = new Pown(piecesTex[p],
Indexes2Position(sf::Vector2i(i, j)), p[0] == 'w', sf::Vector2i(i,
j), p);
				break;
			case 'K':
				pieces[i][j] = new King(piecesTex[p],
Indexes2Position(sf::Vector2i(i, j)), p[0] == 'w', sf::Vector2i(i,
j), p);
				break;
			case 'Q':
```

```
                        pieces[i][j] = new Queen(piecesTex[p],
Indexes2Position(sf::Vector2i(i, j)), p[0] == 'w', sf::Vector2i(i,
j), p);
                        break;
                  case 'R':
                        pieces[i][j] = new Rock(piecesTex[p],
Indexes2Position(sf::Vector2i(i, j)), p[0] == 'w', sf::Vector2i(i,
j), p);
                        break;
                  case 'B':
                        pieces[i][j] = new Bishop(piecesTex[p],
Indexes2Position(sf::Vector2i(i, j)), p[0] == 'w', sf::Vector2i(i,
j), p);
                        break;
                  case 'N':
                        pieces[i][j] = new Knight(piecesTex[p],
Indexes2Position(sf::Vector2i(i, j)), p[0] == 'w', sf::Vector2i(i,
j), p);
                        break;

                  default:
                        pieces[i][j] = NULL;
                        break;
                  }
            }
      }
      if (sqSelected != sf::Vector2i(-1, -1)) {
            cleanSqure(sqSelected);
            sqSelected = { -1,-1 };
      }
      playerClickes.clear();
      possibleCircles.clear();
      whiteTurn = 1;
      checkmate = 0;
      stalemate = 0;
      gameOver = 0;


}
```

```
void Game::Castling(sf::Vector2i indexesBefore, sf::Vector2i
indexesAfter)
{
	RegularMove(indexesBefore, indexesAfter);
	if (indexesAfter.y > indexesBefore.y)//	king castling
	{
		pieces[indexesAfter.x][indexesAfter.y + 1]-
>SetIndexes(sf::Vector2i(indexesAfter.x, indexesAfter.y - 1));
		pieces[indexesAfter.x][indexesAfter.y + 1]-
>SetPosition(Indexes2Position(sf::Vector2i(indexesAfter.x,
indexesAfter.y - 1)));
		pieces[indexesAfter.x][indexesAfter.y - 1] =
pieces[indexesAfter.x][indexesAfter.y + 1];
		pieces[indexesAfter.x][indexesAfter.y + 1] = NULL;
		moveTypeVec.push_back(KING_CASTLING);
	}
	else	// queen castling
	{
		pieces[indexesAfter.x][indexesAfter.y - 2]-
>SetIndexes(sf::Vector2i(indexesAfter.x, indexesAfter.y + 1));
		pieces[indexesAfter.x][indexesAfter.y - 2]-
>SetPosition(Indexes2Position(sf::Vector2i(indexesAfter.x,
indexesAfter.y + 1)));
		pieces[indexesAfter.x][indexesAfter.y + 1] =
pieces[indexesAfter.x][indexesAfter.y - 2];
		pieces[indexesAfter.x][indexesAfter.y - 2] = NULL;
		moveTypeVec.push_back(QUEEN_CASTLING);
	}
}

void Game::Promotion(sf::Vector2i indexesBefore, sf::Vector2i
indexesAfter, bool isWhite)
{
	if(isWhite)
		switch (promotionTypeVec.back())
		{
		case QUEEN:
			pieces[indexesAfter.x][indexesAfter.y] = new
Queen(piecesTex["wQ"], Indexes2Position(indexesAfter), 1,
indexesAfter, "wQ");
			break;
		case KNIGHT:
			pieces[indexesAfter.x][indexesAfter.y] = new
Knight(piecesTex["wN"], Indexes2Position(indexesAfter), 1,
indexesAfter, "wN");
			break;
		case ROCK:
			pieces[indexesAfter.x][indexesAfter.y] = new
Rock(piecesTex["wR"], Indexes2Position(indexesAfter), 1,
indexesAfter, "wR");
			break;
		case BISHOP:
```

```
                    pieces[indexesAfter.x][indexesAfter.y] = new
Bishop(piecesTex["wB"], Indexes2Position(indexesAfter), 1,
indexesAfter, "wB");
                    break;
              }
        else
              switch (promotionTypeVec.back())
              {
              case QUEEN:
                    pieces[indexesAfter.x][indexesAfter.y] = new
Queen(piecesTex["bQ"], Indexes2Position(indexesAfter), 0,
indexesAfter, "bQ");
                    break;
              case KNIGHT:
                    pieces[indexesAfter.x][indexesAfter.y] = new
Knight(piecesTex["bN"], Indexes2Position(indexesAfter), 0,
indexesAfter, "bN");
                    break;
              case ROCK:
                    pieces[indexesAfter.x][indexesAfter.y] = new
Rock(piecesTex["bR"], Indexes2Position(indexesAfter), 0,
indexesAfter, "bR");
                    break;
              case BISHOP:
                    pieces[indexesAfter.x][indexesAfter.y] = new
Bishop(piecesTex["bB"], Indexes2Position(indexesAfter), 0,
indexesAfter, "bB");
                    break;
              }
        pieces[indexesBefore.x][indexesBefore.y] = NULL;
}

void Game::EnPassent(bool isWhite, sf::Vector2i indexesBefore,
sf::Vector2i indexesAfter)
{
        pieces[indexesBefore.x][indexesBefore.y]-
>SetIndexes(indexesAfter);
        pieces[indexesBefore.x][indexesBefore.y]-
>SetPosition(Indexes2Position(indexesAfter));
        pieces[indexesAfter.x][indexesAfter.y] =
pieces[indexesBefore.x][indexesBefore.y];
        pieces[indexesBefore.x][indexesBefore.y] = NULL;
        if (isWhite) {
              pieces[indexesAfter.x + 1][indexesAfter.y] = NULL; //
delete the pown eaten.
        }
        else {
              pieces[indexesAfter.x - 1][indexesAfter.y] = NULL; //
delete the pown eaten.
        }


}
```

```
int Game::IsCastlingLegal(sf::Vector2i indexesBefore, sf::Vector2i
indexesAfter)
{
	if (abs(indexesAfter.y - indexesBefore.y) == 2) // check if
the move was castling.
	{
		if (InCheck())
			return 0;
		if (indexesAfter.y > indexesBefore.y)  // king castling
		{
			if (!squreUnderAttack(sf::Vector2i(indexesAfter.x,
indexesAfter.y - 1)))
				return 1;
		}
		else if (!squreUnderAttack(sf::Vector2i(indexesAfter.x,
indexesAfter.y + 1))) //queen castling
		{
			return 1;
		}
		return 0;
	}
	else
	{
		return 1;
	}
}
-------------------------------------------------------------

void Game::Move(sf::Vector2i indexesBefore, sf::Vector2i
indexesAfter)
{
	lastTypeVec.push_back(pieces[indexesBefore.x][indexesBefore.y]
->GetType());
	lastMoveVec.push_back(std::pair<sf::Vector2i,
sf::Vector2i>(indexesBefore, indexesAfter));
	for (int i = 0; i < DIMENSIONS; i++)
	{
		for (int j = 0; j < DIMENSIONS; j++)
		{
			if (pieces[i][j]) pieces[i][j]-
>SetLastMove(lastMoveVec.back());
		}
	}
	pieceMovedVec.push_back(pieces[indexesBefore.x][indexesBefore.
y]->PieceMoved());
	Piece* pieceCaptured = pieces[indexesAfter.x][indexesAfter.y];
	capturedSqure = pieces[indexesAfter.x][indexesAfter.y];
	// check for special moves :
	if (pieces[indexesBefore.x][indexesBefore.y]->GetType() ==
"bp" && indexesAfter.x == 7)  // check for bp promotion.
	{
		Promotion(indexesBefore, indexesAfter, false);
		moveTypeVec.push_back(PROMOTION);
	}
```

```
	else if (pieces[indexesBefore.x][indexesBefore.y]->GetType()
== "wp" && indexesAfter.x == 0)  // check for wp promotion.
	{
		Promotion(indexesBefore, indexesAfter, true);
		moveTypeVec.push_back(PROMOTION);
	}
	else if (pieces[indexesBefore.x][indexesBefore.y]->GetType()
== "wp" &&
		indexesBefore.x == 3 && indexesAfter.y !=
indexesBefore.y && !pieces[indexesAfter.x][indexesAfter.y]) // check
white en passent
	{
		pieceCaptured = pieces[indexesAfter.x +
1][indexesAfter.y];
		EnPassent(true, indexesBefore, indexesAfter);
		moveTypeVec.push_back(WHITE_ENPASSENT);
	}
	else if (pieces[indexesBefore.x][indexesBefore.y]->GetType()
== "bp" && indexesBefore.x == 4 &&
		indexesAfter.y != indexesBefore.y &&
!pieces[indexesAfter.x][indexesAfter.y])  // check black en passent
	{
		pieceCaptured = pieces[indexesAfter.x -
1][indexesAfter.y];
		EnPassent(false, indexesBefore, indexesAfter);
		moveTypeVec.push_back(BLACK_ENPASSENT);
	}
	else if (pieces[indexesBefore.x][indexesBefore.y]-
>GetType()[1] == 'K' && abs(indexesAfter.y-indexesBefore.y) == 2 )
// check castling
	{
		Castling(indexesBefore, indexesAfter);

	}
	else // A regular move :
	{
		RegularMove(indexesBefore, indexesAfter);
		moveTypeVec.push_back(REGULARMOVE);
	}
	capturedPieces.push_back(pieceCaptured);

}
```

```
void Game::UpdatesAfterMove()
{
	cleanSqure(playerClickes[0]);
	cleanSqure(playerClickes[1]);
	possibleCircles.clear();

	playerClickes.clear();
	if (whiteTurn) {
		sf::Vector2i wKLoc = GetKingsLocation()[0];
		pieces[wKLoc.x][wKLoc.y]-
>SetSpriteColor(sf::Color::White);
	}
	else
	{
		sf::Vector2i bKLoc = GetKingsLocation()[1];
		pieces[bKLoc.x][bKLoc.y]-
>SetSpriteColor(sf::Color::White);
	}
	whiteTurn = !whiteTurn;

	HandleChecks();

	checkmateOrStalemate();
	isUndoPossible = true;
	if(moveTypeVec[moveTypeVec.size()-1] == PROMOTION)
		promtionChoice = 1;
}

std::vector<sf::CircleShape>
Game::GetPossibleCircles(std::vector<sf::Vector2i> possibleMoves)
{
	std::vector<sf::CircleShape> circles;
	sf::CircleShape circle;
	circle.setFillColor(sf::Color::Red);
	circle.setRadius(15);
	for (auto move : possibleMoves) {
		sf::Vector2f pos = Indexes2Position(move);
		pos.x += 23;
		pos.y += 25;
		circle.setPosition(pos);
		circles.push_back(circle);
	}
	return circles;
}
```

```
std::vector<sf::Vector2i> Game::GetLegalMoves(Piece* piece)
{
	// naive soluation:
	//std::vector<sf::Vector2i> moves = piece-
>GetPossibleMoves(pieces);
	//sf::Vector2i indexesBefore = piece->GetIndexes();
	//sf::Vector2i indexesAfter;
	//
	//for (int i = moves.size() - 1; i >= 0; i--)
	//{
	//	// Do the move.
	//	indexesAfter = moves[i];
	//	Move(indexesBefore, indexesAfter);
	//
	//	if (InCheck())
	//		moves.erase(moves.begin() + i);
	//	UndoMove(indexesBefore, indexesAfter);
	//}
	//

	//
	//return moves;
	// advance solution:
	return piece->GetValidMoves(pieces, GetKingsLocation());
}

int Game::InBoard(sf::Vector2i indexes)
{

	return indexes.x >= 0 && indexes.x < DIMENSIONS && indexes.y
>= 0 && indexes.y < DIMENSIONS;
}

int Game::InCheck()
{
	std::vector<sf::Vector2i> locations = GetKingsLocation();
	if (whiteTurn)
		return squreUnderAttack(locations[0]); // White king`s
indexes.
	else
		return squreUnderAttack(locations[1]); // Black king`s
indexes.
}

int Game::squreUnderAttack(sf::Vector2i location)
{

	int dirX, dirY;
	int directions[][8] = { {-1,-1}, {-1, 1}, {1, -1}, {1, 1}, {0,
-1}, {0, 1}, {1, 0}, {-1, 0} };
	sf::Vector2i temp = location;
	Piece* locPiece = pieces[location.x][location.y];
	int isWhite = locPiece->IsWhite();
	for (int i = 0; i < 8; i++)
```

```
	{
		dirX = directions[i][0];
		dirY = directions[i][1];
		temp.x += dirX;
		temp.y += dirY;
		int dis = 1;
		while (InBoard(temp))
		{
			if (pieces[temp.x][temp.y])
			{
				if (pieces[temp.x][temp.y]->IsWhite() !=
pieces[location.x][location.y]->IsWhite())
				{
					switch (pieces[temp.x][temp.y]-
>GetType()[1])
					{
					case 'Q':
						return 1;
					case 'B':
						if (i < 4)
							return 1;
						break;
					case 'R':
						if (i >= 4)
							return 1;
						break;
					case 'p':
						if (dis == 1)
							if (pieces[temp.x][temp.y]-
>IsWhite() && i >= 2 && i <= 3)
								return 1;
							else if
(!pieces[temp.x][temp.y]->IsWhite() && i < 2)
								return 1;
						break;
					case 'K':
						if (dis == 1)
							return 1;
						break;
					} // end switch

				}
				break;
			}
			temp.x += dirX;
			temp.y += dirY;
			dis++;
		}// end while
		temp = location;
	}
	// check for Knights :
	std::vector<sf::Vector2i> possibleMoves = { {location.x + 2,
location.y + 1}, {location.x + 2, location.y - 1 },{location.x - 2,
location.y + 1},
```

```
            {location.x - 2, location.y - 1}, {location.x + 1,
location.y + 2}, {location.x - 1, location.y + 2},
      {location.x + 1, location.y - 2}, {location.x - 1, location.y
- 2} };
      for (auto move : possibleMoves)
            if (InBoard(move) && pieces[move.x][move.y] &&
pieces[move.x][move.y]->GetType()[1] == 'N' &&
pieces[move.x][move.y]->IsWhite() != pieces[location.x][location.y]-
>IsWhite())
                  return 1;
      return 0;


      // naive solution

      //for (int i = 0; i < DIMENSIONS; i++)
      //{
      //    for (int j = 0; j < DIMENSIONS; j++)
      //    {
      //          if (pieces[i][j] && pieces[i][j]->IsWhite() !=
pieces[location.x][location.y]->IsWhite()) // check if its white`s
turn and the piece is white or if it` s black turn and the piece is
black.
      //          {
      //                auto opponentPossibleMoves = pieces[i][j]-
>GetPossibleMoves(pieces);
      //                for (auto move : opponentPossibleMoves)
      //                {
      //                      if (move == location) {
      //                            return 1;
      //                      }
      //                }
      //          }
      //    }
      //}
      //return 0;
}

std::vector<sf::Vector2i> Game::GetKingsLocation()
{
      std::vector<sf::Vector2i> locations;
      locations.resize(2);
      for (int i = 0; i < DIMENSIONS; i++)
      {
            for (int j = 0; j < DIMENSIONS; j++)
            {
                  if (pieces[i][j] && pieces[i][j]->GetType() ==
"wK")
                        locations[0] = pieces[i][j]->GetIndexes();
                  if (pieces[i][j] && pieces[i][j]->GetType() ==
"bK")
                        locations[1] = pieces[i][j]->GetIndexes();
            }
      }
```

```
	return locations;
}


void Game::HandleChecks()
{
	if (InCheck()) {
		if (whiteTurn) {
			sf::Vector2i wKLoc = GetKingsLocation()[0];
			pieces[wKLoc.x][wKLoc.y]-
>SetSpriteColor(sf::Color::Red);
		}
		else
		{
			sf::Vector2i bKLoc = GetKingsLocation()[1];
			pieces[bKLoc.x][bKLoc.y]-
>SetSpriteColor(sf::Color::Red);
		}
		inCheck = true;
	}
	else
	{

		inCheck = false;
	}

}

void Game::checkmateOrStalemate()
{
	auto moves = AllLegalMoves(whiteTurn);
	if (moves.empty()) {
		if (inCheck)
			checkmate = 1;
		else
			stalemate = 1;
	}
}
```

```
void Game::DrawNumbersLetters(sf::RenderWindow* window)
{
	sf::Vector2f pos = { 0.1*DELTA_POS , SCREEN_SIZE - 22 };
	std::string letters[] = { "a", "b", "c", "d", "e", "f", "g",
"h" };
	text_board.setPosition(pos);
	for (auto letter : letters)
	{
		text_board.setString(letter);
		window->draw(text_board);
		pos.x += DELTA_POS;
		text_board.setPosition(pos);
	}
	pos = { 785 , SCREEN_SIZE - DELTA_POS };
	text_board.setPosition(pos);
	std::string numbers[] = { "1", "2", "3", "4", "5", "6", "7",
"8" };
	for (auto number : numbers)
	{
		text_board.setString(number);
		window->draw(text_board);
		pos.y -= DELTA_POS;
		text_board.setPosition(pos);
	}



}



void Game::RotatePieces()
{
	for (int i = 0; i < DIMENSIONS; i++)
	{
		for (int j = 0; j < DIMENSIONS; j++)
		{
			if (pieces[i][j])
				pieces[i][j]->RotatePiece(180.f);
		}
	}
}
```

```
std::vector<std::pair<sf::Vector2i, sf::Vector2i>>
Game::AllLegalMoves(bool isWhite)
{
	std::vector<std::pair<sf::Vector2i, sf::Vector2i>> allMoves;
	std::vector<sf::Vector2i> moves;
	for (int i = 0; i < DIMENSIONS; i++)
	{
		for (int j = 0; j < DIMENSIONS; j++)
		{
			if (!pieces[i][j]) continue;
			if (pieces[i][j]->IsWhite() == isWhite)
			{
				moves = GetLegalMoves(pieces[i][j]);
				for (auto move : moves)

	allMoves.push_back((std::pair<sf::Vector2i,
sf::Vector2i>(sf::Vector2i(i, j), move)));
			}

		}
	}
	return allMoves;
}
```

יצירת המחלקה OFFLINEGAME

DECLARATION

```
class OfflineGame :
    public Game
{
public:
    OfflineGame(); // Ctor.
    void Init(); // loop function.
    void Update();
    void Print(); // print all needed sprites to the screen.
    bool PlayerTurn(sf::Vector2i indexes);


};
```

IMPLEMENTATION

```
#include "OfflineGame.h"

OfflineGame::OfflineGame()
{
      mReleased = 0;
      sqSelected = { -1,-1 };
      whiteTurn = 1;
      isUndoPossible = false;
      mReleased = true;
      checkmate = 0;
      stalemate = 0;
      gameOver = 0;
      promtionChoice = 0;
      inCheck = false;
      promotionTypeVec.push_back(QUEEN);
}
void OfflineGame::Init()
{
      while (window->isOpen())
      {
            UpdateEvents();
            Update();
            Print();
      }
}


void OfflineGame::Update()
{

      if (!gameOver && !gamePaused)
            UpdateBoard();
      else
            if (sf::Mouse::isButtonPressed(sf::Mouse::Left) &&
buttons[0]->IsMouseHover(*window))
```

```
                CleanBoard();
        if (gamePaused)
        {
                for (auto btn : buttons)
                        btn->HandleMouseHover(*window);
                if (sf::Mouse::isButtonPressed(sf::Mouse::Left)) {
                        if (buttons[0]->IsMouseHover(*window)) {
                                CleanBoard();
                                gamePaused = false;
                        }
                        else if (buttons[1]->IsMouseHover(*window)) {
                                gamePaused = false;
                        }
                        else if (buttons[2]->IsMouseHover(*window)) {
                                window->close();
                        }

                }

        }
        if (checkmate) {
                gameOver = 1;
                if (whiteTurn)
                        text_msg.setString("Black won !");
                else
                        text_msg.setString("White won !");
        }
        if (stalemate)
        {
                gameOver = 1;
                text_msg.setString("Stalemate !");
        }

        if (sf::Keyboard::isKeyPressed(sf::Keyboard::Space))
        {
                CleanBoard(); // restart the game.
        }
}
```

```
void OfflineGame::Print()
{
	window->clear();

	for (int i = 0; i < DIMENSIONS; i++)
	{
		for (int j = 0; j < DIMENSIONS; j++)
		{
			window->draw(squres[i * DIMENSIONS + j]);
		}
	}

	for (int i = 0; i < DIMENSIONS; i++)
	{
		for (int j = 0; j < DIMENSIONS; j++)
		{
			if (pieces[i][j])
			{
				pieces[i][j]->Print(window);
			}
		}
	}
	DrawNumbersLetters(window);
	for (auto circle : possibleCircles) {
		window->draw(circle);
	}
	if (gamePaused)
		for (auto btn : buttons)
			btn->DrawTo(*window);

	if (gameOver)
	{
		//window->clear();
		window->draw(text_msg);
		buttons[0]->DrawTo(*window);
		buttons[0]->HandleMouseHover(*window);
	}
	window->display();
}

bool OfflineGame::PlayerTurn(sf::Vector2i indexes)
{
	return pieces[indexes.x][indexes.y]->IsWhite() == whiteTurn;
}
```

יצירת המחלקה AIGAME

DECLARATION

```
#pragma once
#include "Game.h"

class AiGame :
    public Game
{
public:
    // basic functions:
    AiGame();
    void Init(); // loop function.
    void Update();
    void Print(); // print all needed sprites to the screen.
    bool PlayerTurn(sf::Vector2i indexes);
    // evaluations:
    int evaluate(); // evaluate the board.
    int CountMaterial(bool isWhite);
    int CountPieceType(std::string type);
    // minimax
    int minimax(int depth, int isMax);
    int minimax(int depth, int isMax, int alpha, int beta);
    std::pair<sf::Vector2i, sf::Vector2i> findBestMove();
    void SmartAi();
    // analisys
    int MoveGenerarionTest(int depth, int isWhite, int startDepth);
    // A search function
    // order moves for pure alpha beta purning function
    std::vector<std::pair<sf::Vector2i, sf::Vector2i>>
OrderMoves(std::vector<std::pair<sf::Vector2i, sf::Vector2i>>
moves);
   // A map of the pieces scores
    std::map<char, int> scores;
private:
    std::map<std::string, int> moveCounter;
    int a = 1;
};
```

```
#include "AiGame.h"
#include <random>
#include <ctime>
AiGame::AiGame()
{
	sqSelected = { -1,-1 };
	whiteTurn = 1;
	scores['p'] = 100;
	scores['B'] = 300;
	scores['N'] = 320;
	scores['R'] = 500;
	scores['Q'] = 900;
	isUndoPossible = false;
	mReleased = true;
	checkmate = 0;
	stalemate = 0;
	gameOver = 0;
	promtionChoice = 0;
	promotionTypeVec.push_back(QUEEN);

}

void AiGame::Init()
{
	while (window->isOpen())
	{
		UpdateEvents();
		Update();
		Print();
	}
}


void AiGame::Update()
{
	if (checkmate) {
		gameOver = 1;
		if (whiteTurn)
			text_msg.setString("Black won !\nPress space to
restart.");
		else
			text_msg.setString("White won !\nPress space to
restart.");
	}
	if (stalemate)
	{
		gameOver = 1;
		text_msg.setString("Stalemate !\nPress space to
restart.");
	}
	if (gamePaused)
	{
```

```
            for (auto btn : buttons)
                  btn->HandleMouseHover(*window);
            if (sf::Mouse::isButtonPressed(sf::Mouse::Left)) {
                  if (buttons[0]->IsMouseHover(*window)) {
                        CleanBoard();
                        gamePaused = false;
                  }
                  else if (buttons[1]->IsMouseHover(*window)) {
                        gamePaused = false;
                  }
                  else if (buttons[2]->IsMouseHover(*window)) {
                        window->close();
                  }

            }

      }
      if (!gameOver && !gamePaused)
      {
            int a = MoveGenerarionTest(4, whiteTurn, 4);
            std::cout << "-------------------" << a << "------------
----" << std::endl;
            /*for (int i = 1; i < 10; i++)
            {
                  int a = MoveGenerarionTest(i, whiteTurn, i);
                  std::cout << "-------------------" << a << "------
----------" << std::endl;
            }*/
            if (whiteTurn)
                  UpdateBoard();
            else
            {
                  SmartAi();
                  whiteTurn = !whiteTurn;
                  HandleChecks();
                  checkmateOrStalemate();
            }


      }

      if (sf::Keyboard::isKeyPressed(sf::Keyboard::Space))
      {
            CleanBoard(); // restart the game.
      }

}

void AiGame::Print()
{
      window->clear();

      for (int i = 0; i < DIMENSIONS; i++)
      {
```

```
            for (int j = 0; j < DIMENSIONS; j++)
            {
                  window->draw(squres[i * DIMENSIONS + j]);
            }
      }

      for (int i = 0; i < DIMENSIONS; i++)
      {
            for (int j = 0; j < DIMENSIONS; j++)
            {
                  if (pieces[i][j])
                  {
                        pieces[i][j]->Print(window);
                  }
            }
      }
      DrawNumbersLetters(window);
      for (auto circle : possibleCircles) {
            window->draw(circle);
      }
      if (gamePaused)
            for (auto btn : buttons)
                  btn->DrawTo(*window);
      if (gameOver)
      {
            window->draw(text_msg);
            buttons[0]->DrawTo(*window);
            buttons[0]->HandleMouseHover(*window);
      }
      window->display();
}

bool AiGame::PlayerTurn(sf::Vector2i indexes)
{
      if (pieces[indexes.x][indexes.y]->IsWhite())
            return true;
      return false;
}


int AiGame::evaluate()
{
      /*
                  if white won -> negative infinity
                  if black won -> positive infinity
                  if stalemate -> 0
                  else -> blackMetirial - whiteMetirial
      */

      int metirial = 0, mul;

      for (int i = 0; i < DIMENSIONS; i++)
      {
```

```
            for (int j = 0; j < DIMENSIONS; j++)
            {
                  if (!pieces[i][j]) continue;
                  mul = pieces[i][j]->IsWhite() ? -1 : 1;
                  char type = pieces[i][j]->GetType()[1]; // the
letter of the piece.
                  metirial += mul * scores[type];
            }
}

return metirial; }

-----------------------------------------------------------------

int AiGame::minimax(int depth, int isMax)
{
      if (depth == 0)
            return evaluate();
      std::vector<std::pair<sf::Vector2i, sf::Vector2i>> moves =
AllLegalMoves(!isMax);
      if (isMax)
      {
            int bestEval = -INFINITY;
            if (moves.size() == 0) {
                  if (InCheck())
                        return bestEval;
                  return 0;
            }
            for (auto move : moves)
            {
                  // do the move :
                  Move(move.first, move.second);
                  int eval = minimax(depth - 1, !isMax);
                  if (eval > bestEval)
                        bestEval = eval;
                  // undo the move :
                  UndoMove(move.first, move.second);
            }
            return bestEval;
      }
      else
      {
            int bestEval = +INFINITY;
            if (moves.size() == 0) {
                  if (InCheck())
                        return bestEval;
                  return 0;
            }
            for (auto move : moves)
            {
                  // do the move :
                  Move(move.first, move.second);
                  int eval = minimax(depth - 1, !isMax);
                  if (eval < bestEval)
```

```
                        bestEval = eval;
                    // undo the move :
                    UndoMove(move.first, move.second);
                }
                return bestEval;
        }
        return 0;
}


-----------------------------------------------------------------

int AiGame::minimax(int depth, int isMax, int alpha, int beta)
{
        if (depth == 0) {
                int a = evaluate();
                //std::cout << a << std::endl;
                return a;
        }
        //std::vector<std::pair<sf::Vector2i, sf::Vector2i>> moves =
AllLegalMoves(!isMax);
        std::vector<std::pair<sf::Vector2i, sf::Vector2i>> moves =
OrderMoves(AllLegalMoves(!isMax));


        if (isMax)
        {
                int bestEval = -INFINITY;
                if (moves.size() == 0) {
                        if (InCheck())
                                return bestEval;
                        return 0;
                }
                for (auto move : moves)
                {
                        // do the move :
                        Move(move.first, move.second);
                        whiteTurn = !whiteTurn;
                        int eval = minimax(depth - 1, !isMax, alpha,
beta);
                        whiteTurn = !whiteTurn;
                        bestEval = std::max(bestEval, eval);
                        alpha = std::max(alpha, eval);
                        if (beta <= alpha) {
                                // undo the move :
                                UndoMove(move.first, move.second);
                                /*printf("%d\n", a++);*/
                                break;
                        }

                        // undo the move :
                        UndoMove(move.first, move.second);
                }
                return bestEval;
```

```
	}
	else
	{
		int bestEval = +INFINITY;
		if (moves.size() == 0) {
			if (InCheck())
				return bestEval;
			return 0;
		}
		for (auto move : moves)
		{
			// do the move :
			Move(move.first, move.second);
			whiteTurn = !whiteTurn;
			int eval = minimax(depth - 1, !isMax, alpha,
beta);
			whiteTurn = !whiteTurn;
			bestEval = std::min(bestEval, eval);
			beta = std::min(beta, eval);
			if (beta <= alpha) {
				// undo the move :
				UndoMove(move.first, move.second);
				/*printf("%d\n", a++);*/
				break;
			}
			// undo the move :
			UndoMove(move.first, move.second);
		}
		return bestEval;
	}
	return 0;
}
```

```
std::pair<sf::Vector2i, sf::Vector2i> AiGame::findBestMove()
{
	auto moves = AllLegalMoves(false);
	//auto moves = OrderMoves(AllLegalMoves(false));

	std::pair<sf::Vector2i, sf::Vector2i> bestMove;
	int bestScore = -INFINITY;
	for (auto move : moves)
	{
		Move(move.first, move.second);
```

```
            std::string str = TranslateMove(move);
            std::cout << str << std::endl;
            whiteTurn = !whiteTurn;
            int score = minimax(2, 0, -INFINITY, INFINITY);
            whiteTurn = !whiteTurn;
            //int score = Search(2, 0);
            if (score > bestScore)
            {
                  bestScore = score;
                  bestMove = move;
            }
            UndoMove(move.first, move.second);
      }
      a = 0;
      return bestMove;
}



void AiGame::SmartAi()
{
      auto move = findBestMove();
      Move(move.first, move.second);
}
```

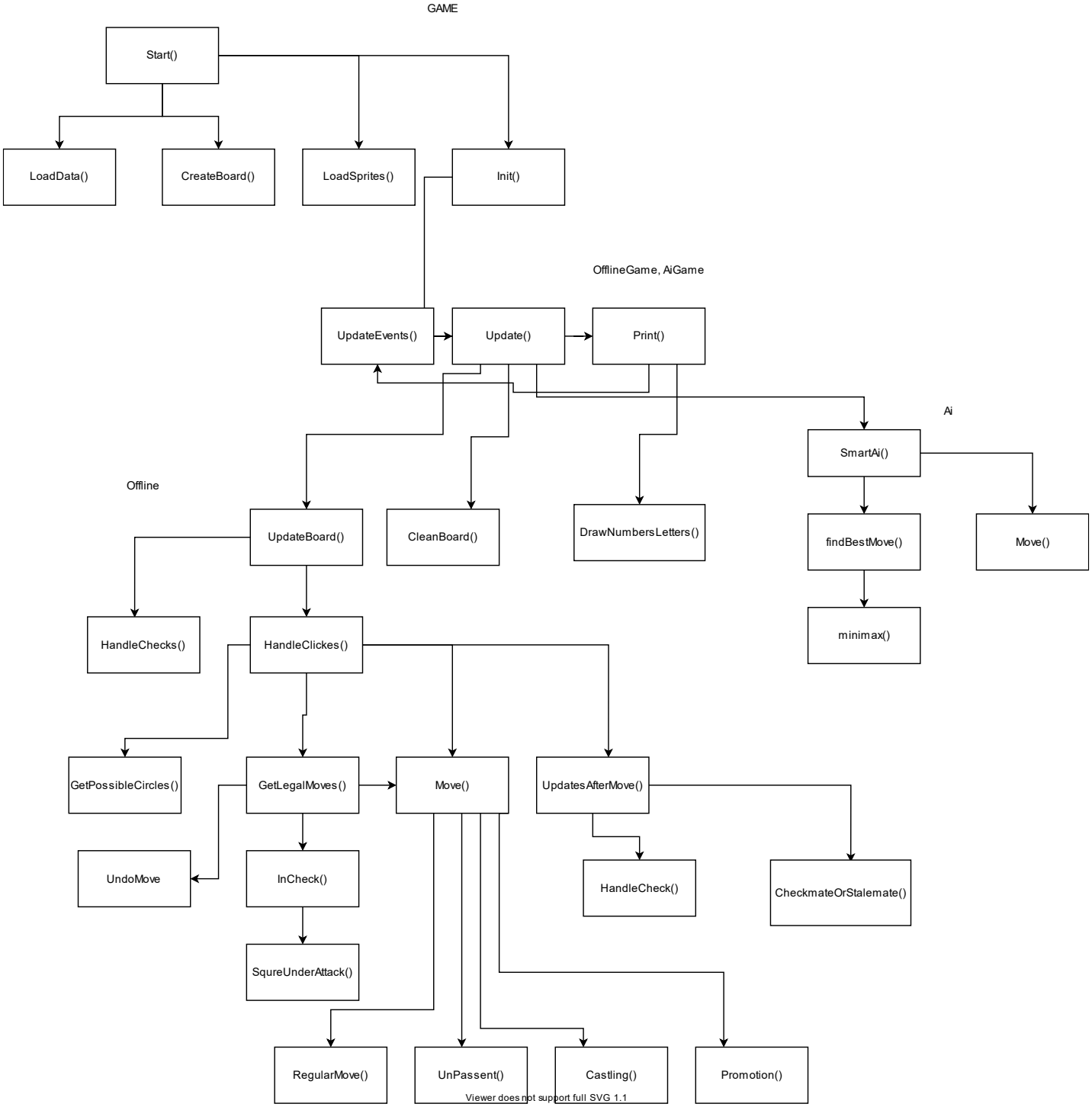
```
int AiGame::MoveGenerarionTest(int depth, int isWhite, int
startDepth)
{
      if (depth == 0)
            return 1;
      auto moves = AllLegalMoves(isWhite);
      //moves = OrderMoves(moves);
      int numPositions = 0;
      int numPositionsEachMove = 0;
      std::string str;
      for (auto move : moves)
```

```
	{
		/*if (depth == startDepth) {
			if(TranslateMove(move) == )
		}*/
		Move(move.first, move.second);
		whiteTurn = !whiteTurn;
		numPositionsEachMove = MoveGenerarionTest(depth - 1,
!isWhite, startDepth);
		whiteTurn = !whiteTurn;
		numPositions += numPositionsEachMove;
		str = TranslateMove(move);
		if (depth == startDepth) {
			// check if the move was promotion and calc for
each option.
			if (moveTypeVec.back() == PROMOTION) {
				char t =
pieces[move.second.x][move.second.y]->GetType()[1];
				str.push_back(t);
			}
			moveCounter[str] = numPositionsEachMove;
			std::cout << str << " : " << numPositionsEachMove
<< std::endl;
		}
		if (moveTypeVec.back() == PROMOTION)
		{
			UndoMove(move.first, move.second);
			for (int i = 0; i < 3; i++)
			{
				promotionTypeVec.push_back(i + 1);
				Move(move.first, move.second);
				whiteTurn = !whiteTurn;
				numPositionsEachMove =
MoveGenerarionTest(depth - 1, !isWhite, startDepth);
				whiteTurn = !whiteTurn;
				numPositions += numPositionsEachMove;
				if (depth == startDepth) {
					// check if the move was promotion and
calc for each option.
					str = TranslateMove(move);
					if (moveTypeVec.back() == PROMOTION) {
						char t =
pieces[move.second.x][move.second.y]->GetType()[1];
						str.push_back(t);
					}
					moveCounter[str] =
numPositionsEachMove;
					std::cout << str << " : " <<
numPositionsEachMove << std::endl;
				}
				UndoMove(move.first, move.second);
				promotionTypeVec.pop_back();
			}

		}
```

```
            else
                  UndoMove(move.first, move.second);


      }
      return numPositions;
}


std::vector<std::pair<sf::Vector2i, sf::Vector2i>>
AiGame::OrderMoves(std::vector<std::pair<sf::Vector2i,
sf::Vector2i>> moves)
{
      std::vector<int> scoreGuesses;
      for (auto move : moves)
      {
            int moveScoreGuess = 0;
            sf::Vector2i movePieceIndexes = move.first;
            sf::Vector2i capturedPieceIndexes = move.second;
            Piece* movePiece =
pieces[movePieceIndexes.x][movePieceIndexes.y];
            Piece* capturedPiece =
pieces[capturedPieceIndexes.x][capturedPieceIndexes.y];
            if (capturedPiece)
                  moveScoreGuess = 10 * scores[capturedPiece-
>GetType()[1]] - scores[movePiece->GetType()[1]];
            if (movePiece->GetType()[1] == 'p')
                  if (movePiece->IsWhite() && capturedPieceIndexes.x
== 0)
                        moveScoreGuess += scores['Q'];
                  else if(!movePiece->IsWhite() &&
capturedPieceIndexes.x == DIMENSIONS-1)
                        moveScoreGuess += scores['Q'];
            Move(move.first, move.second);
            if (squreUnderAttack(capturedPieceIndexes))
                  moveScoreGuess -= scores[movePiece->GetType()[1]];
            UndoMove(move.first, move.second);
            scoreGuesses.push_back(moveScoreGuess);
      }
      // sort the moves by there move score guesses:
      // bubble sort :
      for (int i = 0; i < scoreGuesses.size()-1; i++)
      {
            for (int j = 0; j < scoreGuesses.size() - 1 - i; j++)
            {
                  if (scoreGuesses[j] < scoreGuesses[j + 1])
                  {
                        std::swap(moves[j], moves[j + 1]);
                  }
            }
      }
      return moves;
}
```

```
int AiGame::CountMaterial(bool isWhite)
{
	int material = 0;
	char color = isWhite ? 'w' : 'b';
	std::string type;
	type = color;
	material += CountPieceType(type + 'p') /** scores['p']*/; //
pown material
	material += CountPieceType(type + 'B') /** scores['B']*/; //
Bishop material
	material += CountPieceType(type + 'Q') /** scores['Q']*/; //
Queen material
	material += CountPieceType(type + 'N') /** scores['N']*/; //
Knight material
	material += CountPieceType(type + 'R') /** scores['R']*/; //
Rock material
	return material;
}



int AiGame::CountPieceType(std::string type)
{
	int count = 0;
	for (int i = 0; i < DIMENSIONS; i++)
	{
		for (int j = 0; j < DIMENSIONS; j++)
		{
			if (pieces[i][j] && pieces[i][j]->GetType() ==
type)
				count += 1;
		}
	}
	return count;
}
```

היררכיית פעולות מחלקת GAME *והמחלקות היורשות:*



מחלקת PIECE:

DECLARATION

```
#pragma once
#include <SFML/Graphics.hpp>
#include <map>
#include "Constants.h"



class Piece
{
public:
	// constructor
	Piece(sf::Texture& pieceTex, sf::Vector2f position,
		int isWhite, sf::Vector2i indexes, std::string type); //
Ctor
	Piece(); // defualt Ctor
	//piece functions
	void Print(sf::RenderWindow* window); // gets the window
pointer and prints the piece.
	// get functions
	sf::Sprite GetSprite();
	sf::Vector2f GetPosition();
	sf::Vector2i GetIndexes();
	std::string GetType();
	int IsWhite();
	// set functions
	void SetPosition(sf::Vector2f position);
	void SetIndexes(sf::Vector2i indexes);
	void SetSpriteColor(sf::Color color);
	void SetLastMove(std::pair<sf::Vector2i, sf::Vector2i>
lastMove);
	void setPieceMoved(bool pieceMoved);
	// piece functions :
	/* the function gets source click and dest click and the
pieces.
	the function returns 1 if the dest click has no piece or have
a piece with an opposite color.
	else it returns 0.
	*/
	int OppositeColors(sf::Vector2i dest, Piece*
pieces[DIMENSIONS][DIMENSIONS]);
	int InRange(sf::Vector2i move);// get a move and returns 1 if
the move in the range of the board. else 0.
	int PieceMoved();
	void RotatePiece(float rot);
	int InBoard(sf::Vector2i indexes);
	void UpdatePossibleMoves(std::vector<sf::Vector2i>
*possibleMoves, int xDir, int yDir, Piece*
pieces[DIMENSIONS][DIMENSIONS], int limit);
```

```
	int squreUnderAttack(Piece* pieces[DIMENSIONS][DIMENSIONS], 
sf::Vector2i location);
	int IsEnPassent(sf::Vector2i dest, Piece* 
pieces[DIMENSIONS][DIMENSIONS]);
	int IsMovePinned(sf::Vector2i move, sf::Vector2i kingIndexes, 
Piece* pieces[DIMENSIONS][DIMENSIONS]);
	std::vector<sf::Vector2i> GetSqureesBetween(sf::Vector2i a, 
sf::Vector2i b); // gets to squres and returns all the squres 
between them.
	// virtual functions
	virtual int IsPossibleMove(sf::Vector2i dest, Piece* 
pieces[DIMENSIONS][DIMENSIONS]) = 0;
	/* A virtual function.
	 Each class that inherits piece implemnt it differently.
	 The function gets the dest move and checks if the move is 
legal.
	 returns 1 or 0.*/
	virtual std::vector<sf::Vector2i> GetPossibleMoves(Piece* 
pieces[DIMENSIONS][DIMENSIONS]) ;
	/* A virtual function.
	 The function gets the array of all the pieces and returns all 
the possible moves.
	 This function uses IsLegalMove.
	 The function is virtual becuse pown has a different 
implementation than the other pieces.
	 returns a vector off all the possible moves.*/

	std::vector<sf::Vector2i> GetValidMoves(Piece* 
pieces[DIMENSIONS][DIMENSIONS], std::vector<sf::Vector2i> kings);



	virtual std::vector<sf::Vector2i> GetCastlingMoves(Piece* 
pieces[DIMENSIONS][DIMENSIONS]);
	/* This virtual function will get possible moves for castling.
	Only king will implement this function.*/
protected:
	sf::Sprite piece; // sprite of the piece.
	sf::Vector2f position; // current position.
	int isWhite; // 1 if white, 0 if black.
	sf::Vector2i indexes; // the indexes in the board of the 
current position.
	sf::Vector2i first_indexes; // save the first position of the 
piece.
	/*the type of a piece which is represented by two letters :
	1. for color.
	2. for type of piece.
	example: black Queen -> bQ.*/
	std::string type;
	int pieceMoved;
	sf::Vector2i lastMove[2];

private:
```

```
	std::vector<sf::Vector2i> attackSqures;
	std::vector<sf::Vector2i> pinSqures;

};
```

IMPLEMENTATION

```
#include "Piece.h"



Piece::Piece(sf::Texture& pieceTex, sf::Vector2f position, int
isWhite, sf::Vector2i indexes, std::string type)
{
	this->position = position;
	this->piece.setTexture(pieceTex);
	//this->piece.setScale(1.2f, 1.2f);
	this->piece.setPosition(position.x, position.y);
	this->isWhite = isWhite;
	this->indexes = indexes;
	this->type = type;
	this->pieceMoved = 0;

}

Piece::Piece()
{
}




void Piece::Print(sf::RenderWindow* window)
{
	window->draw(piece);
}


sf::Sprite Piece::GetSprite()
{
	return this->piece;
}
sf::Vector2f Piece::GetPosition()
{
	return this->piece.getPosition();
}
void Piece::SetPosition(sf::Vector2f position)
{
	this->position.x = position.x;
	this->position.y = position.y;
	this->piece.setPosition(this->position);
}
```

```
void Piece::SetIndexes(sf::Vector2i indexes)
{
	this->indexes = indexes;
}

void Piece::SetSpriteColor(sf::Color color)
{
	this->piece.setColor(color);
}

void Piece::SetLastMove(std::pair<sf::Vector2i, sf::Vector2i>
lastMove)
{
	this->lastMove[0] = lastMove.first;
	this->lastMove[1] = lastMove.second;
}

void Piece::setPieceMoved(bool pieceMoved)
{
	this->pieceMoved = pieceMoved;
}

int Piece::OppositeColors(sf::Vector2i dest, Piece*
pieces[DIMENSIONS][DIMENSIONS])
{
	if (pieces[dest.x][dest.y]) {
		if (pieces[dest.x][dest.y]->GetType()[0] !=
			pieces[indexes.x][indexes.y]->GetType()[0]) {
			return 1;
		}
		else {
			return 0;
		}
	}
	else {
		return 1;
	}
}

int Piece::InRange(sf::Vector2i move)
{
	return move.x >= 0 && move.x < 8 && move.y >= 0 && move.y < 8;
}

int Piece::PieceMoved()
{
	return this->pieceMoved;
}

void Piece::RotatePiece(float rot)
{
	piece.rotate(rot);
}
```

```
int Piece::InBoard(sf::Vector2i indexes)
{
	return indexes.x >= 0 && indexes.x < DIMENSIONS && indexes.y
>= 0 && indexes.y < DIMENSIONS;
}

void Piece::UpdatePossibleMoves(std::vector<sf::Vector2i>
*possibleMoves, int xDir, int yDir, Piece*
pieces[DIMENSIONS][DIMENSIONS], int limit)
{
	sf::Vector2i temp = indexes;
	temp.x += xDir;
	temp.y += yDir;
	while (InBoard(temp) && limit)
	{
		if (!pieces[temp.x][temp.y])
			possibleMoves->push_back(temp);
		else
		{
			if (OppositeColors(temp, pieces))
				possibleMoves->push_back(temp);
			break;
		}
		temp.x += xDir;
		temp.y += yDir;
		limit--;
	}
}

int Piece::squreUnderAttack(Piece* pieces[DIMENSIONS][DIMENSIONS],
sf::Vector2i location)
{
	attackSqures.clear();
	pinSqures.clear();
	int dirX, dirY;
	int directions[][8] = { {-1,-1}, {-1, 1}, {1, -1}, {1, 1}, {0,
-1}, {0, 1}, {1, 0}, {-1, 0} };
	sf::Vector2i temp = location;
	//Piece* locPiece = pieces[location.x][location.y];
	//int isWhite = locPiece->IsWhite();
	int count = 0;
	for (int i = 0; i < 8; i++)
	{
		dirX = directions[i][0];
		dirY = directions[i][1];
		temp.x += dirX;
		temp.y += dirY;
		int dis = 1;
		int checkForPins = 0;
		int pinFound = 0;
		while (InBoard(temp))
		{
			if (pieces[temp.x][temp.y])
			{
```

```
                        if (pieces[temp.x][temp.y]->IsWhite() !=
isWhite)
                        {
                              int check = checkForPins;
                              switch (pieces[temp.x][temp.y]-
>GetType()[1])
                              {
                              case 'Q':
                                    if (!pinFound && checkForPins) {
                                          pinFound = 1;
                                          pinSqures.push_back(temp);
                                    }
                                    if (!checkForPins) {
                                          count++;

      attackSqures.push_back(temp);
                                    }
                                    break;
                              case 'B':
                                    if (i < 4) {
                                          if (!pinFound &&
checkForPins) {
                                                pinFound = 1;

      pinSqures.push_back(temp);
                                          }
                                          if (!checkForPins) {
                                                count++;

      attackSqures.push_back(temp);
                                          }
                                    }
                                    break;
                              case 'R':
                                    if (i >= 4) {
                                          if (!pinFound &&
checkForPins) {
                                                pinFound = 1;

      pinSqures.push_back(temp);
                                          }
                                          if (!checkForPins) {
                                                count++;

      attackSqures.push_back(temp);
                                          }
                                    }
                                    break;
                              case 'p':
                                    if (dis == 1)
                                          if (pieces[temp.x][temp.y]-
>IsWhite() && i >= 2 && i <= 3) {
                                                if (!checkForPins) {
                                                      count++;
```

```
    attackSqures.push_back(temp);
                                }
                            }
                            else if
(!pieces[temp.x][temp.y]->IsWhite() && i < 2) {
                                if (!checkForPins) {
                                    count++;

    attackSqures.push_back(temp);
                                }
                            }
                        break;
                    case 'K':
                        if (dis == 1) {
                            if (!checkForPins) {
                                count++;

    attackSqures.push_back(temp);
                            }
                        }

                    } // end switch
                    if (check == checkForPins &&
!pinFound)
                        break;
                }
                else
                {
                    if (checkForPins)
                        break;
                    checkForPins = 1;
                }

                if (pinFound)
                    break;
            }
            temp.x += dirX;
            temp.y += dirY;
            dis++;
        }// end while
        temp = location;
    }// end for
    // check for Knights :
    std::vector<sf::Vector2i> possibleMoves = { {location.x + 2,
location.y + 1}, {location.x + 2, location.y - 1 },{location.x - 2,
location.y + 1},
        {location.x - 2, location.y - 1}, {location.x + 1,
location.y + 2}, {location.x - 1, location.y + 2},
    {location.x + 1, location.y - 2}, {location.x - 1, location.y
- 2} };
    for (auto move : possibleMoves)
```

```
            if (InBoard(move) && pieces[move.x][move.y] &&
pieces[move.x][move.y]->GetType()[1] == 'N' &&
pieces[move.x][move.y]->IsWhite() != isWhite) {
                  count++;
                  attackSqures.push_back(move);
            }
      return count;
}



int Piece::IsEnPassent(sf::Vector2i dest, Piece*
pieces[DIMENSIONS][DIMENSIONS])
{

      if (isWhite)
      {
            if (this->indexes.x != 3)
                  return 0;
            if (lastMove[0] == sf::Vector2i(dest.x - 1, dest.y) &&
lastMove[1] == sf::Vector2i(dest.x + 1, dest.y) && pieces[dest.x +
1][dest.y]->GetType() == "bp")
            {
                  return 1;
            }
      }
      else
      {
            if (this->indexes.x != 4)
                  return 0;
            if (lastMove[0] == sf::Vector2i(dest.x + 1, dest.y) &&
lastMove[1] == sf::Vector2i(dest.x - 1, dest.y) && pieces[dest.x -
1][dest.y]->GetType() == "wp")
            {
                  return 1;
            }
      }
      return 0;
}
int Piece::IsMovePinned(sf::Vector2i move, sf::Vector2i kingIndexes,
Piece* pieces[DIMENSIONS][DIMENSIONS])
{
      if (GetType()[1] == 'p' && IsEnPassent(move, pieces))
      {
            auto attacker = pieces[this->indexes.x][this-
>indexes.y];
            auto defender = pieces[this->indexes.x][move.y];
            pieces[this->indexes.x][this->indexes.y] = nullptr;
            pieces[this->indexes.x][move.y] = nullptr;
            if (squreUnderAttack(pieces, kingIndexes))
            {
                  pieces[this->indexes.x][this->indexes.y] =
attacker;
```

```
                pieces[this->indexes.x][move.y] = defender;
                return true;
            }
            pieces[this->indexes.x][this->indexes.y] = attacker;
            pieces[this->indexes.x][move.y] = defender;
        }
        for (int i = 0; i < pinSqures.size(); i++)
        {
            std::vector<sf::Vector2i> squresBetween =
GetSquresBetween(kingIndexes, pinSqures[i]);
            if (std::find(squresBetween.begin(),
squresBetween.end(), this->indexes) != squresBetween.end()) {
                if (move != pinSqures[i] &&
std::find(squresBetween.begin(), squresBetween.end(), move) ==
squresBetween.end())
                    return true;
            }

        }
        return false;
}

std::vector<sf::Vector2i> Piece::GetSquresBetween(sf::Vector2i a,
sf::Vector2i b)
{
        std::vector<sf::Vector2i> squresBetween =
std::vector<sf::Vector2i>();
        int dirX, dirY;
        dirX = a.x - b.x;
        dirY = a.y - b.y;
        if (dirX) dirX /= abs(dirX);
        if (dirY) dirY /= abs(dirY);
        sf::Vector2i temp = b;
        temp.x += dirX;
        temp.y += dirY;
        while (temp != a)
        {
            squresBetween.push_back(temp);
            temp.x += dirX;
            temp.y += dirY;
        }
        return squresBetween;
}



std::vector<sf::Vector2i> Piece::GetPossibleMoves(Piece*
pieces[DIMENSIONS][DIMENSIONS])
{
        std::vector<sf::Vector2i> possibleMoves;

        for (int i = 0; i < DIMENSIONS; i++)
        {
```

```
            for (int j = 0; j < DIMENSIONS; j++)
            {
                  if (pieces[indexes.x][indexes.y]-
>IsPossibleMove(sf::Vector2i(i, j), pieces))
                  {
                        possibleMoves.push_back(sf::Vector2i(i, j));
                  }
            }
      }
      return possibleMoves;
}
```

```
std::vector<sf::Vector2i> Piece::GetValidMoves(Piece*
pieces[DIMENSIONS][DIMENSIONS], std::vector<sf::Vector2i> kings)
{
      sf::Vector2i kingIndexes;
      if (isWhite)
            kingIndexes = kings.front();
      else
            kingIndexes = kings.back();
      std::vector<sf::Vector2i> possibleMoves =
GetPossibleMoves(pieces);
      if (kingIndexes == this->indexes)
      {
            // if the piece is the king than return from all the
possible squres to move, the squres that not under attack.
            Piece* kingPiece = pieces[kingIndexes.x][kingIndexes.y];
            pieces[kingIndexes.x][kingIndexes.y] = nullptr;
            for (int i = possibleMoves.size() - 1; i >= 0; i--)
            {
                  if(squreUnderAttack(pieces, possibleMoves[i]))
                        possibleMoves.erase(possibleMoves.begin() +
i);

            }
            pieces[kingIndexes.x][kingIndexes.y] = kingPiece;
            return possibleMoves;
      }
      int count = squreUnderAttack(pieces, kingIndexes);
      // from now we check for other pieces than the king:
      if (count == 2)// in a case of double check only the king can
move.
      {
            pinSqures.clear();
            attackSqures.clear();
            return std::vector<sf::Vector2i>();
      }
      else
```

```
      {
            if (count == 1)
            {
                  // if the king is in check and the piece is not
the king, the piece can only move in order to:
                  // 1. block the check
                  // 2. capture the attacking
                  // the piece can only move if it is not a pin.
                  sf::Vector2i squre = attackSqures.back();
                  attackSqures.pop_back();

                  if (pieces[squre.x][squre.y]->GetType()[1] == 'N')
                  {
                        // if the attacking piece is knight than the
piece can only move to capture the attacking piece (only if the
piece is not pinned).
                        for (int i = possibleMoves.size() - 1; i >=
0; i--)
                        {
                              if (possibleMoves[i] != squre ||
IsMovePinned(possibleMoves[i], kingIndexes, pieces)) // check if the
possible move is not a capture.

      possibleMoves.erase(possibleMoves.begin() + i);
                        }

                  }
                  else
                  {
                        std::vector<sf::Vector2i> squresBetween =
GetSqueresBetween(kingIndexes, squre);
                        for (int i = possibleMoves.size() - 1; i >=
0; i--)
                        {
                              if (!IsMovePinned(possibleMoves[i],
kingIndexes, pieces) && possibleMoves[i] == squre) // check if the
possible move is capture.
                                    continue;
                              if (std::find(squresBetween.begin(),
squresBetween.end(), possibleMoves[i]) == squresBetween.end())

      possibleMoves.erase(possibleMoves.begin() + i);
                        }

                  }

                  pinSqures.clear();
                  return possibleMoves;
            }
            else
            {
                  // if the king is not in check than the move is
valid unless it pinned by an enemy piece..
```

```
                    for (int j = possibleMoves.size() - 1; j >= 0; j--
)
                    {
                          if(IsMovePinned(possibleMoves[j],
kingIndexes, pieces))

      possibleMoves.erase(possibleMoves.begin() + j);
                    }
                    return possibleMoves;
              }
      }

}

std::vector<sf::Vector2i> Piece::GetCastlingMoves(Piece*
pieces[DIMENSIONS][DIMENSIONS])
{
      return std::vector<sf::Vector2i>();
}



int Piece::IsWhite()
{
      return this->isWhite;
}

sf::Vector2i Piece::GetIndexes()
{
      return this->indexes;
}

std::string Piece::GetType()
{
      return this->type;
                                                                    }
```

מחלקת BISHOP

DECLARATION

```
#pragma once
#include "Piece.h"
class Bishop :
	public Piece
{
public:
	Bishop(sf::Texture& pieceTex, sf::Vector2f position, int
isWhite, sf::Vector2i indexes, std::string type);
	int IsPossibleMove(sf::Vector2i dest, Piece*
pieces[DIMENSIONS][DIMENSIONS]);
	std::vector<sf::Vector2i> GetPossibleMoves(Piece*
pieces[DIMENSIONS][DIMENSIONS]);
};
```

IMPLEMENTATION

```
#include "Bishop.h"

Bishop::Bishop(sf::Texture& pieceTex, sf::Vector2f position, int
isWhite, sf::Vector2i indexes, std::string type)
{
	this->position = position;
	this->piece.setTexture(pieceTex);
	//this->piece.setScale(0.78125f, 0.78125f);
	//this->piece.setScale(1.2f, 1.2f);
	this->piece.setPosition(position.x, position.y);
	this->isWhite = isWhite;
	this->indexes = indexes;
	this->type = type;
	this->pieceMoved = 0;
}

int Bishop::IsPossibleMove(sf::Vector2i dest, Piece*
pieces[DIMENSIONS][DIMENSIONS])
{

	sf::Vector2i temp = indexes;
	int emptyBetween = 1;
	for (int j = 0; j < DIMENSIONS; j++)
	{
		temp.x++;
		temp.y++;
		if (pieces[temp.x][temp.y] && temp != dest) {
			emptyBetween = 0;
		}
		if (temp == dest && emptyBetween)
		{
			return OppositeColors(dest, pieces);
		}
```

```
        }
        temp = indexes;
        emptyBetween = 1;
        for (int j = 0; j < DIMENSIONS; j++)
        {
                temp.x--;
                temp.y++;
                if (pieces[temp.x][temp.y] && temp != dest) {
                        emptyBetween = 0;
                }
                if (temp == dest && emptyBetween)
                {
                        return OppositeColors(dest, pieces);
                }
        }
        temp = indexes;
        emptyBetween = 1;
        for (int j = 0; j < DIMENSIONS; j++)
        {
                temp.x++;
                temp.y--;
                if (pieces[temp.x][temp.y] && temp != dest) {
                        emptyBetween = 0;
                }
                if (temp == dest && emptyBetween)
                {
                        return OppositeColors(dest, pieces);
                }
        }
        temp = indexes;
        emptyBetween = 1;
        for (int j = 0; j < DIMENSIONS; j++)
        {
                temp.x--;
                temp.y--;
                if (pieces[temp.x][temp.y] && temp != dest) {
                        emptyBetween = 0;
                }
                if (temp == dest && emptyBetween)
                {
                        return OppositeColors(dest, pieces);
                }
        }
        return 0;
}

std::vector<sf::Vector2i> Bishop::GetPossibleMoves(Piece*
pieces[DIMENSIONS][DIMENSIONS])
{
        std::vector<sf::Vector2i> possibleMoves;
        UpdatePossibleMoves(&possibleMoves, 1, 1, pieces, DIMENSIONS);
        UpdatePossibleMoves(&possibleMoves, -1, 1, pieces,
DIMENSIONS);
```

```
	UpdatePossibleMoves(&possibleMoves, 1, -1, pieces,
DIMENSIONS);
	UpdatePossibleMoves(&possibleMoves, -1, -1, pieces,
DIMENSIONS);
	return possibleMoves;
}
```

מחלקת KING

DECLARATION

```
#pragma once
#include "Piece.h"
class King :
    public Piece
{
public:
    King(sf::Texture& pieceTex, sf::Vector2f position, int isWhite,
sf::Vector2i indexes, std::string type);
    int IsPossibleMove(sf::Vector2i dest, Piece*
pieces[DIMENSIONS][DIMENSIONS]);
    std::vector<sf::Vector2i> GetPossibleMoves(Piece*
pieces[DIMENSIONS][DIMENSIONS]);
private:

};
```

IMPLEMENTATION

```
#include "King.h"

King::King(sf::Texture& pieceTex, sf::Vector2f position, int
isWhite, sf::Vector2i indexes, std::string type)
{
      this->position = position;
      this->piece.setTexture(pieceTex);
      //this->piece.setScale(0.78125f, 0.78125f);
      //this->piece.setScale(1.2f, 1.2f);
      this->piece.setPosition(position.x, position.y);
      this->isWhite = isWhite;
      this->indexes = this->first_indexes = indexes;
      this->type = type;
      this->pieceMoved = 0;
}

int King::IsPossibleMove(sf::Vector2i dest, Piece*
pieces[DIMENSIONS][DIMENSIONS])
{
      if (this->indexes != this->first_indexes)
            pieceMoved = 1;

      std::vector<sf::Vector2i> possibleMoves = {
            {this->indexes.x + 1, this->indexes.y},
            {this->indexes.x + 1, this->indexes.y + 1},
            {this->indexes.x + 1, this->indexes.y - 1},
            {this->indexes.x - 1, this->indexes.y},
            {this->indexes.x - 1, this->indexes.y + 1},
            {this->indexes.x - 1, this->indexes.y - 1},
```

```
            {this->indexes.x, this->indexes.y + 1},
            {this->indexes.x, this->indexes.y - 1},
      };
      for (auto move : possibleMoves) {
            if (move == dest)
            {
                  if (OppositeColors(dest, pieces)) {
                        return 1;
                  }
                  else return 0;
            }
      }
      possibleMoves.clear();
      possibleMoves.push_back({ this->indexes.x, this->indexes.y + 2
});
      possibleMoves.push_back({ this->indexes.x, this->indexes.y - 2
});
      if (possibleMoves[0] == dest) // check right side castling
      {
            if (!pieces[this->indexes.x][this->indexes.y + 1] &&
!pieces[this->indexes.x][this->indexes.y + 2]
                  && !pieceMoved && pieces[this->indexes.x][this-
>indexes.y + 3] && !pieces[this->indexes.x][this->indexes.y + 3]-
>PieceMoved())
                  return 2;
      }
      if (possibleMoves[1] == dest) // check left side castling
      {
            if (!pieces[this->indexes.x][this->indexes.y - 1] &&
!pieces[this->indexes.x][this->indexes.y - 2] && !pieces[this-
>indexes.x][this->indexes.y - 3]
                  && !pieceMoved && pieces[this->indexes.x][this-
>indexes.y - 4] && !pieces[this->indexes.x][this->indexes.y - 4]-
>PieceMoved())
                  return 2;
      }

      return 0;
}

std::vector<sf::Vector2i> King::GetPossibleMoves(Piece*
pieces[DIMENSIONS][DIMENSIONS])
{
      if (this->indexes != this->first_indexes)
            pieceMoved = 1;
      std::vector<sf::Vector2i> possibleMoves;
      UpdatePossibleMoves(&possibleMoves, 1, 0, pieces, 1);
      UpdatePossibleMoves(&possibleMoves, -1, 0, pieces, 1);
      UpdatePossibleMoves(&possibleMoves, 0, 1, pieces, 1);
      UpdatePossibleMoves(&possibleMoves, 0, -1, pieces, 1);
      UpdatePossibleMoves(&possibleMoves, 1, -1, pieces, 1);
      UpdatePossibleMoves(&possibleMoves, 1, 1, pieces, 1);
      UpdatePossibleMoves(&possibleMoves, -1, -1, pieces, 1);
      UpdatePossibleMoves(&possibleMoves, -1, 1, pieces, 1);
```

```
	int yDir = isWhite ? 1 : -1;
	sf::Vector2i checkSqure = this->indexes;
	checkSqure.y += yDir;
	if (!squreUnderAttack(pieces, this->indexes) &&
!squreUnderAttack(pieces, checkSqure))
	{

		// check right side castling :
		if (!pieces[this->indexes.x][this->indexes.y + 1] &&
!pieces[this->indexes.x][this->indexes.y + 2]
			&& !pieceMoved && pieces[this->indexes.x][this-
>indexes.y + 3] && pieces[this->indexes.x][this->indexes.y + 3]-
>GetType()[1] == 'R' &&
			!pieces[this->indexes.x][this->indexes.y + 3]-
>PieceMoved())
			possibleMoves.push_back({ this->indexes.x, this-
>indexes.y + 2 });
		// check queen side castling :
		if (!pieces[this->indexes.x][this->indexes.y - 1] &&
!pieces[this->indexes.x][this->indexes.y - 2] && !pieces[this-
>indexes.x][this->indexes.y - 3]
			&& !pieceMoved && pieces[this->indexes.x][this-
>indexes.y - 4] && pieces[this->indexes.x][this->indexes.y - 4]-
>GetType()[1] == 'R' &&
			!pieces[this->indexes.x][this->indexes.y - 4]-
>PieceMoved())
			possibleMoves.push_back({ this->indexes.x, this-
>indexes.y - 2 });
	}

	return possibleMoves;
}
```

מחלקת QUEEN

DECLARATION

```
#pragma once
#include "Piece.h"
class Queen :
    public Piece
{
public:
    Queen(sf::Texture& pieceTex, sf::Vector2f position, int isWhite,
sf::Vector2i indexes, std::string type);
    int IsPossibleMove(sf::Vector2i dest, Piece*
pieces[DIMENSIONS][DIMENSIONS]);
    std::vector<sf::Vector2i> GetPossibleMoves(Piece*
pieces[DIMENSIONS][DIMENSIONS]);
};
```

IMPLEMENTATION

```
#include "Queen.h"


Queen::Queen(sf::Texture& pieceTex, sf::Vector2f position, int
isWhite, sf::Vector2i indexes, std::string type)
{
      this->position = position;
      this->piece.setTexture(pieceTex);
      //this->piece.setScale(0.75f, 0.75f);
      //this->piece.setScale(1.2f, 1.2f);
      this->piece.setPosition(position.x, position.y);
      this->isWhite = isWhite;
      this->indexes = indexes;
      this->type = type;
}

int Queen::IsPossibleMove(sf::Vector2i dest, Piece*
pieces[DIMENSIONS][DIMENSIONS])
{
      sf::Vector2i temp = indexes;
      int emptyBetween = 1;
      for (int j = 0; j < DIMENSIONS; j++)
      {
            temp.x++;
            temp.y++;
            if (pieces[temp.x][temp.y] && temp != dest) {
                  emptyBetween = 0;
            }
            if (temp == dest && emptyBetween)
            {
                  return OppositeColors(dest, pieces);
```

```
        }
    }
    temp = indexes;
    emptyBetween = 1;
    for (int j = 0; j < DIMENSIONS; j++)
    {
        temp.x--;
        temp.y++;
        if (pieces[temp.x][temp.y] && temp != dest) {
            emptyBetween = 0;
        }
        if (temp == dest && emptyBetween)
        {
            return OppositeColors(dest, pieces);
        }
    }
    temp = indexes;
    emptyBetween = 1;
    for (int j = 0; j < DIMENSIONS; j++)
    {
        temp.x++;
        temp.y--;
        if (pieces[temp.x][temp.y] && temp != dest) {
            emptyBetween = 0;
        }
        if (temp == dest && emptyBetween)
        {
            return OppositeColors(dest, pieces);
        }
    }
    temp = indexes;
    emptyBetween = 1;
    for (int j = 0; j < DIMENSIONS; j++)
    {
        temp.x--;
        temp.y--;
        if (pieces[temp.x][temp.y] && temp != dest) {
            emptyBetween = 0;
        }
        if (temp == dest && emptyBetween)
        {
            return OppositeColors(dest, pieces);
        }
    }
    temp = indexes;
    emptyBetween = 1;
    for (int j = 0; j < DIMENSIONS; j++)
    {
        temp.x++;
        if (pieces[temp.x][temp.y] && temp != dest) {
            emptyBetween = 0;
        }
        if (temp == dest && emptyBetween)
        {
```

```
                return OppositeColors(dest, pieces);
            }
        }
        temp = indexes;
        emptyBetween = 1;
        for (int j = 0; j < DIMENSIONS; j++)
        {
            temp.x--;
            if (pieces[temp.x][temp.y] && temp != dest) {
                emptyBetween = 0;
            }
            if (temp == dest && emptyBetween)
            {
                return OppositeColors(dest, pieces);
            }
        }
        temp = indexes;
        emptyBetween = 1;
        for (int j = 0; j < DIMENSIONS; j++)
        {
            temp.y--;
            if (pieces[temp.x][temp.y] && temp != dest) {
                emptyBetween = 0;
            }
            if (temp == dest && emptyBetween)
            {
                return OppositeColors(dest, pieces);
            }
        }
        temp = indexes;
        emptyBetween = 1;
        for (int j = 0; j < DIMENSIONS; j++)
        {
            temp.y++;
            if (pieces[temp.x][temp.y] && temp != dest) {
                emptyBetween = 0;
            }
            if (temp == dest && emptyBetween)
            {
                return OppositeColors(dest, pieces);
            }
        }
        return 0;
}

std::vector<sf::Vector2i> Queen::GetPossibleMoves(Piece*
pieces[DIMENSIONS][DIMENSIONS])
{
        std::vector<sf::Vector2i> possibleMoves;
        UpdatePossibleMoves(&possibleMoves, 1, 0, pieces, DIMENSIONS);
        UpdatePossibleMoves(&possibleMoves, -1, 0, pieces,
DIMENSIONS);
        UpdatePossibleMoves(&possibleMoves, 0, 1, pieces, DIMENSIONS);
```

```
	UpdatePossibleMoves(&possibleMoves, 0, -1, pieces,
DIMENSIONS);
	UpdatePossibleMoves(&possibleMoves, 1, 1, pieces, DIMENSIONS);
	UpdatePossibleMoves(&possibleMoves, 1, -1, pieces,
DIMENSIONS);
	UpdatePossibleMoves(&possibleMoves, -1, 1, pieces,
DIMENSIONS);
	UpdatePossibleMoves(&possibleMoves, -1, -1, pieces,
DIMENSIONS);
	return possibleMoves;
}
```

מחלקת KNIGHT

DECLARATION

```
#pragma once
#include "Piece.h"
class Knight :
    public Piece
{
public:
    Knight(sf::Texture& pieceTex, sf::Vector2f position, int
isWhite, sf::Vector2i indexes, std::string type);
    int IsPossibleMove(sf::Vector2i dest, Piece*
pieces[DIMENSIONS][DIMENSIONS]);
    std::vector<sf::Vector2i> GetPossibleMoves(Piece*
pieces[DIMENSIONS][DIMENSIONS]);
};
```

IMPLEMENTATION

```
#include "Knight.h"

Knight::Knight(sf::Texture& pieceTex, sf::Vector2f position, int
isWhite, sf::Vector2i indexes, std::string type)
{
      this->position = position;
      this->piece.setTexture(pieceTex);
      //this->piece.setScale(0.78125f, 0.78125f);
      //this->piece.setScale(1.2f, 1.2f);
      this->piece.setPosition(position.x, position.y);
      this->isWhite = isWhite;
      this->indexes = indexes;
      this->type = type;
      this->pieceMoved = 0;
}

int Knight::IsPossibleMove(sf::Vector2i dest, Piece*
pieces[DIMENSIONS][DIMENSIONS])
{
      std::vector<sf::Vector2i> possibleMoves = { {this->indexes.x +
2, this->indexes.y + 1}, {this->indexes.x + 2, this->indexes.y - 1
},
      {this->indexes.x - 2, this->indexes.y + 1}, {this->indexes.x -
2, this->indexes.y - 1},
      {this->indexes.x + 1, this->indexes.y + 2}, {this->indexes.x -
1, this->indexes.y + 2},
      {this->indexes.x + 1, this->indexes.y - 2}, {this->indexes.x -
1, this->indexes.y - 2} };

      for (auto move : possibleMoves) {
            if (move == dest)
```

```
            {
                  return OppositeColors(dest, pieces);
            }
      }


      return 0;
}

std::vector<sf::Vector2i> Knight::GetPossibleMoves(Piece*
pieces[DIMENSIONS][DIMENSIONS])
{
      std::vector<sf::Vector2i> possibleMoves = { {this->indexes.x +
2, this->indexes.y + 1}, {this->indexes.x + 2, this->indexes.y - 1
},
      {this->indexes.x - 2, this->indexes.y + 1}, {this->indexes.x -
2, this->indexes.y - 1},
      {this->indexes.x + 1, this->indexes.y + 2}, {this->indexes.x -
1, this->indexes.y + 2},
      {this->indexes.x + 1, this->indexes.y - 2}, {this->indexes.x -
1, this->indexes.y - 2} };
      for (int i = possibleMoves.size()-1; i >= 0; i--)
      {
            if (!InBoard(possibleMoves[i]) ||
!OppositeColors(possibleMoves[i], pieces))
                  possibleMoves.erase(possibleMoves.begin() + i);
      }
      return possibleMoves;
}
```

מחלקת POWN

DECLARATION

```
#pragma once
#include "Piece.h"
class Pown :
    public Piece
{
public:
    Pown(sf::Texture& pieceTex, sf::Vector2f position, int isWhite,
sf::Vector2i indexes, std::string type);
    int IsPossibleMove(sf::Vector2i dest, Piece*
pieces[DIMENSIONS][DIMENSIONS]);
    std::vector<sf::Vector2i> GetPossibleMoves(Piece*
pieces[DIMENSIONS][DIMENSIONS]);
```

IMPLEMENTATION

```
#include "Pown.h"


Pown::Pown(sf::Texture& pieceTex, sf::Vector2f position, int
isWhite, sf::Vector2i indexes, std::string type)
{
      this->position = position;
      this->piece.setTexture(pieceTex);
      //this->piece.setScale(0.78125f, 0.78125f);
      //this->piece.setScale(1.2f, 1.2f);
      this->piece.setPosition(position.x, position.y);
      this->isWhite = isWhite;
      this->indexes = this->first_indexes = indexes;
      this->type = type;
      if (isWhite)
            this->first_indexes.x = 6;
      else
            this->first_indexes.x = 1;
      this->pieceMoved = 0;

}

int Pown::IsPossibleMove(sf::Vector2i dest, Piece*
pieces[DIMENSIONS][DIMENSIONS])
{
      sf::Vector2i possibleMoves[4];
      if (isWhite)
      {
            possibleMoves[0] = { this->indexes.x - 1, this-
>indexes.y + 1 };
```

```
            possibleMoves[1] = { this->indexes.x - 1, this-
>indexes.y - 1 };
            possibleMoves[2] = { this->indexes.x - 1, this-
>indexes.y };
            possibleMoves[3] = { this->indexes.x - 2, this-
>indexes.y };
            if (this->indexes.x != 6)
                  pieceMoved = 1;
      }
      else
      {
            possibleMoves[0] = { this->indexes.x + 1, this-
>indexes.y + 1 };
            possibleMoves[1] = { this->indexes.x + 1, this-
>indexes.y - 1 };
            possibleMoves[2] = { this->indexes.x + 1, this-
>indexes.y };
            possibleMoves[3] = { this->indexes.x + 2, this-
>indexes.y };
            if (this->indexes.x != 1)
                  pieceMoved = 1;

      }

      for (int i = 0; i < 2; i++)
      {
            if (possibleMoves[i] == dest)
            {
                  if (pieces[dest.x][dest.y]) // check if dest squre
is not empty
                  {
                        if (pieces[indexes.x][indexes.y]->IsWhite()
!= pieces[dest.x][dest.y]->IsWhite())
                        {
                              //pieceMoved = 1;
                              return 1;
                        }

                  }
                  else
                  {
                        // check for potential en passent:
                        return IsEnPassent(dest, pieces);
                  }
            }
      }
      for (int i = 2; i < 4; i++)
      {
            if (possibleMoves[i] == dest)
            {
                  if (i == 2)
                  {
                        if (!pieces[dest.x][dest.y]) {
```

```
                                return 1; // returns 1 only if dest
squre is empty.
                        }
                }
                else
                {
                        if (!pieceMoved)
                        {
                                if(isWhite)
                                        return !pieces[dest.x][dest.y]
&& !pieces[dest.x + 1][dest.y];
                                else
                                        return !pieces[dest.x][dest.y]
&& !pieces[dest.x - 1][dest.y];
                        }
                }
            }
        }
        return 0;
}

std::vector<sf::Vector2i> Pown::GetPossibleMoves(Piece*
pieces[DIMENSIONS][DIMENSIONS])
{
        std::vector<sf::Vector2i> possibleMoves;
        if (!pieceMoved && this->indexes != this->first_indexes)
                pieceMoved = 1;
        /*if(!pieceMoved)
                if(isWhite)
                        if (this->indexes.x != 6)
                                pieceMoved = 1;
                        else
                                if (this->indexes.x != 1)
                                        pieceMoved = 1;*/

        int yDir = 1, xDir;
        if (isWhite)
                xDir = -1;
        else
                xDir = 1;
        sf::Vector2i temp = this->indexes;
        temp.x += xDir;
        if (InBoard(temp) && !pieces[temp.x][temp.y])
                possibleMoves.push_back(temp);
        temp.x += xDir;
        if (InBoard(temp) && !pieceMoved && !pieces[temp.x][temp.y] &&
!pieces[temp.x - xDir][temp.y])
                possibleMoves.push_back(temp);
        temp = this->indexes;
        for (int i = 0; i < 2; i++)
        {
                temp.y += yDir;
                temp.x += xDir;
                if (InBoard(temp) && OppositeColors(temp, pieces))
```

```
                    if (pieces[temp.x][temp.y] || IsEnPassent(temp,
pieces))
                        possibleMoves.push_back(temp);
            yDir *= -1;
            temp = this->indexes;
        }
        return possibleMoves;
}
```

מחלקת ROCK

DECLARATION

```
#pragma once
#include "Piece.h"
class Rock :
    public Piece
{
public:
    Rock(sf::Texture& pieceTex, sf::Vector2f position, int isWhite,
sf::Vector2i indexes, std::string type);
    int IsPossibleMove(sf::Vector2i dest, Piece*
pieces[DIMENSIONS][DIMENSIONS]);
    std::vector<sf::Vector2i> GetPossibleMoves(Piece*
pieces[DIMENSIONS][DIMENSIONS]);
};
```

IMPLEMENTATION

```
#include "Rock.h"



Rock::Rock(sf::Texture& pieceTex, sf::Vector2f position, int
isWhite, sf::Vector2i indexes, std::string type)
{
      this->position = position;
      this->piece.setTexture(pieceTex);
      //this->piece.setScale(0.78125f, 0.78125f);
      //this->piece.setScale(1.2f, 1.2f);
      this->piece.setPosition(position.x, position.y);
      this->isWhite = isWhite;
      this->indexes = this->first_indexes =  indexes;
      this->type = type;
      this->pieceMoved = 0;
}

int Rock::IsPossibleMove(sf::Vector2i dest, Piece*
pieces[DIMENSIONS][DIMENSIONS])
{
      if (this->indexes != this->first_indexes)
            pieceMoved = 1;
      sf::Vector2i temp = indexes;
      int emptyBetween = 1;
      for (int j = 0; j < DIMENSIONS; j++)
      {
            temp.x++;
```

```
            if (pieces[temp.x][temp.y] && temp != dest) {
                emptyBetween = 0;
            }
            if (temp == dest && emptyBetween)
            {
                if (OppositeColors(dest, pieces)) {
                    return 1;
                }
                else return 0;

            }
        }
        temp = indexes;
        emptyBetween = 1;
        for (int j = 0; j < DIMENSIONS; j++)
        {
            temp.x--;
            if (pieces[temp.x][temp.y] && temp != dest) {
                emptyBetween = 0;
            }
            if (temp == dest && emptyBetween)
            {
                if (OppositeColors(dest, pieces)) {
                    return 1;
                }
                else return 0;
            }
        }
        temp = indexes;
        emptyBetween = 1;
        for (int j = 0; j < DIMENSIONS; j++)
        {
            temp.y--;
            if (pieces[temp.x][temp.y] && temp != dest) {
                emptyBetween = 0;
            }
            if (temp == dest && emptyBetween)
            {
                if (OppositeColors(dest, pieces)) {
                    return 1;
                }
                else return 0;
            }
        }
        temp = indexes;
        emptyBetween = 1;
        for (int j = 0; j < DIMENSIONS; j++)
        {
            temp.y++;
            if (pieces[temp.x][temp.y] && temp != dest) {
                emptyBetween = 0;
            }
            if (temp == dest && emptyBetween)
            {
```

```
                    if (OppositeColors(dest, pieces)) {
                          return 1;
                    }
                    else return 0;
              }
      }
      return 0;
}

std::vector<sf::Vector2i> Rock::GetPossibleMoves(Piece*
pieces[DIMENSIONS][DIMENSIONS])
{
      if (this->indexes != this->first_indexes)
            pieceMoved = 1;
      std::vector<sf::Vector2i> possibleMoves;
      UpdatePossibleMoves(&possibleMoves, 1, 0, pieces, DIMENSIONS);
      UpdatePossibleMoves(&possibleMoves, -1, 0, pieces,
DIMENSIONS);
      UpdatePossibleMoves(&possibleMoves, 0, 1, pieces, DIMENSIONS);
      UpdatePossibleMoves(&possibleMoves, 0, -1, pieces,
DIMENSIONS);
      return possibleMoves;
}
```

מחלקת MENU:

DECLARATION

```
#pragma once
#include <SFML/Graphics.hpp>
#include <iostream>
#include "Game.h"
#include "AiGame.h"
#include "OfflineGame.h"
#include "Constants.h"



class Menu
{
public:
	Menu();
	void Start();
	void LoadData();
	void LoadSprites();
	void UpdateEvents();
	void Print();
	enum States { OFFLINE, AI};
private:
	// window
	sf::RenderWindow* window;
	// game
	Game* game;
	// variables
	std::vector<Button*> buttons;

	sf::Color btnColor;
	int limitFrom;
	int limitTo;
	// textures
	sf::Texture chessBGTex;
	sf::Texture chessLogoTex;
	// sprites
	sf::Sprite chessBG;
	sf::Sprite chessLogo;



};
```

IMPLEMENTATION

```
#include "Menu.h"


Menu::Menu()
{
	window = new sf::RenderWindow(sf::VideoMode(SCREEN_SIZE,
SCREEN_SIZE), "Menu");
	btnColor = sf::Color(255, 248, 220);
}

void Menu::Start()
{
	LoadData();
	LoadSprites();
	while (window->isOpen())
	{
		UpdateEvents();
		Print();
	}

}

void Menu::LoadData()
{
	buttons.push_back(new Button("Play Offline", 20, { SCREEN_SIZE
/ 4, SCREEN_SIZE / 16 }, sf::Color::Black, btnColor, { SCREEN_SIZE *
0.15, SCREEN_SIZE * 0.01 }));
	buttons.push_back(new Button("Play Ai", 20, { SCREEN_SIZE / 4,
SCREEN_SIZE / 16 }, sf::Color::Black, btnColor, { SCREEN_SIZE *
0.85, SCREEN_SIZE * 0.01 }));
	chessBGTex.loadFromFile("Assets/ChessBG1.jpg");
	chessLogoTex.loadFromFile("Assets/ChessLogo1.png");
}

void Menu::LoadSprites()
{
	chessBG.setTexture(chessBGTex);
	chessLogo.setTexture(chessLogoTex);
	chessLogo.setPosition({ SCREEN_SIZE * 0.5f -
chessLogo.getGlobalBounds().width / 2 }, { SCREEN_SIZE * 0.05 });
}

void Menu::UpdateEvents()
{
	sf::Event event;
	while (window->pollEvent(event))
	{
		if (event.type == sf::Event::Closed)
			window->close();
		if (event.type == sf::Event::MouseButtonPressed)
			for (int i = 0; i < buttons.size(); i++)
				if ((buttons[i]->IsMouseHover(*window)))
```

```
                        {
                              window->close();
                              if (i == States::OFFLINE) {
                                    game = new OfflineGame();
                                    game->Start();
                              }
                              else if (i == States::AI) {
                                    game = new AiGame();
                                    game->Start();
                              }


                        }
            if (event.type == sf::Event::MouseMoved)
                  for (auto btn : buttons)
                        btn->HandleMouseHover(*window);
            if (event.type == sf::Event::Resized)
            {
                  // update the view to the new size of the window
                  sf::View view;
                  if (event.size.width < event.size.height)
                  {
                        sf::FloatRect visibleArea(0.f, 0.f,
SCREEN_SIZE, (SCREEN_SIZE * event.size.height) / event.size.width);
                        view = sf::View(visibleArea);
                        window->setView(view);

                  }
                  else
                  {
                        sf::FloatRect visibleArea(0.f, 0.f,
(SCREEN_SIZE * event.size.width) / event.size.height, SCREEN_SIZE);
                        view = sf::View(visibleArea);
                        window->setView(view);
                  }
                  view.setCenter(sf::Vector2f(SCREEN_SIZE / 2,
SCREEN_SIZE / 2));
                  window->setView(view);


            }

      }
}

void Menu::Print()
{
      window->clear();
      window->draw(chessBG);
      window->draw(chessLogo);
      for (auto btn : buttons)
            btn->DrawTo(*window);
      window->display();
}
```

MAIN.CPP

```
#include "Menu.h"


int main()
{
	Menu menu;
	menu.Start();
	return 0;
}
```

ביבליוגרפיה

Minimax Algorithm in Game Theory | Set 1 (Introduction) - GeeksforGeeks

Alpha-Beta - Chessprogramming wiki

Artificial Intelligence | Alpha-Beta Pruning - Javatpoint

Minimax Algorithm in Game Theory | Set 4 (Alpha-Beta Pruning) - GeeksforGeeks

chess (cornell.edu)

Alpha–beta pruning - Wikipedia

Alpha Beta Pruning in AI - Great Learning (mygreatlearning.com)

Alpha Beta Pruning in Minimax Algorithm (opengenus.org)

Algorithms Explained – minimax and alpha-beta pruning - Bing video

6. Search: Games, Minimax, and Alpha-Beta - Bing video

Perft Results - Chessprogramming wiki