**עבודת גמר**

**התמחות: תכנות משחקי מחשב**

**שם הפרויקט: שחמט**

**מגיש : עזרא גבאי**

**מנחה : איציק גולדברגר**

**שנה''ל תשע"ח**

**תוכן עניינים**

**הצעת פרויקט**

|  |  |
| --- | --- |
| **פרק** | **מס' עמוד** |
| 1. מבוא |  |
| * 1. מטרה | 2 |
| * 1. תיאור המערכת | 2 |
| * 1. מבנה המסמך | 3 |
| * 1. ביבליוגרפיה | 3 |
| 1. מפרטי תוכנה |  |
| * 1. תיאור כללי | 3 |
| * 1. ממשק אדם מכונה MMI | 4 |
| * 1. יכולות פונקציונליות | 9 |
| * 1. תיאור אלגוריתמים   2. רשימת נתונים | 10  18 |
| 1. תכנון |  |
| * 1. חלוקה למודולים | 21 |
| * 1. פירוט המודולים | 23 |
| * 1. מבנה נתונים | 28 |
| * 1. Coding Style | 28 |
| 1. מדריך למשתמש |  |
| * 1. הוראות התקנה (Setup) | 30 |
| * 1. הוראות הפעלה | 30 |
| נספח 1 Listing | 31 |

# 1**. מבוא**

## **1.1 מטרה**

המשחק שנבחר לפרויקט הוא שחמט. מטרת המשחק הינה חשיבה אסטרטגית לטווח ארוך של המשתמש. המשחק מיישם את הידע שרכשתי ולמדתי במהלך שנת הלימודים בשפת C# ועל תכנות מונחה אובייקטים.

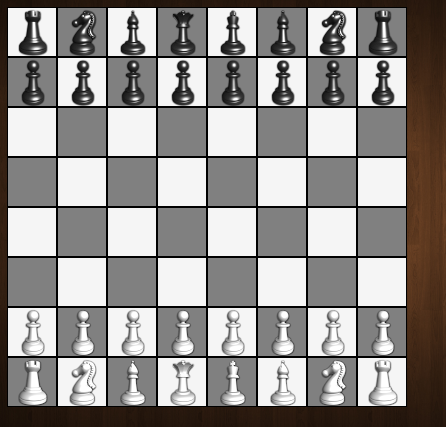
**1.2 תיאור המערכת**

משחק השחמט משוחק על לוח ריבועי המחולק ל-64 משבצות, הצבועות בצבעי שחור ולבן לסירוגין (או, בכלליות, צבע בהיר וצבע כהה) - כל ארבע הצלעות של משבצת לבנה גובלות במשבצת שחורה ולהפך. מניחים את הלוח כך שהמשבצת הימנית ביותר בשורה הקרובה לכל שחקן היא לבנה.

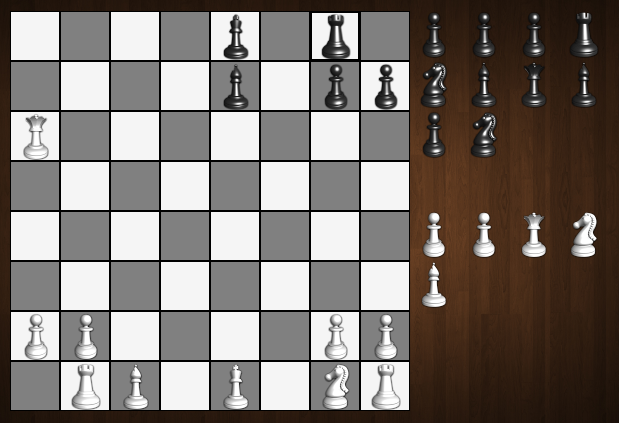
על הלוח מניחים את הכלים בשורה הקרובה ביותר לכל שחקן בכל צד, מהחוץ לפנים:

2 צריחים על המשבצות החיצוניות ביותר, לאחר מכן 2 פרשים, לאחר מכן 2 רצים, ובשתי המשבצות המרכזיות הנותרות בשורה: מניחים את המלכה על הצבע שלה (מלכה לבנה על משבצת לבנה, ומלכה שחורה על משבצת שחורה), ואת המלך מניחים על המשבצת היחידה שנותרה בשורה.

על השורה השנייה בכל צד מניחים את הרגלים לכל רוחב השורה (על כל שמונת המשבצות בשורה).



במהלך המשחק כל חלק נע כפי יכולתו (פירוט יכולת תזוזת החלקים בהמשך), וכאשר חלק זז אל משבצת בה כבר נמצא חלק של היריב, החלק "אוכל" את החלק של היריב. פירוש פעולה זו, היא שהחלק מחליף את חלקו של היריב במשבצת, וחלקו של היריב מוצא מהלוח, ומונח לצידו:



**1.3 מבנה המסמך**

הספר כולל ארבעה פרקים:

הפרק הראשון - מציג מבוא למשחק עצמו: מטרת המשחק, תיאור כללי של המשחק, מבנה מסמך ורשימת מקורות שנעזרתי לצורך הכנת הפרויקט.

הפרק השני - מפרט את התוכנה. הוא כולל תיאור מפורט יותר של המשחק (מצבי משחק שונים ותיאור נקודות יתרון ותורפה של שחקנים) מממשק האדם מכונה, יכולות של המערכת, תיאור האלגוריתם ורשימת הנתונים.

הפרק השלישי - מפרט את התכנון של המערכת. הוא מתאר את המודולים השונים ואת מבנה נתונים.

הפרק הרביעי - מכיל הוראות הפעלה. הוא מפרט את הוראות ההתקנה ואת הוראות ההפעלה.

## בסוף ארבעת הפרקים מופיע נספח הכולל את קוד המקור המלא של התוכנה.

## **1.4 ביבליוגרפיה**

* מדריך לשחמט באתר ויקיפדיה: https://he.wikipedia.org/wiki/שחמט

# **2. מפרטי תוכנה**

## **2.1 תיאור כללי**

יישום זה מדמה את המשחק המוכר שחמט, במערכת ההפעלה Windows. יישום זה נכתב בסביבת עבודה Visual Studio C# 2015. המשחק משוחק על ידי שני שחקנים על לוח אחד (אדם נגד מחשב, או אדם נגד אדם באותה תוכנה), לוח המראה את חלקי המשחק הפועלים של השחקנים, ולידו החלקים ה"אכולים" של שני היריבים.

## **2.1.1 שלבי המשחק**

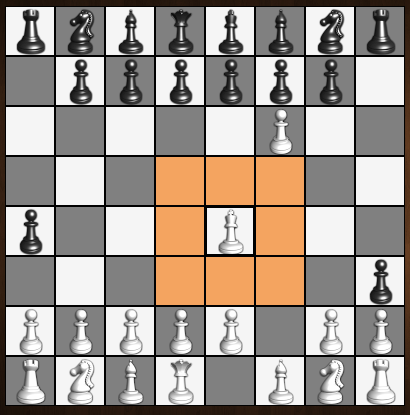
כאשר המלך נמצא במצב של "שח", כלומר במצב שבו היריב יוכל "לאכלו" בתורו הבא, השחקן המאוים חייב לשנות מצב זה על ידי הזזת המלך, חסימת קו ההתקפה של הכלי המאיים, או על ידי "אכילת" הכלי המאיים. כאשר השחקן אינו יכול למלט את מלכו ממצב השח, נוצר מצב הנקרא "שחמט".

המשחק משוחק בתורות בין השחקנים כאשר האסטרטגיה היא ליצור מצב של "שחמט" למלך של היריב. השחקן צריך לחשב את צעדיו ולתכנן את שלבי יריביו, ובכך לקבוע את מהלכו בכל תור.

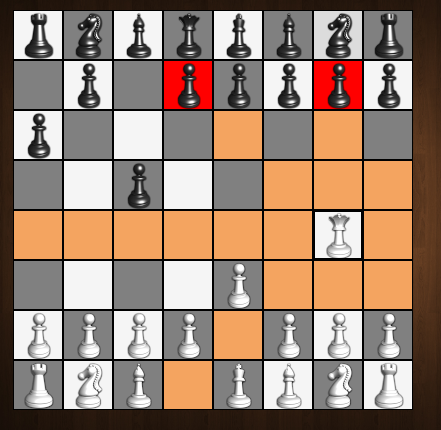
## **2.1.1.1 כללי הזזת החלקים**

במשחק שישה סוגים שונים של חלקים, בעלי כללי תזוזה שונים:

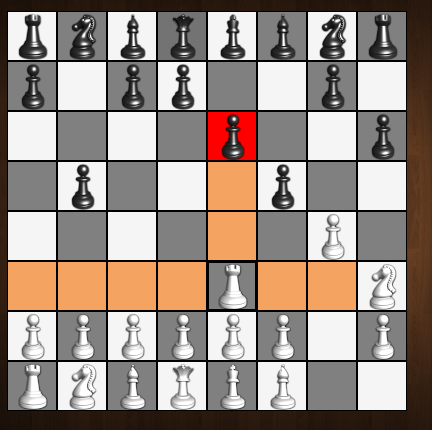
1. **מלך –** הכלי החשוב ביותר. ביכולתו לנוע לכל מקום משבצת אחת בכל כיוון למעט "הצרחה" (ראו בהמשך).



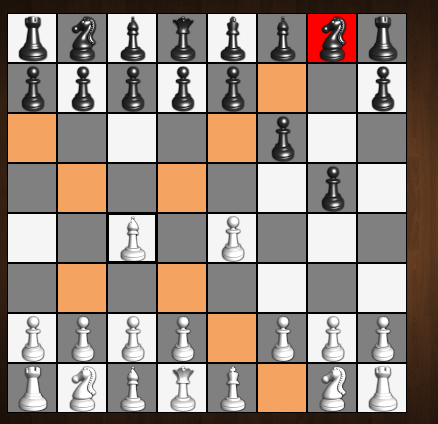
1. **מלכה –** הכלי החשוב ביותר אחרי המלך. יתרונה טעון בגמישות התנועה שלה, שכן היא יכולה לנוע בכל קו ישר (טור, שורה או אלכסון).



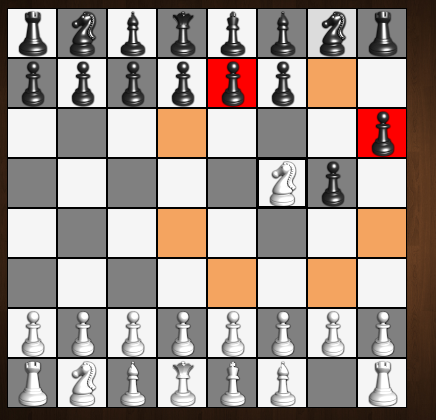
1. **צריח –** הצריח יכול לנוע רק לאורך הטורים או השורות. תפקיד מיוחד נועד לצריח בהגנת המלך במהלך המכונה "הצרחה" (ראו בהמשך).



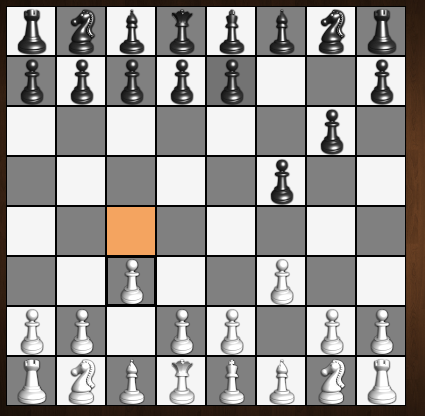
1. **רץ –** הרץ יכול לנוע רק באלכסונים.



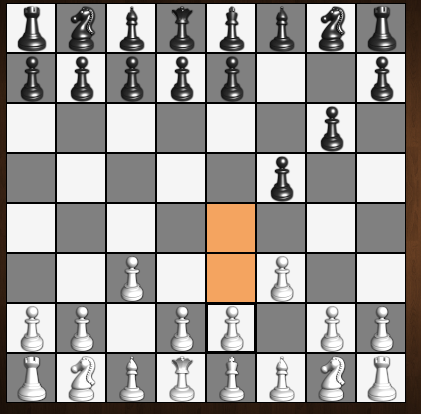
1. **פרש –** לפרש תנועה מיוחדת שמשלבת תנועה ישר ובאלכסון. לחלופין ניתן לתאר את תנועתו כאות 'ך' המסובבת בכיוונים שונים. כלים אינם חוסמים את תנועתו, משום שהוא יכול לדלג מעליהם.



1. **חייל –** האחרון בהיררכיה והמוגבל ביותר בתנועתו הוא החייל. יכול לנוע רק משבצת אחת בכל פעם ורק לכיוון אחד ("קדימה", ז"א, הכיוון הנגדי לצד הלוח בו הוא מתחיל), מה שמביא לכך שהוא צפוי להיתקע כאשר את המשבצת לפניו תופס כלי אחר. אופן ההכאה של הרגלי מתבצע באופן שונה מאופן תנועתו הרגיל, על ידי מעבר משבצת אחת באלכסון, אך ורק בכיוון התקדמותו. למרות היותו כלי חלש (בשל תנועתו המוגבלת), לרגלי יש יתרון אחד על פני שאר הכלים: אם הוא מגיע לשורה האחרונה של הלוח הוא הופך לכל כלי אחר, פרט למלך, באותו הצבע, על-פי בחירת השחקן (גם אם הכלי הנבחר טרם יצא מהמשחק). - מצב זה קרוי **"הכתרה"**. נהוג להעריך את הרגלי בנקודה אחת (ביחס אליה מוערך ערכם של שאר הכלים). מהלך מיוחד של הרגלי הוא **הכאה דרך הילוכו**.



במהלכו הראשון של הרגלי, ורק במהלך זה, הוא יכול לנוע שתי משבצות קדימה או משבצת אחת בהתאם לרצון השחקן.



במסכים המוצגים לעיל, ניתן לראות את התנועות האפשריות לכל כלי המשחק של השחקנים.

ישנם שלושה מהלכים מיוחדים בשחמט:

1. **הצרחה –** מהלך משותף של המלך והצריח הנחשב כמהלך אחד ומתבצע בשני שלבים:

**א.1.** המלך מוזז שתי משבצות לכיוון הצריח

**א.2.** הצריח מוזז למשבצת בצמודה למלך – ומעבר לו.

**הצרחה תתבצע בתנאים הבאים:**

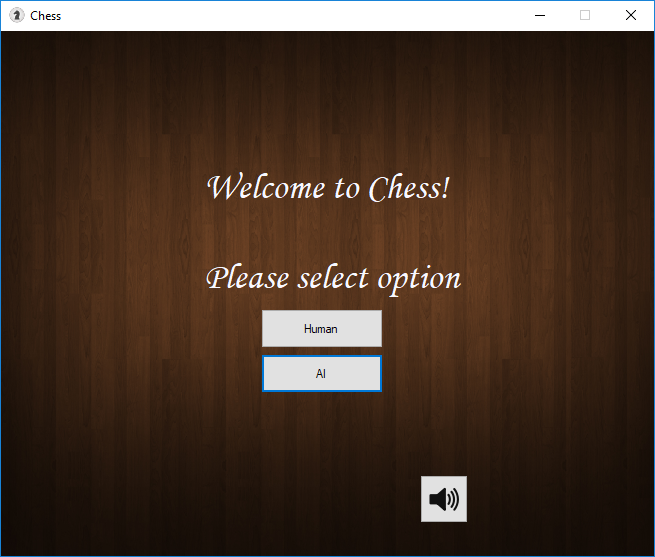
1. המלך והצריח לא הוזזו כלל מתחילת המשחק ועד כה.
2. המלך אינו נמצא באותו רגע באיום שח.
3. המלך אינו עובר על פני משבצות המאויימות על ידי כלי היריב.
4. המלך לא יימצא במצב של שח בתום ההצרחה.
5. אין כלי הנמצא בין המלך לצריח.
6. הצריח והמלך נמצאים באותה שורה.
7. **הכתרה -** **הכתרה** היא עמדה בה רגלי מגיע לשורה האחרונה בלוח (כלומר לצד הנגדי לצד ממנו הגיע). במצב זה, הרגלי מוסר מהלוח ובמקומו מציבים באופן מידי ובאותו תור כלי אחר, מאותו צבע (לפי בחירת השחקן): מלכה, צריח, רץ או פרש, אך לא מלך. ניתן להציב כלי גם אם הוא עוד לא "נאכל".
8. **"אכילה" דרך הילוכו –** כאשר חייל של היריב מבצע מהלך כפול (נע שתי משבצות קדימה, כאשר זז בפעם הראשונה), אם יש לשחקן חייל העומד במשבצת לצד החייל שהוזז, החייל של השחקן רשאי "לאכול" את חייל היריב, כאשר הוא נע האלכסון אל המשבצת מאחורי חייל היריב.

## **2.1.2 אסטרטגיית משחק**

למרות שכללי המשחק פשוטים להבנה, המשחק מצריך רמת חשיבה גבוהה, ויכולת חשיבה אסטרטגית טובה ביותר.אמנם ניתן לחשוב כמה צעדים קדימה במשחק ולחשב טקטיקות כדי ליצור מצב של שחמט נגד היריב, אך יש לזכור כי היריב מנסה לעשות את אותו הדבר בו זמנית, ויש לפעול לסכל את פעולותיו גם כן. היריב אף מנסה לסכל את אסטרטגיית השחקן, ולכן יש צורך בחשיבה מתוחכמת במיוחד, לחשב היטב את המהלכים, ובמקביל לנסות ולחזות את מהלכי היריבים, כל זאת על מנת לנסות להבטיח ניצחון במשחק.

## **2.2 ממשק אדם מכונה MMI**

## **2.2.1 תפריט ראשי של המשחק**



תצלום מסך של התפריט הראשי של המשחק.

## **2.2.2 מסך המשחק**



תצלום מסך משחק לדוגמה.

## **2.2.3 הודעות מערכת**

|  |  |
| --- | --- |
| **מתי מוצגת ההודעה?** | **ההודעה** |
| כאשר נוצר מצב של שחמט, ובכך אחד מהשחקנים מנצחים. הצבע המתאים למנצח מופיע בהודעת המערכת. |  |

## **2.2.4 הודעות מערכת קוליות**

1. מוזיקת רקע תישמע בעת הפעלת התוכנה.
2. מוזיקת רקע שנייה תישמע לאורך כל המשחק.
3. מוזיקת רקע מרגשת של ניצחון תישמע כאשר המשתמש מנצח את ה-AI, או ששחקן אחד מנצח במשחק רגיל.
4. מוזיקת רקע של הפסד (מוזיקה עצובה) תישמע כאשר המשתמש מפסיד ל-AI.

## **2.3 יכולות פונקציונאליות**

1. המערכת מספקת אפשרות של התחלת משחק חדש בין שתי משתמשים אנושיים.
2. המערכת מספקת אפשרות של התחלת משחק חדש בין המשתמש למחשב (AI).
3. בסוף כל משחק המערכת מכריזה על המנצח.
4. המערכת מאפשרת למשתמש לחזור לאחור, ולבטל את מהלכו הנוכחי (Undo Button).
5. המערכת מאפשרת התחלת המשחק מחדש (Restart Button).

## **2.4 תיאור אלגוריתמים**

**2.4.1 אלגוריתם מהלכים חוקיים**

**מטרת האלגוריתם:**

למצוא את המהלכים החוקיים שיכול כלי משחק נתון לבצע במצב המשחק הנוכחי.

**תיאור האלגוריתם:**

לפי תכונת סוג החלק, Type, האלגוריתם מחשב את המהלכים החוקיים של אותו הכלי הנתון לפי חוקי שחמט המוכרים.

**מלך –** בודק מחשב את המשבצות על הלוח שנמצאים ממש ליד המלך מסביבו. ב"צעד" אחד מהמלך. בודק גם אם המלך אינו זז, צריח מאותו הצבע של המלך לא זז, ואם ישנם חלקיםהנמצאים בין המלך לצריח, ובכך מחשב אפשרות של הצרחה.

**צריח –** מחשב את כל המשבצות באותה שורה של הצריח וכל המשבצות באותה עמודה של הצריח.

**רץ –** מחשב את החלקים הנמצאים באותו ישרים אלכסוניים עליהם עומד הרץ.

**מלכה –** מחשב את האלגוריתם של הרץ והצריח ביחד.

**פרש –** מחשב את המשבצות הנמצאים בצורת "ך" מהפרש.

**חייל / רגלי –** מחשב את המשבצת הראשונה לפני הרגלי בכיוון תנועתו. אם הרגלי לא נע עד עכשיו מחשב גם משבצת אחת נוספת בכיוון תנועתו. אם הרגלי נמצא ליד רגלי של היריב, אשר בתור האחרון נע שתי משבצות בכיוון תנועת הרגלי של היריב, מחשב גם את המשבצת הריקה באלכסון מהרגלי של השחקן הנוכחי, אשר נמצאת מאחורי הרגלי המדובר של היריב.

**2.4.2 אלגוריתם בחר מהלך (אינטליגנציה מלאכותית)**

**מטרת האלגוריתם:**

בחירת מהלך אשר האינטליגנציה המלאכותית תבצע.

**תיאור האלגוריתם:**

בניית "עץ מהלכים" אשר מתאר סדרות מהלכים אפשריים, ומחשב לפי האלגוריתם **תן ציון למצב הלוח**

את ציון המהלך שהוא יכול לעשות, ובוחר את המהלך בעל הציון הכי גבוה.

**2.4.3 אלגוריתם תן ציון למצב הלוח (אינטליגנציה מלאכותית)**

**מטרת האלגוריתם:**

לכמת את טיב מצב הלוח **לטובת המחשב**, כדי להעריך כמה כדאי למחשב לפעול דרך פעולה קודמת שביצע.

**תיאור האלגוריתם:**

האלגוריתם מעריך את הכמות וההפסד של החלקים שבבעלות המחשב, וכן לחלקים שבבעלות השחקן האנושי, ומכפיל את הפרשם במשקל כלשהו.

לכל סוג חלק בשחמט יש ערך משלו:

**מלכה – 9**

**צריח – 5**

**רץ – 3**

**פרש – 3**

**חייל / רגלי – 1**

***\*המלך הינו חסר-ערך, זאת משום שהניצחון במשחק טעון במפלתו.***

האלגוריתם מכמת את שליטת המחשב על ארבעת ריבועי מרכז הלוח, וכן את שליטת השחקן האנושי באותה הצורה, ומכפיל את הפרשם במשקל אחר מהקודם. לאחר מכן, האלגוריתם מכמת את פגיעות המלך של המחשב ואת פגיעות המלך של השחקן, ומכפיל את הפרשם במשקל שלישי.

לבסוף האלגוריתם מחשב את ערך המצב של הלוח **לטובת המחשב**, על ידי סיכום שלושת הכמויות הקודמות

## **2.5 רשימת נתונים**

**2.5.01 מחלקת Logic**

|  |  |  |  |
| --- | --- | --- | --- |
|  | **שם התכונה** | **טיפוס התכונה** | **תיאור התכונה** |
| 1 | grid | Cell[,] | מייצג את לוח משחק השחמט |
| 2 | CurrentTurn | Color | צבע של החלקים השייכים לשחקן שזהו תורו לשחק. |
| 3 | castlingHasMoved | bool[] | מערך המחזיק לכל מלך והצריחים אם זזו או לא |
| 4 | ep | EnPassant[] | מערך המחזיק את כל החיילים אשר נעו מהלך כפול, ואשר אם יש לידם חייל של היריב, הוא יוכל לבצע "אכילה" דרך הילוכו. |

**2.5.02 מחלקת MainForm**

|  |  |  |  |
| --- | --- | --- | --- |
|  | **שם התכונה** | **טיפוס התכונה** | **תיאור התכונה** |
| 1 | grid | GraphicalCell[,] | מטריצה של תאים גרפיים עליה המחלקה מציגה את החלקים שאמורים להיראות בתאים המתאימים להם, בתיאום עם המטריצה של מחלקת Logic. |
| 2 | EatenBlack | EatenPieceHolder[,] | מטריצה של תאים גרפיים, המציגה את החלקים **השחורים** ה"אכולים". |
| 3 | EatenWhite | EatenPieceHolder[,] | מטריצה של תאים גרפיים, המציגה את החלקים **הלבנים** ה"אכולים". |
| 4 | nextBlack | EatenPieceHolder | תא גרפי ריק, שהוא המקום הבא במטריצה EatenWhite, אשר בו יש להציב את החלק **הלבן** האכול הבא. |
| 5 | nextWhite | EatenPieceHolder | תא גרפי ריק, שהוא המקום הבא במטריצה EatenBlack, אשר בו יש להציב את החלק **השחור** האכול הבא. |
| 6 | bgGame | System.Media.SoundPlayer | מוזיקת הרקע במהלך המשחק. |
| 7 | undoStackLogic | Stack<Cell[,]> | ערימה של כל המטריצות grid של logic, מכל מהלך שנעשה מתחילת המשחק. |
| 8 | undoStackGraphic | Stack<GraphicCell[,]> | ערימה של כל המטריצות grid של MainForm, מכל מהלך שנעשה מתחילת המשחק. |
| 9 | logic | Logic | עצם מטיפוס Logic, אשר תפקידו לנהל את החלקים התאורטיים של המשחק, כגון: היכן מותר להזיז חלק, האם יש מנצח, וכו'. |
| 10 | ai | AI | האינטליגנציה המלאכותית, אשר מנהלת את מהלכיה לבד. במשחקים נגד שחקנים אנושיים, ערך התכונה הזו הוא null. |
| 11 | ValidMoves | LinkedList<Cell> | רשימה המכילה את כל המקומות בלוח השחמט אליו יכול החלק אשר נמצא ב-lastChosen, לזוז. |
| 12 | lastChosen | GraphicalCell | תא גרפי, אשר בו התא הגרפי האחרון שהשמשתמש לחץ עליו. |
| 13 | bgGameMusic | System.Media.SoundPlayer | מוזיקת רקע במהלך המשחק |
| 14 | OpeningWarMusic | System.Media.SoundPlayer | מוזיקת רקע בתפריט הראשי |
| 15 | DefeatMusic | System.Media.SoundPlayer | מוזיקת תבוסה |
| 16 | VictoryMusic | System.Media.SoundPlayer | מוזיקת ניצחון |
| 17 | bgGamePlaying | bool | אם מוזיקת הרקע שבמהלך המשחק מושמעת, התכונה היא אמת, אחרת היא שקר. |
| 18 | OpeningWarPlaying | bool | אם מוזיקת הרקע שבתפריט הראשי מושמעת, התכונה היא אמת, אחרת היא שקר. |
| 19 | DefeatPlaying | bool | אם מוזיקת התבוסה מושמעת, התכונה היא אמת, אחרת היא שקר. |
| 20 | VictoryPlaying | bool | אם מוזיקת הניצחון מושמעת, התכונה היא אמת, אחרת היא שקר. |

**2.5.03 מחלקת GraphicCell**

|  |  |  |  |
| --- | --- | --- | --- |
|  | **שם התכונה** | **טיפוס התכונה** | **תיאור התכונה** |
| 1 | I | int | השורה בה ממקום התא הגרפי הנוכחי |
| 2 | J | int | העמודה בה ממקום התא הגרפי הנוכחי |
| 3 | Papa | MainForm | מחלקת ה-MainForm, אשר ב-grid שלה נמצא התא הגרפי הנוכחי |
| 4 | OriginalColor | System.Drawing.Color | צבא הרקע של התא הגרפי |
| 5 | Icon | System.Drawing.Icon | הסמל של התא הגרפי |

**2.5.04 מחלקת AI**

|  |  |  |  |
| --- | --- | --- | --- |
|  | **שם התכונה** | **טיפוס התכונה** | **תיאור התכונה** |
| 1 | Color | Color | הצבע בו ה-AI משחק |
| 2 | game | Logic | המנהל של המשחק הנוכחי |
| 3 | worths | Dictionary<Type, int> | מילון ערכים מספריים של כל סוג של כלי משחק. |

**2.5.05 מחלקת EatenPieceHolder**

|  |  |  |  |
| --- | --- | --- | --- |
|  | **שם התכונה** | **טיפוס התכונה** | **תיאור התכונה** |
| 1 | I | int | השורה בה ממקום התא הגרפי הנוכחי |
| 2 | J | int | העמודה בה ממקום התא הגרפי הנוכחי |
| 3 | Papa | MainForm | מחלקת ה-MainForm, אשר ב-grid שלה נמצא התא הגרפי הנוכחי |
| 4 | Icon | System.Drawing.Icon | הסמל של התא הגרפי |

**2.5.06 מחלקת ChooseDubbedPieceWhite**

|  |  |  |  |
| --- | --- | --- | --- |
|  | **שם התכונה** | **טיפוס התכונה** | **תיאור התכונה** |
| 1 | chosen | String | מחזיק את סוג החלק אליו אנו רוצים להכתיר את החייל הלבן. |

**2.5.07 מחלקת ChooseDubbedPieceBlack**

|  |  |  |  |
| --- | --- | --- | --- |
|  | **שם התכונה** | **טיפוס התכונה** | **תיאור התכונה** |
| 1 | chosen | String | מחזיק את סוג החלק אליו אנו רוצים להכתיר את החייל השחור. |

**2.5.08 מחלקת Cell**

|  |  |  |  |
| --- | --- | --- | --- |
|  | **שם התכונה** | **טיפוס התכונה** | **תיאור התכונה** |
| 1 | I | Int | מספר השורה של התא בלוח השחמט. |
| 2 | J | Int | מספר העמודה של התא בלוח השחמט. |
| 3 | piece | Piece | כלי המשחק שעומד על המשבצת אותה מייצגת המחלקה. |

**2.5.09 מחלקת Piece**

|  |  |  |  |
| --- | --- | --- | --- |
|  | **שם התכונה** | **טיפוס התכונה** | **תיאור התכונה** |
| 1 | Type | Type | סוג החלק. |
| 2 | Color | Color | צבע החלק. |

**2.5.10 מחלקת EnPassant**

|  |  |  |  |
| --- | --- | --- | --- |
|  | **שם התכונה** | **טיפוס התכונה** | **תיאור התכונה** |
| 1 | cell | Cell | התא בו יש חייל עם |
| 2 | isActive | bool |  |

**2.5.11 מחלקת Move**

|  |  |  |  |
| --- | --- | --- | --- |
|  | **שם התכונה** | **טיפוס התכונה** | **תיאור התכונה** |
| 1 | from | Cell | מחזיק את סוג החלק אליו אנו רוצים להכתיר את החייל השחור. |
| 2 | to | Cell |  |

**2.5.12 מחלקת TreeNode**

|  |  |  |  |
| --- | --- | --- | --- |
|  | **שם התכונה** | **טיפוס התכונה** | **תיאור התכונה** |
| 1 | sons | List<TreeNode> | מחזיק את סוג החלק אליו אנו רוצים להכתיר את החייל השחור. |
| 2 | score | Int |  |
| 3 | whatGotUsHere | Move |  |

# **3. תכנון**

## **3.1 חלוקה למודולים**

**3.1.1 תפקיד כללי של מודולים**

|  |  |  |
| --- | --- | --- |
|  | **שם מודול** | **תפקיד כללי** |
| 1 | OpeningPanel | מסך פתיחה- התפריט הראשי במשחק. המסך מספק תפריט ראשי שבאמצעותו ניתן להיכנס לשאר המסכים : מסך משחק נגד AI, ומסך משחק נגד שחקן אנושי. |
| 2 | MainForm (Windows Form) | החלון הראשי.על החלון הראשי ישנם הדברים הבאים:   * מסך OpeningPanel. * כפתור אתחול משחק. * כפתור שחזור מצב משחק קודם. |
| 3 | Logic | מחלקת "לוגיקת מחשב". אחראית לניהול המשחק על פי כללי שחמט. |
| 4 | Cell | מחלקת "תא". המחלקה אחראית לייצוג משבצת על לוח השחמט של Logic. |
| 5 | EnPassant | מחלקת "אכילה דרך הילוכו". המחלקה אחראית לשמירת מיקומים של חיילים אשר יכולים "להיאכל" בידי חיילים של היריב, במהלך המיוחד: "אכילה" דרך הילוכו. |
| 6 | Piece | מחלקת "חלק". המחלקה אחראית לנהל חלק משחק. |
| 7 | Color | טיפוס "צבע". מייצג או לבן או שחור. |
| 8 | Type | טיפוס "סוג". מייצג את סוג החלק: מלך, מלכה, צריח, רץ, פרש או חייל. |
| 9 | MainForm (C# class) | מחלקת "חלון ראשי". המחלקה אחראית על ניהול הגרפיקה במשחק. |
| 10 | AI | מחלקת "אינטליגנציה מלאכותית". המחלקה אחראית לניהול שחקן המחשב. |
| 11 | Move | מחלקת "מהלך". עצם המייצג מהלך במשחק שחמט. |
| 12 | TreeNode | מחלקת "צומת עץ". מייצג צומת בעץ. |
| 13 | GraphicCell | מחלקת "תא גרפי". אחראי על הצגת משבצת בלוח שחמט. |
| 14 | EatenPieceHolder | מחלקת "מחזיק חלק-אכול". אחראית על ניהול והצגת חלק "אכול" בשחמט. |
| 15 | ChooseDubbedPieceBlack | מחלקת "בחר-כינוי שחור". אחראית על ניהול הגרפיקה של הכתרת חייל שחור. |
| 16 | ChooseDubbedPieceWhite | מחלקת "בחר-כינוי לבן". אחראית על ניהול הגרפיקה של הכתרת חייל לבן. |
| 17 | Program | מחלקת "תכנית". מריצה את MainForm. |

**3.1.2 עץ ראשי של מודולים**

**3.2 פירוט המודולים**

**3.2.01 מודול Logic**

|  |  |  |
| --- | --- | --- |
|  | **שם פונקציה** | **תפקיד כללי** |
| 1 | EnPassantLeft | בודקת אם ביכולתו חייל לבצע "אכילה" בדרך הילוכו, בחלק של לוח השחמט הנמצא משמאל לחייל. |
| 2 | EnPassantRight | בודקת אם ביכולתו חייל לבצע "אכילה" בדרך הילוכו, בחלק של לוח השחמט הנמצא מימין לחייל. |
| 3 | FindKing | מוצאת את המלך של אחד מן הצבעים בלוח השחמט. |
| 4 | GetCell | מחזירה משבצת בלוח השחמט. |
| 5 | getCellContent | מחזירה את החלק שנמצא במשבצת של לוח שחמט. |
| 6 | GetTurn | מחזירה את הצבע של החלקים השייכים לשחקן אשר כרגע תורו לשחק. |
| 7 | GetValidMoves | מקבלת מספר שורה ומספר עמודה, ואם ישנו חלק כלשהו במשבצת המתאימה לשני מספרים אלו, מחזירה את המהלכים החוקיים שהחלק יכול לבצע, null אחרת. |
| 8 | GetValidMoves | מקבלת משבצת בלוח השחמט, ומחזירה רשימת מהלכים חוקיים שהחלק באותו לוח שחמט יכול לבצע, null במידה ואין במשבצת הנתונה חלק. |
| 9 | InCheck | מחזירה אמת אם השחקן שכליו הינם בצבע הנתון נמצא בשח, ושקר אחרת. |
| 10 | InCheckMate | מחזירה אמת אם השחקן שכליו הינם בצבע הנתון נמצא בשחמט, ושקר אחרת. |
| 11 | initGrid | מאתחלת את לוח השחמט התאורטי. |
| 12 | isThisColorToPlay | מחזירה אמת אם הצבע הנתון הינו הצבע שתורו לשחק, שקר אחרת. |
| 13 | MakeMove | מבצעת מהלך. |
| 14 | NextTurn | מעדכנת את התור הנוכחי להיות הצבע האחר. |
| 15 | Promotion | מכתירה חייל בלוח השחמט התאורטי. |
| 16 | RefineRookByPieces | מוחקת מרשימת המהלכים החוקיים שצריח מסוגל לבצע, את המהלכים הכוללים אכילת חלקים מאותו צבע או דילוג מעל חלקים משני הצבעים. |
| 17 | validIndexes | מחזירה אמת אם ערכי שתי מספרים נתונים, הינם אכן מספרי שורה ועמודה הקיימים בלוח השחמט התיאורטי, שקר אחרת. |
| 18 | WhitePerformingEnPassant | מחזירה אמת אם חייל לבן מבצע "אכילה" בדרך הילוכו, ושקר אחרת. |
| 19 | BlackPerformingEnPassant | מחזירה אמת אם חייל שחור מבצע "אכילה" בדרך הילוכו, ושקר אחרת. |
| 20 | copyLogicGrid | מחזירה העתק של לוח השחמט התיאורטי. |
| 21 | Castling | אם מלך כלשהו מבצע הצרחה, הפעולה מעדכנת את הצריחים שיזוזו כפי התנאי. |
| 22 | CheckMovesPawn | מעדכנת את רשימת המהלכים לכלול את כל המהלכים החוקיים לרגלי לבצע. |
| 23 | CheckMovesRook | מעדכנת את רשימת המהלכים לכלול את כל המהלכים החוקיים לצריח לבצע. |
| 25 | CheckMovesKnight | מעדכנת את רשימת המהלכים לכלול את כל המהלכים החוקיים לפרש לבצע. |
| 26 | CheckMovesBishop | מעדכנת את רשימת המהלכים לכלול את כל המהלכים החוקיים לרץ לבצע. |
| 27 | CheckMovesQueen | מעדכנת את רשימת המהלכים לכלול את כל המהלכים החוקיים למלכה לבצע. |
| 28 | CheckMovesKing | מעדכנת את רשימת המהלכים לכלול את כל המהלכים החוקיים למלך לבצע. |

**3.2.02 מודול MainForm**

|  |  |  |
| --- | --- | --- |
|  | **שם פונקציה** | **תפקיד כללי** |
| 1 | init | מאתחל את לוח השחמט. |
| 2 | resurrectGrid | מחזיר את לוח השחמט למצבו ההתחלתי. מתחיל משחק מחדש. |
| 3 | initEaten | מאתחל את המטריצה אשר אחראית על הצגת חלקים "אכולים". |
| 4 | CheckMyColor | משחק מהלך אם המשתמש שמנסה לשחק, רשאי לשחק, או אם הוא בוחר מקום להעביר אליו חלק שהוא כבר משחק. |
| 5 | CellClicked | משחק את המהלך בצורה גרפית. צובע את המשבצות אליהם מותר לשחקן להעביר את הכלי המשחק שלו, וקורא למחלקת "לוגיקת מחשב" לשחק את המהלך בצורה תאורטית. |
| 6 | copyGraphicGrid | מחזירה העתק של לוח השחמט הגרפי. |
| 7 | showEaten | מציגה את החלק שכעת "נאכל". |
| 8 | graphicallyMovePiece | מעבירה את כלי המשחק אל יעדו. |
| 9 | Promote | מכתירה בצורה גרפית (משנה תמונה) את החייל שהרגע הגיע לסוף, נותנת למשתמש לבחור סוג חלק אליו החייל יוכתר, וקוראת למחלקת "לוגיקת מחשב" להכתיר בצורה תאורטית את החייל. |
| 10 | letAIPlay | אומרת ל-AI להחזיר מהלך שהוא רוצה לשחק, ומשחקת לו את המהלך. |
| 11 | EndGame | מציגה הודעת סוף משחק, ומודיעה מי המנצח. מחליפה את מוזיקת הרקע למוזיקת הפסד במידה והמשתמש הפסיד ל-AI, ולמוזיקת רקע, אחרת. |

**3.2.03 מודול AI**

|  |  |  |
| --- | --- | --- |
|  | **שם פונקציה** | **תפקיד כללי** |
| 1 | ChooseMove | מחזירה את המהלך שה-AI רוצה לשחק. |
| 2 | Assess | מחזירה ציון (ערך מספרי) המדרג את טיב המצב של ה-AI על לוח השחמט הנתון. |
| 3 | occupyCenter | מחזירה ערך מספרי המתאר את שליטת ה-AI על מרכז הלוח, ביחס לשליטת המשתמש על מרכז הלוח. |
| 4 | hasValidMoves | מחזירה אמת אם לשחקן הנתון ישנו מהלכים אפשריים לעשות, ושקר אחרת. |
| 5 | buildTree | מחזירה עץ מטיפוס Move, המתארת את המהלכים האפשריים של ה-AI והמשתמש, עם דירוג של טיב המצב של ה-AI על לוח השחמט בכל צומת בעץ. |
| 6 | CloneGame | מחזירה העתק של "לוגיקת המחשב" הנתון. |

**3.2.04 מודול EatenPieceHolder**

|  |  |  |
| --- | --- | --- |
|  | **שם פונקציה** | **תפקיד כללי** |
| 1 | PlaceEaten | מציגה את תמונת כלי המשחק הנתון. |

**3.2.05 מודול EnPassant**

|  |  |  |
| --- | --- | --- |
|  | **שם פונקציה** | **תפקיד כללי** |
| 1 | GetCell | פעולת אחזור לתא אליה מצביעה המחלקה. |
| 2 | GetIsActive | פעולת אחזור למשתנה המחלקה המכריז אם פעולת ה"אכילה" בדרך הילוכו אותה מייצגת המחלקה, קיימת. |
| 3 | SetCell | פעולת עדכון לתא אליה מצביעה המחלקה. |
| 4 | SetIsActive | פעולת עדכון למשתנה המחלקה המכריז אם פעולת ה"אכילה" בדרך הילוכו אותה מייצגת המחלקה, קיימת. |

**3.2.06 מודול Cell**

|  |  |  |
| --- | --- | --- |
|  | **שם פונקציה** | **תפקיד כללי** |
| 1 | Cell | פעולה בונה המאתחלת את כל תכונות המחלקה |
| 2 | Equals | מחזירה אמת אם התא הנתון שווה בנקודות המיקום שלו לתא הנוכחי. |

**3.2.07 מודול Piece**

|  |  |  |
| --- | --- | --- |
|  | **שם פונקציה** | **תפקיד כללי** |
| 1 | Piece | פעולה בונה המקבלת סוג מטיפוס Type, וצבע מטיפוס Color, ומאתחלת את תכונות המחלקה. |
| 2 | GetType | פעולה מאחזרת לתכונה Type |
| 3 | GetColor | פעולה מאחזרת לתכונה Color |

**3.2.08 מודול Move**

|  |  |  |
| --- | --- | --- |
|  | **שם פונקציה** | **תפקיד כללי** |
| 1 | Move | פעולה בונה המקבלת תא יעד מטיפוס Cell, ותא מוצא מטיפוס Cell, ומאתחלת את תכונות המחלקה. |

**3.2.09 מודול TreeNode**

|  |  |  |
| --- | --- | --- |
|  | **שם פונקציה** | **תפקיד כללי** |
| 1 | GetMaxSonScore | מחזירה את ה-score הגדול ביותר מבין הבנים **המידיים** של צומת העץ. |
| 2 | GetMinSonScore | מחזירה את ה-score הנמוך ביותר מבין הבנים **המידיים** של צומת העץ. |

**3.2.10 מודול Program**

|  |  |  |
| --- | --- | --- |
|  | **שם פונקציה** | **תפקיד כללי** |
| 1 | Main | הפעולה הראשית של התוכנה (פעולת ה-Main). |

**3.3 מבנה נתונים**

**3.3.1 לוח משחק גרפי מסוג GraphicCell[,]**

לוח משחק שמכיל את כל משבצות הלוח **הגרפי** מסוג GraphicCell,ישנם שני אזורים ליד הלוח להשמת החלקים ה"אכולים" מסוג EatenPieceHolder[,], בשם WhiteEaten ן-BlackEaten.

**3.3.2 עץ מהלכי משחק מסוג TreeNode**

עץ אשר בכל צומת ישנו מהלך אחר שיכול להיעשות על ידי אחד מהצדדים. הצומת המקורי של העץ הינו תמיד המהלך של ה-AI, ולאחר מכן הצמתים מתחלפים בין מהלכי שחקן אנושי ו-AI, לסירוגין.

**3.3.3 רשימת מהלכים חוקיים מסוג LinkedList<Cell>**

רשימה מקושרת של כל המהלכים החוקיים ששחקן מסוגל לקיים עם חלק נתון כלשהו.

**3.3.4 ערימת לוחות שחמט מסוג Stack<Cell[,]>**

ערימה של לוחות שחמט של מחלקת Logic , של כל מצבי הלוח **התאורטיים** הקודמים.

שימוש הערימה הוא לצורך פעולת "שחזר" (“Undo”).

**3.3.5 ערימת לוחות שחמט מסוג Stack<GraphicCell[,]>**

ערימה של לוחות שחמט של מחלקת MainForm , של כל מצבי הלוח **הגרפיים** הקודמים.

שימוש הערימה הוא לצורך פעולת "שחזר" (“Undo”).

**3.3.6 לוח משחק תאורטי מסוג Cell[,]**

לוח משחק שמכיל את כל משבצות הלוח **התאורטי** מסוג Cell.

**3.3.7 לוח חלקי משחק "אכולים" מסוג EatenPieceHolder[,]**

שתי לוחות משחק אשר מציגים את חלקי המשחק שכל צד "אכל" לשני.

**3.4 Coding Style**

במהלך כתיבת הקוד, הקפדתי לשמור על הכללים הבאים:

* יש לבחור שמות משמעותיים עבור המשתנים.
* שם מחלקה מתחיל תמיד באות גדולה.
* שם מתודה יתחיל תמיד באות גדולה. במידה והוא מורכב מכמה מילים, כל שאר המילים (מלבד המילה הפותחת) תתחלנה באות גדולה.
* קבוצות השייכות לתחום משותף (כגון: סוג חלק, צבע חלק וכו') ייוצגו ע"י enum.
* יש לכתוב הערות לאורך הקוד שמסבירות את תוכנו.
* יש לוודא שיש הזחות כנדרש בקוד.

# 4. מדריך למשתמש

**4.1 הוראות התקנה**

* צור ספרייה חדשה בדיסק קשיח בשםChess .
* היכנס לדפדפן כלשהו כרצונך.
* הקלד כ-URL: https:\\github.com\NullByte16\Chess-Project-With-ai.
* הורד את כל הקבצים בפרויקט לתיקיית Chess, שיצרת קודם לכן.

**4.2 הוראות הפעלה**

דרישות במערכת להפעלת המשחק הם:

* מעבד מסוג133 MHz Pentium ומעלה
* זיכרון פנימי 16 MB RAM ומעלה
* כרטיס מסך MB 4 ומעלה
* מסך SVGA
* .NET Framework 4
* מערכת הפעלה Windows 7 ומעלה

לצורך הפעלת המשחק הכנס לספרייה Chess שיצרת, והפעל את הקובץ Chess.exe.

# **נספח Listing**

|  |  |
| --- | --- |
| **שם הקובץ** | **מס' עמוד** |
| מחלקת Logic | 25 |
| מחלקת EnPassant | 39 |
| מחלקת Cell | 39 |
| מחלקת Piece | 40 |
| מחלקת MainForm | 41 |
| מחלקת AI | 52 |
| מחלקת Move | 57 |
| מחלקת TreeNode | 57 |
| מחלקת GraphicCell | 59 |
| מחלקת EatenPieceHolder | 60 |
| מחלקת ChooseDubbedPieceWhite | 61 |
| מחלקת ChooseDubbedPieceBlack | 62 |
| מחלקת Program | 63 |

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace Chess

{

public class Logic

{

public Cell[,] grid = new Cell[8, 8]; //todo private?

private Color CurrentTurn = Color.White;

private bool[] castlingHasMoved = new bool[6];

private EnPassant[] ep = new EnPassant[8];

public Logic()

{

// todo init all cells. some of the cell shall include a piece

for (int i = 0; i < this.castlingHasMoved.Length; i++)

{

castlingHasMoved[i] = false;

}

for (int i = 0; i < ep.Length; i++)

{

ep[i] = new EnPassant();

}

InitGrid();

}

public Piece GetCellContent(int i, int j)

{

return this.grid[i, j].piece;

}

public void InitGrid()

{

CurrentTurn = Color.White;

for (int i = 0; i < 8; i++)

for (int j = 0; j < 8; j++)

grid[i, j] = new Cell(i, j, null);

for (int j = 0; j < 8; j++)

{

//Init Pawns

grid[1, j].piece = new Piece(Type.Pawn, Color.Black);

grid[6, j].piece = new Piece(Type.Pawn, Color.White);

}

//Init Rooks

grid[0, 0].piece = new Piece(Type.Rook, Color.Black);

grid[0, 7].piece = new Piece(Type.Rook, Color.Black);

grid[7, 0].piece = new Piece(Type.Rook, Color.White);

grid[7, 7].piece = new Piece(Type.Rook, Color.White);

//Init Knights

grid[0, 1].piece = new Piece(Type.Knight, Color.Black);

grid[0, 6].piece = new Piece(Type.Knight, Color.Black);

grid[7, 1].piece = new Piece(Type.Knight, Color.White);

grid[7, 6].piece = new Piece(Type.Knight, Color.White);

//Init Bishops

grid[0, 2].piece = new Piece(Type.Bishop, Color.Black);

grid[0, 5].piece = new Piece(Type.Bishop, Color.Black);

grid[7, 2].piece = new Piece(Type.Bishop, Color.White);

grid[7, 5].piece = new Piece(Type.Bishop, Color.White);

//Init Queens

grid[0, 3].piece = new Piece(Type.Queen, Color.Black);

grid[7, 3].piece = new Piece(Type.Queen, Color.White);

//Init Kings

grid[0, 4].piece = new Piece(Type.King, Color.Black);

grid[7, 4].piece = new Piece(Type.King, Color.White);

}

/// <summary>

/// Returns the current color who's turn it is to play.

/// </summary>

public Color GetTurn()

{

return CurrentTurn;

}

/// <summary>

/// Switches the turn over to the opposite color.

/// </summary>

public void NextTurn()

{

if (GetTurn() == Color.Black) this.CurrentTurn = Color.White;

else this.CurrentTurn = Color.Black;

}

/// <summary>

/// Returns true if color of piece in cell at index [i, j] is the same as the color who's turn it currently is.

/// </summary>

public bool IsThisColorToPlay(int i, int j)

{

if (grid[i, j].piece != null && grid[i, j].piece.GetColor() == GetTurn())

return true;

return false;

}

/// <summary>

/// Moves the piece at index [fromI, fromJ] to the cell at index [toI, toJ]. Returns true if there is checkmate,

/// else returns false. \* If checkmate exists, will present appropriate MessageBox and initialize the grid. \*

/// </summary>

public bool MakeMove(int toI, int toJ, int fromI, int fromJ)

{

// Also switch turns if success

if (grid[fromI, fromJ].piece == null)

throw new Exception("attempt to move a null piece");

if (grid[fromI, fromJ].piece.GetType() == Type.King)

{

if (grid[fromI, fromJ].piece.GetColor() == Color.White)

castlingHasMoved[0] = true;

else

castlingHasMoved[1] = true;

}

else if (grid[fromI, fromJ].piece.GetType() == Type.Rook)

{

if (grid[fromI, fromJ].piece.GetColor() == Color.White && castlingHasMoved[2]) //HERE WAS STUPIDITY

castlingHasMoved[3] = true;

else if (grid[fromI, fromJ].piece.GetColor() == Color.White && !castlingHasMoved[2])

castlingHasMoved[2] = true;

else if (castlingHasMoved[4])

castlingHasMoved[5] = true;

else

castlingHasMoved[4] = true;

}

for (int i = 0; i < ep.Length; i++)

{

if (ep[i].GetIsActive()) //If the current EnPassant is active, deactivate it, and set cell to null.

{

ep[i].SetCell(null);

ep[i].SetIsActive(false);

}

else

i = ep.Length; //If we find an EnPassant object that is inactive, we have reached the last active EnPassant.

}

if (grid[fromI, fromJ].piece.GetType() == Type.Pawn && (toI == fromI + 2 || toI == fromI - 2))

{

int i = 0;

while (ep[i].GetIsActive()) i++;

ep[i].SetCell(grid[toI, toJ]);

ep[i].SetIsActive(true);

}

if(grid[fromI, fromJ].piece.GetType() == Type.Pawn)

{

if(WhitePerformingEnPassant(toI, toJ, fromJ))

grid[toI + 1, toJ].piece = null;

if(BlackPerformingEnPassant(toI, toJ, fromJ))

grid[toI - 1, toJ].piece = null;

}

if (grid[fromI, fromJ].piece.GetType() == Type.King && Math.Abs(toJ - fromJ) == 2)

{

Castling(toI, toJ, fromJ);

NextTurn();

}

grid[toI, toJ].piece = grid[fromI, fromJ].piece;

grid[fromI, fromJ].piece = null;

NextTurn();

if (InCheck(GetTurn()))

{

if (InCheckMate(GetTurn()))

{

NextTurn();

InitGrid();

return true;

}

}

return false;

}

public void Promotion(int i, int j, Type type)

{

this.grid[i, j].piece = new Piece(type, this.grid[i, j].piece.GetColor());

}

public void Castling(int toI, int toJ, int fromJ)

{

if (toI > 4)

{

if (toJ > fromJ)

{

MakeMove(7, 5, 7, 7);

}

else

{

MakeMove(7, 3, 7, 0);

}

}

else

{

if (toJ > fromJ)

{

MakeMove(0, 5, 0, 7);

}

else

{

MakeMove(0, 3, 0, 0);

}

}

}

public bool WhitePerformingEnPassant(int toI, int toJ, int fromJ)

{

if ((toJ == fromJ - 1 || toJ == fromJ + 1) && grid[toI, toJ].piece == null)

return true;

return false;

}

public bool BlackPerformingEnPassant(int toI, int toJ, int fromJ)

{

if ((toJ == fromJ - 1 || toJ == fromJ + 1) && grid[toI, toJ].piece == null)

return true;

return false;

}

public Cell GetCell(int i, int j)

{

return grid[i, j];

}

public LinkedList<Cell> GetValidMoves(Cell cell)

{

return GetValidMoves(cell.I, cell.J);

}

/// <summary>

/// Returns a LinkedList with all Cells to which the piece in the cell at index [IndexerCell.I, IndexerCell.J]

/// is allowed to move.

/// </summary>

public LinkedList<Cell> GetValidMoves(int i, int j)

{

Cell cell = grid[i, j];

if (cell.piece == null)

return null;

LinkedList<Cell> result = new LinkedList<Cell>();

// if not check, dont inclucde moves that lead to check

// todo

switch (cell.piece.GetType())

{

case Type.Pawn:

CheckMovesPawn(cell, result); // its a void function. it fills up our list.

break;

case Type.Knight:

CheckMovesKnight(cell, result);

break;

case Type.Rook:

CheckMovesRook(cell, result);

break;

case Type.Bishop:

CheckMovesBishop(cell, result);

break;

case Type.Queen:

CheckMovesQueen(cell, result);

break;

case Type.King:

CheckMovesKing(cell, result);

break;

}

#region move\_simulation

LinkedList<Cell> remover = new LinkedList<Cell>();

Piece tempPiece = cell.piece;

cell.piece = null;

Piece occupied\_piece = null;// To save any existing pieces in the current move of the current piece, from being deleted.

foreach (var target\_cell in result)

{

occupied\_piece = target\_cell.piece;

target\_cell.piece = tempPiece;

if (InCheck(target\_cell.piece.GetColor()))

{

remover.AddLast(target\_cell);

}

target\_cell.piece = occupied\_piece;

}

cell.piece = tempPiece;

#endregion

foreach (Cell remove in remover)

{

result.Remove(remove);

}

return result;

}

#region MoveCheckers

//Checkers. Different assisstive methods for determining valid moves for each piece type.

/// <summary>

/// Updates the LinkedList result to contain all Cells to which the pawn in cell is allowed to move.

/// </summary>

private void CheckMovesPawn(Cell cell, LinkedList<Cell> result)

{

if (cell.piece.GetColor() == Color.Black && cell.I + 1 < 8)

{

if (cell.I == 1 && grid[cell.I + 1, cell.J].piece == null && grid[cell.I + 2, cell.J].piece == null)

result.AddLast(grid[cell.I + 2, cell.J]);

if (grid[cell.I + 1, cell.J].piece == null)

result.AddLast(grid[cell.I + 1, cell.J]);

if (cell.I == 4)//RECENT CODE

{

if (ValidIndexes(cell.I, cell.J - 1) && EnPassantLeft(cell))

result.AddLast(grid[cell.I + 1, cell.J - 1]);

if (ValidIndexes(cell.I, cell.J + 1) && EnPassantRight(cell))

result.AddLast(grid[cell.I + 1, cell.J + 1]);

}

if (ValidIndexes(cell.I + 1, cell.J + 1) && grid[cell.I + 1, cell.J + 1].piece != null && grid[cell.I + 1, cell.J + 1].piece.GetColor() == Color.White)

result.AddLast(grid[cell.I + 1, cell.J + 1]);

if (ValidIndexes(cell.I + 1, cell.J - 1) && grid[cell.I + 1, cell.J - 1].piece != null && grid[cell.I + 1, cell.J - 1].piece.GetColor() == Color.White)

result.AddLast(grid[cell.I + 1, cell.J - 1]);

}

else if (cell.I - 1 > -1)

{

if (cell.I == 6 && grid[cell.I - 1, cell.J].piece == null && grid[cell.I - 2, cell.J].piece == null)

result.AddLast(grid[cell.I - 2, cell.J]);

if (grid[cell.I - 1, cell.J].piece == null)

result.AddLast(grid[cell.I - 1, cell.J]);

if (cell.I == 3)//RECENT CODE

{

if (ValidIndexes(cell.I, cell.J - 1) && EnPassantLeft(cell))

result.AddLast(grid[cell.I - 1, cell.J - 1]);

if (ValidIndexes(cell.I, cell.J + 1) && EnPassantRight(cell))

result.AddLast(grid[cell.I - 1, cell.J + 1]);

}

if (ValidIndexes(cell.I - 1, cell.J + 1) && grid[cell.I - 1, cell.J + 1].piece != null && grid[cell.I - 1, cell.J + 1].piece.GetColor() == Color.Black)

result.AddLast(grid[cell.I - 1, cell.J + 1]);

if (ValidIndexes(cell.I - 1, cell.J - 1) && grid[cell.I - 1, cell.J - 1].piece != null && grid[cell.I - 1, cell.J - 1].piece.GetColor() == Color.Black)

result.AddLast(grid[cell.I - 1, cell.J - 1]);

}

}

public bool EnPassantLeft(Cell cell)

{

EnPassant epTemp = new EnPassant();

epTemp.SetCell(grid[cell.I, cell.J - 1]);

epTemp.SetIsActive(true);

if (grid[cell.I, cell.J - 1].piece != null)

{

for (int i = 0; i < ep.Length; i++)

{

if (ep[i].GetCell() == epTemp.GetCell())

return true;

}

}

return false;

}

public bool EnPassantRight(Cell cell)

{

EnPassant epTemp = new EnPassant();

epTemp.SetCell(grid[cell.I, cell.J + 1]);

epTemp.SetIsActive(true);

if (grid[cell.I, cell.J + 1].piece != null)

{

for (int i = 0; i < ep.Length; i++)

{

if (ep[i].GetCell() == epTemp.GetCell())

return true;

}

}

return false;

}

/// <summary>

/// Updates the LinkedList result to contain all Cells to which the knight in cell is allowed to move.

/// </summary>

public void CheckMovesKnight(Cell cell, LinkedList<Cell> result)

{

if (ValidIndexes(cell.I - 1, cell.J - 2))

result.AddLast(grid[cell.I - 1, cell.J - 2]);

if (ValidIndexes(cell.I - 1, cell.J + 2))

result.AddLast(grid[cell.I - 1, cell.J + 2]);

if (ValidIndexes(cell.I + 1, cell.J - 2))

result.AddLast(grid[cell.I + 1, cell.J - 2]);

if (ValidIndexes(cell.I + 1, cell.J + 2))

result.AddLast(grid[cell.I + 1, cell.J + 2]);

if (ValidIndexes(cell.I - 2, cell.J - 1))

result.AddLast(grid[cell.I - 2, cell.J - 1]);

if (ValidIndexes(cell.I - 2, cell.J + 1))

result.AddLast(grid[cell.I - 2, cell.J + 1]);

if (ValidIndexes(cell.I + 2, cell.J - 1))

result.AddLast(grid[cell.I + 2, cell.J - 1]);

if (ValidIndexes(cell.I + 2, cell.J + 1))

result.AddLast(grid[cell.I + 2, cell.J + 1]);

//Refine result to only include cells that do not contain friendly pieces.

LinkedList<Cell> remover = new LinkedList<Cell>();

foreach (Cell checker in result)

{

if (checker.piece != null && checker.piece.GetColor() == cell.piece.GetColor()) remover.AddLast(checker);

}

foreach (Cell remove in remover)

result.Remove(remove);

}

/// <summary>

/// Updates the LinkedList result to contain all Cells to which the rook in cell is allowed to move.

/// </summary>

public void CheckMovesRook(Cell cell, LinkedList<Cell> result)

{

for (int count = 0; count <= 7; count++)

{

if (grid[count, cell.J] != cell)

result.AddLast(grid[count, cell.J]);

if (grid[cell.I, count] != cell)

result.AddLast(grid[cell.I, count]);

}

//Refine rook valid moves to only contain empty cells or cells with enemy pieces. //TODO TODO TODO

RefineRookByPieces(cell, result);

}

/// <summary>

/// Checks for moves in result that contain other pieces. Moves that contain friendly pieces, will be removed,

/// as well as all moves that require a rook to 'skip' over the friendly piece. For moves that contain the opponent's

/// pieces, the case is the same as with friendly pieces, except that the move with the opponent piece will remain.

/// </summary>

private void RefineRookByPieces(Cell cell, LinkedList<Cell> result)

{

LinkedList<Cell> remover = new LinkedList<Cell>();

for (int i = 0; i < 8; i++)

{

if (grid[i, cell.J].piece != null && i != cell.I)

{

if (grid[i, cell.J].piece.GetColor() == cell.piece.GetColor() && i > cell.I)

{

foreach (Cell checker in result)

{

if (checker.I >= i) remover.AddLast(checker);

}

}

else if (grid[i, cell.J].piece.GetColor() == cell.piece.GetColor() && i < cell.I)

{

foreach (Cell checker in result)

{

if (checker.I <= i) remover.AddLast(checker);

}

}

else if (grid[i, cell.J].piece.GetColor() != cell.piece.GetColor() && i > cell.I)

{

foreach (Cell checker in result)

{

if (checker.I > i) remover.AddLast(checker);

}

}

else if (grid[i, cell.J].piece.GetColor() != cell.piece.GetColor() && i < cell.I)

{

foreach (Cell checker in result)

{

if (checker.I < i) remover.AddLast(checker);

}

}

}

}

for (int j = 0; j < 8; j++)

{

if (grid[cell.I, j].piece != null && j != cell.J)

{

if (grid[cell.I, j].piece.GetColor() == cell.piece.GetColor() && j > cell.J)

{

foreach (Cell checker in result)

{

if (checker.J >= j) remover.AddLast(checker);

}

}

else if (grid[cell.I, j].piece.GetColor() == cell.piece.GetColor() && j < cell.J)

{

foreach (Cell checker in result)

{

if (checker.J <= j) remover.AddLast(checker);

}

}

else if (grid[cell.I, j].piece.GetColor() != cell.piece.GetColor() && j > cell.J)

{

foreach (Cell checker in result)

{

if (checker.J > j) remover.AddLast(checker);

}

}

else if (grid[cell.I, j].piece.GetColor() != cell.piece.GetColor() && j < cell.J)

{

foreach (Cell checker in result)

{

if (checker.J < j) remover.AddLast(checker);

}

}

}

}

foreach (Cell remove in remover)

{

Cell temp = remove;

result.Remove(temp);

//System.Windows.Forms.MessageBox.Show(temp.ToString());

}

}

/// <summary>

/// Updates the LinkedList result to contain all Cells to which the bishop in cell is allowed to move.

/// </summary>

public void CheckMovesBishop(Cell cell, LinkedList<Cell> result)

{

int i = cell.I - 1, j = cell.J - 1;

//Check for available moves in all four diagonal directions from the Bishop.

while (i > -1 && j > -1)

{

if (grid[i, j].piece != null)

{

if (grid[i, j].piece.GetColor() == cell.piece.GetColor()) break;

else

{

result.AddLast(grid[i, j]);

break;

}

}

result.AddLast(grid[i, j]);

i--;

j--;

}

i = cell.I + 1;

j = cell.J + 1;

while (i < 8 && j < 8)

{

if (grid[i, j].piece != null)

{

if (grid[i, j].piece.GetColor() == cell.piece.GetColor()) break;

else

{

result.AddLast(grid[i, j]);

break;

}

}

result.AddLast(grid[i, j]);

i++;

j++;

}

i = cell.I + 1;

j = cell.J - 1;

while (i < 8 && j > -1)

{

if (grid[i, j].piece != null)

{

if (grid[i, j].piece.GetColor() == cell.piece.GetColor()) break;

else

{

result.AddLast(grid[i, j]);

break;

}

}

result.AddLast(grid[i, j]);

i++;

j--;

}

i = cell.I - 1;

j = cell.J + 1;

while (i > -1 && j < 8)

{

if (grid[i, j].piece != null)

{

if (grid[i, j].piece.GetColor() == cell.piece.GetColor()) break;

else

{

result.AddLast(grid[i, j]);

break;

}

}

result.AddLast(grid[i, j]);

i--;

j++;

}

}

/// <summary>

/// Updates the LinkedList result to contain all Cells to which the queen in cell is allowed to move.

/// </summary>

public void CheckMovesQueen(Cell cell, LinkedList<Cell> result)

{

CheckMovesRook(cell, result);

CheckMovesBishop(cell, result);

}

/// <summary>

/// Updates the LinkedList result to contain all Cells to which the king in cell is allowed to move.

/// </summary>

public void CheckMovesKing(Cell cell, LinkedList<Cell> result)

{

LinkedList<Cell> remover = new LinkedList<Cell>();

//Add spaces king can move to, do not include pieces of the same color.

for (int i = cell.I - 1; i <= cell.I + 1; i++)

{

if (ValidIndexes(i, cell.J - 1) && ((grid[i, cell.J - 1].piece != null && grid[i, cell.J - 1].piece.GetColor() != GetTurn()) || grid[i, cell.J - 1].piece == null))

result.AddLast(grid[i, cell.J - 1]);

if (ValidIndexes(i, cell.J) && ((grid[i, cell.J].piece != null && grid[i, cell.J].piece.GetColor() != GetTurn()) || grid[i, cell.J].piece == null))

result.AddLast(grid[i, cell.J]);

if (ValidIndexes(i, cell.J + 1) && ((grid[i, cell.J + 1].piece != null && grid[i, cell.J + 1].piece.GetColor() != GetTurn()) || grid[i, cell.J + 1].piece == null))

result.AddLast(grid[i, cell.J + 1]);

}

result.Remove(cell);

//castling.

if (cell.piece.GetColor() == Color.White && !castlingHasMoved[0])

{

bool clear = true;

if (grid[7, 5].piece != null || grid[7, 6].piece != null)

clear = false;

if (!castlingHasMoved[2] && ValidIndexes(7, cell.J + 2) && clear) result.AddLast(grid[7, cell.J + 2]);

clear = true;

if (grid[7, 3].piece != null || grid[7, 2].piece != null || grid[7, 1].piece != null)

clear = false;

if (!castlingHasMoved[3] && ValidIndexes(7, cell.J - 2) && clear) result.AddLast(grid[7, cell.J - 2]);

}

if (cell.piece.GetColor() == Color.Black && !castlingHasMoved[1])

{

bool clear = true;

if (grid[0, 5].piece != null || grid[0, 6].piece != null)

clear = false;

if (!castlingHasMoved[4] && ValidIndexes(0, cell.J + 2) && clear) result.AddLast(grid[0, cell.J + 2]);

clear = true;

if (grid[0, 3].piece != null || grid[0, 2].piece != null || grid[0, 1].piece != null)

clear = false;

if (!castlingHasMoved[5] && ValidIndexes(0, cell.J - 2) && clear) result.AddLast(grid[0, cell.J - 2]);

}

}

/// <summary>

/// Receives the index [i, j]. Returns true if they exist in grid, else returns false.

/// </summary>

public bool ValidIndexes(int i, int j)

{

if (i > -1 && i < 8 && j > -1 && j < 8)

return true;

return false;

}

/// <summary>

/// Return true if current player to play is under check. Else returns false.

/// </summary>

public bool InCheck(Color color)

{

Cell king = FindKing(color);

LinkedList<Cell> moves = new LinkedList<Cell>();

// does any of the opponent pices' GetValidMoves contain the our king\s cell.

//System.Windows.Forms.MessageBox.Show(king.piece + " " + king.I + " " + king.J);

foreach (Cell checker in grid)

{

if (checker.piece != null && checker.piece.GetColor() != king.piece.GetColor())

{

switch (checker.piece.GetType())

{

case Type.Pawn:

CheckMovesPawn(checker, moves);

break;

case Type.Knight:

CheckMovesKnight(checker, moves);

break;

case Type.Rook:

CheckMovesRook(checker, moves);

break;

case Type.Bishop:

CheckMovesBishop(checker, moves);

break;

case Type.Queen:

CheckMovesQueen(checker, moves);

break;

case Type.King:

CheckMovesKing(checker, moves);

break;

}

if (moves.Contains(king)) return true;

}

}

return false;

}

/// <summary>

/// Returns a cell with one of the Kings, White if Color is White, and Black if Color is Black.

/// </summary>

public Cell FindKing(Color Color)

{

foreach (Cell cell in grid)

{

if (cell.piece != null && cell.piece.GetType() == Type.King && cell.piece.GetColor() == Color)

return cell;

}

return null;

}

/// <summary>

/// Return true if current player to play is under checkmate. Else returns false.

/// </summary>

public bool InCheckMate(Color color)

{

/\*if (!IsCheck())

return false;\*/

// for each king's valid move:

// move the king,

// check if check

// get the king back

LinkedList<Cell> ValidMovesKing = GetValidMoves(FindKing(color));

if (ValidMovesKing.Count > 0)

{

foreach (Cell cell in grid)

{

if (cell.piece != null && cell.piece.GetColor() == color)

{

if (GetValidMoves(cell) != null)

return false;

}

}

}

return true;

}

public Cell[,] CopyLogicGrid()

{

Cell[,] gc = new Cell[8, 8];

for (int i = 0; i < 8; i++)

for (int j = 0; j < 8; j++)

{

gc[i, j] = new Cell(i, j, grid[i, j].piece);

}

return gc;

}

}

public class EnPassant

{

private Cell cell;

private bool isActive;

public EnPassant()

{

this.cell = null;

this.isActive = false;

}

//Getters

public Cell GetCell()

{

return this.cell;

}

public bool GetIsActive()

{

return this.isActive;

}

//Setters

public void SetCell(Cell cell)

{

this.cell = cell;

}

public void SetIsActive(bool isActive)

{

this.isActive = isActive;

}

}

public class Cell

{

public readonly int I, J; // Coordinates of the cell.

public Piece piece; // The piece that the cell contains.

public Cell(int I, int J, Piece piece)

{

this.I = I;

this.J = J;

this.piece = piece;

}

public override bool Equals(object obj)

{

Cell other = (Cell)obj;

return other.J == this.J && other.I == this.I;

}

}

public class Piece

{

readonly Type Type; // The type of chess piece that this piece is.

readonly Color Color; // The color to which this piece belongs.

public Piece(Type Type, Color Color)

{

this.Type = Type;

this.Color = Color;

}

/// <summary>

/// Returns the Type of the current piece.

/// </summary>

public Type GetType()

{

return this.Type;

}

/// <summary>

/// Returns the Color of the current piece.

/// </summary>

public Color GetColor()

{

return this.Color;

}

}

}

public enum Color

{

White,

Black

}

public enum Type

{

King,

Queen,

Rook,

Bishop,

Knight,

Pawn

}

#endregion

using System;

using System.Collections.Generic;

using System.ComponentModel;

using System.Data;

using System.Drawing;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

using System.Windows.Forms;

namespace Chess

{

public partial class MainForm : Form

{

private GraphicCell[,] grid = new GraphicCell[8, 8]; // corresponding to the original matrix. to show whats there

private EatenPieceHolder[,] EatenBlack = new EatenPieceHolder[4, 4];

private EatenPieceHolder[,] EatenWhite = new EatenPieceHolder[4, 4];

private EatenPieceHolder nextBlack;

private EatenPieceHolder nextWhite;

private System.Media.SoundPlayer bgGameMusic = new System.Media.SoundPlayer(Properties.Resources.di\_evantile\_charming\_life); //AUDIO

private System.Media.SoundPlayer OpeningWarMusic = new System.Media.SoundPlayer(Properties.Resources.Epic\_battle\_music\_grzegorz\_majcherczyk\_heroica);

private System.Media.SoundPlayer DefeatMusic = new System.Media.SoundPlayer(Properties.Resources.Stephen\_Rippy\_\_\_Age\_of\_Mythology\_OST\_\_\_Ma\_am\_\_\_Some\_Other\_Sunset\_Defeat\_Theme\_\_mp3\_pm\_);

private System.Media.SoundPlayer VictoryMusic = new System.Media.SoundPlayer(Properties.Resources.Two\_Steps\_From\_Hell\_\_\_Victory\_\_320\_\_kbps\_\_\_YouTube\_2\_MP3\_Converter\_);

private bool bgGamePlaying = false, OpeningWarPlaying = false, DefeatPlaying = false, VictoryPlaying = false;

private Stack<Cell[,]> undoStackLogic = new Stack<Cell[,]>();

private Stack<GraphicCell[,]> undoStackGraphic = new Stack<GraphicCell[,]>();

private Logic logic;

private bool withSound = true;

private AI ai = null; // null indicates its a human playing

public MainForm()

{

InitializeComponent();

ToMainMenu.Visible = false;

buttonRestart.Visible = false;

undoButton.Visible = false;

OpeningWarMusic.PlayLooping();

OpeningWarPlaying = true;

}

private void Init()

{

logic = new Chess.Logic();

InitGrid();

InitEaten();

ToMainMenu.Visible = true;

buttonRestart.Visible = true;

undoButton.Visible = true;

}

private void InitGrid()

{

var images = new Dictionary<Color, Dictionary<Type, Bitmap>>();

#region filling up images

var whites = new Dictionary<Type, Bitmap>();

var blacks = new Dictionary<Type, Bitmap>();

images[Color.White] = whites;

images[Color.Black] = blacks;

whites[Type.Bishop] = Properties.Resources.BishopW;

whites[Type.Pawn] = Properties.Resources.PawnW;

whites[Type.Knight] = Properties.Resources.KnightW;

whites[Type.Queen] = Properties.Resources.QueenW;

whites[Type.King] = Properties.Resources.KingW;

whites[Type.Rook] = Properties.Resources.RookW;

blacks[Type.Bishop] = Properties.Resources.BishopB;

blacks[Type.Pawn] = Properties.Resources.PawnB;

blacks[Type.Knight] = Properties.Resources.KnightB;

blacks[Type.Queen] = Properties.Resources.QueenB;

blacks[Type.King] = Properties.Resources.KingB;

blacks[Type.Rook] = Properties.Resources.RookB;

#endregion

for (int i = 0; i < 8; i++)

{

for (int j = 0; j < 8; j++)

{

grid[i, j] = new GraphicCell(i, j, this);

Piece piece = logic.GetCellContent(i, j);

if (piece != null)

grid[i, j].Image = images[piece.GetColor()][piece.GetType()];

}

}

}

private void RessurrectGrid()

{

lastChosen = null;

while (undoStackGraphic.Count > 0 && undoStackLogic.Count > 0)

{

GraphicCell[,] gc = undoStackGraphic.Pop();

Cell[,] c = undoStackLogic.Pop();

for (int i = 0; i < 8; i++)

for (int j = 0; j < 8; j++)

{

this.grid[i, j].Image = gc[i, j].Image;

this.grid[i, j].BackColor = this.grid[i, j].OriginalColor;

logic.grid[i, j].piece = c[i, j].piece;

}

if (this.ai == null)

logic.NextTurn();

}

for (int i = 0; i < 4; i++)

{

for (int j = 0; j < 4; j++)

{

this.Controls.Remove(EatenWhite[i, j]);

this.Controls.Remove(EatenBlack[i, j]);

EatenWhite[i, j] = null;

EatenBlack[i, j] = null;

}

}

/\*logic.InitGrid();

this.InitGrid();\*/

this.InitEaten();

}

public void InitEaten()

{

for (int i = 0; i < 4; i++)

{

for (int j = 0; j < 4; j++)

{

EatenWhite[i, j] = new EatenPieceHolder(i, j, this, null, Color.White);

EatenBlack[i, j] = new EatenPieceHolder(i, j, this, null, Color.Black);

}

}

nextWhite = EatenWhite[0, 0];

nextBlack = EatenBlack[0, 0];

}

public void CheckMyColor(int i, int j)

{

ToMainMenu.Enabled = false;

// todo names that mean something

if (lastChosen != null || logic.IsThisColorToPlay(i, j))

CellClicked(i, j);

}

private LinkedList<Cell> ValidMoves = null;

private GraphicCell lastChosen = null;

public void CellClicked(int i, int j) // todo there's a bug - after choosing a target once, then choosing another target, it won't show valid moves

{

if (lastChosen == null) // choosing source

{

ValidMoves = logic.GetValidMoves(i, j);

// would return null if no piece

if (ValidMoves != null)

{

foreach (Cell cell in ValidMoves)

{

if (cell.piece == null && logic.grid[i, j].piece.GetType() == Type.King && Math.Abs(j - cell.J) == 2)

grid[cell.I, cell.J].BackColor = System.Drawing.Color.Aqua;

else if (cell.piece == null)

grid[cell.I, cell.J].BackColor = System.Drawing.Color.SandyBrown;

else

grid[cell.I, cell.J].BackColor = System.Drawing.Color.Red;

}

lastChosen = grid[i, j];

}

}

else // choosing target

{

foreach (Cell cell in ValidMoves)

{

grid[cell.I, cell.J].BackColor = grid[cell.I, cell.J].OriginalColor;

}

if (!ValidMoves.Contains(this.logic.GetCell(i, j)))

{

if (logic.GetCellContent(i, j) != null && logic.GetCellContent(i, j).GetColor() == logic.GetTurn())

{

lastChosen = null;

CellClicked(i, j);

return;

}

lastChosen = null;

return;

}

ShowEaten(i, j);

//Save in undo stacks.

undoStackLogic.Push(logic.CopyLogicGrid());

undoStackGraphic.Push(CopyGraphicGrid());

GraphicallyMovePiece(lastChosen, grid[i, j]); //BREAKPOINT

if (logic.MakeMove(i, j, lastChosen.I, lastChosen.J)) // means if checkmate

{

EndGame();

}

lastChosen = null;

// todo

// if move was successfully done, and not checkmate... blah blah if's

if (ai != null)

{

LetAIplay();

}

}

}

private GraphicCell[,] CopyGraphicGrid() //NEWLY ADDED

{

GraphicCell[,] gc = new GraphicCell[8, 8];

for (int i = 0; i < 8; i++)

for (int j = 0; j < 8; j++)

{

gc[i, j] = new GraphicCell(i, j, this);

gc[i, j].Image = grid[i, j].Image;

}

return gc;

}

private void ShowEaten(int i, int j) //RECENT CODE

{

if (logic.GetCellContent(i, j) != null && !logic.IsThisColorToPlay(i, j)) // target is about get eaten!

{

if (logic.GetTurn() == Color.White)

{

nextBlack.PlaceEaten(grid[i, j].Image);

if (nextBlack.J == 3) // todo its a magic number. at least explain

nextBlack = EatenBlack[nextBlack.I + 1, 0];

else nextBlack = EatenBlack[nextBlack.I, nextBlack.J + 1];

}

else

{

nextWhite.PlaceEaten(grid[i, j].Image);

if (nextWhite.J == 3)

nextWhite = EatenWhite[nextWhite.I + 1, 0];

else nextWhite = EatenWhite[nextWhite.I, nextWhite.J + 1];

}

}

}

private void GraphicallyMovePiece(GraphicCell source, GraphicCell target)

{

int i, j;

Image temp = source.Image;

source.Image = null;

target.Image = temp;

if (logic.grid[source.I, source.J].piece != null)

{

i = source.I;

j = source.J;

}

else

{

i = target.I;

j = target.J;

}

if (logic.GetCellContent(i, j).GetType() == Type.King && Math.Abs(target.J - source.J) == 2)

{

if (target.J < 4)

{

temp = grid[target.I, 0].Image;

grid[target.I, 0].Image = null;

grid[target.I, 3].Image = temp;

}

if (target.J > 4)

{

temp = grid[target.I, 7].Image;

grid[target.I, 7].Image = null;

grid[target.I, 5].Image = temp;

}

}

//En Passant

if (logic.GetCellContent(i, j).GetType() == Type.Pawn)

{

if (logic.GetCellContent(i, j).GetColor() == Color.White && logic.WhitePerformingEnPassant(target.I, target.J, source.J))

{

ShowEaten(target.I + 1, target.J);

grid[target.I + 1, target.J].Image = null;

}

else if (logic.GetCellContent(i, j).GetColor() == Color.Black && logic.BlackPerformingEnPassant(target.I, target.J, source.J))

{

ShowEaten(target.I - 1, target.J);

grid[target.I - 1, target.J].Image = null;

}

}

//Promotion needed?

if (logic.GetCellContent(i, j).GetType() == Type.Pawn && (target.I == 0 || target.I == 7))

{

string promotion = Promote(target.I);

switch (promotion)

{

case "Knight":

logic.Promotion(source.I, source.J, Type.Knight);

if (target.I == 0)

grid[target.I, target.J].Image = Properties.Resources.KnightW;

else

grid[target.I, target.J].Image = Properties.Resources.KnightB;

break;

case "Rook":

logic.Promotion(source.I, source.J, Type.Rook);

if (target.I == 0)

grid[target.I, target.J].Image = Properties.Resources.RookW;

else

grid[target.I, target.J].Image = Properties.Resources.RookB;

break;

case "Bishop":

logic.Promotion(source.I, source.J, Type.Bishop);

if (target.I == 0)

grid[target.I, target.J].Image = Properties.Resources.BishopW;

else

grid[target.I, target.J].Image = Properties.Resources.BishopB;

break;

case "Queen":

logic.Promotion(source.I, source.J, Type.Queen);

if (target.I == 0)

grid[target.I, target.J].Image = Properties.Resources.QueenW;

else

grid[target.I, target.J].Image = Properties.Resources.QueenB;

break;

}

}

}

private string Promote(int i)

{

var form1 = new ChooseDubbedPieceWhite();

var form2 = new ChooseDubbedPieceBlack();

string chosen;

if (i == 0)

{

form1.Location = this.Location;

form1.ShowDialog();

chosen = form1.chosen;

}

else

{

form2.Location = this.Location;

form2.ShowDialog();

chosen = form2.chosen;

}

// code will freeze here until that form is closed

return chosen;

}

const int AI\_DELAY\_SECONDS = 2;

private void LetAIplay()

{

this.Update();

long start = DateTime.Now.Ticks / TimeSpan.TicksPerMillisecond;

Move move = ai.ChooseMove();

if (move != null)

{

ShowEaten(move.to.I, move.to.J);

GraphicallyMovePiece(grid[move.from.I, move.from.J], grid[move.to.I, move.to.J]);

if (logic.MakeMove(move.to.I, move.to.J, move.from.I, move.from.J))

EndGame();

long end = DateTime.Now.Ticks / TimeSpan.TicksPerMillisecond;

int sleep\_time = 1000 \* AI\_DELAY\_SECONDS - (int)(end - start);

if (sleep\_time > 0)

System.Threading.Thread.Sleep(sleep\_time);

}

else

EndGame();

}

public void EndGame()

{

bgGameMusic.Stop(); //AUDIO

bgGamePlaying = false;

if (ai != null && logic.GetTurn() == Color.Black)

{

DefeatMusic.Play();

DefeatPlaying = true;

}

else

{

VictoryMusic.Play();

VictoryPlaying = true;

}

System.Windows.Forms.MessageBox.Show("Checkmate! " + logic.GetTurn() + " wins the game!");

DefeatMusic.Stop();

DefeatPlaying = false;

VictoryMusic.Stop();

VictoryPlaying = false;

this.RessurrectGrid();

}

private void help\_Paint(object sender, PaintEventArgs e)

{

}

private void button4\_Click(object sender, EventArgs e)

{

RessurrectGrid();

}

private void button5\_Click(object sender, EventArgs e)

{

OpeningPanel.Visible = false;

Init();

OpeningWarMusic.Stop();

if (withSound)

{

bgGamePlaying = true;

bgGameMusic.PlayLooping();

}

ai = new AI(Color.Black, logic);

}

private void button6\_Click(object sender, EventArgs e)

{

OpeningPanel.Visible = false;

Init();

OpeningWarMusic.Stop();

if (withSound)

{

bgGamePlaying = true;

bgGameMusic.PlayLooping();

}

ai = null;

}

private void MainForm\_Load(object sender, EventArgs e)

{

}

private void undoButton\_Click(object sender, EventArgs e)

{

if (undoStackGraphic.Count > 0 && undoStackLogic.Count > 0)

{

GraphicCell[,] gc = undoStackGraphic.Pop();

Cell[,] c = undoStackLogic.Pop();

for (int i = 0; i < 8; i++)

for (int j = 0; j < 8; j++)

{

this.grid[i, j].Image = gc[i, j].Image;

logic.grid[i, j].piece = c[i, j].piece;

}

if (this.ai == null)

logic.NextTurn();

}

}

private void button7\_Click(object sender, EventArgs e)

{

var form = new ChooseDubbedPieceWhite();

form.Location = this.Location;

form.ShowDialog();

// code will freeze here until that form is closed

string chosen = form.chosen;

}

private void ToMainMenu\_Click(object sender, EventArgs e)

{

bgGameMusic.Stop();

bgGamePlaying = false;

lastChosen = null;

for (int i = 0; i < 4; i++)

for (int j = 0; j < 4; j++)

{

this.Controls.Remove(EatenBlack[i, j]);

this.Controls.Remove(EatenWhite[i, j]);

}

for (int i = 0; i < 8; i++)

{

for (int j = 0; j < 8; j++)

{

this.Controls.Remove(grid[i, j]);

}

}

/\*for(int i = 0; i < 8; i++)

for (int j = 0; j < 8; j++)

{

this.grid[i, j].Image = null;

}

for (int i = 0; i < 8; i++)

for (int j = 0; j < 8; j++)

{

this.Controls.Remove(grid[i, j]);

}\*/

/\*for (int i = 0; i < 8; i++)

for (int j = 0; j < 8; j++)

{

this.Controls.Remove(this.grid[i, j]);

}\*/

while (undoStackLogic.Count > 0 && undoStackGraphic.Count > 0)

{

undoStackLogic.Pop();

undoStackGraphic.Pop();

}

if (withSound)

{

OpeningWarPlaying = true;

OpeningWarMusic.PlayLooping();

}

ToMainMenu.Visible = false;

buttonRestart.Visible = false;

undoButton.Visible = false;

OpeningPanel.Visible = true;

this.logic = null;

}

private void soundOnOff\_Click(object sender, EventArgs e)

{

if (bgGamePlaying || OpeningWarPlaying)

{

bgGameMusic.Stop();

OpeningWarMusic.Stop();

bgGamePlaying = false;

OpeningWarPlaying = false;

soundOnOff.Image = Properties.Resources.\_32px\_Mute\_Icon\_svg;

withSound = false;

}

else if (!bgGamePlaying && !OpeningPanel.Visible)

{

bgGameMusic.PlayLooping();

bgGamePlaying = true;

soundOnOff.Image = Properties.Resources.\_32px\_Speaker\_Icon\_svg;

withSound = true;

}

else if (!OpeningWarPlaying && OpeningPanel.Visible)

{

OpeningWarMusic.PlayLooping();

OpeningWarPlaying = true;

soundOnOff.Image = Properties.Resources.\_32px\_Speaker\_Icon\_svg;

withSound = true;

}

}

}

}

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace Chess

{

public class AI

{

Color Color;

Logic game;

Dictionary<Type, int> worths = new Dictionary<Type, int>();

public Move ChooseMove()

{

var root = BuildTree(2, game.GetTurn(), this.CloneGame(game));

Console.WriteLine(root.score); //ERASE

// now we have to find the son with max score

// we can use the fact the root's score == max score of its sons

// so we will iterate over sons to find a son with score eq to ours

List<TreeNode> BestSons = new List<TreeNode>();

foreach (var son in root.sons)

{

if (son.score == root.score)

{

BestSons.Add(son);

/\*Console.WriteLine(son.score); //ERASE

game.NextTurn(); //DEBUGGG

return son.whatGotUsHere;\*/

}

}

Random rnd = new Random();

int ChosenOne = rnd.Next(0, BestSons.Count - 1);

return BestSons[ChosenOne].whatGotUsHere;

throw new Exception("ai did not find any move to play. at all"); //Unreachable Code?

}

public AI(Color color, Logic game)

{

this.Color = color;

this.game = game;

this.worths[Type.Pawn] = 1;

this.worths[Type.Knight] = 3;

this.worths[Type.Bishop] = 3;

this.worths[Type.Rook] = 5;

this.worths[Type.Queen] = 9;

this.worths[Type.King] = 0;

}

/// <summary>

/// Assess the current state of the game, in favor of a certain color (whoIsToPlay).

/// Returns an integer between -100 and 100, ranking how good the state of the game is for specified color.

/// </summary>

private int Assess(Logic game, Color Color) // pretty much heuristic

{

/\*

\* L = my pieces sum - his pices sum (on start: 39 - 39 = 0, but after we lost the queen -9)

\* we multiply L by a large wieght (say 10,000) to make it matter very much. -> queenloss = -90,000

\*

\* C = for each piece in the center: if its ours sum its value (1, 3, 5, 9) if its his piece, sum his minus value

\* multiply this by, say, 1.

\* if we are not in the center but hes too, then c = 0. if we both control it equally, then C = 0

\* C becomes somethinh when there is a diference.

\*

\* V = our king safety - his. this one's weight = 100

\*/

// todo return a number from -inf(we lose) to inf(we win)

// optional aspects:

// are we (or they) close to the centre

// are any piece lost

// is the king vulnerable

int centerControl, piecesLost, kingVulnerability;

const int a = 1, b = 10000, c = 100;

//First we'll check whoIsToPlay's control over the center.

//1: How many pieces from each side physically occupy the center?

int countWhite = OccupyCenter(game, Color.White);

int countBlack = OccupyCenter(game, Color.Black);

centerControl = this.Color == Color.White ? a \* (countWhite - countBlack) : a \* (countBlack - countWhite);

//Next, we will compare our losses against the opponent's.

countWhite = 0;

countBlack = 0;

int WhiteVulnerability = 0, BlackVulnerability = 0; // num of opponent's pieces that can get into king palace

Cell whiteKing = game.FindKing(Color.White), blackKing = game.FindKing(Color.Black);

HashSet<Cell> surroundingWhiteKing = new HashSet<Cell>();

HashSet<Cell> surroundingBlackKing = new HashSet<Cell>();

for (int i = whiteKing.I - 2; i < whiteKing.I + 2; i++)

{

for(int j = whiteKing.J -2; j < whiteKing.J + 2; j++)

{

if (game.ValidIndexes(i, j))

surroundingWhiteKing.Add(game.grid[i, j]);

}

}

for (int i = blackKing.I - 2; i < blackKing.I + 2; i++)

{

for (int j = blackKing.J - 2; j < blackKing.J + 2; j++)

{

if (game.ValidIndexes(i, j))

surroundingBlackKing.Add(game.grid[i, j]);

}

}

for (int i = 0; i < 8; i++)

{

for (int j = 0; j < 8; j++)

{

if (game.grid[i, j].piece != null)

{

if (game.grid[i, j].piece.GetColor() == Color.White)

{

countWhite += worths[game.grid[i, j].piece.GetType()];

foreach (var move in game.GetValidMoves(game.grid[i, j]))

{

if (surroundingWhiteKing.Contains(move))

{

WhiteVulnerability++;

break;

}

}

}

else

{

countBlack += worths[game.grid[i, j].piece.GetType()];

foreach (var move in game.GetValidMoves(game.grid[i, j]))

{

if (surroundingBlackKing.Contains(move))

{

BlackVulnerability++;

break;

}

}

}

}

}

}

piecesLost = this.Color == Color.White ? b \* (countWhite - countBlack) : b \* (countBlack - countWhite);

kingVulnerability = this.Color == Color.White ? c \* (countWhite - countBlack) : b \* (countBlack - countWhite);

return a \* centerControl + b \* piecesLost + c \* kingVulnerability;

}

//Receives a game and a color. Returns how many pieces of that color physically occupy the center.

private int OccupyCenter(Logic game, Color Color)

{

int count = 0;

if (game.grid[3, 3].piece != null && game.grid[3, 3].piece.GetColor() == Color)

count++;

if (game.grid[4, 4].piece != null && game.grid[4, 4].piece.GetColor() == Color)

count++;

if (game.grid[3, 4].piece != null && game.grid[3, 4].piece.GetColor() == Color)

count++;

if (game.grid[4, 3].piece != null && game.grid[4, 3].piece.GetColor() == Color)

count++;

return count;

}

private bool HasValidMoves(Logic state, Color WhoIsToPlay)

{

for (int i = 0; i < 8; i++)

{

for (int j = 0; j < 8; j++)

{

if (state.grid[i, j].piece != null && state.grid[i, j].piece.GetColor() == WhoIsToPlay && state.GetValidMoves(state.grid[i, j]).Count > 0)

{

return true;

}

}

}

return false;

}

int idFactory = 0;

TreeNode BuildTree(int levelsLeftToBuild, Color WhoIsToPlay, Logic state) // the function's final result would be the tree's root

{

TreeNode resultNode = new TreeNode();

bool hasAnyValidMoves; // todo check if checkmate or stalemate.

hasAnyValidMoves = HasValidMoves(state, WhoIsToPlay);

if (levelsLeftToBuild > 0 && hasAnyValidMoves) // recursion end

{

/\* for each possible move, named m:

\* clone the current game (deep copy)

\* in the clone, play m. alternatively - remember what you have done to reמשהו it

\* son (with all its descendants along) = buildTree(levels - 1, opposite\_color)

\* result.sons.add(son);

\*/

for (int i = 0; i < 8; i++)

{

for (int j = 0; j < 8; j++)

{

if (state.grid[i, j].piece != null && state.grid[i, j].piece.GetColor() == WhoIsToPlay)

{

// Console.WriteLine(idFactory + " before recur:");

// Console.WriteLine(state);

LinkedList<Cell> ValidMoves = state.GetValidMoves(state.grid[i, j]);

Piece movingPiece = state.grid[i, j].piece;

state.grid[i, j].piece = null;

Piece occupied\_piece;// To save any existing pieces in the current move of the current piece, from being deleted.

foreach (var target\_cell in ValidMoves)

{

occupied\_piece = target\_cell.piece;

target\_cell.piece = movingPiece;

// todo handle castling (move rook in addition), promotion always to queen implement PromoteToQueen

var son = BuildTree(levelsLeftToBuild - 1, WhoIsToPlay == Color.Black ? Color.White : Color.Black, state);

son.whatGotUsHere = new Move(state.grid[i, j], target\_cell);

resultNode.sons.Add(son);

target\_cell.piece = occupied\_piece;

}

state.grid[i, j].piece = movingPiece;

// Console.WriteLine(idFactory + " after recur:");

// Console.WriteLine(state);

idFactory++;

}

}

}

// now after recur is done,

if (WhoIsToPlay == this.Color)

resultNode.score = resultNode.GetMaxSonScore();

else

resultNode.score = resultNode.GetMinSonScore();

}

else // we are a leaf!

{

if (!hasAnyValidMoves)

{

if (state.InCheck(WhoIsToPlay))

resultNode.score = WhoIsToPlay == this.Color ? int.MinValue : int.MaxValue; // won or lost

else

resultNode.score = 0; // stale, mate!

}

else

resultNode.score = Assess(state, WhoIsToPlay);

}

return resultNode;

}

public Logic CloneGame(Logic game)

{

Cell[,] Grid = new Cell[8, 8];

for (int i = 0; i < 8; i++)

{

for (int j = 0; j < 8; j++)

{

if (game.grid[i, j].piece != null)

Grid[i, j] = new Cell(i, j, new Piece(game.grid[i, j].piece.GetType(), game.grid[i, j].piece.GetColor()));

else

Grid[i, j] = new Cell(i, j, null);

}

}

Logic Clone = new Logic();

Clone.grid = Grid;

//if (game.GetTurn() != Color.White)

// game.NextTurn();

return Clone;

}

}

public class Move

{

readonly public Cell from, to;

public Move(Cell from, Cell to)

{

this.from = from;

this.to = to;

}

}

public class TreeNode

{

public List<TreeNode> sons = new List<TreeNode>();

public int score;

public Move whatGotUsHere = null;

public int GetMaxSonScore()

{

int MaxScore = int.MinValue;

foreach (TreeNode son in this.sons)

{

if (MaxScore < son.score)

{

MaxScore = son.score;

}

}

return MaxScore;

}

public int GetMinSonScore()

{

int MinScore = int.MaxValue; // todo make it like the above function

foreach (TreeNode son in this.sons)

{

if (MinScore > son.score)

{

MinScore = son.score;

}

}

return MinScore;

}

}

}

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

using System.Windows.Forms;

namespace Chess

{

public class GraphicCell : Button

{

public readonly int I, J;

readonly MainForm Papa; // container

public readonly System.Drawing.Color OriginalColor;

public System.Drawing.Icon Icon;

public GraphicCell(int i, int j, MainForm papaForm)

{

this.FlatStyle = System.Windows.Forms.FlatStyle.Flat;

this.Location = new System.Drawing.Point(j \* 50 + 30, i \* 50 + 25);

this.Size = new System.Drawing.Size(50, 50);

this.UseVisualStyleBackColor = true;

this.OriginalColor = i % 2 == j % 2 ? System.Drawing.Color.WhiteSmoke : System.Drawing.Color.Gray;

papaForm.Controls.Add(this);

this.Click += new System.EventHandler(this.OnClick);

this.BackColor = OriginalColor;

I = i;

J = j;

Papa = papaForm;

}

private void OnClick(object sender, EventArgs e)

{

Papa.CheckMyColor(I, J);

}

}

}

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

using System.Windows.Forms;

namespace Chess

{

class EatenPieceHolder : PictureBox

{

public readonly int I, J;

readonly MainForm Papa;

public System.Drawing.Icon Icon;

public EatenPieceHolder(int I, int J, MainForm PapaForm, System.Drawing.Icon Icon, Color color)

{

if (color == Color.Black)

this.Location = new System.Drawing.Point(J \* 50 + 430, I \* 50 + 25);

else

this.Location = new System.Drawing.Point(J \* 50 + 430, I \* 50 + 225);

this.Size = new System.Drawing.Size(50, 50);

this.Icon = Icon;

PapaForm.Controls.Add(this);

this.I = I;

this.J = J;

Papa = PapaForm;

this.BackColor = System.Drawing.Color.Transparent;

}

public void PlaceEaten(System.Drawing.Image img)

{

this.Image = img;

}

}

}

using System;

using System.Collections.Generic;

using System.ComponentModel;

using System.Data;

using System.Drawing;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

using System.Windows.Forms;

namespace Chess

{

public partial class ChooseDubbedPieceWhite : Form

{

public ChooseDubbedPieceWhite()

{

InitializeComponent();

}

public String chosen = null;

private void pictureBox1\_Click(object sender, EventArgs e)

{

chosen = "Knight";

this.Close();

}

private void pictureBox2\_Click(object sender, EventArgs e)

{

chosen = "Queen";

this.Close();

}

private void pictureBox3\_Click(object sender, EventArgs e)

{

chosen = "Bishop";

this.Close();

}

private void pictureBox4\_Click(object sender, EventArgs e)

{

chosen = "Rook";

this.Close();

}

private void ChooseDubbedPieceWhite\_Load(object sender, EventArgs e)

{

}

}

}

using System;

using System.Collections.Generic;

using System.ComponentModel;

using System.Data;

using System.Drawing;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

using System.Windows.Forms;

namespace Chess

{

public partial class ChooseDubbedPieceBlack : Form

{

public ChooseDubbedPieceBlack()

{

InitializeComponent();

}

public string chosen = null;

private void ChooseDubbedPieceBlack\_Load(object sender, EventArgs e)

{

}

private void pictureBox1\_Click(object sender, EventArgs e)

{

chosen = "Queen";

this.Close();

}

private void pictureBox2\_Click(object sender, EventArgs e)

{

chosen = "Bishop";

this.Close();

}

private void pictureBox3\_Click(object sender, EventArgs e)

{

chosen = "Knight";

this.Close();

}

private void pictureBox4\_Click(object sender, EventArgs e)

{

chosen = "Rook";

this.Close();

}

}

}

using System;

using System.Collections.Generic;

using System.Linq;

using System.Threading.Tasks;

using System.Windows.Forms;

namespace Chess

{

static class Program

{

/// <summary>

/// The main entry point for the application.

/// </summary>

[STAThread]

static void Main()

{

Application.EnableVisualStyles();

Application.SetCompatibleTextRenderingDefault(false);

Application.Run(new MainForm());

}

}

}