View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by University of Louisville

University of Louisville

ThinkIR: The University of Louisville's Institutional Repository

Electronic Theses and Dissertations

2-2012

Embedded non-interactive CAPTCHA for Fischer Random Chess.

Ryan McDaniel
University of Louisville

Follow this and additional works at: https://ir.library.louisville.edu/etd

Recommended Citation

McDaniel, Ryan, "Embedded non-interactive CAPTCHA for Fischer Random Chess." (2012). *Electronic Theses and Dissertations.* Paper 944.
https://doi.org/10.18297/etd/944

This Master's Thesis is brought to you for free and open access by ThinkIR: The University of Louisville's Institutional Repository. It has been accepted for inclusion in Electronic Theses and Dissertations by an authorized administrator of ThinkIR: The University of Louisville's Institutional Repository. This title appears here courtesy of the author, who has retained all other copyrights. For more information, please contact thinkir@louisville.edu.

EMBEDDED NON-INTERACTIVE CAPTCHA FOR FISCHER RANDOM CHESS

By

Ryan McDaniel
B.S., University of Louisville, 2006

A Thesis
Submitted to the Faculty of the
University of Louisville
J.B. Speed School of Engineering
as Partial Fulfillment of the Requirements
for the Professional Degree

MASTER OF ENGINEERING

Department of Computer Engineering and Computer Science

February 2012

NON-INTERACTIVE EMBEDDED CAPTCHA
FOR FISCHER RANDOM CHESS

Submitted by:___________________________________
Ryan McDaniel

A Thesis Approved on

____
(Date)

By the Following Reading and Examination Committee:

____
Dr. Roman V. Yampolskiy, Thesis Director

____
Dr. Tim Hardin, Member

____
Dr. Dar-Jen Chang, Member

____

____

____

ACKNOWLEDGMENTS

I am very thankful to my thesis advisor, Dr. Roman Yampolskiy, whose encouragement, guidance and support from the initial to the final level enabled me to develop a better understanding of the subject.

I would also like to make a special thanks to Dr. Annette Littrell, who provided help and support every step of the way. It wouldn't have been possible without you.

ABSTRACT

Cheating in chess can take many forms and has existed almost as long as the game itself. The advent of computers has introduced a new form of cheating into the game. Thanks to the computational power of modern-day computers, a player can use a program to calculate thousands of moves for him or her, and determine the best possible scenario for each move and counter-move. These programs are often referred to as "bots," and can even play the game without any user interaction. In this paper, we describe a methodology aimed at preventing bots from participating in online chess games. The proposed approach is based on the integration of a CAPTCHA protocol into a game scenario, and the subsequent inability of bots to accurately track the game states. Preliminary experimental results provide favorable feedback for further development of the proposed algorithm.

TABLE OF CONTENTS

APPROVAL PAGE ii

ACKNOWLEDGMENTS iii

ABSTRACT iv

TABLE OF CONTENTS v

LIST OF FIGURES vi

I. THE IMPACT OF BOTS IN ONLINE GAMES 1

II. A LOOK INSIDE THE BEHAVIOR OF A BOT / PREVIOUS WORK 8

III. PROCEDURE 15

IV. RESULTS 22

V. CONCLUSION 25

REFERENCES 27

APPENDIX I. PROGRAM CODE 29

APPENDIX II. FEEDBACK FORM 68

VITA 69

LIST OF FIGURES

FIGURE 1 - A POKER BOT BEING USED IN MULTIPLE GAMES [7]. 3
FIGURE 2 - AN EXAMPLE OF FISCHER RANDOM CHESS STARTING POSITIONS [15]. 6
FIGURE 3 - A FULLY AUTONOMOUS CHESS BOT IN ACTION [16]. 9
FIGURE 4 - A TYPICAL CHESS LOG [13]. 10
FIGURE 5 - A BOT COMPARES AN IMAGE IN ITS DATABASE WITH THE IMAGES IT SEES ON THE BOARD [17]. ..11
FIGURE 6 - SEVERAL EXAMPLES OF A TEXT-BASED CAPTCHA TEST [18]. 12
FIGURE 7 - A GRAPH SHOWING THE TYPE OF DATA THAT CAN BE GATHERED USING SERVER-SIDE BOT DETECTION. THE BOTS DEMONSTRATE MUCH MORE REPETITIONS IN THEIR MOVEMENT PATTERN [5]. ..13
FIGURE 8 - STANDARD CHESS BOARD LAYOUT. 16
FIGURE 9 - DISTORTION USING THE TEXT BOX. 17
FIGURE 10 - DISTORTION USING THE TRACK BAR. 18
FIGURE 11 - FISCHER RANDOM CHESS 21
FIGURE 12 - ALTERNATE SET OF PIECES. 21
FIGURE 13 - A CAPTCHA VERSION OF THE KING OF HEARTS [4]. 23
FIGURE 14 - THERE ARE MULTIPLE WAYS TO PRESENT THE SAME CHESS PIECE OR POKER CARD. MORE IMAGE SETS WILL COMPLICATE THE MATCHING PROCESS FOR A BOT [4]. 24

I. THE IMPACT OF BOTS IN ONLINE GAMES

Chess programs have been designed and implemented on computers since the 1950's. In 1950, Claude E. Shannon published "Programming a Computer for Playing Chess," in which he presented a chess computer as possible proof of artificial intelligence:

"The chess machine is an ideal one to start with, since: (1) the problem is sharply defined both in allowed operations (the moves) and in the ultimate goal (checkmate); (2) it is neither so simple as to be trivial nor too difficult for satisfactory solution; (3) chess is generally considered to require "thinking" for skillful play; a solution of this problem will force us either to admit the possibility of a mechanized thinking or to further restrict our concept of "thinking"; (4) the discrete structure of chess fits well into the digital nature of modern computers." [1]

At first, these chess programs were created only to test the waters of what computing could do to enhance the game. However, over the years, programs like *Rybka* have become very powerful [2]. In 1997, a computer built by IBM, called Deep Blue, beat then-world champion Garry Kasparov, marking the first time a computer was able to beat a reigning world champion [3]. Some of the chess programs available today include

databases of past games, and provide numerous ways for players to learn the game and improve their skills. These aspects are certainly positive; however, there are other forms of computer-assisted chess which are not.

While cheating in chess can take many forms and has existed almost as long as the game itself, the advent of computers has introduced a new form of cheating into the game. Robots, or “bots,” are computer programs that can read a chessboard and the pieces, determine the best possible move to make, and either recommend the move to a player or make the move for them [4]. These bots are easily accessible, and can be very difficult to detect. Chess is not the only game plagued by bots, however. These technology cheats are very common in online games today, from traditional games such as poker and chess, all the way up to complex Massively Multiplayer Online Roleplaying Games (MMORPG) games like Blizzard Entertainment’s World of Warcraft [5]. Keeping bots from ruining the game for honest players requires a constant effort, since whenever a game update to eliminate bots is implemented, the bot creators update their bot to circumvent the latest fix [6].

FIGURE 1 - A poker bot being used in multiple games [7].

Although there are different motives for a person to use a bot in an online game, the most obvious is money. Poker is an easy target in this case, as money can be won by simply breaking even, thanks to the rewards programs offered through various sites [8]. Chess, on the other hand, is not normally played for the purpose of winning money. This would not be the situation if it was not for bots. Ranked chess matches are quite popular, though, and cheating to attain a higher rank is not unheard of. Perhaps the answer lies in the psychology of winning, which makes people feel better about themselves, even if they had to cheat to do it [9].

Cheating in online gaming can have far-reaching impact on honest players. For example, online poker is played for money, and only one player wins per match. If someone is cheating with a bot, then they are having a direct impact on the other players

by taking money from them [10]. Poker is gambling, however, and whenever money is involved, you can expect dishonesty as well. Chess, on the other hand, has traditionally been about the spirit of the game. Quintessentially, chess is a war campaign, with two players battling it out, planning their short-term and long-term strategies, and utilizing either an offensive or defensive game plan. Inevitably, some plans end in defeat, some in victory, and yet still some end with a draw. In this regard, it is easy to see how one player planning his moves with a computer has a very unfair advantage over the other. The game is not intended to be played this way, so the chess-playing community gets frustrated with cheating players. Unknowingly playing a game of chess against a bot may have varied repercussions. Some players may just brush off the loss, attributing it to a stronger opponent, and trying to learn from it; they may or may not be suspicious that cheating was involved. Others may get so frustrated that they quit playing chess altogether after a few encounters [11]. It is the latter group that makes preventing cheating so very important. Quickly and accurately identifying a bots' presence in an online game is crucial; false positives cannot be tolerated.

Not everyone agrees that using a computer to assist you in a chess game is considered cheating. In fact, Gary Kasparov, the same man that played against IBM's Deep Blue computer, created a form of chess involving computers called Advanced Chess. The idea of Advanced Chess is that a human player teams up with a computer to challenge another human-computer team to a game of chess.

Some advocates of Advanced Chess believe that it can increase competitiveness in chess matches and show how impressive the game can be when no mistakes are made and tactics are used at the highest level. The computer's ability to avoid mistakes and

analyze tactics, combined with the strategy of a human player, gives the audience a very compelling look at the game of chess [12].

Kasparov played against Veselin Topalov in the first Advanced Chess event in 1998, with Kasparov using *Fritz 5* [13] to assist him and Topalov using *Chessbase 7* [12]. The match went six games, ending in a 3-3 tie [14]. Recognizing there are strengths and weaknesses in both human and computer players, Advanced Chess attempts to counterbalance the disparities to make a very impressive game of chess.

A computer chess program has advantages over a human in certain areas, such as being able to process and evaluate data in fractions of a second. A computer program may also have access to large databases of opening moves, called opening books, which give a significant advantage at the beginning of a match. Hash tables can also be used by a computer opponent towards the end of the game to look up from the database the best possible moves to make, allowing for a flawless finish to a match [12].

A human chess player has some advantages over a computer as well, though. A strong human chess player is able to develop long term strategies early on in the match that a computer will not be able to analyze. Human players also only analyze moves that will have an impact on the game, as opposed to a computer analyzing every possible move. The long term effects of a move can be understood by a human player, while computers still can’t calculate far enough ahead to tell the consequences of moves during the middle of the game [12]. Advanced Chess illustrates the fact that having computers involved with chess is not necessarily always a bad thing.

Another interesting style of chess was created by one of the greatest chess players, Bobby Fischer, in 1996. Fischer Random Chess, or Chess960, changes the initial starting position of the pieces. The rules are as follows:

- Pawns are placed on the second row, just like in normal chess
- The king must be placed between the two rooks
- The bishops must be placed on opposite colored squares
- All other pieces are placed randomly
- The black pieces are placed in the same order as the white pieces

The idea is that the back row will be randomized, with the bishops on opposite colored squares and the king is between the two rooks, as in basic chess. Once one back row is randomized and set, the other back row is set in the same order.

FIGURE 2 - An example of Fischer Random Chess starting positions [15].

This style of chess places more value on creativity than memorizing opening moves, as in normal chess. Since computer chess programs rely heavily on huge databases of opening moves to predict and strategize, this style of chess can confuse a bot. There are 960 possible starting positions in Fischer Random Chess, so the size of a database with all possible opening moves would be staggering and prohibitive. Fischer Random Chess, therefore, is a strong candidate for bot prevention.

This research attempts to use Fischer Random Chess and some graphical User Interface elements to prevent a bot from having an impact on a game. The focus of chess should be on strategy and tactics, without concern as to whether or not the other player is cheating. In standard chess, every piece begins at the same position every game, making it easy for a bot to simply track the movements and not need to know what a piece looks like. By using Fischer Random Chess, the bot loses the advantage of knowing where each piece will start.

II. A LOOK INSIDE THE BEHAVIOR OF A BOT / PREVIOUS WORK

A robot or "bot", for the purposes of this research, can be described as an artificially intelligent program, with either partial or full autonomy, that assists a player in an online game [4]. One must understand how a bot works before one can discuss how to combat it. A typical bot program will go through three basic steps, the first of which is collecting data for the input. The second step is the heart of the program, where the collected data will be used to create a course of action, predicated by the bots' purpose and design. For example, in this step, a poker bot would determine the action a player should take, while a chess bot would determine which piece should be moved. In step three, the bot will output the desired action to the player, or even perform that action for the player, in the case of a fully autonomous bot.

FIGURE 3 - A fully autonomous chess bot in action [16].

Bots commonly collect input data in one of two ways. A chess server may give the location of pieces in a log file, possibly even in real time, making it very easy to gather the information needed for a bot to process the locations and determine a move.

FIGURE 4 - A typical chess log [13].

If the data is not available via a log file, a second data collection option is called screen scraping [8]. In the case of chess, the bot will compare the images on the board with images in its database. The bot can then essentially know which piece is a king, which is a queen, etc. The position of each piece is also easily determined, since the board is an image as well. As a result, the bot can look at the board, identify each piece,

and its respective location, and process that information to determine the best move to make.

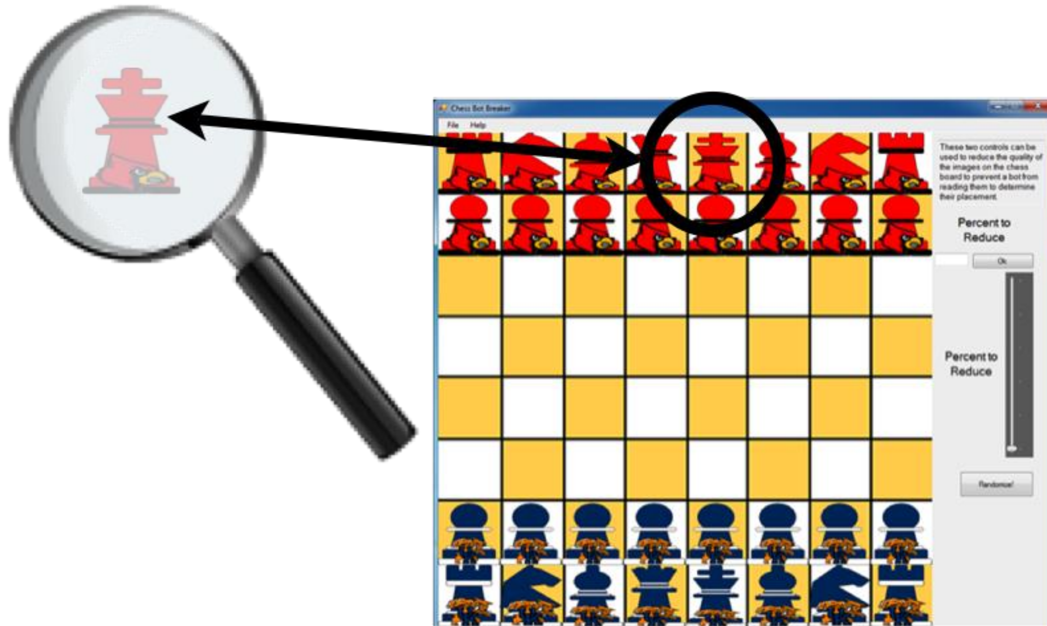

FIGURE 5 - A bot compares an image in its database with the images it sees on the board [17].

A popular method of ensuring the players involved in a game are, in fact, human players is "Completely Automated Public Turing test to tell Computers and Humans Apart" (CAPTCHA). A typical CAPTCHA test will present the player with distorted text, and then require them to type that text into a box in order to continue. A computer program will be unable to read the text and respond correctly, preventing the bot from continuing beyond that point [4].

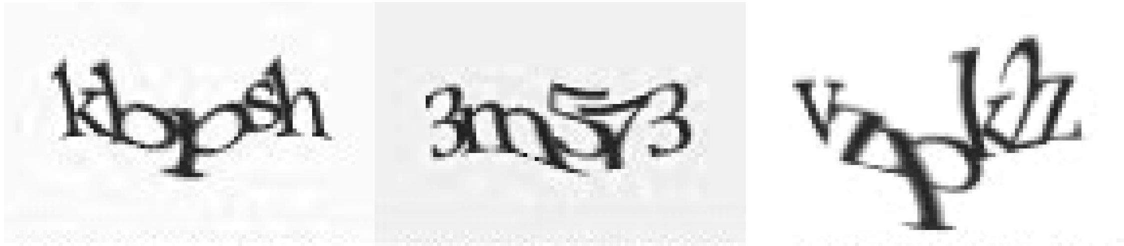

FIGURE 6 - Several examples of a text-based CAPTCHA test [18].

There are, however, ways to beat a CAPTCHA test. A bug in the CAPTCHA may be exploited to bypass the CAPTCHA test completely. Also, Optical Character Recognition (OCR) software is improving, allowing the bot to “read” the CAPTCHA text. Finally, the bot may present the CAPTCHA to the player as a part of the program. Due to the limitations of a CAPTCHA test, it is not an ideal solution for bot prevention; the test can be easily defeated if it is presented at the beginning of the game, and the test becomes an inconvenience to the player if it is presented during the game [19].

Research on bot detection and prevention in online games has expanded over the past few years, with methodologies ranging from direct impact on the player’s game experience, to total transparency, with varying degrees of success. Input devices could be used as a type of hardware-based CAPTCHA; for example, a joystick could be used as a CAPTCHA device, or a specially designed keypad could be used to input a series of characters at certain points during a game [19]. However, both of these options require special hardware, and keying in characters on a keypad would have a direct negative impact on the game experience, as it draws the players’ attention away.

Embedding a CAPTCHA into the game itself is a clever idea. However, it can be difficult to implement properly. Randomizing certain aspects of the game can make it much more difficult for a bot to participate. Randomization creates a non-interactive CAPTCHA type of test, as the bot will have to analyze options; however, it is not a particularly powerful deterrent since it can be solved. A more formal test could be presented to a player as well, in the form of a simple text-based or image-based CAPTCHA, in order to allow access to various aspects of the game [20]. This type of test would at least force some human interaction, adding only minor disruption to the playing experience, ensuring that a bot cannot operate completely autonomously.

Server-side bot detection is a method that is transparent to users, and typically focuses on the behavioral patterns of game clients. For example, the movement pattern of a character can be analyzed for overly repetitious actions to determine whether a bot or a human is in control [5].

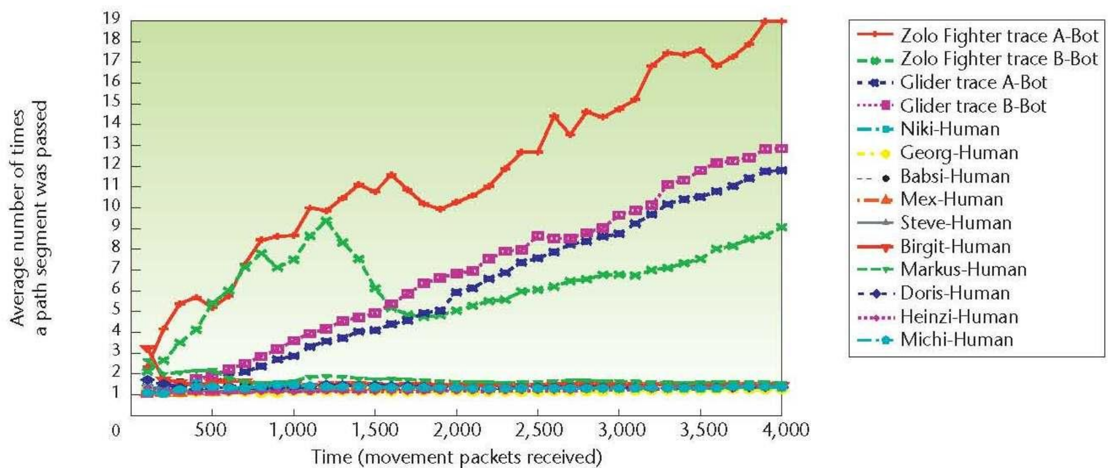

FIGURE 7 - A graph showing the type of data that can be gathered using server-side bot detection. The bots demonstrate much more repetitions in their movement pattern [5].

Also, input data from devices such as a mouse or keyboard could be analyzed for button-press-length and interval to determine if a bot is controlling a character [6]. Server-side bot detection requires some resources to analyze the data that is collected, however, and could possibly be circumvented by a bot program [21]. Once a bot is removed from the game, the bot creator can easily change the program to avoid the behavior that resulted in detection.

This research focuses on a type of embedded CAPTCHA, which can be non-interactive during gameplay. This is achieved by creating a chess interface which incorporates user-determined deterrents for bots, including skewing the resolution and/or rotating the chess pieces. In this type of environment, bot detection actually becomes secondary to bot prevention. By skewing the resolution and rotation of the chess pieces, the pieces no longer “fit” into a bot’s repertoire of known chess entities. A further complication for bots is introduced with a randomization option, which rearranges the back row of both team’s pieces according to the rules for Fisher Random Chess [15]. Also known as Chess960, this semi-random variant adds a new element to the beginning of a match.

Conversely, in classical chess, opening moves, or opening books, are finite and can be studied and memorized. Computers have clear advantages in a classical match, as they can have access to huge databases of opening books; with that access, a computer can evaluate the possibilities and choose the optimal move in nanoseconds. The nature of Fischer Random Chess prevents tracking of pieces from the initial board position, which would be possible in classical chess.

III. <u>PROCEDURE</u>

Written in C#, the software contains menu, grid, and options elements. The grid consists of sixty-four separate panels, each representing a single space on a chess board. Each 100x100 pixel panel is added to a two-dimensional array, arranged in an 8 panel x 8 panel square. This layout makes it simple to place a chess board image behind the panels, allowing the panels themselves to contain the chess piece images. While standard functions are used to determine the movability of pieces, it's the resolution, rotation, and randomization which provide additional ammunition to prevent bot play, and protect the players who simply wish to pursue a challenging game of chess against another human opponent.

The algorithm is based on two morphing functions:

Rotate Image Function:

- *Accepts image data type as argument*
- *Generate random number between -35 and +35*
- *Create new bitmap from image passed in*

- *Create new graphics object from bitmap and rotate*
- *Draw image back to bitmap form and return it*

Reduce Resolution Function:

- *Accepts image and integer data types as arguments*
- *Pass image to Rotate Image Function*
- *Create new bitmap image from rotated image with new size based on input*

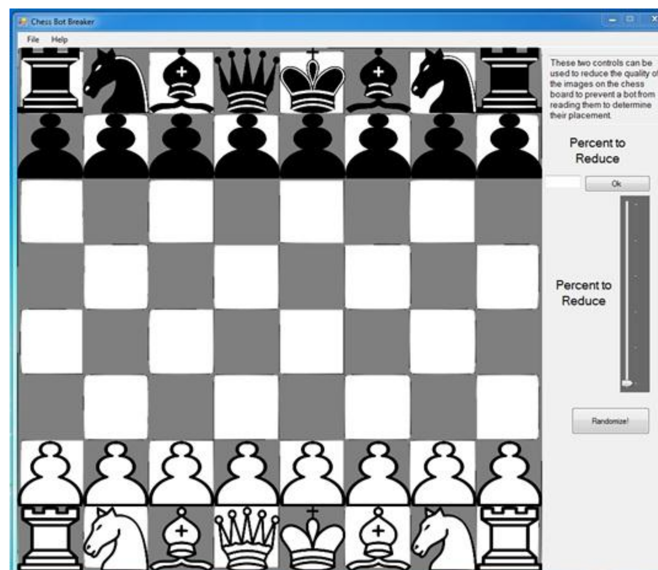

FIGURE 8 - Standard chess board layout in the developed software.

As shown in Figure 8, functions to adjust the resolution and rotation of the chess pieces have been added to the user interface, using a text box on the right-hand side of the form to accept entry of an integer between zero and ninety-nine. Once a valid number is entered into the text box and the adjacent "Ok" button is pressed, the resolution of all

pieces is decreased by the value entered, as a percentage, and a rotation between -35 degrees and +35 degrees is applied to each piece individually. This will result in all pieces having the same resolution reduction, but a different rotation for each piece (see Figure 9).

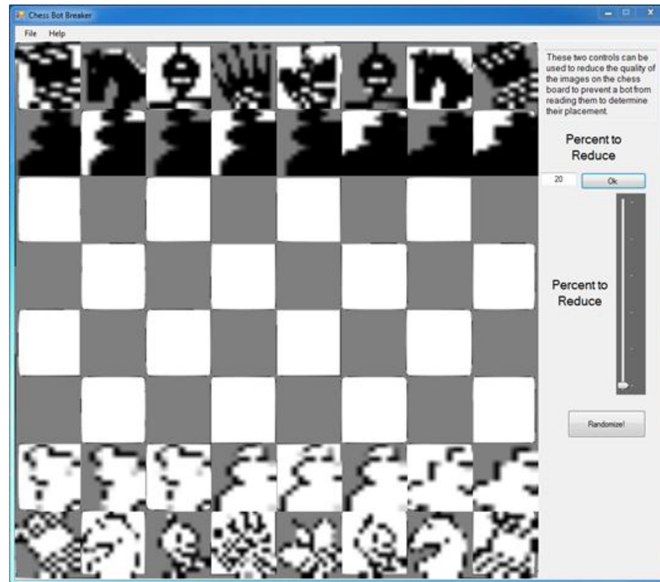

FIGURE 9 - Distortion using the text box.

Track Bar Function:

- *Track bar minimum is 0, maximum is 5, increments by 1*
- *Generate random number between 5 and 10 and multiply it by track bar value*
- *Pass image and random number to Resolution Reduction Function*
- *Repeat steps 2 and 3 for each image to give every piece a different rotation and reduction*

Below the text box is a track bar that can be used to increment the distortion of the pieces in a slightly different way (Figure 10). The track bar consists of six values, starting with zero at the bottom and incrementing by one to five at the very top notch. The track bar starts at zero by default. When incremented, the track bar value is multiplied by a random number, labeled *randNum*, between five and ten, and passed on to the resolution reduction function. A new random number is generated for each chess piece, giving a certain amount of randomness to the resolution reduction of each individual piece. Incrementing the track bar increases the value to be multiplied by the random number, somewhat guaranteeing an increase in distortion as the track bar is incremented.

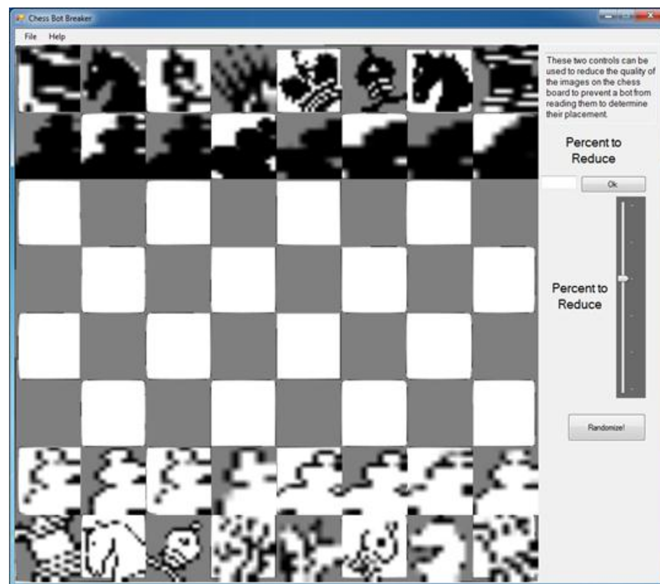

FIGURE 10 - Distortion using the track bar.

Fischer Random Chess Function:

- *Assign const integers to pieces (example: EMPTY = 0, KING = 1, QUEEN = 2, etc.)*
- *Create two lists to keep track of empty spaces in the back row, one for odd and one for even spaces*
- *Create an array to hold piece positions in the back row*
- *Generate a random number between 1 and 6 to place the KING*
- *Place KING into back row array at index just generated*
- *Generate 2 random numbers for placing ROOK. These must be between 0 and KING index, KING index and 7*
- *Place ROOK into back row array, 1 at each index just generated*
- *Update even and odd lists so no pieces are placed on occupied spaces*
- *Generate random number between 0 and even list size*
- *Place BISHOP into back row array at index just generated*
- *Generate random number between 0 and odd list size*
- *Place BISHOP into back row array at index just generated*
- *Update odd and even lists*

- *Consolidate odd and even lists into 1 empty spaces list since no remaining pieces have an odd or even requirement*
- *Generate random number between 0 and empty spaces list size*
- *Place QUEEN into back row array at index just generated*
- *Update empty spaces list*
- *Place KNIGHT into back row array at last 2 remaining indices*

As illustrated in Figure 11, when the "Randomize!" button below the track bar is selected, the program rearranges the back row of both team's pieces according to the rules set forth for Fischer Random Chess. Another function, *RandomResolution*, is called to distort the images as well. A random value, between five and thirty percent, is chosen for resolution reduction and applied to every piece on the board. Rotation is again applied to each piece individually. The "Randomize!" button may be pressed as many times as desired; however, once a piece has been moved, the game is officially started and the button is disabled. Distortion of the pieces can still be controlled via the other two methods at any point during the game.

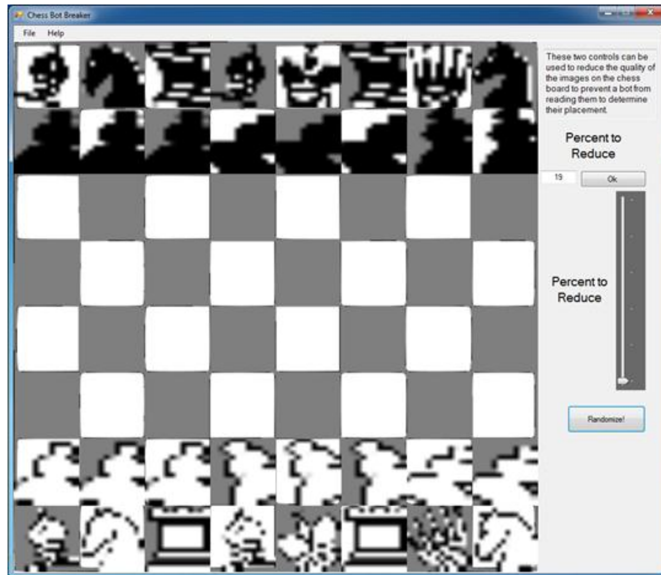

FIGURE 11 - Fischer Random Chess as implemented in the developed software.

Finally, Figure 12 shows that different sets of pieces can be selected to replace the standard black and white pieces and increase the difficulty of a bot successfully identifying them.

FIGURE 12 - Alternate set of pieces.

IV. RESULTS

Each user filled out a feedback form during testing, located in Appendix II, and the results were consistent. The form was used to collect some background information from the players to get an idea of their skill level at chess and determine how long it took them to adjust to the distorted pieces. Adjusting to the altered appearance of the pieces took most users a few seconds regardless of their skill level; after an average of four moves, none of the players had any trouble differentiating between pieces. Very few mistakes were made by the players. Some users did mention that additional changes could be implemented to make distinguishing pieces easier. For example, a letter representing the piece could be added to the image in a distorted way as well, similar to the poker card below.

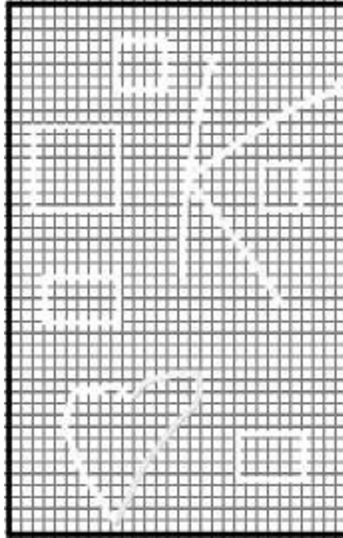

FIGURE 13 - A CAPTCHA version of the King of Hearts [4].

Humans are much better than computers at identifying patterns in an image [22]. Most modern text-based CAPTCHAs rely on letters which have been distorted. Therefore, distorting and rotating an image should prove very difficult for a bot to detect. Including additional distortion effects will increase the difficulty of programming a bot to read the pieces. For example, a skewing function would further help to prevent detection.

However, there are weaknesses to this approach to bot prevention. If there is only one available set of images for the chess pieces, then a bot simply has to compare the known images to distorted ones and make a guess based on similarity. This method could be fairly accurate, so it would be important to include multiple sets of images that can be used. Tracking piece movements could also give a bot clues as to what the pieces are; the bot may be fooled at the start, but as the game progresses, the movement trail left by the opponent may allow the bot to identify the pieces and resume control of the board.

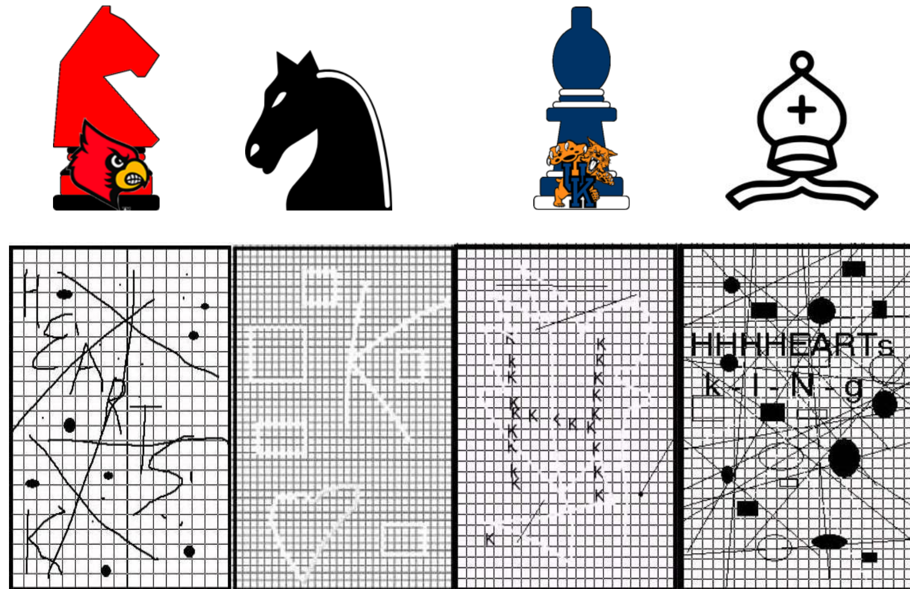

FIGURE 14 - There are multiple ways to present the same chess piece or poker card. More image sets will complicate the matching process for a bot [4].

The feedback, gathered from players with skill levels ranging from beginner to advanced, is promising. Players rated the difficulty of reading the pieces on a scale of one to ten, with an average rank of 6.5. However, the average number of moves to adjust to the distorted pieces was only 3.75, indicating that although this approach has a direct impact on the game experience, the user's ability to play the game is not hindered. The relatively high level of difficulty to read the pieces can be seen as a good indicator that a bot will have a hard time determining what the pieces are as well. The low number of moves to adjust to the distortion is a good sign that the player's experience will not be affected a great deal.

V. CONCLUSION

The program is designed to prevent a bot's ability to read a chess board, which renders the bot harmless and unable to suggest or make moves for the opposing player. This is accomplished by altering the visual aspects of the chess pieces on the board via user-controlled changes in resolution and/or rotation of the pieces; this skewing makes the pieces unrecognizable by a bot, while allowing human players to identify the pieces. With the added ability to play the game using the rules of Fischer Random Chess, a bot's inability to read the piece positions would prevent unfair advantages. However, this method is not without its disadvantages. For example, a bot may be able to identify a piece simply by the movement: a piece moving two places forward and one to the left is clearly a knight. In this manner, a bot could eventually identify each piece on the board. This is mitigated somewhat by the fact that the game would be well under way by this point.

Additional research into bot prevention is clearly needed. One path may be further altering the area of CAPTCHA tests. For example, added altering of visual elements -- such as skewing or stretching -- could be tested. Other user-interface changes could include multiple image sets for swapping. These options continually evolve as standard

CAPTCHA research moves forward. Additionally, although distorted audio or program-initiated questioning are alternatives to standard CAPTCHA tests, those methods could not be used with chess pieces. However, some of these methods are very creative; it is possible future research could find a way to incorporate an alternative method into the game of chess.

Breaking down a CAPTCHA is not always considered a total loss, as there are some positives that arise from it. For one, a weakness in the CAPTCHA has to be exposed, which can be fixed to strengthen the test in future revisions. But also, it is important to note that programming a bot to break a CAPTCHA test can be considered an advancement in Artificial Intelligence, as a bot has to try to emulate how a human would think in order to pass the test. This kind of competition is very important for promoting advancement in the fields of both Artificial Intelligence and security.

REFERENCES

[1] C. E. Shannon, "Programming a Computer for Playing Chess," Phil. Mag., 41, March 1950.

[2] "Rybka, for the Serious Chess Player", rybkachess.com, http://rybkachess.com/ (accessed June 2011).

[3] "Ibm Research | Deep Blue | Overview", IBM, http://www.research.ibm.com/deepblue (accessed January 2011).

[4] Yampolskiy, R. and Govindaraju, V. "Embedded Noninteractive Continuous Bot Detection." *ACM Computers in Entertainment* 5, no. 4 (2008).

[5] Mitterhofer, S., Platzer, C., Kruegel, C., and Kirda, E. "Server Side Bot Detection in Massively Multiplayer Online Games." *IEEE Security and Privacy* 7, no. 3 (2009): 29-36.

[6] Gianvecchio, S., Wu, Z., Xie, M., Wang, H. "Battle of Botcraft: Fighting Bots in Online Games with Human Observational Proofs." In *16th ACM conference on Computer and Communications Security*. New York, NY, USA, 2009.

[7] Michalski, Dan, "Rise of the (Real) Poker Bots", Pokerati.com, http://pokerati.com/2008/06/10/programmer-reveals-his-secrets-rise-of-the-real-poker-bots-artificial-opponents-emerge-from-dallas-underground-collude-online/ (accessed June 2011).

[8] Devlin, James, "How I Built a Working Poker Bot, Part 1", Coding the Wheel, http://www.codingthewheel.com/archives/how-i-built-a-working-poker-bot (accessed January 2011).

[9] Whitbourne, Susan. "Excuses, Excuses, Excuses: Why People Lie, Cheat, and Procrastinate." In *Fulfillment at Any Age*: Psychology Today, 2010.

[10] Kushner, D. "On the Internet, Nobody Knows You're a Bot." *Wired Magazine* 13, no. 9 (2005). [accessed January 2011].

[11] "Ethical Cheating in Online Chess", The Chess Corner, http://amirbagheri.virtuaboard.com/t34-ethical-cheating-in-online-chess (accessed January 2011).

[12] Friedel, Frederic, "Advanced Chess – General Info", ChessBase, http://www.chessbase.com/EvenTS/events.asp?pid=133 (accessed January 2011).

[13] "Fritz Chess", Chessbase.com http://www.chessbase.com/shop/product.asp?pid=467 (accessed January 2011).

[14] Friedel, Frederic, "Advanced Chess Match Leon 1998", ChessBase, http://www.chessbase.com/events/events.asp?pid=68 (accessed June 2011).

[15] "Fischer Random Chess Starting Positions", frcec.tripod.com, http://frcec.tripod.com/fischerrandomchessstartingpositions/ (accessed June 2011).

[16] "Chess Buddy", Playbuddy.com http://www.playbuddy.com/chess.php (accessed January 2011).

[17] Embedded Captcha for Fischer Random Chess. University of Louisville.

[18] "Captcha." 2011. wikipedia.org, http://en.wikipedia.org/wiki/CAPTCHA.

[19] Golle, P. and Ducheneaut, N. "Preventing Bots from Playing Online Games." *ACM Computers in Entertainment* 3, no. 3 (2005).

[20] Bushell, David. "In Search of the Perfect Captcha." *Smashing Magazine* (2011). http://coding.smashingmagazine.com/2011/03/04/in-search-of-the-perfect-captcha [accessed June 2011].

[21] Bethea, D., Cochran, R. and Reiter, M. "Server-Side Verification of Client Behavior in Online Games." In *Proceedings of the 17th Annual Network and Distributed System Security Symposium of the Internet Society*. San Diego, California, 2010.

[22] Strickland, Jonathan, "How Captcha Works", Howstuffworks.com, http://computer.howstuffworks.com/captcha.htm/printable (accessed June 2011).

APPENDIX I.   PROGRAM CODE

```
/* Written by Ryan McDaniel for M Eng. thesis in *
 * Computer Engineering and Computer Science     *
 * University of Louisville                      *
 * Email: ryan.mcdaniel@louisville.edu           *
 * Last updated March 10, 2011                   */


using System;
using System.Drawing;
using System.Drawing.Imaging;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Text;
using System.Windows.Forms;
using System.Drawing.Drawing2D;


namespace ChessBotBreaker
{
    public partial class Form1 : Form
    {
        static string workingDirectory;
        static Image king_reduced_bk;
        static Image queen_reduced_bk;
        static Image bishop_reduced_bk;
        static Image knight_reduced_bk;
        static Image rook_reduced_bk;
        static Image pawn_reduced_bk;

        static Image king_reduced_wh;
        static Image queen_reduced_wh;
        static Image bishop_reduced_wh;
        static Image knight_reduced_wh;
        static Image rook_reduced_wh;
        static Image pawn_reduced_wh;

        static Image board;

        int begin;                                  //Flag to disable
Randomize button and Game Type options
        private string gameType = "Standard";

        private int[] backRow = new int[8];         //This array is
used for randomizing the starting positions of the back rows
        private Panel[,] grid = new Panel[8, 8];    //This 8x8 2D array
will represent a 64 square chess board
        private Panel[] moveArray = new Panel[0];   //This array will
represent all possible legal moves a piece can make
        const int EMPTY = 0;
```

```
        const int KING = 1;
        const int QUEEN = 2;
        const int BISHOP = 3;
        const int KNIGHT = 4;
        const int ROOK = 5;
        ChessBoard chBoard;
        Rook bkRK1, bkRK2, whRK1, whRK2;
        Knight bkKN1, bkKN2, whKN1, whKN2;
        Bishop bkBI1, bkBI2, whBI1, whBI2;
        King bkKI, whKI;
        Queen bkQU, whQU;
        Pawn bkPA1, bkPA2, bkPA3, bkPA4, bkPA5, bkPA6, bkPA7, bkPA8;
        Pawn whPA1, whPA2, whPA3, whPA4, whPA5, whPA6, whPA7, whPA8;

        ChessPiece tempPieceSource, tempPieceDest;      //These are
used for picking up and moving a chess piece
        Panel tempSource, tempDest;                     //These are
used for picking up and moving a chess piece
        bool isMoveValid, isOpposingPiece;
        bool isNorthOrNWFinished, isSouthOrSEFinished,
isEastOrNEFinished;      //Used during the creation of the array that
holds all valid moves a piece can make
        int turnFlag = 1;                               //0 = black
turn, 1 = white turn
        private Rectangle imageBox;                     //Used in the
RotateImage function to adjust the size of the image

        public Form1()
        {
            InitializeComponent();
            InitializeImages();

            backRow[0] = ROOK;
            backRow[1] = KNIGHT;
            backRow[2] = BISHOP;
            backRow[3] = QUEEN;
            backRow[4] = KING;
            backRow[5] = BISHOP;
            backRow[6] = KNIGHT;
            backRow[7] = ROOK;
            begin = 0;

            BoardSetup();
            //Begin the game with pieces randomly placed according to
Fischer Random Chess rules
            //RandomChess();
            //RandomResolution();
            GridSetup();
        }

        private void GridSetup()
        {
            //Creates a grid to represent a 64 square chess board
            Panel[] rowOne = new Panel[8] { panel2, panel3, panel4,
panel5, panel6, panel7, panel8, panel9 };
            Panel[] rowTwo = new Panel[8] { panel10, panel11, panel12,
panel13, panel14, panel15, panel16, panel17 };
```

```
            Panel[] rowThree = new Panel[8] { panel18, panel19,
panel20, panel21, panel22, panel23, panel24, panel25 };
            Panel[] rowFour = new Panel[8] { panel26, panel27, panel28,
panel29, panel30, panel31, panel32, panel33 };
            Panel[] rowFive = new Panel[8] { panel34, panel35, panel36,
panel37, panel38, panel39, panel40, panel41 };
            Panel[] rowSix = new Panel[8] { panel42, panel43, panel44,
panel45, panel46, panel47, panel48, panel49 };
            Panel[] rowSeven = new Panel[8] { panel50, panel51,
panel52, panel53, panel54, panel55, panel56, panel57 };
            Panel[] rowEight = new Panel[8] { panel58, panel59,
panel60, panel61, panel62, panel63, panel64, panel65 };
            for (int i = 0; i < 8; i++)
            {
                grid[0, i] = rowOne[i];
            }
            for (int i = 0; i < 8; i++)
            {
                grid[1, i] = rowTwo[i];
            }
            for (int i = 0; i < 8; i++)
            {
                grid[2, i] = rowThree[i];
            }
            for (int i = 0; i < 8; i++)
            {
                grid[3, i] = rowFour[i];
            }
            for (int i = 0; i < 8; i++)
            {
                grid[4, i] = rowFive[i];
            }
            for (int i = 0; i < 8; i++)
            {
                grid[5, i] = rowSix[i];
            }
            for (int i = 0; i < 8; i++)
            {
                grid[6, i] = rowSeven[i];
            }
            for (int i = 0; i < 8; i++)
            {
                grid[7, i] = rowEight[i];
            }

        }

        private void InitializeImages()
        {
            //This function changes the images of the pieces and the
board
            if (gameType == "Standard")
            {
                workingDirectory = @"C:\Users\RCM-MAC\Documents\Visual
Studio 2008\Projects\ChessBotBreaker\ChessBotBreaker\bin\Standard\";
                king_reduced_bk = Image.FromFile(workingDirectory +
@"king_bk.png");
```

```
                    queen_reduced_bk = Image.FromFile(workingDirectory +
@"queen_bk.png");
                    bishop_reduced_bk = Image.FromFile(workingDirectory +
@"bishop_bk.png");
                    knight_reduced_bk = Image.FromFile(workingDirectory +
@"knight_bk.png");
                    rook_reduced_bk = Image.FromFile(workingDirectory +
@"rook_bk.png");
                    pawn_reduced_bk = Image.FromFile(workingDirectory +
@"pawn_bk.png");

                    king_reduced_wh = Image.FromFile(workingDirectory +
@"king_wh.png");
                    queen_reduced_wh = Image.FromFile(workingDirectory +
@"queen_wh.png");
                    bishop_reduced_wh = Image.FromFile(workingDirectory +
@"bishop_wh.png");
                    knight_reduced_wh = Image.FromFile(workingDirectory +
@"knight_wh.png");
                    rook_reduced_wh = Image.FromFile(workingDirectory +
@"rook_wh.png");
                    pawn_reduced_wh = Image.FromFile(workingDirectory +
@"pawn_wh.png");

                    imageBox = new Rectangle(0, 0, 300, 300);

                    board = Image.FromFile(workingDirectory +
@"board.png");
                    BoardSetup();
                }

                if (gameType == "UL/UK" && begin == 0)
                {
                    workingDirectory = @"C:\Users\RCM-MAC\Documents\Visual
Studio 2008\Projects\ChessBotBreaker\ChessBotBreaker\bin\ULUK\";
                    king_reduced_bk = Image.FromFile(workingDirectory +
@"king_ul.png");
                    queen_reduced_bk = Image.FromFile(workingDirectory +
@"queen_ul.png");
                    bishop_reduced_bk = Image.FromFile(workingDirectory +
@"bishop_ul.png");
                    knight_reduced_bk = Image.FromFile(workingDirectory +
@"knight_ul.png");
                    rook_reduced_bk = Image.FromFile(workingDirectory +
@"rook_ul.png");
                    pawn_reduced_bk = Image.FromFile(workingDirectory +
@"pawn_ul.png");

                    king_reduced_wh = Image.FromFile(workingDirectory +
@"king_uk.png");
                    queen_reduced_wh = Image.FromFile(workingDirectory +
@"queen_uk.png");
                    bishop_reduced_wh = Image.FromFile(workingDirectory +
@"bishop_uk.png");
                    knight_reduced_wh = Image.FromFile(workingDirectory +
@"knight_uk.png");
```

```
                rook_reduced_wh = Image.FromFile(workingDirectory +
@"rook_uk.png");
                pawn_reduced_wh = Image.FromFile(workingDirectory +
@"pawn_uk.png");

                imageBox = new Rectangle(0, 0, 600, 600);

                board = Image.FromFile(workingDirectory +
@"chessboard_golden2b.png");
                BoardSetup();
            }

            if (gameType == "UL/UK2" && begin == 0)
            {
                workingDirectory = @"C:\Users\RCM-MAC\Documents\Visual
Studio 2008\Projects\ChessBotBreaker\ChessBotBreaker\bin\ULUK\";
                king_reduced_bk = Image.FromFile(workingDirectory +
@"king_ul.png");
                queen_reduced_bk = Image.FromFile(workingDirectory +
@"queen_ul.png");
                bishop_reduced_bk = Image.FromFile(workingDirectory +
@"bishop_ul.png");
                knight_reduced_bk = Image.FromFile(workingDirectory +
@"knight_ul.png");
                rook_reduced_bk = Image.FromFile(workingDirectory +
@"rook_ul.png");
                pawn_reduced_bk = Image.FromFile(workingDirectory +
@"pawn_ul.png");

                king_reduced_wh = Image.FromFile(workingDirectory +
@"king_uk.png");
                queen_reduced_wh = Image.FromFile(workingDirectory +
@"queen_uk.png");
                bishop_reduced_wh = Image.FromFile(workingDirectory +
@"bishop_uk.png");
                knight_reduced_wh = Image.FromFile(workingDirectory +
@"knight_uk.png");
                rook_reduced_wh = Image.FromFile(workingDirectory +
@"rook_uk.png");
                pawn_reduced_wh = Image.FromFile(workingDirectory +
@"pawn_uk.png");

                imageBox = new Rectangle(0, 0, 600, 600);

                board = Image.FromFile(workingDirectory +
@"chessboard_2.png");
                BoardSetup();
            }

            this.Refresh();
        }

        public void RandomChess()
        {
            /* Fischer random chess function. Rules are as follows:
             * The black king is placed somewhere between the two black
rooks.
```

```
             * Bishops are placed on opposite colored squares.
             * White pieces are place in the same positions as the
black pieces.
            */

            //Create lists to keep track of where the empty spaces are
in the back row. Each is updated after a piece is placed
            List<int> emptyOdd = new List<int>();
            List<int> emptyEven = new List<int>();

            //Initialize backRow to all empty and the odd and even
lists to all empty
            for (int i = 0; i < 8; i++)
            {
                backRow[i] = EMPTY;
                if (i % 2 == 0)
                    emptyEven.Add(i);
                else
                    emptyOdd.Add(i);
            }

            Random rand = new Random();

            //place king in array at index between 1 and 6
            int kingIndex = rand.Next(1, 7);
            backRow[kingIndex] = KING;

            //place rooks at random spaces higher and lower than kings
place
            int rookOneIndex = rand.Next(0, kingIndex - 1);
            int rookTwoIndex = rand.Next(kingIndex + 1, 8);
            backRow[rookOneIndex] = ROOK;
            backRow[rookTwoIndex] = ROOK;

            //Update the odd and even lists so no pieces will be placed
on top of the already placed pieces
            emptyEven.Remove(kingIndex);
            emptyOdd.Remove(kingIndex);
            emptyEven.Remove(rookOneIndex);
            emptyOdd.Remove(rookOneIndex);
            emptyEven.Remove(rookTwoIndex);
            emptyOdd.Remove(rookTwoIndex);

            //place bishops,1 on odd index,1 on even index
            int bishopOneIndex = rand.Next(0, emptyEven.Count - 1);
            int bishopTwoIndex = rand.Next(0, emptyOdd.Count - 1);
            backRow[emptyEven[bishopOneIndex]] = BISHOP;
            backRow[emptyOdd[bishopTwoIndex]] = BISHOP;

            //Update the odd and even lists so no pieces will be placed
on top of the already placed pieces
            emptyEven.RemoveAt(bishopOneIndex);
            emptyOdd.RemoveAt(bishopTwoIndex);

            //Since the Bishops are placed, only 1 list of empty spaces
is needed
            List<int> emptySpaces = new List<int>();
```

```
            emptySpaces.AddRange(emptyEven);
            emptySpaces.AddRange(emptyOdd);

            //Place queen
            int queenIndex = rand.Next(0, emptySpaces.Count - 1);
            backRow[emptySpaces[queenIndex]] = QUEEN;

            //Update the emptyspaces list so no pieces will be placed
on top of the already placed pieces
            emptySpaces.RemoveAt(queenIndex);

            //Place knights
            int knightOneIndex = emptySpaces[0];
            int knightTwoIndex = emptySpaces[1];
            backRow[knightOneIndex] = KNIGHT;
            backRow[knightTwoIndex] = KNIGHT;
        }

        public Image ReducePieces(Image imgPhoto, int percent)
        {
            //Scales the size of original image by percent
            if (percent == 0)
                return imgPhoto;
            if (gameType == "UL/UK" || gameType == "UL/UK2")
            {
                percent = percent * 2;
            }
            if (imgPhoto == null)
                return null;
            Image tempImage = RotateImage(imgPhoto);
            /*
            If each piece is to be reduced by a different percentage:
            Random rand = new Random();
            int randNum = rand.Next(0, percent);
            tempImage = new Bitmap(tempImage, new Size(imgPhoto.Width /
randNum, imgPhoto.Height / randNum));
            percent becomes the maximum resolution in this case
            */
            tempImage = new Bitmap(tempImage, new Size(imgPhoto.Width /
percent, imgPhoto.Height / percent));
            return tempImage;
        }

        private Image RotateImage(Image imgPhoto)
        {
            //Used to rotate an image randomly up to 35 degrees
            Random rand = new Random();
            int randNum = (int)rand.Next(-35, 35);
            Image tempImage = null;
            if (gameType == "UL/UK" || gameType == "UL/UK2")
            {
                //Some adjustment for size is done here to make the
pieces fit in the squares a little better
                if (imgPhoto == pawn_reduced_wh || imgPhoto ==
pawn_reduced_bk)
                {
                    tempImage = new Bitmap(750, 750);
```

```
                }
                else
                {   tempImage = new Bitmap(700, 700); }
            }

            else
            {
                tempImage = new Bitmap(imgPhoto.Width,
imgPhoto.Height);
            }
            Graphics g = Graphics.FromImage(tempImage);
            g.TranslateTransform((float)imgPhoto.Width / 2,
(float)imgPhoto.Height / 2);
            g.RotateTransform(randNum);
            g.TranslateTransform(-(float)imgPhoto.Width / 2, -
(float)imgPhoto.Height / 2);
            g.DrawImage(imgPhoto, imageBox);
            return tempImage;
        }

        public void Rotate()
        {
            //Invokes the RotateImage function to tilt a piece up to 35
degrees
            bkKI.SetImage(bkKI.GetPosition(),
RotateImage(king_reduced_bk));
            bkQU.SetImage(bkQU.GetPosition(),
RotateImage(queen_reduced_bk));
            bkRK1.SetImage(bkRK1.GetPosition(),
RotateImage(rook_reduced_bk));
            bkRK2.SetImage(bkRK2.GetPosition(),
RotateImage(rook_reduced_bk));
            bkKN1.SetImage(bkKN1.GetPosition(),
RotateImage(knight_reduced_bk));
            bkKN2.SetImage(bkKN2.GetPosition(),
RotateImage(knight_reduced_bk));
            bkBI1.SetImage(bkBI1.GetPosition(),
RotateImage(bishop_reduced_bk));
            bkBI2.SetImage(bkBI2.GetPosition(),
RotateImage(bishop_reduced_bk));

            bkPA1.SetImage(bkPA1.GetPosition(),
RotateImage(pawn_reduced_bk));
            bkPA2.SetImage(bkPA2.GetPosition(),
RotateImage(pawn_reduced_bk));
            bkPA3.SetImage(bkPA3.GetPosition(),
RotateImage(pawn_reduced_bk));
            bkPA4.SetImage(bkPA4.GetPosition(),
RotateImage(pawn_reduced_bk));
            bkPA5.SetImage(bkPA5.GetPosition(),
RotateImage(pawn_reduced_bk));
            bkPA6.SetImage(bkPA6.GetPosition(),
RotateImage(pawn_reduced_bk));
            bkPA7.SetImage(bkPA7.GetPosition(),
RotateImage(pawn_reduced_bk));
            bkPA8.SetImage(bkPA8.GetPosition(),
RotateImage(pawn_reduced_bk));
```

```
            whKI.SetImage(whKI.GetPosition(),
RotateImage(king_reduced_wh));
            whQU.SetImage(whQU.GetPosition(),
RotateImage(queen_reduced_wh));
            whRK1.SetImage(whRK1.GetPosition(),
RotateImage(rook_reduced_wh));
            whRK2.SetImage(whRK2.GetPosition(),
RotateImage(rook_reduced_wh));
            whKN1.SetImage(whKN1.GetPosition(),
RotateImage(knight_reduced_wh));
            whKN2.SetImage(whKN2.GetPosition(),
RotateImage(knight_reduced_wh));
            whBI1.SetImage(whBI1.GetPosition(),
RotateImage(bishop_reduced_wh));
            whBI2.SetImage(whBI2.GetPosition(),
RotateImage(bishop_reduced_wh));

            whPA1.SetImage(whPA1.GetPosition(),
RotateImage(pawn_reduced_wh));
            whPA2.SetImage(whPA2.GetPosition(),
RotateImage(pawn_reduced_wh));
            whPA3.SetImage(whPA3.GetPosition(),
RotateImage(pawn_reduced_wh));
            whPA4.SetImage(whPA4.GetPosition(),
RotateImage(pawn_reduced_wh));
            whPA5.SetImage(whPA5.GetPosition(),
RotateImage(pawn_reduced_wh));
            whPA6.SetImage(whPA6.GetPosition(),
RotateImage(pawn_reduced_wh));
            whPA7.SetImage(whPA7.GetPosition(),
RotateImage(pawn_reduced_wh));
            whPA8.SetImage(whPA8.GetPosition(),
RotateImage(pawn_reduced_wh));

            this.Refresh();
        }

        public void BoardSetup()
        {
            //Creates an object for each piece on the board. The object
stores position and image.

            //Arrays of type Panel for setting up the board initially
            Panel[] backRowPanelsBK = new Panel[8] { panel2, panel3,
panel4, panel5, panel6, panel7, panel8, panel9 };
            Panel[] backRowPanelsBKpawns = new Panel[8] { panel10,
panel11, panel12, panel13, panel14, panel15, panel16, panel17 };
            Panel[] backRowPanelsWHpawns = new Panel[8] { panel50,
panel51, panel52, panel53, panel54, panel55, panel56, panel57 };
            Panel[] backRowPanelsWH = new Panel[8] { panel58, panel59,
panel60, panel61, panel62, panel63, panel64, panel65 };

            int rookFlag = 0;
            int knightFlag = 0;
            int bishopFlag = 0;
```

```
            chBoard = new ChessBoard(panel66, board);

            //Place pieces in their proper spot on the board depending
on the RandomChess function
            int temp = 0;
            foreach(Panel P in backRowPanelsBK)
            {
                if (backRow[temp] == ROOK)
                {
                    if (rookFlag == 1)
                    {
                        bkRK2 = new Rook(P, rook_reduced_bk, "rook",
"black");
                    }
                    if (rookFlag == 0)
                    {
                        bkRK1 = new Rook(P, rook_reduced_bk, "rook",
"black");
                        rookFlag = 1;
                    }
                }

                if (backRow[temp] == KNIGHT)
                {
                    if (knightFlag == 1)
                    {
                        bkKN2 = new Knight(P, knight_reduced_bk,
"knight", "black");
                    }
                    if (knightFlag == 0)
                    {
                        bkKN1 = new Knight(P, knight_reduced_bk,
"knight", "black");
                        knightFlag = 1;
                    }
                }

                    if (backRow[temp] == BISHOP)
                    {
                        if (bishopFlag == 1)
                        {
                            bkBI2 = new Bishop(P, bishop_reduced_bk,
"bishop", "black");
                        }
                        if (bishopFlag == 0)
                        {
                            bkBI1 = new Bishop(P, bishop_reduced_bk,
"bishop", "black");
                            bishopFlag = 1;
                        }
                    }

                if (backRow[temp] == KING)
                    bkKI = new King(P, king_reduced_bk, "king",
"black");
                if (backRow[temp] == QUEEN)
```

```
                    bkQU = new Queen(P, queen_reduced_bk, "queen",
"black");
                temp++;
            }

            bkPA1 = new Pawn(backRowPanelsBKpawns[0], pawn_reduced_bk,
"pawn", "black");
            bkPA2 = new Pawn(backRowPanelsBKpawns[1], pawn_reduced_bk,
"pawn", "black");
            bkPA3 = new Pawn(backRowPanelsBKpawns[2], pawn_reduced_bk,
"pawn", "black");
            bkPA4 = new Pawn(backRowPanelsBKpawns[3], pawn_reduced_bk,
"pawn", "black");
            bkPA5 = new Pawn(backRowPanelsBKpawns[4], pawn_reduced_bk,
"pawn", "black");
            bkPA6 = new Pawn(backRowPanelsBKpawns[5], pawn_reduced_bk,
"pawn", "black");
            bkPA7 = new Pawn(backRowPanelsBKpawns[6], pawn_reduced_bk,
"pawn", "black");
            bkPA8 = new Pawn(backRowPanelsBKpawns[7], pawn_reduced_bk,
"pawn", "black");

            whPA1 = new Pawn(backRowPanelsWHpawns[0], pawn_reduced_wh,
"pawn", "white");
            whPA2 = new Pawn(backRowPanelsWHpawns[1], pawn_reduced_wh,
"pawn", "white");
            whPA3 = new Pawn(backRowPanelsWHpawns[2], pawn_reduced_wh,
"pawn", "white");
            whPA4 = new Pawn(backRowPanelsWHpawns[3], pawn_reduced_wh,
"pawn", "white");
            whPA5 = new Pawn(backRowPanelsWHpawns[4], pawn_reduced_wh,
"pawn", "white");
            whPA6 = new Pawn(backRowPanelsWHpawns[5], pawn_reduced_wh,
"pawn", "white");
            whPA7 = new Pawn(backRowPanelsWHpawns[6], pawn_reduced_wh,
"pawn", "white");
            whPA8 = new Pawn(backRowPanelsWHpawns[7], pawn_reduced_wh,
"pawn", "white");

            rookFlag = 0;
            knightFlag = 0;
            bishopFlag = 0;
            temp = 0;
            foreach (Panel P in backRowPanelsWH)
            {
                if (backRow[temp] == ROOK)
                {
                    if (rookFlag == 1)
                    {
                        whRK2 = new Rook(P, rook_reduced_wh, "rook",
"white");
                    }
                    if (rookFlag == 0)
                    {
                        whRK1 = new Rook(P, rook_reduced_wh, "rook",
"white");
                        rookFlag = 1;
```

```
                }
            }

            if (backRow[temp] == KNIGHT)
            {
                if (knightFlag == 1)
                {
                    whKN2 = new Knight(P, knight_reduced_wh,
"knight", "white");
                }
                if (knightFlag == 0)
                {
                    whKN1 = new Knight(P, knight_reduced_wh,
"knight", "white");
                    knightFlag = 1;
                }
            }

            if (backRow[temp] == BISHOP)
            {
                if (bishopFlag == 1)
                {
                    whBI2 = new Bishop(P, bishop_reduced_wh,
"bishop", "white");
                }
                if (bishopFlag == 0)
                {
                    whBI1 = new Bishop(P, bishop_reduced_wh,
"bishop", "white");
                    bishopFlag = 1;
                }
            }

            if (backRow[temp] == KING)
                whKI = new King(P, king_reduced_wh, "king",
"white");
            if (backRow[temp] == QUEEN)
                whQU = new Queen(P, queen_reduced_wh, "queen",
"white");
            temp++;
        }

    }

    public void RandomResolution()
    {
        //Raduces the resolution of images by 5 - 30 percent
        Random rand = new Random();
        int randNum = rand.Next(10, 21);
        textBox1.Text = Convert.ToString(randNum);

        //Reduce the resolution of the images to make it harder for
a bot to detect the pieces
        bkKI.SetImage(bkKI.GetPosition(),
ReducePieces(king_reduced_bk, randNum));
        bkQU.SetImage(bkQU.GetPosition(),
ReducePieces(queen_reduced_bk, randNum));
```

```
            bkRK1.SetImage(bkRK1.GetPosition(),
ReducePieces(rook_reduced_bk, randNum));
            bkRK2.SetImage(bkRK2.GetPosition(),
ReducePieces(rook_reduced_bk, randNum));
            bkKN1.SetImage(bkKN1.GetPosition(),
ReducePieces(knight_reduced_bk, randNum));
            bkKN2.SetImage(bkKN2.GetPosition(),
ReducePieces(knight_reduced_bk, randNum));
            bkBI1.SetImage(bkBI1.GetPosition(),
ReducePieces(bishop_reduced_bk, randNum));
            bkBI2.SetImage(bkBI2.GetPosition(),
ReducePieces(bishop_reduced_bk, randNum));

            bkPA1.SetImage(bkPA1.GetPosition(),
ReducePieces(pawn_reduced_bk, randNum));
            bkPA2.SetImage(bkPA2.GetPosition(),
ReducePieces(pawn_reduced_bk, randNum));
            bkPA3.SetImage(bkPA3.GetPosition(),
ReducePieces(pawn_reduced_bk, randNum));
            bkPA4.SetImage(bkPA4.GetPosition(),
ReducePieces(pawn_reduced_bk, randNum));
            bkPA5.SetImage(bkPA5.GetPosition(),
ReducePieces(pawn_reduced_bk, randNum));
            bkPA6.SetImage(bkPA6.GetPosition(),
ReducePieces(pawn_reduced_bk, randNum));
            bkPA7.SetImage(bkPA7.GetPosition(),
ReducePieces(pawn_reduced_bk, randNum));
            bkPA8.SetImage(bkPA8.GetPosition(),
ReducePieces(pawn_reduced_bk, randNum));

            whKI.SetImage(whKI.GetPosition(),
ReducePieces(king_reduced_wh, randNum));
            whQU.SetImage(whQU.GetPosition(),
ReducePieces(queen_reduced_wh, randNum));
            whRK1.SetImage(whRK1.GetPosition(),
ReducePieces(rook_reduced_wh, randNum));
            whRK2.SetImage(whRK2.GetPosition(),
ReducePieces(rook_reduced_wh, randNum));
            whKN1.SetImage(whKN1.GetPosition(),
ReducePieces(knight_reduced_wh, randNum));
            whKN2.SetImage(whKN2.GetPosition(),
ReducePieces(knight_reduced_wh, randNum));
            whBI1.SetImage(whBI1.GetPosition(),
ReducePieces(bishop_reduced_wh, randNum));
            whBI2.SetImage(whBI2.GetPosition(),
ReducePieces(bishop_reduced_wh, randNum));

            whPA1.SetImage(whPA1.GetPosition(),
ReducePieces(pawn_reduced_wh, randNum));
            whPA2.SetImage(whPA2.GetPosition(),
ReducePieces(pawn_reduced_wh, randNum));
            whPA3.SetImage(whPA3.GetPosition(),
ReducePieces(pawn_reduced_wh, randNum));
            whPA4.SetImage(whPA4.GetPosition(),
ReducePieces(pawn_reduced_wh, randNum));
            whPA5.SetImage(whPA5.GetPosition(),
ReducePieces(pawn_reduced_wh, randNum));
```

```
            whPA6.SetImage(whPA6.GetPosition(),
ReducePieces(pawn_reduced_wh, randNum));
            whPA7.SetImage(whPA7.GetPosition(),
ReducePieces(pawn_reduced_wh, randNum));
            whPA8.SetImage(whPA8.GetPosition(),
ReducePieces(pawn_reduced_wh, randNum));

            this.Refresh();
        }

        protected override void OnPaint(PaintEventArgs e)
        {

        }

        private void trackBar1_Scroll(object sender, EventArgs e)
        {
            //Scroll bar for selecting how much to reduce the
resolution of the pieces.
            //The goal here is to reduce the resolution of each piece
differently

            trackBar1.Maximum = 5;
            trackBar1.Minimum = 0;
            trackBar1.TickFrequency = 1;

            //Create a random number
            Random rand = new Random();
            //Multiply the random number by the trackbar value to
somewhat guarantee
            //an increase in distortion as the trackbar moves up.
            //Values should end up between 5% (trackbar1.Value = 1 and
randNum = 5) and 50% (trackbar1.Value = 5 and randNum = 10).
            int randNum = rand.Next(5, 11) * trackBar1.Value;

            textBox1.Text = null;
            //textBox1.Text = Convert.ToString(randNum);

            //Reduce the resolution of the images to make it harder for
a bot to detect the pieces
            bkKI.SetImage(bkKI.GetPosition(),
ReducePieces(king_reduced_bk, randNum));
            randNum = rand.Next(5, 10) * trackBar1.Value;
            bkQU.SetImage(bkQU.GetPosition(),
ReducePieces(queen_reduced_bk, randNum));
            randNum = rand.Next(5, 10) * trackBar1.Value;
            bkRK1.SetImage(bkRK1.GetPosition(),
ReducePieces(rook_reduced_bk, randNum));
            randNum = rand.Next(5, 10) * trackBar1.Value;
            bkRK2.SetImage(bkRK2.GetPosition(),
ReducePieces(rook_reduced_bk, randNum));
            randNum = rand.Next(5, 10) * trackBar1.Value;
            bkKN1.SetImage(bkKN1.GetPosition(),
ReducePieces(knight_reduced_bk, randNum));
            randNum = rand.Next(5, 10) * trackBar1.Value;
            bkKN2.SetImage(bkKN2.GetPosition(),
ReducePieces(knight_reduced_bk, randNum));
```

```
            randNum = rand.Next(5, 10) * trackBar1.Value;
            bkBI1.SetImage(bkBI1.GetPosition(),
ReducePieces(bishop_reduced_bk, randNum));
            randNum = rand.Next(5, 10) * trackBar1.Value;
            bkBI2.SetImage(bkBI2.GetPosition(),
ReducePieces(bishop_reduced_bk, randNum));

            randNum = rand.Next(5, 10) * trackBar1.Value;
            bkPA1.SetImage(bkPA1.GetPosition(),
ReducePieces(pawn_reduced_bk, randNum));
            randNum = rand.Next(5, 10) * trackBar1.Value;
            bkPA2.SetImage(bkPA2.GetPosition(),
ReducePieces(pawn_reduced_bk, randNum));
            randNum = rand.Next(5, 10) * trackBar1.Value;
            bkPA3.SetImage(bkPA3.GetPosition(),
ReducePieces(pawn_reduced_bk, randNum));
            randNum = rand.Next(5, 10) * trackBar1.Value;
            bkPA4.SetImage(bkPA4.GetPosition(),
ReducePieces(pawn_reduced_bk, randNum));
            randNum = rand.Next(5, 10) * trackBar1.Value;
            bkPA5.SetImage(bkPA5.GetPosition(),
ReducePieces(pawn_reduced_bk, randNum));
            randNum = rand.Next(5, 10) * trackBar1.Value;
            bkPA6.SetImage(bkPA6.GetPosition(),
ReducePieces(pawn_reduced_bk, randNum));
            randNum = rand.Next(5, 10) * trackBar1.Value;
            bkPA7.SetImage(bkPA7.GetPosition(),
ReducePieces(pawn_reduced_bk, randNum));
            randNum = rand.Next(5, 10) * trackBar1.Value;
            bkPA8.SetImage(bkPA8.GetPosition(),
ReducePieces(pawn_reduced_bk, randNum));

            randNum = rand.Next(5, 10) * trackBar1.Value;
            whKI.SetImage(whKI.GetPosition(),
ReducePieces(king_reduced_wh, randNum));
            randNum = rand.Next(5, 10) * trackBar1.Value;
            whQU.SetImage(whQU.GetPosition(),
ReducePieces(queen_reduced_wh, randNum));
            randNum = rand.Next(5, 10) * trackBar1.Value;
            whRK1.SetImage(whRK1.GetPosition(),
ReducePieces(rook_reduced_wh, randNum));
            randNum = rand.Next(5, 10) * trackBar1.Value;
            whRK2.SetImage(whRK2.GetPosition(),
ReducePieces(rook_reduced_wh, randNum));
            randNum = rand.Next(5, 10) * trackBar1.Value;
            whKN1.SetImage(whKN1.GetPosition(),
ReducePieces(knight_reduced_wh, randNum));
            randNum = rand.Next(5, 10) * trackBar1.Value;
            whKN2.SetImage(whKN2.GetPosition(),
ReducePieces(knight_reduced_wh, randNum));
            randNum = rand.Next(5, 10) * trackBar1.Value;
            whBI1.SetImage(whBI1.GetPosition(),
ReducePieces(bishop_reduced_wh, randNum));
            randNum = rand.Next(5, 10) * trackBar1.Value;
            whBI2.SetImage(whBI2.GetPosition(),
ReducePieces(bishop_reduced_wh, randNum));
```

```
            randNum = rand.Next(5, 10) * trackBar1.Value;
            whPA1.SetImage(whPA1.GetPosition(),
ReducePieces(pawn_reduced_wh, randNum));
            randNum = rand.Next(5, 10) * trackBar1.Value;
            whPA2.SetImage(whPA2.GetPosition(),
ReducePieces(pawn_reduced_wh, randNum));
            randNum = rand.Next(5, 10) * trackBar1.Value;
            whPA3.SetImage(whPA3.GetPosition(),
ReducePieces(pawn_reduced_wh, randNum));
            randNum = rand.Next(5, 10) * trackBar1.Value;
            whPA4.SetImage(whPA4.GetPosition(),
ReducePieces(pawn_reduced_wh, randNum));
            randNum = rand.Next(5, 10) * trackBar1.Value;
            whPA5.SetImage(whPA5.GetPosition(),
ReducePieces(pawn_reduced_wh, randNum));
            randNum = rand.Next(5, 10) * trackBar1.Value;
            whPA6.SetImage(whPA6.GetPosition(),
ReducePieces(pawn_reduced_wh, randNum));
            randNum = rand.Next(5, 10) * trackBar1.Value;
            whPA7.SetImage(whPA7.GetPosition(),
ReducePieces(pawn_reduced_wh, randNum));
            randNum = rand.Next(5, 10) * trackBar1.Value;
            whPA8.SetImage(whPA8.GetPosition(),
ReducePieces(pawn_reduced_wh, randNum));

            this.Refresh();
        }

        private void Ok_Click(object sender, EventArgs e)
        {
            //Reduces resolution by a percentage input by user
            string input = textBox1.Text;
            try
            {
                int intInput = Convert.ToInt32(input);

                //Reduce the resolution of the images by percent
                bkKI.SetImage(bkKI.GetPosition(),
ReducePieces(king_reduced_bk, intInput));
                bkQU.SetImage(bkQU.GetPosition(),
ReducePieces(queen_reduced_bk, intInput));
                bkRK1.SetImage(bkRK1.GetPosition(),
ReducePieces(rook_reduced_bk, intInput));
                bkRK2.SetImage(bkRK2.GetPosition(),
ReducePieces(rook_reduced_bk, intInput));
                bkKN1.SetImage(bkKN1.GetPosition(),
ReducePieces(knight_reduced_bk, intInput));
                bkKN2.SetImage(bkKN2.GetPosition(),
ReducePieces(knight_reduced_bk, intInput));
                bkBI1.SetImage(bkBI1.GetPosition(),
ReducePieces(bishop_reduced_bk, intInput));
                bkBI2.SetImage(bkBI2.GetPosition(),
ReducePieces(bishop_reduced_bk, intInput));

                bkPA1.SetImage(bkPA1.GetPosition(),
ReducePieces(pawn_reduced_bk, intInput));
```

```
                bkPA2.SetImage(bkPA2.GetPosition(),
ReducePieces(pawn_reduced_bk, intInput));
                bkPA3.SetImage(bkPA3.GetPosition(),
ReducePieces(pawn_reduced_bk, intInput));
                bkPA4.SetImage(bkPA4.GetPosition(),
ReducePieces(pawn_reduced_bk, intInput));
                bkPA5.SetImage(bkPA5.GetPosition(),
ReducePieces(pawn_reduced_bk, intInput));
                bkPA6.SetImage(bkPA6.GetPosition(),
ReducePieces(pawn_reduced_bk, intInput));
                bkPA7.SetImage(bkPA7.GetPosition(),
ReducePieces(pawn_reduced_bk, intInput));
                bkPA8.SetImage(bkPA8.GetPosition(),
ReducePieces(pawn_reduced_bk, intInput));

                whKI.SetImage(whKI.GetPosition(),
ReducePieces(king_reduced_wh, intInput));
                whQU.SetImage(whQU.GetPosition(),
ReducePieces(queen_reduced_wh, intInput));
                whRK1.SetImage(whRK1.GetPosition(),
ReducePieces(rook_reduced_wh, intInput));
                whRK2.SetImage(whRK2.GetPosition(),
ReducePieces(rook_reduced_wh, intInput));
                whKN1.SetImage(whKN1.GetPosition(),
ReducePieces(knight_reduced_wh, intInput));
                whKN2.SetImage(whKN2.GetPosition(),
ReducePieces(knight_reduced_wh, intInput));
                whBI1.SetImage(whBI1.GetPosition(),
ReducePieces(bishop_reduced_wh, intInput));
                whBI2.SetImage(whBI2.GetPosition(),
ReducePieces(bishop_reduced_wh, intInput));

                whPA1.SetImage(whPA1.GetPosition(),
ReducePieces(pawn_reduced_wh, intInput));
                whPA2.SetImage(whPA2.GetPosition(),
ReducePieces(pawn_reduced_wh, intInput));
                whPA3.SetImage(whPA3.GetPosition(),
ReducePieces(pawn_reduced_wh, intInput));
                whPA4.SetImage(whPA4.GetPosition(),
ReducePieces(pawn_reduced_wh, intInput));
                whPA5.SetImage(whPA5.GetPosition(),
ReducePieces(pawn_reduced_wh, intInput));
                whPA6.SetImage(whPA6.GetPosition(),
ReducePieces(pawn_reduced_wh, intInput));
                whPA7.SetImage(whPA7.GetPosition(),
ReducePieces(pawn_reduced_wh, intInput));
                whPA8.SetImage(whPA8.GetPosition(),
ReducePieces(pawn_reduced_wh, intInput));

                this.Refresh();
            }

            //Error checking
            catch
            {
                MessageBox.Show("Please input an integer between 0 and
99.");
```

```
            }
        }

        private void textBox1_KeyPress(object sender, KeyPressEventArgs
e)
        {
            //Ensures that only numbers and "delete" are accepted in
the text box
            const char Delete = (char)8;
            e.Handled = !char.IsDigit(e.KeyChar) && e.KeyChar !=
Delete;
        }

        private void Randomize_Click(object sender, EventArgs e)
        {
            //Calls the RandomChess function
            if (begin == 0)
            {
                RandomChess();
                BoardSetup();
                RandomResolution();
                this.Refresh();
            }
        }

        ChessPiece MovePiece(Panel input)
        {
            //Function to handle moving a chess piece
            if (bkRK1.GetPosition() == input)
                return bkRK1;
            if (bkRK2.GetPosition() == input)
                return bkRK2;
            if (whRK1.GetPosition() == input)
                return whRK1;
            if (whRK2.GetPosition() == input)
                return whRK2;
            if (bkKN1.GetPosition() == input)
                return bkKN1;
            if (bkKN2.GetPosition() == input)
                return bkKN2;
            if (whKN1.GetPosition() == input)
                return whKN1;
            if (whKN2.GetPosition() == input)
                return whKN2;
            if (bkBI1.GetPosition() == input)
                return bkBI1;
            if (bkBI2.GetPosition() == input)
                return bkBI2;
            if (whBI1.GetPosition() == input)
                return whBI1;
            if (whBI2.GetPosition() == input)
                return whBI2;
            if (bkKI.GetPosition() == input)
                return bkKI;
            if (bkQU.GetPosition() == input)
                return bkQU;
            if (whKI.GetPosition() == input)
```

```
                return whKI;
            if (whQU.GetPosition() == input)
                return whQU;
            if (bkPA1.GetPosition() == input)
                return bkPA1;
            if (bkPA2.GetPosition() == input)
                return bkPA2;
            if (bkPA3.GetPosition() == input)
                return bkPA3;
            if (bkPA4.GetPosition() == input)
                return bkPA4;
            if (bkPA5.GetPosition() == input)
                return bkPA5;
            if (bkPA6.GetPosition() == input)
                return bkPA6;
            if (bkPA7.GetPosition() == input)
                return bkPA7;
            if (bkPA8.GetPosition() == input)
                return bkPA8;
            if (whPA1.GetPosition() == input)
                return whPA1;
            if (whPA2.GetPosition() == input)
                return whPA2;
            if (whPA3.GetPosition() == input)
                return whPA3;
            if (whPA4.GetPosition() == input)
                return whPA4;
            if (whPA5.GetPosition() == input)
                return whPA5;
            if (whPA6.GetPosition() == input)
                return whPA6;
            if (whPA7.GetPosition() == input)
                return whPA7;
            if (whPA8.GetPosition() == input)
                return whPA8;
            return null;
        }

        void PossibleMoves(ChessPiece ch, int row, int column)
        {
            //Function to create an array of valid moves for a chess
piece
            if (ch == null)
                return;
            if (ch.GetName() == "rook")
            {
                ChessPiece testPiece;

                //Check possible moves to the South
                try
                {
                    testPiece = MovePiece(grid[row + 1, column]);
                    if (isSouthOrSEFinished == false)
                    {
                        if (testPiece != null && SameTeam(testPiece) ==
false)
                        {
```

```
                            Array.Resize<Panel>(ref moveArray,
moveArray.Length + 1);
                            moveArray[moveArray.Length - 1] = grid[row
+ 1, column];
                            isOpposingPiece = true;
                        }
                        else if (testPiece == null)
                        {
                            PossibleMoves(ch, row + 1, column);
                            Array.Resize<Panel>(ref moveArray,
moveArray.Length + 1);
                            moveArray[moveArray.Length - 1] = grid[row
+ 1, column];
                        }
                    }
                }
                catch { }
                if (ch.GetPosition() != grid[row, column] &&
isSouthOrSEFinished == false)
                    return;

                //Check possible moves to the North
                isSouthOrSEFinished = true;
                try
                {
                    testPiece = MovePiece(grid[row - 1, column]);
                    if (isNorthOrNWFinished == false)
                    {
                        if (testPiece != null && SameTeam(testPiece) ==
false)
                        {
                           Array.Resize<Panel>(ref moveArray,
moveArray.Length + 1);
                           moveArray[moveArray.Length - 1] = grid[row -
1, column];
                           isOpposingPiece = true;
                        }
                        else if (testPiece == null)
                        {
                            PossibleMoves(ch, row - 1, column);
                            Array.Resize<Panel>(ref moveArray,
moveArray.Length + 1);
                            moveArray[moveArray.Length - 1] = grid[row
- 1, column];
                        }
                    }
                }

                catch { }
                if (ch.GetPosition() != grid[row, column] &&
isNorthOrNWFinished == false)
                    return;

                //Check possible moves to the East
                isNorthOrNWFinished = true;
                try
                {
```

```
                    testPiece = MovePiece(grid[row, column + 1]);
                    if (isEastOrNEFinished == false)
                    {
                        if (testPiece != null && SameTeam(testPiece) ==
false)
                        {
                            Array.Resize<Panel>(ref moveArray,
moveArray.Length + 1);
                            moveArray[moveArray.Length - 1] = grid[row,
column + 1];
                            isOpposingPiece = true;
                        }
                        else if (testPiece == null)
                        {
                            PossibleMoves(ch, row, column + 1);
                            Array.Resize<Panel>(ref moveArray,
moveArray.Length + 1);
                            moveArray[moveArray.Length - 1] = grid[row,
column + 1];
                        }
                    }
                }
                catch { }
                if (ch.GetPosition() != grid[row, column] &&
isEastOrNEFinished == false)
                    return;

                //Check Possible moves to the West
                isEastOrNEFinished = true;
                try
                {
                    testPiece = MovePiece(grid[row, column - 1]);
                    if (testPiece != null && SameTeam(testPiece) ==
false)
                    {
                        Array.Resize<Panel>(ref moveArray,
moveArray.Length + 1);
                        moveArray[moveArray.Length - 1] = grid[row,
column - 1];
                        isOpposingPiece = true;
                    }
                    else if (testPiece == null)
                    {
                        PossibleMoves(ch, row, column - 1);
                        Array.Resize<Panel>(ref moveArray,
moveArray.Length + 1);
                        moveArray[moveArray.Length - 1] = grid[row,
column - 1];
                    }
                }
                catch { }
                if (ch.GetPosition() != grid[row, column])
                    return;

                return;
            }
```

```
            if (ch.GetName() == "knight")
            {
                if (ch == null)
                    return;

                ChessPiece testPiece;
                //Possible moves to the North
                try
                {
                    testPiece = MovePiece(grid[row + 2, column + 1]);
                    if (testPiece != null && SameTeam(testPiece) ==
false)
                    {
                        Array.Resize<Panel>(ref moveArray,
moveArray.Length + 1);
                        moveArray[moveArray.Length - 1] = grid[row + 2,
column + 1];
                        isOpposingPiece = true;
                    }
                    else if (testPiece == null)
                    {
                        Array.Resize<Panel>(ref moveArray,
moveArray.Length + 1);
                        moveArray[moveArray.Length - 1] = grid[row + 2,
column + 1];
                    }
                }
                catch { }

                try
                {
                    testPiece = MovePiece(grid[row + 2, column - 1]);
                    if (testPiece != null && SameTeam(testPiece) ==
false)
                    {
                        Array.Resize<Panel>(ref moveArray,
moveArray.Length + 1);
                        moveArray[moveArray.Length - 1] = grid[row + 2,
column - 1];
                        isOpposingPiece = true;
                    }
                    else if (testPiece == null)
                    {
                        Array.Resize<Panel>(ref moveArray,
moveArray.Length + 1);
                        moveArray[moveArray.Length - 1] = grid[row + 2,
column - 1];
                    }
                }
                catch { }

                try
                {
                    testPiece = MovePiece(grid[row + 1, column + 2]);
                    if (testPiece != null && SameTeam(testPiece) ==
false)
                    {
```

```
                            Array.Resize<Panel>(ref moveArray,
moveArray.Length + 1);
                            moveArray[moveArray.Length - 1] = grid[row + 1,
column + 2];
                            isOpposingPiece = true;
                        }
                        else if (testPiece == null)
                        {
                            Array.Resize<Panel>(ref moveArray,
moveArray.Length + 1);
                            moveArray[moveArray.Length - 1] = grid[row + 1,
column + 2];
                        }
                    }
                    catch { }

                    try
                    {
                        testPiece = MovePiece(grid[row + 1, column - 2]);
                        if (testPiece != null && SameTeam(testPiece) ==
false)
                        {
                            Array.Resize<Panel>(ref moveArray,
moveArray.Length + 1);
                            moveArray[moveArray.Length - 1] = grid[row + 1,
column - 2];
                            isOpposingPiece = true;
                        }
                        else if (testPiece == null)
                        {
                            Array.Resize<Panel>(ref moveArray,
moveArray.Length + 1);
                            moveArray[moveArray.Length - 1] = grid[row + 1,
column - 2];
                        }
                    }
                    catch { }

                    //Diagonal
                    try
                    {
                        testPiece = MovePiece(grid[row - 2, column + 1]);
                        if (testPiece != null && SameTeam(testPiece) ==
false)
                        {
                            Array.Resize<Panel>(ref moveArray,
moveArray.Length + 1);
                            moveArray[moveArray.Length - 1] = grid[row - 2,
column + 1];
                            isOpposingPiece = true;
                        }
                        else if (testPiece == null)
                        {
                            Array.Resize<Panel>(ref moveArray,
moveArray.Length + 1);
                            moveArray[moveArray.Length - 1] = grid[row - 2,
column + 1];
```

```
                    }
                }
                catch { }

                try
                {
                    testPiece = MovePiece(grid[row - 2, column - 1]);
                    if (testPiece != null && SameTeam(testPiece) ==
false)
                    {
                        Array.Resize<Panel>(ref moveArray,
moveArray.Length + 1);
                        moveArray[moveArray.Length - 1] = grid[row - 2,
column - 1];
                        isOpposingPiece = true;
                    }
                    else if (testPiece == null)
                    {
                        Array.Resize<Panel>(ref moveArray,
moveArray.Length + 1);
                        moveArray[moveArray.Length - 1] = grid[row - 2,
column - 1];
                    }
                }
                catch { }

                try
                {
                    testPiece = MovePiece(grid[row - 1, column + 2]);
                    if (testPiece != null && SameTeam(testPiece) ==
false)
                    {
                        Array.Resize<Panel>(ref moveArray,
moveArray.Length + 1);
                        moveArray[moveArray.Length - 1] = grid[row - 1,
column + 2];
                        isOpposingPiece = true;
                    }
                    else if (testPiece == null)
                    {
                        Array.Resize<Panel>(ref moveArray,
moveArray.Length + 1);
                        moveArray[moveArray.Length - 1] = grid[row - 1,
column + 2];
                    }
                }
                catch { }

                try
                {
                    testPiece = MovePiece(grid[row - 1, column - 2]);
                    if (testPiece != null && SameTeam(testPiece) ==
false)
                    {
                        Array.Resize<Panel>(ref moveArray,
moveArray.Length + 1);
```

```
                        moveArray[moveArray.Length - 1] = grid[row - 1,
column - 2];
                        isOpposingPiece = true;
                    }
                    else if (testPiece == null)
                    {
                        Array.Resize<Panel>(ref moveArray,
moveArray.Length + 1);
                        moveArray[moveArray.Length - 1] = grid[row - 1,
column - 2];
                    }
                }
                catch { }

                return;
            }
            if (ch.GetName() == "bishop")
            {
                if (ch == null)
                    return;

                ChessPiece testPiece;

                //Check possible moves to the Southeast
                try
                {
                    testPiece = MovePiece(grid[row + 1, column + 1]);
                    if (isSouthOrSEFinished == false)
                    {
                        if (testPiece != null && SameTeam(testPiece) ==
false)

                        {
                            Array.Resize<Panel>(ref moveArray,
moveArray.Length + 1);
                            moveArray[moveArray.Length - 1] = grid[row
+ 1, column + 1];
                            isOpposingPiece = true;
                        }
                        else if (testPiece == null)
                        {
                            PossibleMoves(ch, row + 1, column + 1);
                            Array.Resize<Panel>(ref moveArray,
moveArray.Length + 1);
                            moveArray[moveArray.Length - 1] = grid[row
+ 1, column + 1];
                        }
                    }
                }
                catch { }
                if (ch.GetPosition() != grid[row, column] &&
isSouthOrSEFinished == false)
                    return;

                //Check possible moves to the Northwest
                isSouthOrSEFinished = true;
                try
                {
```

```
                        testPiece = MovePiece(grid[row - 1, column - 1]);
                        if (isNorthOrNWFinished == false)
                        {
                            if (testPiece != null && SameTeam(testPiece) ==
false)
                            {
                                Array.Resize<Panel>(ref moveArray,
moveArray.Length + 1);
                                moveArray[moveArray.Length - 1] = grid[row
- 1, column - 1];
                                isOpposingPiece = true;
                            }
                            else if (testPiece == null)
                            {
                                PossibleMoves(ch, row - 1, column - 1);
                                Array.Resize<Panel>(ref moveArray,
moveArray.Length + 1);
                                moveArray[moveArray.Length - 1] = grid[row
- 1, column - 1];
                            }
                        }
                    }

                    catch { }
                    if (ch.GetPosition() != grid[row, column] &&
isNorthOrNWFinished == false)
                        return;

                    //Check possible moves to the Northeast
                    isNorthOrNWFinished = true;
                    try
                    {
                        testPiece = MovePiece(grid[row - 1, column + 1]);
                        if (isEastOrNEFinished == false)
                        {
                            if (testPiece != null && SameTeam(testPiece) ==
false)
                            {
                                Array.Resize<Panel>(ref moveArray,
moveArray.Length + 1);
                                moveArray[moveArray.Length - 1] = grid[row
- 1, column + 1];
                                isOpposingPiece = true;
                            }
                            else if (testPiece == null)
                            {
                                PossibleMoves(ch, row - 1, column + 1);
                                Array.Resize<Panel>(ref moveArray,
moveArray.Length + 1);
                                moveArray[moveArray.Length - 1] = grid[row
- 1, column + 1];
                            }
                        }
                    }
                    catch { }
                    if (ch.GetPosition() != grid[row, column] &&
isEastOrNEFinished == false)
```

```
                    return;

                //Check Possible moves to the Southwest
                isEastOrNEFinished = true;
                try
                {
                    testPiece = MovePiece(grid[row + 1, column - 1]);
                    if (testPiece != null && SameTeam(testPiece) ==
false)
                    {
                            Array.Resize<Panel>(ref moveArray,
moveArray.Length + 1);
                            moveArray[moveArray.Length - 1] = grid[row
+ 1, column - 1];
                            isOpposingPiece = true;
                    }
                    else if (testPiece == null)
                    {
                        PossibleMoves(ch, row + 1, column - 1);
                        Array.Resize<Panel>(ref moveArray,
moveArray.Length + 1);
                        moveArray[moveArray.Length - 1] = grid[row + 1,
column - 1];
                    }
                }
                catch { }
                if (ch.GetPosition() != grid[row, column])
                    return;

                return;
            }
            if (ch.GetName() == "king")
            {
                if (ch == null)
                    return;

                ChessPiece testPiece;
                //Vertical and Horizontal
                try
                {
                    testPiece = MovePiece(grid[row + 1, column]);
                    if (testPiece != null && SameTeam(testPiece) ==
false)
                    {
                        Array.Resize<Panel>(ref moveArray,
moveArray.Length + 1);
                        moveArray[moveArray.Length - 1] = grid[row + 1,
column];
                        isOpposingPiece = true;
                    }
                    else if (testPiece == null)
                    {
                        Array.Resize<Panel>(ref moveArray,
moveArray.Length + 1);
                        moveArray[moveArray.Length - 1] = grid[row + 1,
column];
                    }
```

```
                }
                catch { }

                try
                {
                    testPiece = MovePiece(grid[row - 1, column]);
                    if (testPiece != null && SameTeam(testPiece) ==
false)
                    {
                        Array.Resize<Panel>(ref moveArray,
moveArray.Length + 1);
                        moveArray[moveArray.Length - 1] = grid[row - 1,
column];

                        isOpposingPiece = true;
                    }
                    else if (testPiece == null)
                    {
                        Array.Resize<Panel>(ref moveArray,
moveArray.Length + 1);
                        moveArray[moveArray.Length - 1] = grid[row - 1,
column];
                    }
                }
                catch { }

                try
                {
                    testPiece = MovePiece(grid[row, column + 1]);
                    if (testPiece != null && SameTeam(testPiece) ==
false)
                    {
                        Array.Resize<Panel>(ref moveArray,
moveArray.Length + 1);
                        moveArray[moveArray.Length - 1] = grid[row,
column + 1];

                        isOpposingPiece = true;
                    }
                    else if (testPiece == null)
                    {
                        Array.Resize<Panel>(ref moveArray,
moveArray.Length + 1);
                        moveArray[moveArray.Length - 1] = grid[row,
column + 1];
                    }
                }
                catch { }

                try
                {
                    testPiece = MovePiece(grid[row, column - 1]);
                    if (testPiece != null && SameTeam(testPiece) ==
false)
                    {
                        Array.Resize<Panel>(ref moveArray,
moveArray.Length + 1);
                        moveArray[moveArray.Length - 1] = grid[row,
column - 1];
```

```
                        isOpposingPiece = true;
                    }
                    else if (testPiece == null)
                    {
                        Array.Resize<Panel>(ref moveArray,
moveArray.Length + 1);
                        moveArray[moveArray.Length - 1] = grid[row,
column - 1];
                    }
                }
                catch { }

                //Diagonal
                try
                {
                    testPiece = MovePiece(grid[row + 1, column + 1]);
                    if (testPiece != null && SameTeam(testPiece) ==
false)
                    {
                        Array.Resize<Panel>(ref moveArray,
moveArray.Length + 1);
                        moveArray[moveArray.Length - 1] = grid[row + 1,
column + 1];
                        isOpposingPiece = true;
                    }
                    else if (testPiece == null)
                    {
                        Array.Resize<Panel>(ref moveArray,
moveArray.Length + 1);
                        moveArray[moveArray.Length - 1] = grid[row + 1,
column + 1];
                    }
                }
                catch { }

                try
                {
                    testPiece = MovePiece(grid[row + 1, column - 1]);
                    if (testPiece != null && SameTeam(testPiece) ==
false)
                    {
                        Array.Resize<Panel>(ref moveArray,
moveArray.Length + 1);
                        moveArray[moveArray.Length - 1] = grid[row + 1,
column - 1];
                        isOpposingPiece = true;
                    }
                    else if (testPiece == null)
                    {
                        Array.Resize<Panel>(ref moveArray,
moveArray.Length + 1);
                        moveArray[moveArray.Length - 1] = grid[row + 1,
column - 1];
                    }
                }
                catch { }
```

```
                try
                {
                    testPiece = MovePiece(grid[row - 1, column + 1]);
                    if (testPiece != null && SameTeam(testPiece) ==
false)
                    {
                        Array.Resize<Panel>(ref moveArray,
moveArray.Length + 1);
                        moveArray[moveArray.Length - 1] = grid[row - 1,
column + 1];
                        isOpposingPiece = true;
                    }
                    else if (testPiece == null)
                    {
                        Array.Resize<Panel>(ref moveArray,
moveArray.Length + 1);
                        moveArray[moveArray.Length - 1] = grid[row - 1,
column + 1];
                    }
                }
                catch { }

                try
                {
                    testPiece = MovePiece(grid[row - 1, column - 1]);
                    if (testPiece != null && SameTeam(testPiece) ==
false)
                    {
                        Array.Resize<Panel>(ref moveArray,
moveArray.Length + 1);
                        moveArray[moveArray.Length - 1] = grid[row - 1,
column - 1];
                        isOpposingPiece = true;
                    }
                    else if (testPiece == null)
                    {
                        Array.Resize<Panel>(ref moveArray,
moveArray.Length + 1);
                        moveArray[moveArray.Length - 1] = grid[row - 1,
column - 1];
                    }
                }
                catch { }

                return;

            }
            if (ch.GetName() == "queen")
            {
                //Create array of possible moves for queen by adding
rook moves and bishop moves
                ch.SetName(ch, "rook");
                PossibleMoves(ch, row, column);
                isNorthOrNWFinished = false;
                isEastOrNEFinished = false;
                isSouthOrSEFinished = false;
                ch.SetName(ch, "bishop");
```

```
                    PossibleMoves(ch, row, column);
                    ch.SetName(ch, "queen");

                }
                if (ch.GetName() == "pawn")
                {
                    if (ch == null)
                        return;

                    ChessPiece testPiece;

                    if (ch.GetTeam() == "black")
                    {
                        //Create array of possible moves for black pawn
                        try
                        {
                            testPiece = MovePiece(grid[row + 1, column]);
                            if (row == 1)
                            {
                                if (MovePiece(grid[row + 2, column]) ==
null)

                                {
                                    Array.Resize<Panel>(ref moveArray,
moveArray.Length + 1);
                                    moveArray[moveArray.Length - 1] =
grid[row + 2, column];
                                }
                            }

                            if (testPiece == null)
                            {
                                Array.Resize<Panel>(ref moveArray,
moveArray.Length + 1);
                                moveArray[moveArray.Length - 1] = grid[row
+ 1, column];
                            }
                        }
                        catch{}

                        try
                        {
                            testPiece = MovePiece(grid[row + 1, column +
1]);
                            if (testPiece != null && SameTeam(testPiece) ==
false)
                            {
                                Array.Resize<Panel>(ref moveArray,
moveArray.Length + 1);
                                moveArray[moveArray.Length - 1] = grid[row
+ 1, column + 1];
                                isOpposingPiece = true;
                            }
                        }
                        catch{}

                        try
                        {
```

```
                            testPiece = MovePiece(grid[row + 1, column -
1]);
                            if (testPiece != null && SameTeam(testPiece) ==
false)
                            {
                                Array.Resize<Panel>(ref moveArray,
moveArray.Length + 1);
                                moveArray[moveArray.Length - 1] = grid[row
+ 1, column - 1];
                                isOpposingPiece = true;
                            }
                        }
                        catch { }
                    }

                    if (ch.GetTeam() == "white")
                    {
                        //Create array of possible moves for white pawn
                        try
                        {
                            testPiece = MovePiece(grid[row - 1, column]);
                            if (row == 6)
                            {
                                if (MovePiece(grid[row - 2, column]) ==
null)
                                {
                                    Array.Resize<Panel>(ref moveArray,
moveArray.Length + 1);
                                    moveArray[moveArray.Length - 1] =
grid[row - 2, column];
                                }
                            }

                            if (testPiece == null)
                            {
                                Array.Resize<Panel>(ref moveArray,
moveArray.Length + 1);
                                moveArray[moveArray.Length - 1] = grid[row
- 1, column];
                            }
                        }
                        catch { }

                        try
                        {
                            testPiece = MovePiece(grid[row - 1, column +
1]);
                            if (testPiece != null && SameTeam(testPiece) ==
false)
                            {
                                Array.Resize<Panel>(ref moveArray,
moveArray.Length + 1);
                                moveArray[moveArray.Length - 1] = grid[row
- 1, column + 1];
                                isOpposingPiece = true;
                            }
                        }
```

```
                    catch { }

                    try
                    {
                        testPiece = MovePiece(grid[row - 1, column -
1]);
                        if (testPiece != null && SameTeam(testPiece) ==
false)
                        {
                            Array.Resize<Panel>(ref moveArray,
moveArray.Length + 1);
                            moveArray[moveArray.Length - 1] = grid[row
- 1, column - 1];
                            isOpposingPiece = true;
                        }
                    }
                    catch { }
                }
            }
                return;
        }

        bool SameTeam(ChessPiece ch)
        {
            //Simple check to see if 2 peices are the same color
            if (ch != null && ch.GetTeam() ==
tempPieceSource.GetTeam())
                return true;
            else if (ch == null)
            {
                return false;
            }
            else
            {
                return false;
            }
        }

        private void panel_MouseDown(object sender, MouseEventArgs e)
        {
            //Checks the panel the user has clicked on, determines what
piece is there, and calls
            //the function PossibleMoves to determine what spaces it
can move to.

            Panel source = (Panel)sender;
            tempSource = source;
            tempPieceSource = MovePiece(source);
            if (tempPieceSource == null)
                return;
            if (turnFlag == 0 && tempPieceSource.GetTeam() == "white")
            {
                tempPieceSource = null;
            }
            if (turnFlag == 1 && tempPieceSource.GetTeam() == "black")
            {
                tempPieceSource = null;
```

```
            }
            if (tempPieceSource == null)
                return;
            int row = 0, column = 0;
            for (int i = 0; i < 8; i++)
            {
                for (int j = 0; j < 8; j++)
                {
                    if (grid[i, j] == tempPieceSource.GetPosition())
                    {
                        row = i;
                        column = j;
                    }
                }
            }

            isOpposingPiece = false;
            isNorthOrNWFinished = false;
            isEastOrNEFinished = false;
            isSouthOrSEFinished = false;
            Array.Resize<Panel>(ref moveArray, 0);
            PossibleMoves(tempPieceSource, row, column);

            DoDragDrop(source.BackgroundImage, DragDropEffects.Move);
            //If a piece is moved, set the image for the vacated space
to null
            if (tempSource != tempDest && isMoveValid == true)
            {
                source.BackgroundImage = null;
                if (turnFlag == 0)
                    turnFlag = 1;
                else turnFlag = 0;
                //Disable the randomize function once a game is in
progress
                begin = 1;
            }
        }

        private void panel_DragEnter(object sender, DragEventArgs e)
        {
            //Grabs image data
            if (e.Data.GetDataPresent(typeof(Bitmap)))
            {
                e.Effect = DragDropEffects.Move;
            }
            else
            {
                e.Effect = DragDropEffects.None;
            }
        }

        private void panel_DragDrop(object sender, DragEventArgs e)
        {
            //This executes when the mouse button is realesed after
pickup up a piece and moving it over the desired destination.
            //If the destination panel is in the moveArray, the image
that was picked up replaces the destination image.
```

```
            Panel destination = (Panel)sender;
            tempDest = destination;
            tempPieceDest = MovePiece(destination);
            isMoveValid = false;        //Used to make sure the
background image isn't removed unless the move is valid

            //Check the array of possible moves against the destination
of the Drag/Drop operation and move the piece if allowed
            foreach(Panel P in moveArray)
            {
                if(destination == P)
                {
                    //If user tries to move a piece on top of their own
piece, this statement will prevent it
                    if (SameTeam(tempPieceDest) == true)
                    {
                        break;
                    }
                    //If user captures an opposing piece, remove the
captured piece from the board
                    if (tempPieceDest != null && isOpposingPiece ==
true)
                    {
                        tempPieceDest.SetPosition(panel67);
                    }

                    isMoveValid = true;
                    tempPieceSource.SetPosition(destination);
                    destination.BackgroundImage =
(Bitmap)e.Data.GetData(typeof(Bitmap));
                }
            }
        }

        private void exitToolStripMenuItem_Click(object sender,
EventArgs e)
        {
            Application.Exit();
        }

        private void newToolStripMenuItem_Click(object sender,
EventArgs e)
        {
            Application.Restart();
        }

        private void standardToolStripMenuItem1_Click(object sender,
EventArgs e)
        {
            gameType = standardToolStripMenuItem1.Text;
            textBox1.Text = null;
            InitializeImages();
        }

        private void uLUKToolStripMenuItem1_Click(object sender,
EventArgs e)
        {
```

```
            gameType = uLUKToolStripMenuItem1.Text;
            textBox1.Text = null;
            InitializeImages();
        }

        private void uLUK2ToolStripMenuItem_Click(object sender,
EventArgs e)
        {
            gameType = uLUK2ToolStripMenuItem.Text;
            textBox1.Text = null;
            InitializeImages();
        }

        private void aboutToolStripMenuItem_Click(object sender,
EventArgs e)
        {
            MessageBox.Show(@"      White pawns are placed on their
normal home squares.
       The white king is placed somewhere between the two white rooks.
       The white bishops are placed on opposite-colored squares.
       The black pieces are placed equal-and-opposite to the white
pieces.
       For example, if white's king is placed on b1, then black's king
is placed on b8.", "Fischer Random Chess");
        }

        private void Form1_Load(object sender, EventArgs e)
        {

        }
    }
}
```

```
using System;

using System.Collections.Generic;
using System.Linq;
using System.Windows.Forms;
using System.Drawing.Imaging;
using System.Drawing;

namespace ChessBotBreaker
{
    class ChessPiece : Form
    {
        protected Panel position;
        protected string name;
        protected string teamColor;

        public ChessPiece()
        {

        }

        public void SetPosition(Panel pan)
        {
            this.position = pan;
        }

        public Panel GetPosition()
        {
            return position;
        }

        public void SetImage(Panel pos, Image img)
        {
            pos.BackgroundImage = img;
        }

        public string GetName()
        {
            return name;
        }

        public void SetName(ChessPiece ch, string na)
        {
            ch.name = na;
        }

        public string GetTeam()
        {
            return teamColor;
        }
    }

    //Individual pieces inherit from ChessPiece
    class King : ChessPiece
    {
        public King(Panel pos, Image img, string pieceName, string
team)
```

```
        {
            this.teamColor = team;
            this.name = pieceName;
            this.position = pos;
            pos.BackgroundImage = img;
        }
    }

    class Queen : ChessPiece
    {
        public Queen(Panel pos, Image img, string pieceName, string
team)
        {
            this.teamColor = team;
            this.name = pieceName;
            this.position = pos;
            pos.BackgroundImage = img;
        }
    }

    class Rook : ChessPiece
    {
        public Rook(Panel pos, Image img, string pieceName, string
team)
        {
            this.teamColor = team;
            this.name = pieceName;
            this.position = pos;
            pos.BackgroundImage = img;
        }
    }

    class Knight : ChessPiece
    {
        public Knight(Panel pos, Image img, string pieceName, string
team)
        {
            this.teamColor = team;
            this.name = pieceName;
            this.position = pos;
            pos.BackgroundImage = img;
        }
    }

    class Bishop : ChessPiece
    {
        public Bishop(Panel pos, Image img, string pieceName, string
team)
        {
            this.teamColor = team;
            this.name = pieceName;
            this.position = pos;
            pos.BackgroundImage = img;
        }
    }

    class Pawn : ChessPiece
```

```
    {
        public Pawn(Panel pos, Image img, string pieceName, string
team)
        {
            this.teamColor = team;
            this.name = pieceName;
            this.position = pos;
            pos.BackgroundImage = img;
        }
    }

    class ChessBoard : ChessPiece
    {
        public ChessBoard(Panel pos, Image img)
        {
            this.position = pos;
            pos.BackgroundImage = img;
        }
    }
}
```

APPENDIX II. FEEDBACK FORM

Embedded Non-Interactive
CAPTCHA for Fischer Random Chess

Adjust the distortion to desired level.
Try playing the game for at least 5-10 minutes.

What level of distortion was applied? (eg. 17% or trackbar tick #2)	
Your gender	☐ Male ☐ Female
Your age	
Chess skill (Beginner, Intermediate, Advanced)	☐ Beginner ☐ Intermediate ☐ Advanced
How many times have you played online chess?	☐ 0 ☐ <10 ☐ >10
How recently have you played online chess?	☐ Less than a month ago ☐ 1-6 months ago ☐ >6 months ago
On a scale of 1 – 10, with 10 being very difficult, how difficult was it to recognize pieces after distortion?	
Approximately how long did it take to get used to the look of the distorted pieces?	☐ Less than 3 moves ☐ 4-7 moves ☐ >8 moves
Have you ever played Fischer Random Chess?	☐ Yes ☐ No
Comments	

VITA

Professional Experience

Technology Specialist Senior

- 2007-Present
- Provide first-level user support within the President's Office and Office of Community Engagement for 36 users, including troubleshooting and training
- Provide support for the University of Louisville Foundation and its subsidiaries, which includes NUCLEUS and the University of Louisville Development Corporation (ULDC)
- Manage projects to add and upgrade AV equipment, both on campus and off
- Assist NUCLEUS and ULDC in making decisions on the technology aspects of construction projects

Administrative Support Specialist University of Louisville, Louisville, KY

- 2001-2007
- Provide first-level user support within the President's Office for 23 users, including troubleshooting and training
- Create and maintain websites vital to the mission of the university
- Support and maintain computer and servers within the department's LAN, including disaster recovery preparedness
- Serve as liaison between the President's Office and the university's central IT group
- Plan and coordinate acquisition and management of desktop and LAN technologies, including data, voice, and video
- Set up and maintain enterprise and personal email for Blackberry

Technology Solutions Group Brown-Forman Corp., Louisville, KY

- 2003 (Co-op Position)
- Provided set up and implementation of new machines
- Managed users via Active Directory
- Provided first-level support for a userbase of 4000

Retail Customer Service Kroger Co., Louisville, KY

- 1998-2001
- Customer Service
- Bagging, dairy department, pricing department

Technical Proficiency

- Microsoft Word (2000/XP/2003)
- Microsoft Excel (2000/XP/2003)
- Microsoft Access (2000/XP/2003)
- Microsoft Exchange(2007/2010)
- Microsoft PowerPoint (2000/XP/2003)
- Novell Groupwise
- Symantec Ghost
- MySQL
- Microsoft Visio 2003
- Microsoft Visual Studio .NET

- Macromedia Dreamweaver
- Lotus Notes
- Sungard Advance

- Windows 2000/XP/Vista/7
- Windows 2003/2008 Server
- VMware ESXi
- Macintosh OSX
- Linux
- TCP/IP
- NetBui

Education and Certification

BS in Computer Engineering & Computer Science — University of Louisville, Louisville, KY

MS in Computer Engineering — University of Louisville, Louisville, KY

MCP (2007)
MCSA (In progress)