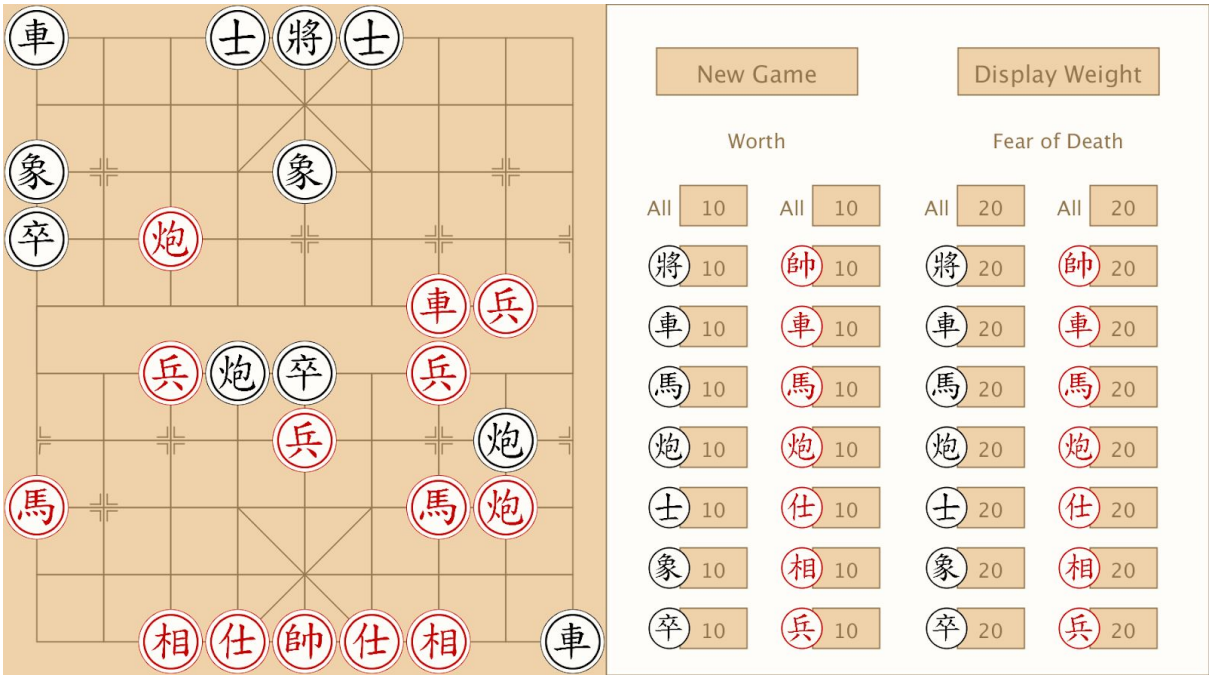
Chan Hiu Yan

Individualist Chess, 2019



Chan Hiu Yan
Individualist Chess, 2019
Mutated Chinese chess game created with Processing

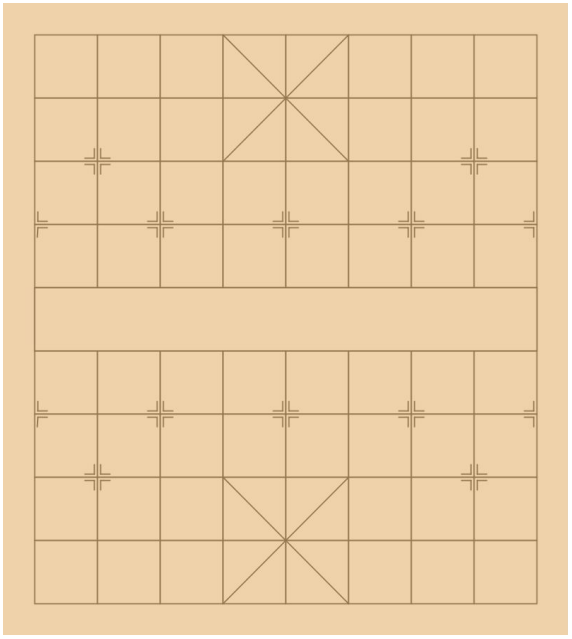
"Individualist Chess" is an explorational artwork based on the question, "What if in a game of chess, every piece made their own decisions based on their own desires and fears, with no single sovereignty?" In the work, each chess piece is its own agent within the system, driven by its desire to eat others and its fear to die. The simulation unveils interesting social phenomenon in an individualistic society where the lack of a central command causes the team to lose direction, strategy, and cooperation. Further interesting observations could be explored by altering the values of each chess piece, or even of a team. Perhaps one team's fear-of-death value is negative, comprising a suicidal society. Perhaps the society upholds equality such that the general is of no greater worth than the soldier, reforming the entire goal of the traditional game. Various situations are possible in each play of the work, as it is a structure for potential situations within the constraints of the system. Even if all values are maintained constant, each play would still vary by the randomness incorporated in each move. A small change early on in the game could change the outcome drastically since the chess world is such a complex system. The scope of this artwork runs from the simulation of agents to the phenomenon of a society to the values of humanity. Ultimately, there is no single premise. The piece invites one to interact with the game, explore the social possibilities, and examine their own worldview.

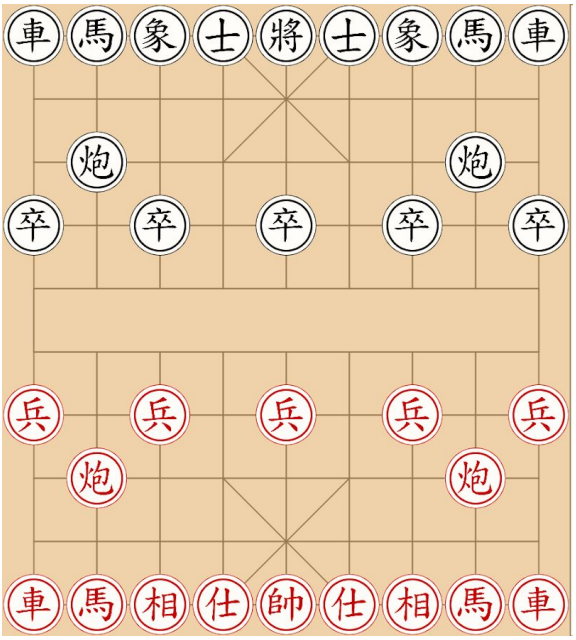
How It Works

The game is based on traditional Chinese chess. It is necessary to understand the basic rules of the Chinese chess to fully appreciate the work. Below is a brief introduction.

The game involves a chess board and 32 chess pieces, 16 on each side. It is played by two players. Each player takes a turn moving one piece at a time. There are 7 different kinds of chess pieces. Each kind has its own way of movements and way of eating. The goal of the game is to eat the other team's general.

Here is the what the board looks like, and how the pieces are placed at the beginning of the game. The two diagonal lines mark the box which the general and advisors must stay within. The long bar down the middle of the board is the river, which marks the half-court. The seven other markings mark the starting positions of the cannons and soldiers.





Chess pieces are moved and placed along the crossing nodes of the grid. Any one kind is allowed to eat any other kind as long as it is within their way of moving and eating. Below are the descriptions of each kind of chess pieces. Although the Chinese characters of the two sides may differ for a particular kind of chess, they are the same in function.

GENERAL

將 帥

move by: vertically and horizontally, 1 unit
eat by: same as move
limit: stay within the box

CHARIOT

車 車

move by: vertically and horizontally, as many units
eat by: same as move

HORSE

馬 馬

move by: diagonally, 2x1 units
eat by: same as move
trip: cannot move to 2x1 or 2x-1 if there is a piece at 1x0

CANNON

炮 炮

move by: vertically and horizontally, as many units
eat by: vertically and horizontally, jump over one piece

ADVISOR

士 仕

move by: diagonally 1x1 unit
eat by: same as move
limit: stay within the box

ELEPHANT

象 相

move by: diagonally, 2x2 units
eat by: same as move
limit: stay within half-court;
trip: cannot move to 2x2 if there is a piece at 1x1

SOLDIER

卒 兵

move by: vertically and horizontally, 1 unit, only forward never backward
eat by: same as move
limit: can only move horizontally once it passes half-court

After a basic understanding of Chinese chess, here is how the game works in *"Individualist Chess."* Unlike chess, this work is played by one person. The person does not control how to move the pieces, but rather how the pieces think and decide how to move. The pieces will play themselves within the rules of Chinese chess, but make individual decisions based on their desires and fears.

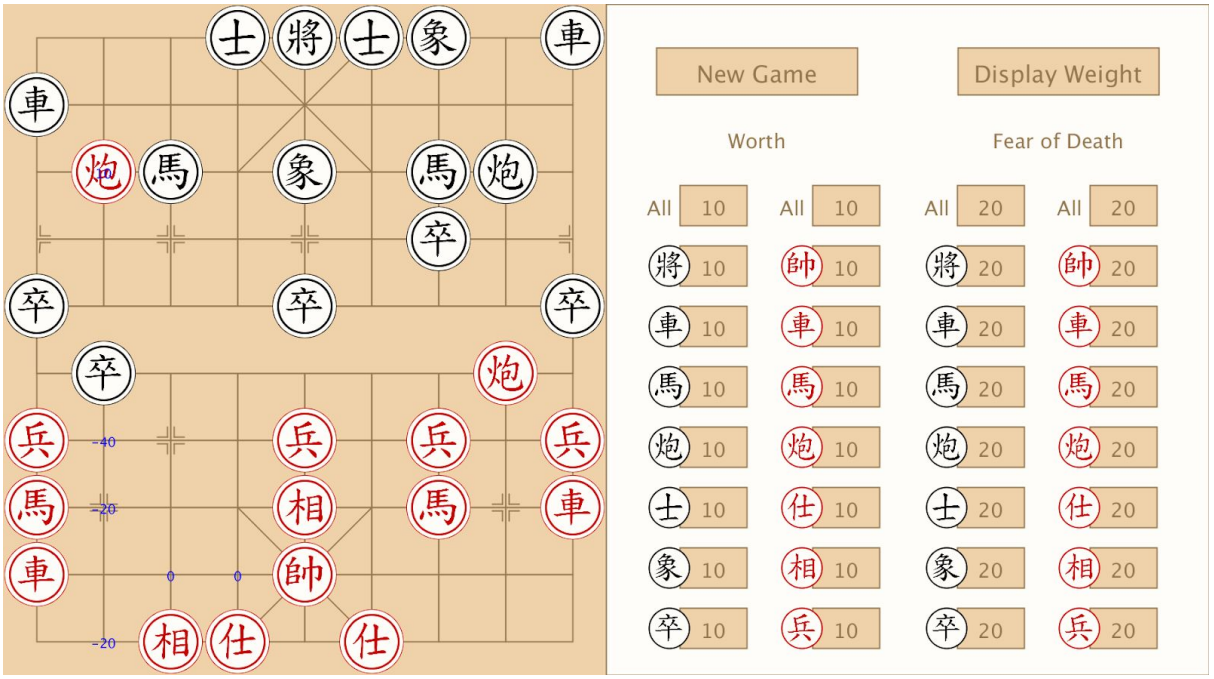
Each chess agent has the following attributes:

- Side (black or red)
- Kind (how it moves and eats)
- Position (where it is on the board)
- State (alive or dead)
- Worth (how much other pieces desire to eat this piece)
- Fear-of-death (how much this piece fears to die)
- Possible moves (positions the piece can go)
- Weight of possible moves (calculated based on worth and fear-of-death)
- Foresee Times (how many times the piece foresees future moves)

Each chess agent has the following behavioural rules:

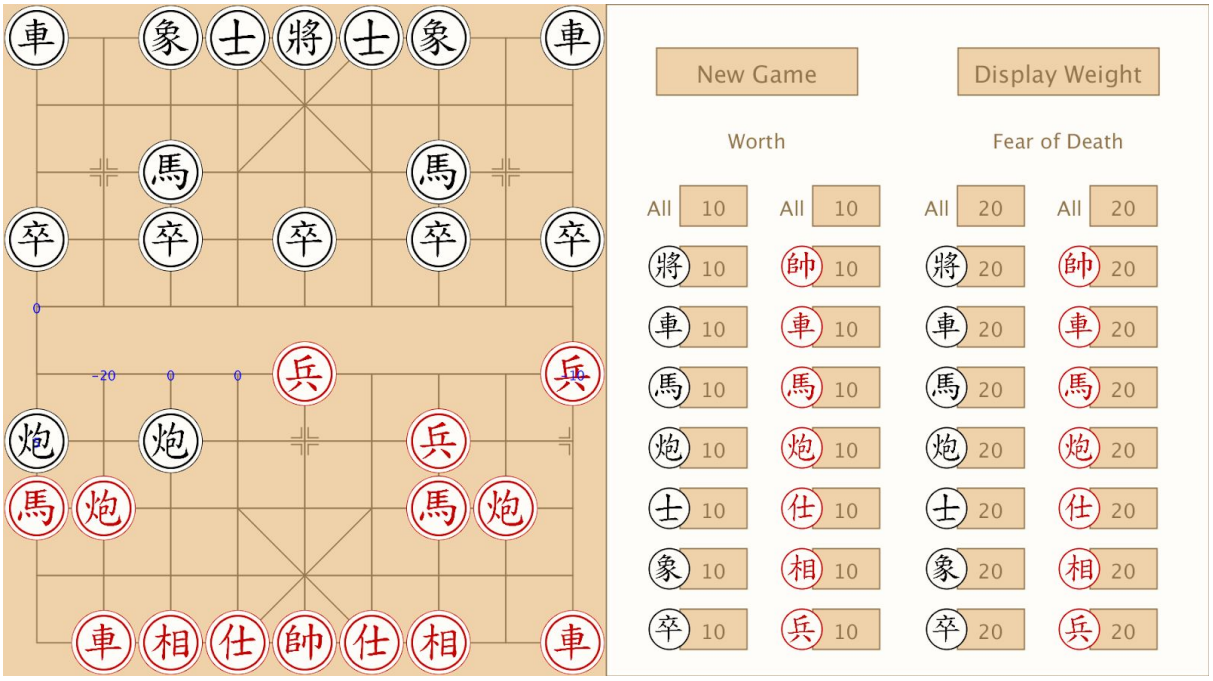
1. When it is this agent's turn, it moves to the possible move that is most weighty.
2. If there are multiple possible moves that are the weightiest, choose randomly among them.

The weight of possible moves are calculated based on worth and fear-of-death values. If a possible move allows the piece to eat another piece, then the worth of the eaten piece is added to the weight. If a possible move puts the piece in danger of being eaten, then the fear-of-death of the piece is subtracted from the weight.



In the above example, the red cannon at (1,2) *(given (0,0) is located at the top left corner)* which had just moved from (1,8) chose the move because it is weighted 10 points. The piece that it had eaten is a black cannon, and since the worth of black cannons is 10, that move adds a weight of 10. On the other hand, the moves (1,6), (1,7), (1,9) have negative weight because those moves puts the piece in danger. If the red cannon moves to those positions, it would've been eaten by the black cannon which originally was at (1,2). Since the fear-of-death of red cannons is 20, those moves will subtract a weight of 20. The move (1,6) has a weight of -40 because not only does it put the red cannon at risk of being eaten by the black cannon, it is endangered by the black soldier at (1,5).

This is the simple weighing algorithm for the game if all chess pieces do not foresee moves. If the chess pieces foresee one time, this is how the weights would be calculated: For every possible move, further calculate the possible moves from that point. Weigh all those moves and get the highest score. Divide that by half and add it to the weight of that move.



In this example, the black cannon at (0,6) just moved from (0,5) to an empty space weighted 5 points. The reason it is weighted 5 points although it cannot eat anyone in that move is that the move allows it to eat a red soldier (6,6) in the next move. Since a red soldier is worth 10 points, and it takes two moves to eat that soldier, the move is scaled by half, yielding 5 points.

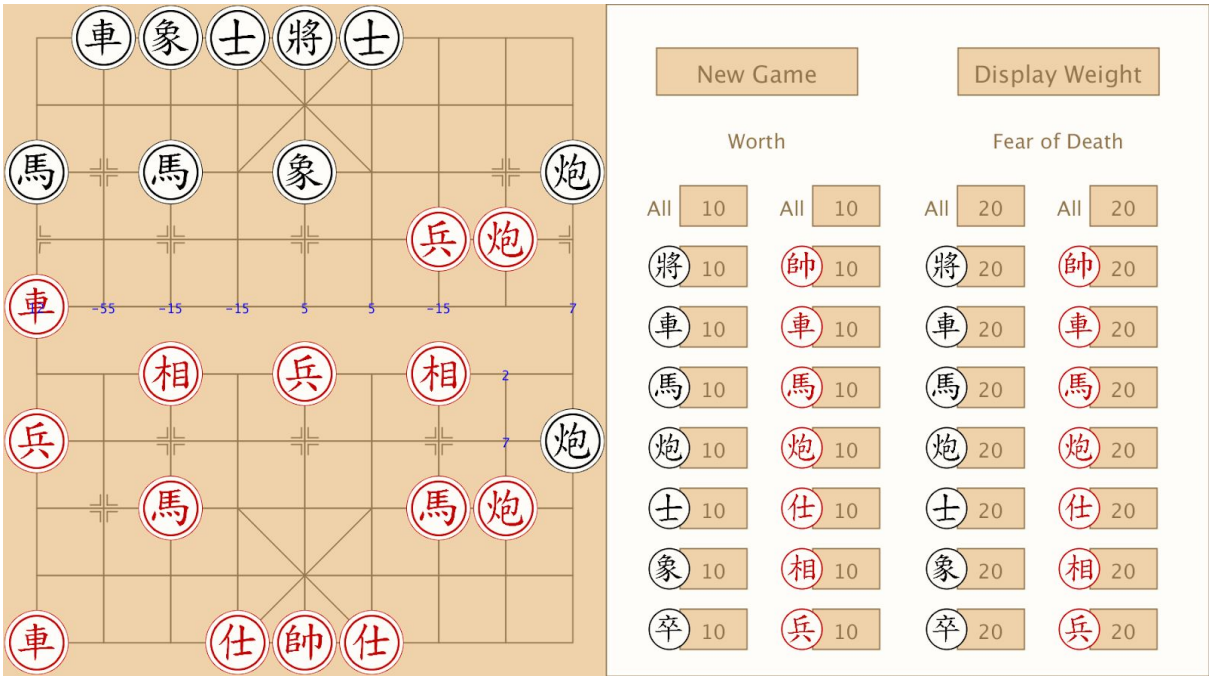
Another thing to point out in this example is that the red cannon did not eat the soldier at (8,5) is because that position also puts it in danger of being eaten by the red chariot (8,9). Even though the red soldier is worth 10 points, the fear-of-death of the red cannon is 20. Thus that move is weighted 10 - 20 = -10, a negative score.

So here is how the weights are calculated when foreseeing moves.
Can eat in one move: add the worth

Can eat in two moves: add the worth/2
Can eat in three moves: add the worth/4
Can eat in four moves: add the worth/8

Since each time the score is further divided by half, the weight being added gets exponentially smaller in each round of foresight.

For the need of this game, foreseeing two times is already plenty. As seen below, it is already difficult to mentally note how each of these weights were calculated.



Now that it is clear how each individual chess agent makes its decision by weighing the best possible move, it is necessary to know how the game knows which piece to move at a given turn. There must be some form of central decision here, or else it would not be chess. Otherwise, the pieces would all update simultaneously just like in cellular automata.

The priorities of the government determine how it decides which piece should move next.

1. Is one piece in danger of being eaten? Move that piece.
2. Is one piece in an advantage for eating another piece? Move that piece.
3. Is one piece least frequently moved in this game? Move that piece.

The above is the logic flow of the government as it chooses the piece to move. First it looks at all the pieces' current positions and see if any is in danger of being eaten. It does so by calculating the weight of the current positions, by subtracting the fear-of-death if it is in danger. It then chooses to move the piece with the lowest weight. However, what if all pieces have the same score of lowest weight because none of them are in danger? This is why the program will first find the minimum score of all the weights. Then it will see if there are three or less pieces that share this minimum

score. If it does, it picks one of them to move. If there are more than three pieces that share this minimum score, it will assume no pieces are in danger and move on to the second criterion.

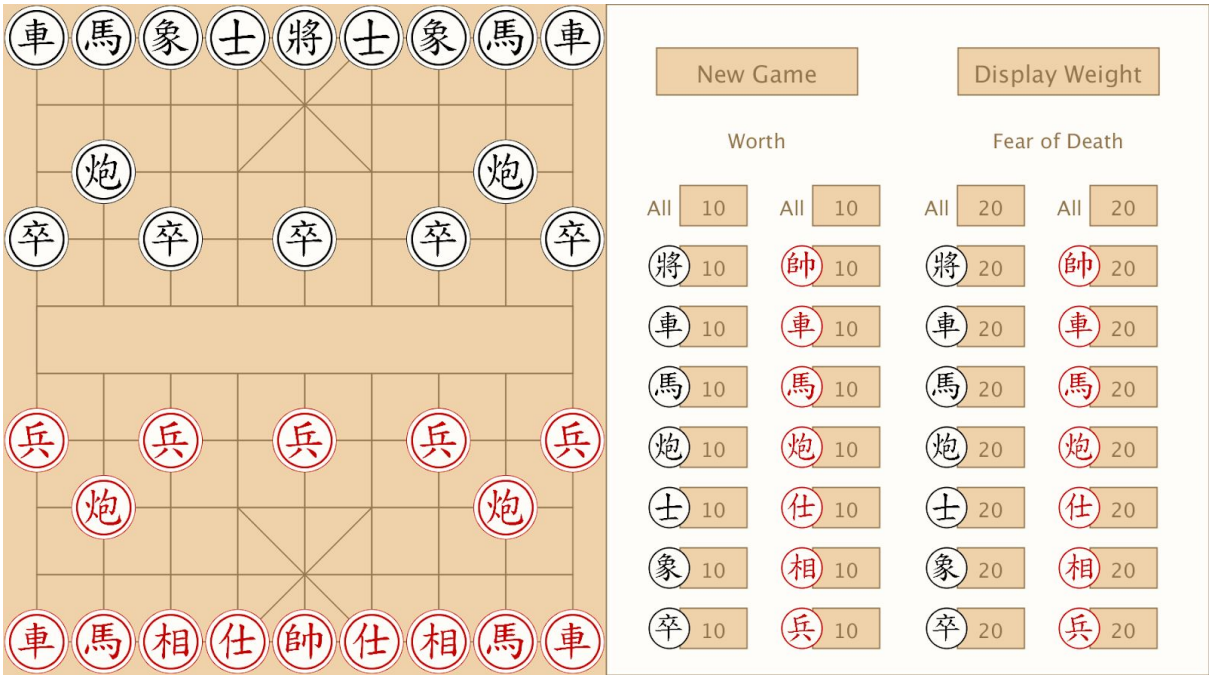
The second criterion asks whether any piece is in advantage for eating another piece. It does so by calculating all the possible moves for each chess piece and all the weights of those moves, finding the maximum. It then moves the piece that has the highest weight of possible move. The same mechanism occurs here as above, however. The piece is only in an advantage if the weight is higher than all the other ones. Therefore, it will only be counted if three or less pieces share that high score. Otherwise, the program will assume there is no advantage for eating and move on to the third criterion.

The third criterion moves the chess piece that is least frequently moved. There is a tally system that keeps track of how many times each chess pieces moved. The program simply has to randomly pick one of the pieces that scored the least on the tally system.

After the explanation of the rules of Chinese chess and the way this game is structured - how each chess piece decides which move to take and how the game decides which piece to move - one should have a clear concept of the way this game functions. Ready to play?

How To Play

Here is a basic user manual for playing the game. There is no “goal” or “winning” in this game, but rather the player is invited to play around the values and observe interesting situations and decisions the game makes.



The game can be downloaded as a zip file. Unzip to find a Processing sketch folder. Make sure Processing is downloaded and the sound library installed before hitting play. The above screen should be displayed at the start.

Click the screen once to make sure the sketch will be receiving keyboard commands.
Hit SPACEBAR to view the next move.
Click NEW GAME to start over.
Click DISPLAY WEIGHT to toggle the display of the weights of the possible moves.

To edit the worth and the fear-of-death values of the chess pieces:
1. First click on the input box
2. Type the desired value. Make sure it is an integer
3. Hit ENTER

Make sure to hit ENTER after typing the desired value, otherwise the program will keep recording the keyboard input. To edit multiple values at once, click on each of the input boxes then type the number, and all of the clicked boxes will update when ENTER is hit. To edit all the values of the same column, edit the box that is marked ALL above that column.

It is possible to edit the values during the game, updating half-way. When NEW GAME is clicked, all chess pieces are revived at their original positions with their default worth and fear-of-death values.

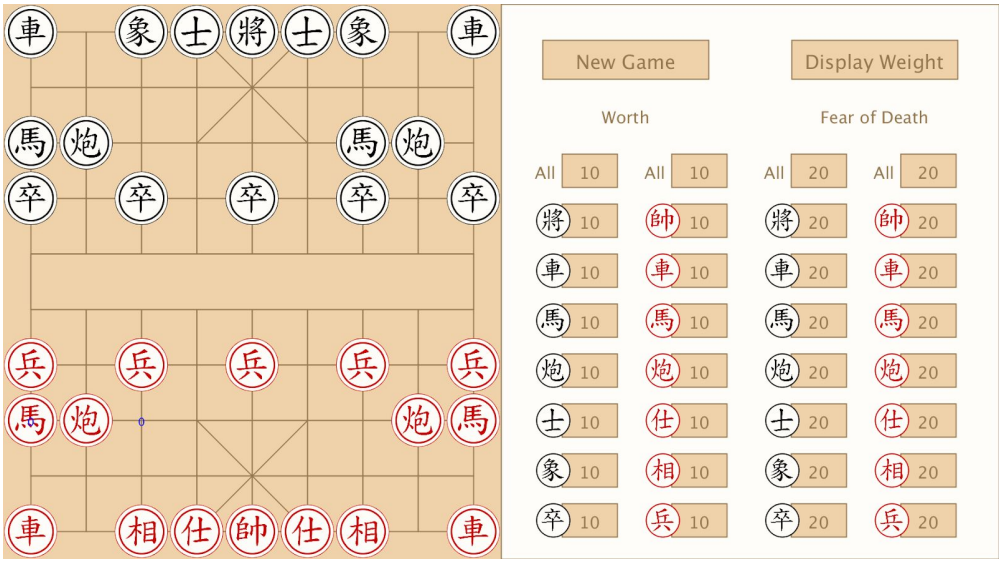
What It Means

The idea of playing chess in an individualistic way first occurred to me when I was pondering on how many people uphold the self - rights, freedom, will, competition, democracy... all grand and positive words which somehow I am skeptical from fully endorsing. I would question the legitimacy and extent of human rights. Certainly if everyone has the right to do whatever one wants, it is paradoxical if one person believes he has the right to listen to loud music at home, and his neighbor believes he has the right to enjoy silence. Perhaps this is a silly example, but already the reader may interpret one neighbor is more right than the other, and thus making a moral judgement, all the more begging to question where the line is drawn for human rights.

How about competition and free trade? Does fulfilling good for self ultimately leads to the good of all? The seed of the concept originated with a kind of game theory camp game, with a room-sized chess board, and people as chess pieces. People would earn points for themselves personally by eating others, and earn points for the whole team by winning the chess game. Would they cooperate to win together or be selfish and earn for themselves? What if everyone only cares about themselves? What would happen? How would it differ from the conventional way of playing chess with a central command? This is what the artwork seeks to simulate.

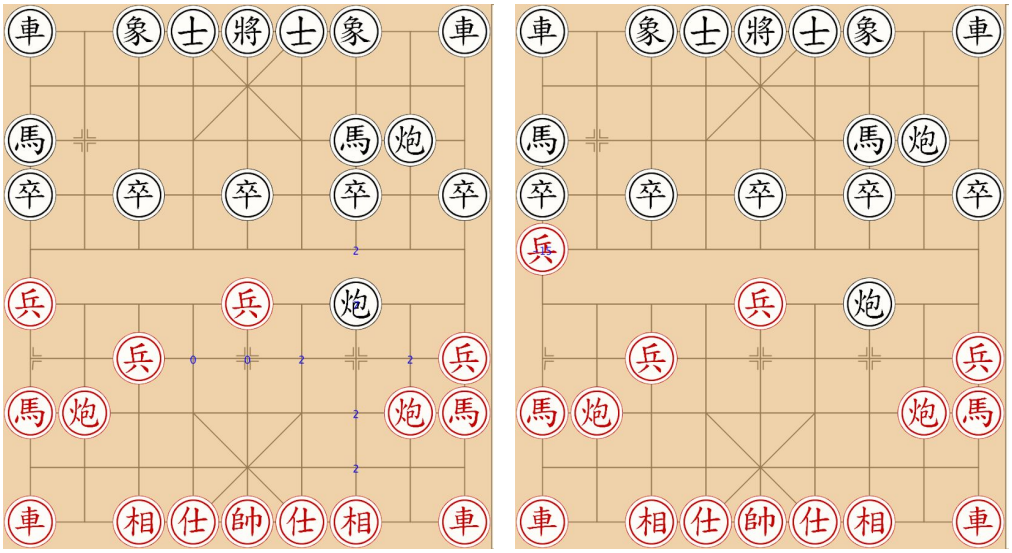
In exploring the potential situations in this work, some interesting phenomenon could be observed. Below are just some examples, however I am sure there could be many more.

- **FEAR**: the fear of death blinds one from seeing the security provided by its teammate and causes it to take unnecessary measures to feel safe.



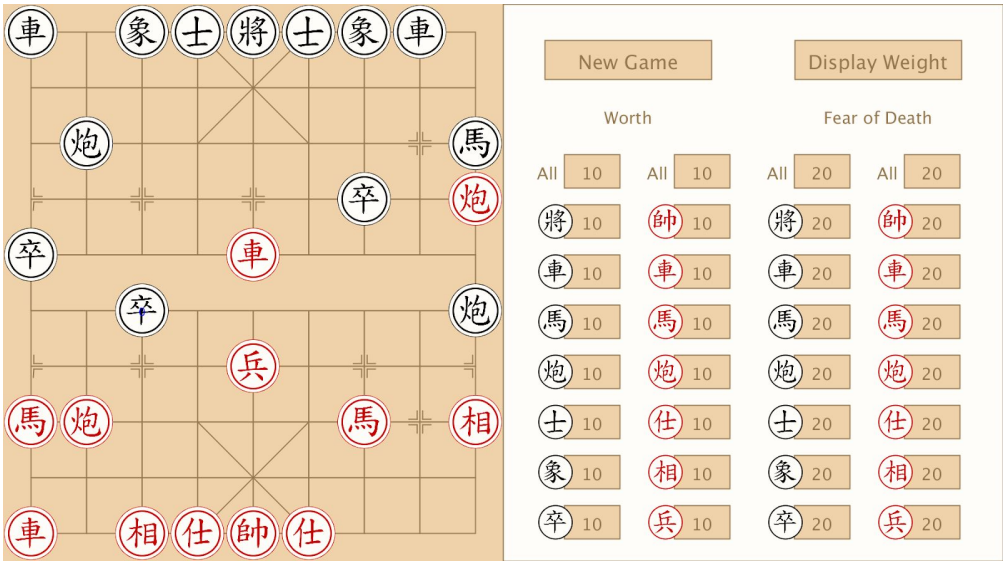
As shown above, the start of every game begins with all the horses moving. The reason is that they feel threatened by the cannons. But in reality they are safe because they are protected by the chariots next to them.

- **FEAR**: Prioritizing fear causes one to make stupid decision by moving the most threatened piece regardless, even though that piece will have no other choice but death.



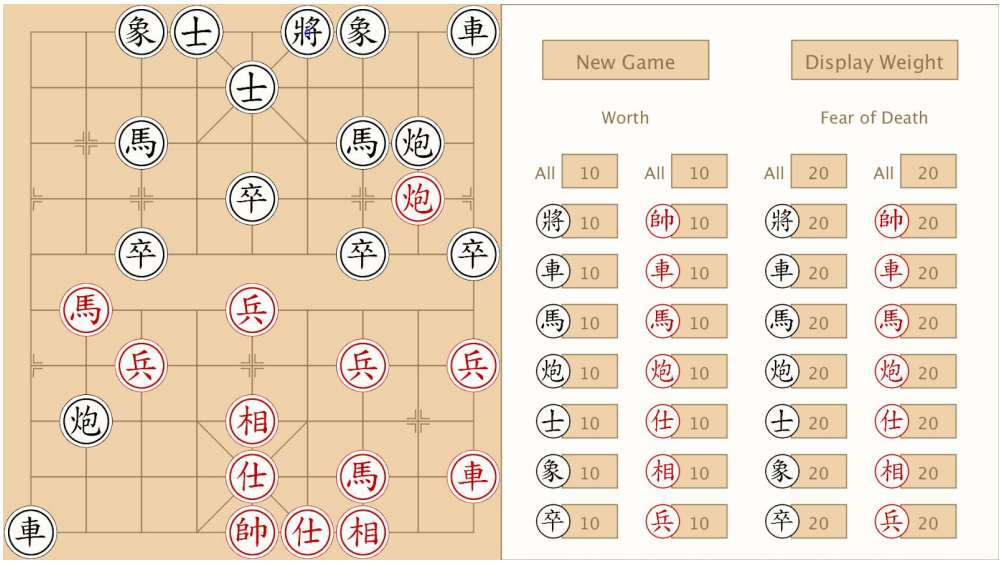
In the above example, the red soldier (0, 5) is threatened by the black cannon (6,5). However, if it moves forward it will be eaten by the black soldier (0,4). Since the government in this game fears death more than anything else, it will move the red soldier anyway. What it doesn't realize is that it is a waste of move and thus disadvantageous for the whole team.

- **EQUALITY**: Each one having equal worth hinders the winning of the game.



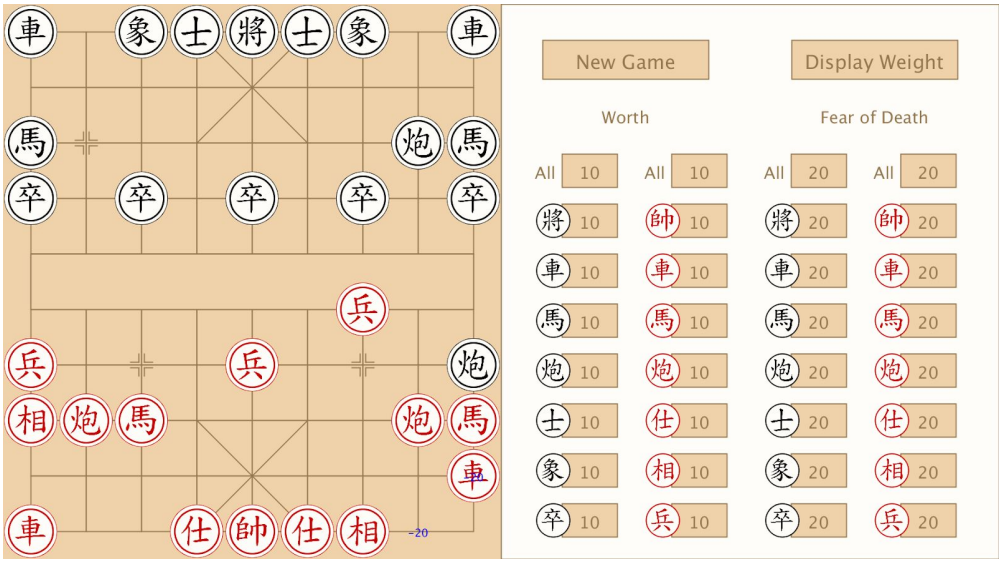
In this example, the black general (4,0) is threatened by the red chariot (4,4). However, instead of doing something to save the general, the black team chose to move the soldier (2,5) because all pieces have equal worth and so it didn't care to save the general before the soldier.

- **FAIRNESS**: When each piece moves equally frequently because all the pieces on a team must have a fair number of turns, it may not advance the team towards its ultimate goal.



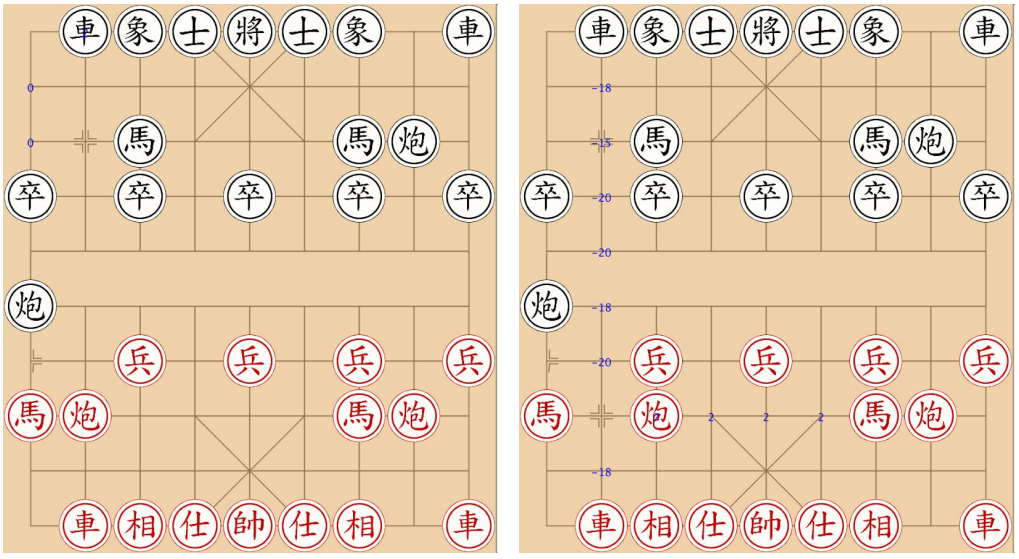
Here, the black general (5,0) moved out of its original position (4,0) only because the team thought it was moved least frequently compared to the others. However, it is not always a good idea to sit out the general because it makes it more exposed to the threats of opponents.

- **COOPERATION**: When one only looks after oneself, there is a lack of cooperation.



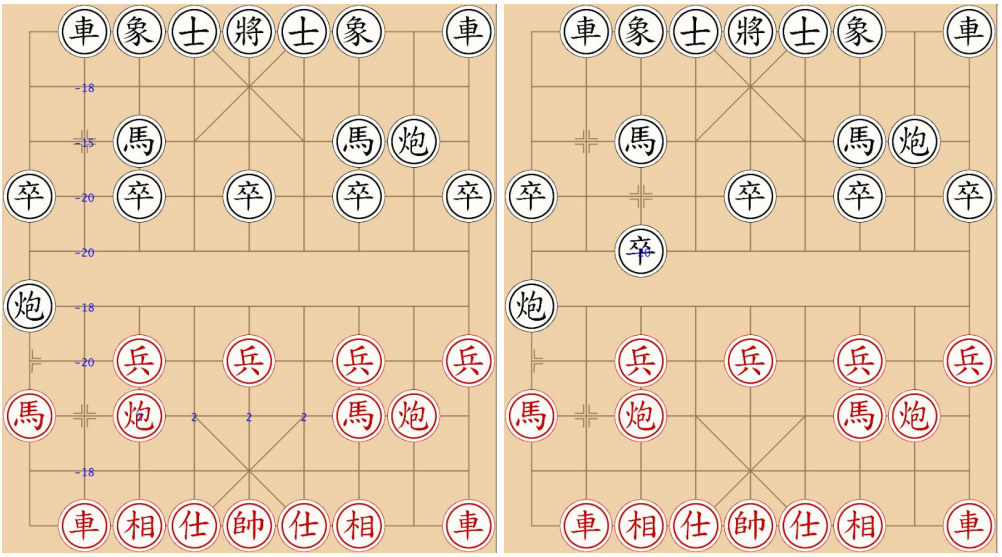
The above is a good example of a waste of a win-win cooperation. The red chariot (8,8) had just advanced from (8,9) because it was threatened by the black cannon (8,6). However, it is silly to move the chariot because moving forward it would still be eaten by the cannon, and moving left it would be eaten by the other cannon. Here is where cooperation could've came in. If instead of moving, the red chariot stays in the corner and the red horse (8,7) moves to (7,9), then the black cannon can no longer leap over and eat the red chariot, but rather the red chariots threatens back the cannon. The red horse, although it moved to a location threatened by the other black cannon, it is safe because it is protected by the chariot.

- **SELFISHNESS**: To protect oneself, one will not care that its move will cause the death of its teammate.



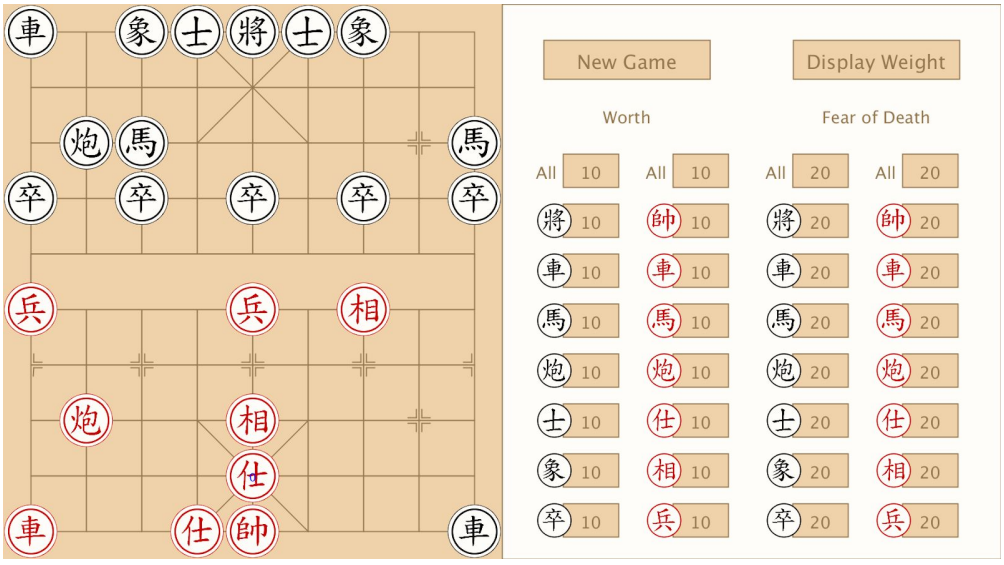
In the above example, the red cannon (1,7) is threatened by the black chariot (1,0). However, since the chariot does not realize it is protected by the red chariot (1,9) but rather fears death, it moves out of the way to protect himself. What he in fact does is sacrifice the red chariot to save himself.

- **INDIVIDUALISM**: The lack of sovereign decision makes one blinded to a better solution.



As continued from the last example, the red chariot (1,9) is now threatened by the black chariot (1,0). However since the fear of death trumps eating, instead of eating the red chariot, the black team decides to move the soldier (2,3) because it is threatened by the red cannon (2,7). This wastes the precious opportunity to eat one of the opponent's most valuable piece and instead, black team ends up losing its chariot to red team. Here, it is observed that the fear of death indirectly causes death.

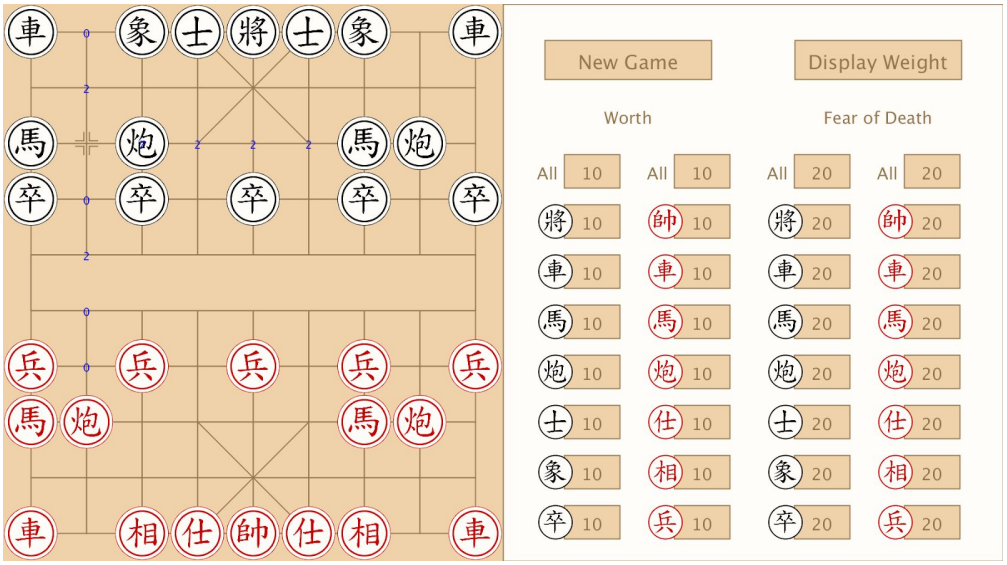
- **DIRECTION**: One forgets the ultimate direction when one is only worried about oneself.



Above is a perfect demonstration of how the red advisor (5,9) moved away for its own fear of death, but thus sacrificing the general (4,9) to the black chariot (8,9). The ultimate goal of the game was

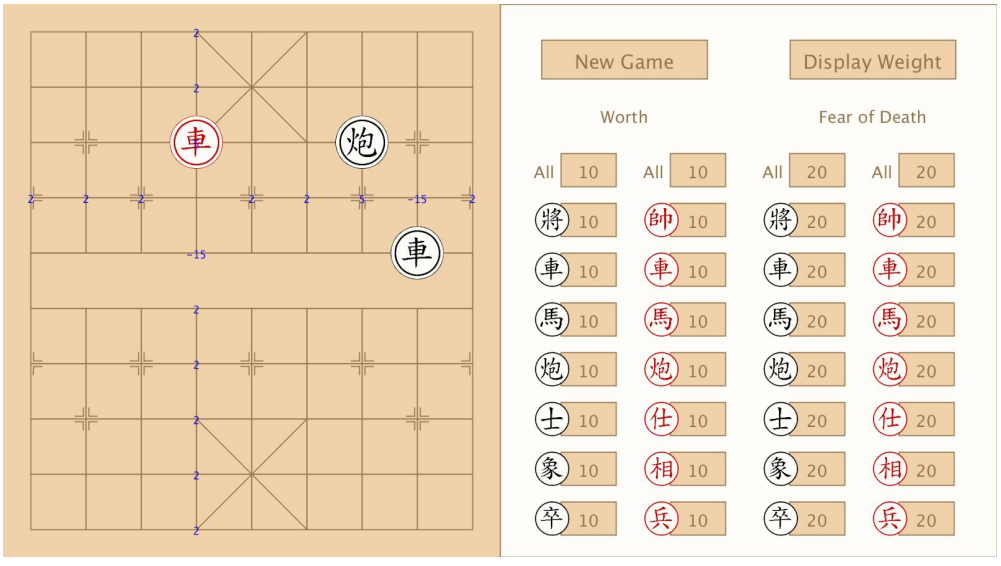
supposed to be to protect one's own general and to eat the other team's general. Protecting oneself was only a means to achieve that end. But the advisor has sacrificed the end for the means.

- **DIRECTION**: One goes for quick and easy, but not always the most valuable.



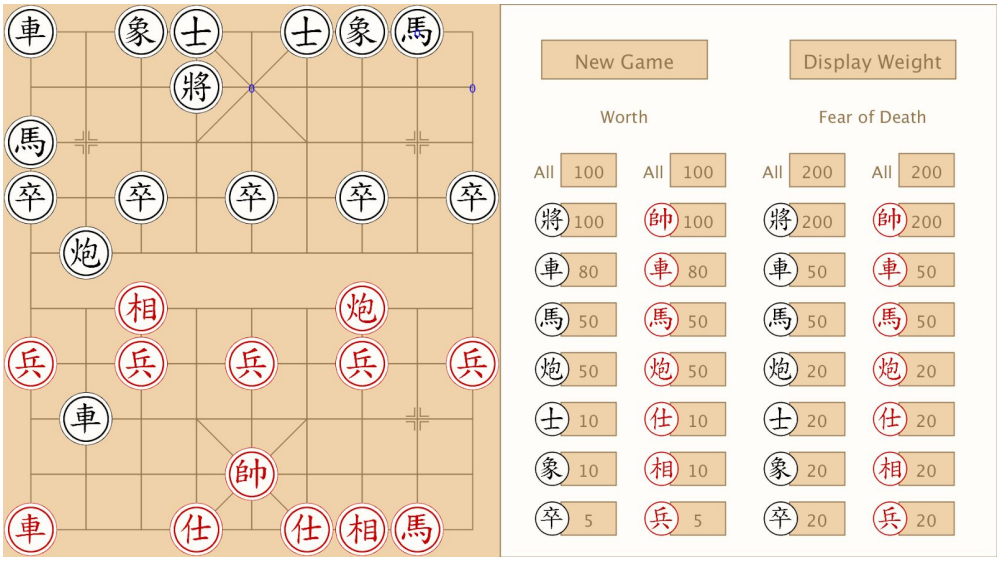
Here, the black cannon (2,2) went for the red soldier (2,6) but eating soldiers is relatively unimportant to winning the game. Had they have in mind to cooperate and eat the general together, perhaps the cannon moving to (4,2) would better drive towards that goal.

- **SURVIVAL OF THE FITTEST**: Equality is not always equity when it comes down to survival.



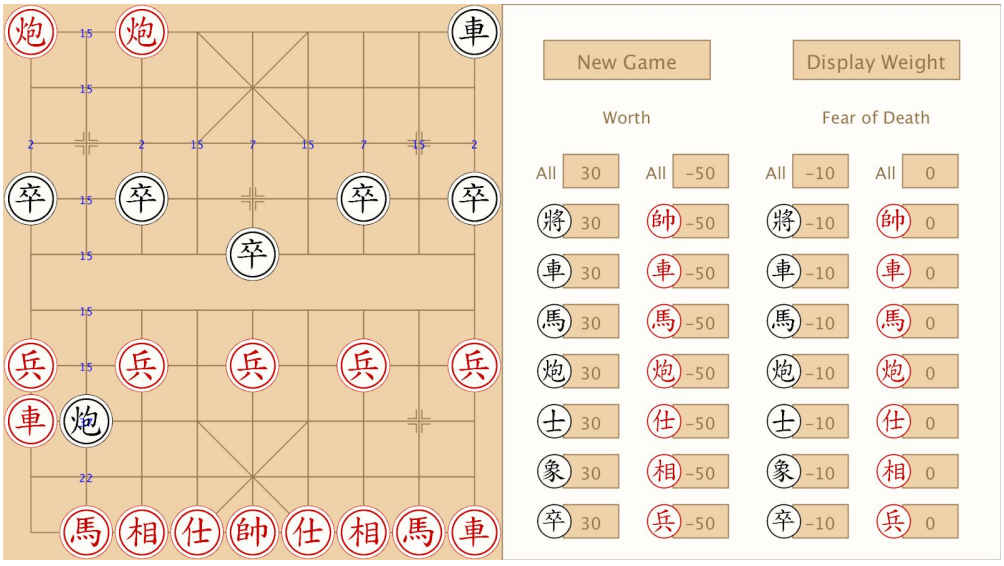
Since the chariot has the most advantageous way of moving, it is usually one of the remaining pieces in the game although all the worth values were kept equal.

RANKING: By ranking the different kinds of pieces with different worth and fear-of-death, a more normal play of chess is observed where the general is most esteemed.



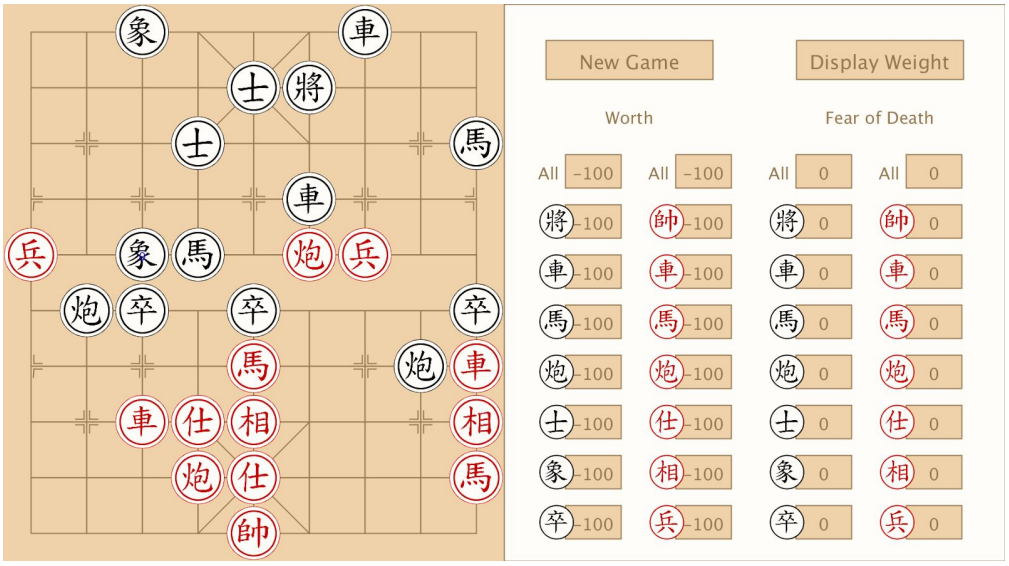
Try inputting the values above for a play that is closer to the normal play in traditional Chinese chess, theoretically. Adjust the values for more variation and accuracy.

- **SUICIDE**: The entire team is suicidal when its fear of death is negative.



Try inputting the values above for an interesting play where the red team wipes out the black team. Here, the black cannon (1,7) moved right next to the red chariot to commit suicide because it desires to die. The black team will never eat the red team because eating red pieces will cost them the worth of -50 points. The red team never has to fear the black team because of this, and with the fear-of-death of 0, it never worries.

- **PEACE**: A relatively peaceful play occurs when no one is greedy to eat another nor fear death.



Try inputting the values above for a relatively peaceful play. Because the worth of eating another is -100, one has no desire to harm others. Because the fear-of-death is 0, one has no fear of being harmed.

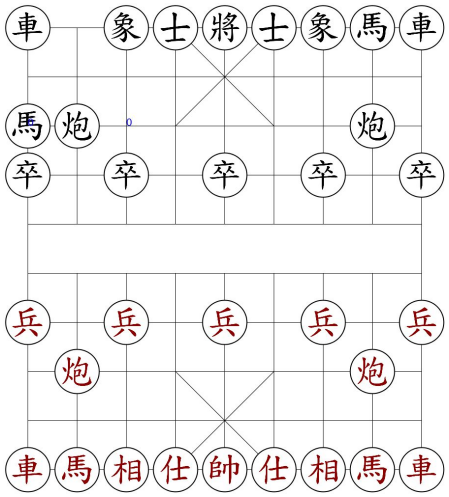
How I Made It

I will briefly discuss the process of coding this game here, although I will not dive into too much detail. For more technical understanding, please read the code and the comments in the appendix.

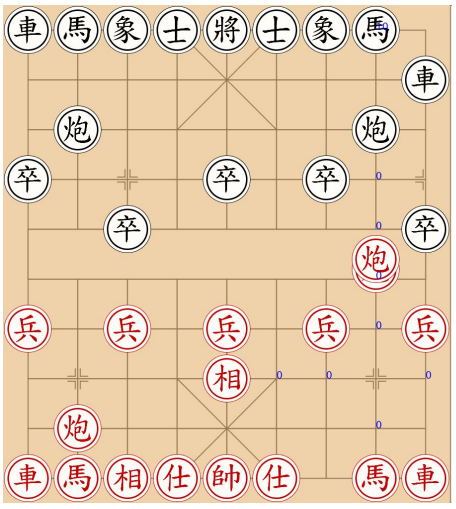
Before I began on coding, I read parts of Daniel Shiffman's *The Nature of Code* for inspiration and guidelines. His lessons on PVectors, classes, and particle systems helped me consolidate how I wanted to go about this project.

I first began the programming from creating a Board class to manage displaying the chess board. I further coded the Chess class to display the chess pieces and manage all the properties and methods associated with it. This includes how the chess piece looks at all its possible moves and weighs each of those moves based on its own worth and fear-of-death. The Chess class is the superclass to the seven other classes (Gen, Cha, Hor, Can, Adv, Ele, Sol) which inherit from the Chess class. I then wrapped a Game class as the ultimate manager in the system hierarchy above all the other classes I intend to create.

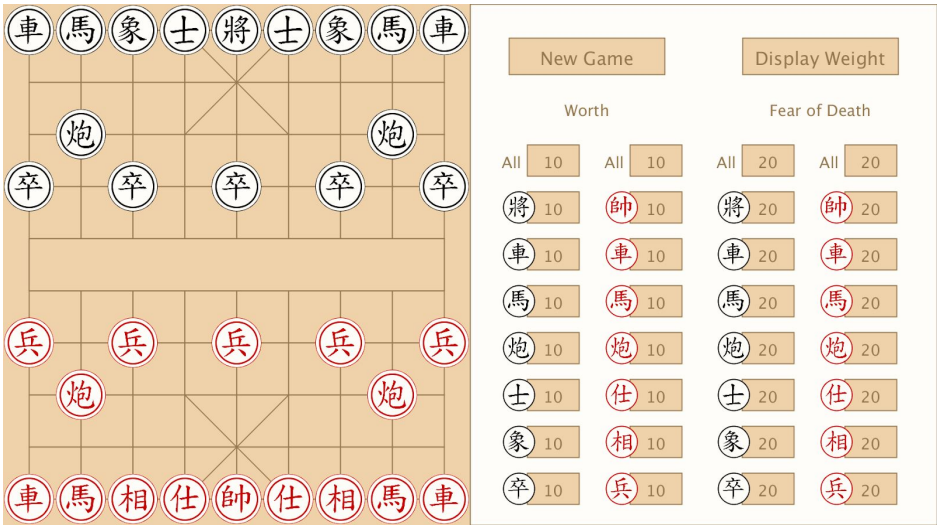
I then coded each of those Chess subclasses for their unique way of movements. Once I programmed Chess class and how the pieces calculate weights, I moved on to the Game class to program how it decides which chess to move next. The Game class is responsible for managing the array of Chess objects and other properties such as whose turn it is, tallying how many times who moved, and communicating with each chess piece the information of the other chess pieces.



With the basic game structure intact, I polished the aesthetics such as the colours of the display, sound effects, as well as the smooth animation of the movements.



After that, I coded the Bar class, along with its Button class and Input class for the user interface.



For coding the input box, I researched some online code for reference, using parts and altering parts.

Finally, the game is nearly finished. I had to debug multiple times for errors such as array out of bounds. I upgraded the intelligence of the chess pieces as I moved on. At first the pieces do not realize they are in danger because they only look at the possible moves. Then I added one more piece of intelligence, which is the ability to foresee future moves. I coded recursively so that by editing one number I can change the times of foresight.

Throughout the process, I tested the game, and asked my family to try it out for feedback and adjustments. I am very proud of how it turned out, and am quite intrigued by all the possible situations the game puts itself it. It is humorous, fascinating, and insightful and I hope it finds well with the engagers too.

Appendix

Below is the code of the game in Processing. Each tab of the sketch is separated by a horizontal line.

TAB: individualist_chess

```
/*
How to play:
Hit SPACEBAR to view the next move
Click NEW GAME to start over
Click DISPLAY WEIGHT to toggle the display of the weights of the possible moves

To edit the worth and the fear-of-death values of the chess pieces:
1. First click on the input box
2. Type the desired value. Make sure it is an integer
3. Hit ENTER

*/

import java.util.Iterator;
import processing.sound.*;

Game game;

void setup(){
  size(1800,1000);
  game = new Game(100, this); //100 is the scale of the whole game display
  game.newGame();
}

void draw(){
  game.display();
}

void keyPressed(){
  if(key == ' '){
    game.move();
  } else if(key == 's'){
    saveFrame("chess_#######.png");
  }
  game.bar.type();
}

void mousePressed(){
  game.bar.click();
}
```

TAB: Game

```
/*
Game is the class in the top hierarchy of the system.
 It creates and contains all the Chess pieces played in the game, the chess Board, and the user interface Bar.

 It manages communication among the pieces by passing information of other chess pieces for each piece.
 It keeps track of whose turn it is.
 For each turn, it determines which piece should move for a given side.
 It does so in a more or less democratic manner, with a preference for the survival of the fittest.
 Its priority for picking who should move in a turn is as follows:
 Is one piece in danger of being eaten? Move that piece.
 Is one piece in an advantage for eating another piece? Move that piece.
 Is one piece least frequently moved in this game? Move that piece.

 */

class Game {
  Board board; //chess board
  Bar bar; //user interface bar
  int s; //scale
  boolean blackTurn; //whose turn it is

  //tally array of how many times each piece has moved
  int[] blackTally;
  int[] redTally;

  Chess[] all; //array storing all the chess objects in the game, a total of 32 pieces
  int chosenMove; //index of the chess that is chosen to move next
  ArrayList<Integer> eating; //list of all the chess index that are in process of eating another
  ArrayList<Integer> dying;  //list of all the chess index that are in process of being eaten
  ArrayList<Integer> moving; //list of all the chess index that are in process of moving

  SoundFile eatSound;
  SoundFile moveSound;
  boolean end; //if the game has ended
  boolean blackWins; //who has won
  boolean displayWeight; //toggle display weight

  Game(int s_, PApplet pa) {
   s = s_;
   eatSound = new SoundFile(pa, "eat.wav");
   moveSound = new SoundFile(pa, "move.wav");
  }

  //starts a new game, creates board, bar, and new chess pieces at starting possition
  void newGame() {
```

```
  board = new Board(s);
  newChess();
  bar = new Bar(this, s);

  blackTurn = true;
  blackTally = new int[16];
  redTally = new int[16];
  eating = new ArrayList<Integer>();
  dying = new ArrayList<Integer>();
  moving = new ArrayList<Integer>();
  end = false;
  displayWeight = false;
}

//creates all the chess pieces at their starting possition and stores them into the all array
void newChess() {
  all = new Chess[32];

  all[0] = new Gen(0, s, 0, new PVector(4, 0));

  all[1] = new Cha(1, s, 0, new PVector(0, 0));
  all[2] = new Cha(2, s, 0, new PVector(8, 0));

  all[3] = new Hor(3, s, 0, new PVector(1, 0));
  all[4] = new Hor(4, s, 0, new PVector(7, 0));

  all[5] = new Can(5, s, 0, new PVector(1, 2));
  all[6] = new Can(6, s, 0, new PVector(7, 2));

  all[7] = new Adv(7, s, 0, new PVector(3, 0));
  all[8] = new Adv(8, s, 0, new PVector(5, 0));

  all[9] = new Ele(9, s, 0, new PVector(2, 0));
  all[10] = new Ele(10, s, 0, new PVector(6, 0));

  for (int i = 0; i<5; i++) {
    all[i+11] = new Sol(i+11, s, 0, new PVector(i*2, 3));
  }

  all[16] = new Gen(16, s, 1, new PVector(4, 9));

  all[17] = new Cha(17, s, 1, new PVector(0, 9));
  all[18] = new Cha(18, s, 1, new PVector(8, 9));

  all[19] = new Hor(19, s, 1, new PVector(1, 9));
  all[20] = new Hor(20, s, 1, new PVector(7, 9));

  all[21] = new Can(21, s, 1, new PVector(1, 7));
```

```
  all[22] = new Can(22, s, 1, new PVector(7, 7));

  all[23] = new Adv(23, s, 1, new PVector(3, 9));
  all[24] = new Adv(24, s, 1, new PVector(5, 9));

  all[25] = new Ele(25, s, 1, new PVector(2, 9));
  all[26] = new Ele(26, s, 1, new PVector(6, 9));

  for (int i = 0; i<5; i++) {
   all[i+27] = new Sol(i+27, s, 1, new PVector(i*2, 6));
  }
 }

 //display everything, the board, bar, and chess pieces at their positions
 void display() {
  board.display();
  bar.display();
  for (int i = 0; i<all.length; i++) {
   all[i].display(); //display all chess pieces
  }

  //display the moving piece again to make sure it is on top of the other pieces
  for (int i = 0; i<all.length; i++) {
   if (all[i].moving) all[i].display();
  }

  //display weight of possible moves of the piece that is moving, if the game is in display weight state
  if (displayWeight) {
   all[chosenMove].visualizeWeight();
  }

  //when the moving piece has arrived at its target, play the move sound and remove from the moving array
  Iterator<Integer> iterMov = moving.iterator();
  while (iterMov.hasNext()) {
   if (!all[iterMov.next()].moving) {
    moveSound.play();
    iterMov.remove();
   }
  }
  Iterator<Integer> iterEat = eating.iterator();
  Iterator<Integer> iterDy = dying.iterator();

  //when the eating piece has arrived at its target, play the eat sound and remove from the eating array and
let the eaten piece die
  while (iterEat.hasNext()) {
   if (!all[iterEat.next()].moving) {
    all[iterDy.next()].alive = false;
    eatSound.play();
```

```
      iterEat.remove();
      iterDy.remove();
    }
  }
  if (end) {
    if (blackWins) {
      message("Black Wins!");
    } else {
      message("Red Wins!");
    }
  }
}

/*choose which chess piece goes next
 Its priority for picking who should move in a turn is as follows:
 1. Is one piece in danger of being eaten? Move that piece.
 2. Is one piece in an advantage for eating another piece? Move that piece.
 3. Is one piece least frequently moved in this game? Move that piece.
 */
int chooseMove() {
  int[] weights = new int[16]; //the best weight for each chess
  int[] current = new int[16]; //how dangerous it is in its current position for each chess
  int start; //index of the first piece of each red or black team
  int chosen = -1; //index of which chess is chosen
  int[] tally; //pointer to which tally array to add to, depending on whose turn it is

  //different start and tally for red and black team
  if (blackTurn) {
    start = 0;
    tally = blackTally;
  } else {
    start = 16;
    tally = redTally;
  }

  //1. Is one piece in danger of being eaten? Move that piece.
  //get the best weight for each piece at its best move
  for (int i = 0; i<weights.length; i++) {
    weights[i] = all[i+start].getBestWeight(all, 0); //do not recur or foresee when calculating for danger moves
  }

  //get the danger weight for each piece at its current position
  for (int i = 0; i<current.length; i++) {
    current[i] = all[i+start].weighCurrent(all);
  }

  //get the weight of the chess that is in the most danger
  //the lower the weight, the more dangerous
```

```
int curMin = 10000;
for (int i = 0; i<current.length; i++) {
 if (all[i+start].alive && all[i+start].hasMove()) {
  if (current[i] < curMin) curMin = current[i];
 }
}

//put every move that has the minimum weight into the array
ArrayList<Integer> fearMove = new ArrayList<Integer>();
for (int i = 0; i<current.length; i++) {
 if (all[i+start].alive) {
  if (current[i] == curMin && all[i+start].hasMove()) {
   fearMove.add(i);
  }
 }
}

//if there are no moves with minimum weight, it means there are no more moves left, so the game is over
if (fearMove.size() == 0) {
 endGame();
 return -1;
}

//only if there are three or less moves with minimum weight (outliers) then choose randomly from the array
if (fearMove.size()<= 3) {
 chosen = fearMove.get((int)random(fearMove.size()));
 println("I'm in danger. It's my turn.");
}

//if no moves are chosen from reason 1. then look at reason 2.
if (chosen == -1) {
 //2. Is one piece in an advantage for eating another piece? Move that piece.

 //get all the highest weights from all the moves again, this time with foresight involved
 for (int i = 0; i<weights.length; i++) {
  weights[i] = all[i+start].getBestWeight(all, all[i+start].foreseeTimes);
 }

 //get the maximum weight from all the moves
 int max = -10000;
 for (int i = 0; i<weights.length; i++) {
  if (all[i+start].alive && all[i+start].hasMove()) {
   if (weights[i] > max) max = weights[i];
  }
 }

 //put all the moves that have that maximum weight in an array
 ArrayList<Integer> maxMove = new ArrayList<Integer>();
```

```
    for (int i = 0; i<weights.length; i++) {
      if (all[i+start].alive && all[i+start].hasMove()) {
        if (weights[i] == max) {
          maxMove.add(i);
        }
      }
    }

    //if there is no move in the array, then the game is over
    if (maxMove.size() == 0) {
      endGame();
      return -1;
    } else if (maxMove.size()<= 3) { //only if there are less than three moves with highest weight, pick randomly
one of the moves
      chosen = maxMove.get((int)random(maxMove.size()));
      println("I can eat. It's my turn.");
    } else {

      //3. Is one piece least frequently moved in this game? Move that piece.
      //get the minimum value of from the tally array
      int min = 10000;
      for (int i = 0; i<tally.length; i++) {
        if (all[i +start].alive && all[i+start].hasMove()) {
          if (tally[i] < min) {
            min = tally[i];
          }
        }
      }

      //add all the moves that moved the least into an array, and pick randomly one of the moves
      ArrayList<Integer> minTally = new ArrayList<Integer>();
      for (int i = 0; i<tally.length; i++) {
        if (all[i+start].alive && all[i+start].hasMove()) {
          if (tally[i] == min) {
            minTally.add(i);
          }
        }
      }
      chosen = minTally.get((int)random(minTally.size()));
      println("I'm least played. It's my turn.");
    }
  }
  tally[chosen]++; //tally up the move that was chosen

  chosen = chosen +start;
  blackTurn = !blackTurn; //toggle the turn
  chosenMove = chosen;
  println(chosen);
```

```
  return chosen;
 }

 //move() is called every time user hits the spacebar
 //get the next move in the game and move that piece
 void move() {
  chooseMove();
  if (!end) { //only run if the game has not yet ended
   moving.add(chosenMove); //keep track of the moving pieces
   int eaten = all[chosenMove].move(all); //see if there are any pieces eaten
   if (eaten != -1) {
    eating.add(chosenMove); //store the moving piece in the eating array
    dying.add(eaten); //store the dying piece in the dying array, so it only dies once the eating piece has
finished moving
   }
  }
 }

 //record who won the game, and make end true, so that user cannot move anymore
 void endGame() {
  if (blackTurn) {
   blackWins = false;
  } else {
   blackWins = true;
  }
  end = true;
 }

 //for displaying end message of who won the game
 void message(String msg) {
  rectMode(CENTER);
  fill(240, 210, 170);
  stroke(150, 120, 80);
  strokeWeight(3);
  rect(width/2, height/2, s*10, s*7);
  fill(150, 120, 80);
  textAlign(CENTER, CENTER);
  textSize(s*1.5);
  text(msg, width/2, height/2);
 }
}
```

TAB: Board

```
/*
Chinese chess board
Displays the board of the Chinese chess
```

```
*/

class Board {
  int s; //scale, pixel distance of one unit
  PVector[][] pos = new PVector[9][10]; //2D array that translates the position (units) into the actual location (pixels).

  Board(int s_) {
    s = s_;
    for (int x = 0; x<9; x++) {
      for (int y = 0; y<10; y++) {
        pos[x][y] = new PVector(s/2+x*s, s/2+y*s); //calculate actual location in pixels
      }
    }
  }

  void display() {
    rectMode(CORNER);
    fill(240,210,170);
    noStroke();
    rect(0,0,9*s, 10*s);
    stroke(150,120,80);
    strokeWeight(2);

    //draw grid
    for (int i = 0; i<9; i++) {
      for (int j = 0; j<8; j++) {
        rect(s/2+j*s, s/2+i*s, s, s);
      }
    }
    rect(pos[0][4].x, pos[0][4].y, 8*s, s); //the half-court bar, aka the "river" in Chinese Chess
    //cross marks of the boxes
    line(pos[3][0].x, pos[3][0].y, pos[5][2].x, pos[5][2].y);
    line(pos[3][2].x, pos[3][2].y, pos[5][0].x, pos[5][0].y);
    line(pos[3][7].x, pos[3][7].y, pos[5][9].x, pos[5][9].y);
    line(pos[3][9].x, pos[3][9].y, pos[5][7].x, pos[5][7].y);

    //marks of cannon position
    star(pos[1][2]);
    star(pos[7][2]);
    star(pos[1][7]);
    star(pos[7][7]);

    //marks of soldier position
    for (int i = 0; i<2; i++) {
      for (int j = 0; j<5; j++) {
        int type = 0;
```

```
      if (j==0) type = -1; //left-hand star
      else if (j==4) type = 1; //right-hand star
        star(pos[j*2][3+i*3],type);
    }
   }
  }
  void star(PVector v){
   star(v, 0);
  }

  //draws the star marks in the starting positions of the cannons and soldiers
  //the type accounts for the half-stars at the edge of the board
  void star(PVector v, int type) {
   float ang = 0;
   int num = 4;
   if (type == -1) {
    num = 2;
    ang = PI*3/2;
   } else if (type == 1) {
    num = 2;
    ang = PI/2;
   }
   pushMatrix();
   translate(v.x, v.y);
   for (int i = 0; i<num; i++) {
    pushMatrix();
    rotate(ang +PI/2*i);
    line(s/20, s/20, s/20, s/5);
    line(s/20, s/20, s/5, s/20);
    popMatrix();
   }
   popMatrix();
  }

}
```

TAB: BAR

```
/*
User Interface bar
contains: 2 buttons and 32 text inputs
User can change the worth values and the fear-of-death values of the chess pieces

*/

class Bar {
  Game game;
```

```
int s; //scale
int left; //x position of left side of the bar
int w; //width of bar
Button[] buttons = new Button[2];
Input[] inputs = new Input[32];

Bar(Game game_, int s_) {
 Game game = game_;
 s = s_;
 left = s*9;
 w = s*9;

 //make two buttons
 buttons[0] = new NewGame(game, left +w/4, s, s, "New Game");
 buttons[1] = new DisplayWeight(game, left +w/4*3, s, s, "Display Weight");

 //make a total of 32 input boxes
 for (int j = 0; j<2; j++) { //loop through twice for making "worth" and "fear-of-death" controls
  for (int i = 0; i<2; i++) { //loop through twice for getting chess from red side and black side pieces

   int x = int(left +s*1.6 +w*0.22*i+w*0.46*j); //x location for input box
   int index = i*8+j*16; //index within the inputs array
   int top = s*3; //y location for the top row
   int d = int(s*0.9); //distances between rows

   ArrayList<Chess> chess = new ArrayList<Chess>();
   for (Chess c : game.all) {
    if (c.side == i) chess.add(c);
   }
   inputs[0+index] = new Input(chess, s, x, top +d*0, (j==0), true); //this one controls all values in the column

   chess = new ArrayList<Chess>();
   for (Chess c : game.all) {
    if (c instanceof Gen && c.side == i) chess.add(c);
   }
   inputs[1+index] = new Input(chess, s, x, top +d*1, (j==0));
   chess = new ArrayList<Chess>();
   for (Chess c : game.all) {
    if (c instanceof Cha && c.side == i) chess.add(c);
   }
   inputs[2+index] = new Input(chess, s, x, top +d*2, (j==0));
   chess = new ArrayList<Chess>();
   for (Chess c : game.all) {
    if (c instanceof Hor && c.side == i) chess.add(c);
   }
   inputs[3+index] = new Input(chess, s, x, top +d*3, (j==0));
   chess = new ArrayList<Chess>();
   for (Chess c : game.all) {
```

```
      if (c instanceof Can && c.side == i) chess.add(c);
    }
    inputs[4+index] = new Input(chess, s, x, top +d*4, (j==0));
    chess = new ArrayList<Chess>();
    for (Chess c : game.all) {
      if (c instanceof Adv && c.side == i) chess.add(c);
    }
    inputs[5+index] = new Input(chess, s, x, top +d*5, (j==0));
    chess = new ArrayList<Chess>();
    for (Chess c : game.all) {
      if (c instanceof Ele && c.side == i) chess.add(c);
    }
    inputs[6+index] = new Input(chess, s, x, top +d*6, (j==0));
    chess = new ArrayList<Chess>();
    for (Chess c : game.all) {
      if (c instanceof Sol && c.side == i) chess.add(c);
    }
    inputs[7+index] = new Input(chess, s, x, top +d*7, (j==0));
   }
  }
}

void display() {
  rectMode(CORNER);
  fill(255, 253, 250);
  stroke(150, 120, 80);
  strokeWeight(2);
  rect(left, 0, w, s*10);
  for (Button b : buttons) {
    b.display();
  }
  for (Input inp : inputs) {
    inp.display();
  }
  fill(150, 120, 80);
  textAlign(CENTER, CENTER);
  textSize(s*0.3);
  text("Worth", left+w/4, s*2);
  text("Fear of Death", left+w/4*3, s*2);
}

//this method is called when mouse is clicked
void click() {
  for (Button b : buttons) {
    if (b.clicked()) {
      b.execute();
    }
  }
```

```
   for (Input inp : inputs) {
     if (inp.clicked()) {
       inp.typing = true;
     }
   }
  }

  //this method is called when key is pressed
  void type() {
   for (Input inp : inputs) {
     inp.type();
   }
  }
}
```

TAB: Button

```
/*
Button on the user interface bar
Mouse click on button to execute its commands

*/

//Button superclass
class Button {
  Game game;
  String name; //as displayed on the button
  int s; //scale
  int x;
  int y;
  int w;
  int h;

  Button(Game game_, int x_, int y_, int s_, String name_) {
    game = game_;
    s = s_;
    x = x_;
    y = y_;
    w = s*3;
    h = s/10*7;
    name = name_;
  }

  void display(){
    rectMode(CENTER);
    fill(240,210,170);
    rect(x, y, w, h);
```

```
   fill(150,120,80);
   textAlign(CENTER, CENTER);
   textSize(s*0.35);
   text(name, x, y);
  }

  //method is called by bar when mouse is clicked
  boolean clicked() {
   //if clicked within borders of the button
   if (mouseX > x - w/2 && mouseX < x + w/2 && mouseY > y - h/2 && mouseY < y + h/2) {
    return true;
   }
   return false;
  }

  //method is called by bar when clicked() returns true
  void execute() {
  }
}

//subclass button starts a new game when clicked
class NewGame extends Button {
  NewGame(Game game, int cornX_, int cornY_, int s_, String name_) {
   super(game, cornX_, cornY_,s_, name_);
  }
  void execute(){
   game.newGame();
  }
}

//subclass button toggles the display on weights when clicked
class DisplayWeight extends Button {
  DisplayWeight(Game game, int cornX_, int cornY_, int s_, String name_) {
   super(game, cornX_, cornY_,s_, name_);
  }
  void execute(){
   game.displayWeight = !game.displayWeight;
  }
}
```

TAB: Input

```
/*
Textbox input on the user interface bar
Mouse click on box to start typing.
When finished typing, hit ENTER to update the new value.
```

User can play around with their own values of worth and fear-of-death for chess pieces.

*/

```
class Input {
 boolean typing; //typing is true if the box is clicked, and the user has not yet hit ENTER
 String result; //stores what is being typed
 int input;
 ArrayList<Chess> chess; //the list of chess that this input controls
 int s; //scale
 int x;
 int y;
 int w;
 int h;
 String displayed; //what is displayed in the box
 boolean worth; //if it controls worth or controls fear-of-death
 boolean all; //if it is a special input box controlling all values of the column

 Input(ArrayList<Chess> chess_, int s_, int x_, int y_, boolean worth_) {
  construct(chess_, s_, x_, y_, worth_);
 }
 Input(ArrayList<Chess> chess_, int s_, int x_, int y_, boolean worth_, boolean all_) {
  construct(chess_, s_, x_, y_, worth_);
  all = all_;
 }
 void construct(ArrayList<Chess> chess_, int s_, int x_, int y_, boolean worth_) {
  typing = false;
  result = "";
  chess = chess_;
  s = s_;
  x = x_;
  y = y_;
  w = s;
  h = s/10*6;
  worth = worth_;
  all = false;
 }

 void display() {
  if (typing) { //if typing, display what is being typed
   displayed = result;
  } else { //if not typing, display the value of the worth or fear-of-death
   int s;
   if (worth) {
    s = chess.get(0).worth;
   } else {
    s = chess.get(0).fearOfDeath;
   }
```

```
    displayed = Integer.toString(s);
  }
  rectMode(CENTER);
  fill(240, 210, 170);
  stroke(150, 120, 80);
  strokeWeight(2);
  rect(x, y, w, h);
  fill(150, 120, 80);
  textAlign(CENTER, CENTER);
  textSize(s*0.3);
  text(displayed, x, y);

  if(all){
    text("All", x-s/5*4, y);
  } else {
    chess.get(0).displayIcon(s/4*3, x-s/3*2, y);
  }
}

//method is called by bar when mouse is clicked
boolean clicked() {
  //if mouse is clicked within the borders of the box
  if (mouseX > x - w/2 && mouseX < x + w/2 && mouseY > y - h/2 && mouseY < y + h/2) {
    result = "";
    return true; //if box is clicked, bar will change typing to true;
  }
  return false;
}

//method is called when key is pressed
void type() {
  if (typing) {

    //hit ENTER to update the value
    if (key==ENTER||key==RETURN) {
      typing = false;
      if (isInt(result)) { //only update the value if what is entered is an integer
        input = Integer.parseInt(result);
        update();
      }
    } else if (key==BACKSPACE||key==DELETE){
      result = result.substring(0,result.length()-1);
    } else {
      result = result + key;
    }
  }
}
```

```
 //update the value of the chess pieces within its control array
 void update() {
  if (worth) {
   for (Chess c : chess) {
    c.worth = input;
   }
  } else {
   for (Chess c : chess) {
    c.fearOfDeath = input;
   }
  }
 }

 //test if the string can be converted into integer
 boolean isInt(String s) {
  if (s == null) {
   return false;
  }
  try {
   int n = Integer.parseInt(s);
  }
  catch (NumberFormatException nfe) {
   return false;
  }
  return true;
 }

 //check if String is number. Reference code from:
 //https://www.baeldung.com/java-check-string-number
}

//input textbox. Reference code from:
//https://forum.processing.org/two/discussion/20882/very-basic-question-how-to-create-an-input-text-box
```

TAB: Chess

```
/*
Chess is the superclass of all chess pieces,
namely Gen, Cha, Hor, Can, Adv, Ele, Sol.

Contains important information of the piece such as its:
id, location, alive-status, list of possible moves, worth, fear-of-death, how many times it foresees etc.

When the chess piece is called upon to move, it looks at all its possible moves and weighs each move.
If a move allows it to eat another piece, that move will have a value of the worth of the eaten piece.
If a move puts it in danger of being eaten, that move will have a value of the negative of fear-of-death.
```

How many times it foresees determines how it weighs the possible moves.
If it foresees once, it will also weigh the possible moves of the possible moves.
This additional weight is added by a scale of half.
For example:
If in the next move, it can eat a piece of worth 10, that move will be worth 10;
If in two moves, it can eat a piece of worth 10, that move will be worth 5.

*/

```
class Chess {
  int id; //its index in the array in game, aka game.all[id]
  PVector xy; //its xy position in units
  PVector targetxy; //its target xy position in units
  PVector loc; //its location in pixels
  int side; //0 is black side, 1 is red side
  int s; //scale, pixel distance of one unit
  boolean alive; //alive status
  boolean moving; //if the piece is moving
  ArrayList<PVector> posMoves; //list of all its possible moves
  ArrayList<Integer> weight; //list of the weights of all its possible moves
  int bestMove; //index of best move in the posMoves
  int bestWeight; //value of the highest weight in weight
  PImage img; //image of the Chinese character on the chess
  int worth; //In a move, the weight someone else will get if it eats this piece
  int fearOfDeath; //In a move, the negative weight if this piece moves to a place of danger
  int foreseeTimes; //number of times it foresees moves

  Chess(int id_, int s_, int side_, PVector xy_) {
    id = id_;
    s = s_;
    side = side_;
    xy = xy_;
    targetxy = xy_;
    alive = true;
    moving = false;
    posMoves = new ArrayList<PVector>();
    weight = new ArrayList<Integer>();
    worth = 10;
    fearOfDeath = 20;
    foreseeTimes = 2;

    //get the name of the png file and load it
    String s = this.getClass().getName();
    s = s.substring(17);
    s = s + "_" + String.valueOf(side) + ".png";
    img = loadImage(s);
  }
```

```
 //this method translates xy values from unit to pixels, to get the location of the piece
 PVector getLoc(PVector xy) {
  return new PVector(s/2+xy.x*s, s/2+xy.y*s);
 }

 //displays the chess piece at its location
 void display() {
  if (alive) { //only display chess piece if it is alive

    //if the piece is in the process of moving to a new location
    if (xy.x != targetxy.x || xy.y != targetxy.y) {
     moving = true;
     PVector go = PVector.sub(targetxy, xy); //PVector it needs to get to target
     float d = go.mag(); //distance between it and target
     if (d>3) { //if it is far from target go faster
      go.setMag(0.2);
      xy.add(go);
     } else if (d>0.1) { //if it is approaching target, slow down
      go.setMag(map(d, 0, 3, 0, 0.2));
      xy.add(go);
     } else { //if it is very near target, reach target
      xy.x = targetxy.x;
      xy.y = targetxy.y;
      moving = false;
     }
    }

    //colours depend on its side, black or red
    if (side == 0)stroke(0);
    if (side == 1)stroke(200, 0, 0);
    fill(255, 253, 250);
    loc = getLoc(xy); //calculate location vector
    strokeWeight(1);
    ellipse(loc.x, loc.y, s*0.95, s*0.95);
    strokeWeight(3);
    ellipse(loc.x, loc.y, s*0.8, s*0.8);
    imageMode(CENTER);
    image(img, loc.x, loc.y, s*0.55, s*0.55);
  }
 }
 /* updatePosMoves takes the current xy position of the piece and calculates all possible moves from that
position.
 It then takes those possible moves and puts them into the arrayList which was inputted in the parameters.

 Each subclass of Chess will have its custom function of updatePosMoves, since each piece moves differently.
 */
 void updatePosMoves(ArrayList<PVector> array, Chess[] all) {
 }
```

```
 /*test takes a PVector (from updatePosMoves) and see if it is a valid possible move.
 If it is a valid possible move, it adds the PVector into the array that was given in the parameter.
 It returns a boolean of whether that space is occupied by another piece.
 This boolean is useful for chariots and cannons to calculate their next possible move.
 */
 boolean test(ArrayList<PVector> array, PVector pv, Chess[] all) {
  boolean overlap = false;
  pv.x = Math.round(pv.x);
  pv.y = Math.round(pv.y);
  if (occupied(pv, all) != null) { //if there is another piece occupying the space in this move
   Chess c = occupied(pv, all);
   overlap = true;
   if (c.side != side) { //if that piece is an opponent, then it is a possible move
    array.add(pv);
   }
  }
  if (!overlap) { //if that space is empty, then it is a possible move
   array.add(pv);
  }
  return overlap;
 }

 /* The foresee method is called by weighMove(), in calculating the weight of each move, by foreseeing the moves that can be taken after taking a certain move.
 Before weighMove() calls this, it temporarily changes the xy PVector to one of the possible moves that it seeks to weigh.
 The foresee method then recursively calls updateweighMove() to find the possible moves from that possible move and return the score of the best move scaled by half.
 The function does not recur endlessly because of the fsTimes parameter.
 */
 int foresee(Chess[] all, int fsTimes){

  ArrayList<PVector> foresight = new ArrayList<PVector>(); //storing possible moves from possible move
  updatePosMoves(foresight, all);
  ArrayList<Integer> fsWeight = new ArrayList<Integer>(); //storing the weight of moves in foresight
  for(PVector fs: foresight){
   fsWeight.add(weighMove(fs, all, fsTimes));
  }
  int max = 0;
  for (int i = 0; i<fsWeight.size(); i++) {
   int w = fsWeight.get(i);
   if (w>max) {
    max = w;
   }
  }
  return max/2; //scaled by half each further foresight
 }
```

```
//a method that sees if that position is occupied by a piece of chess
Chess occupied(PVector pv, Chess[] all) {
  for (Chess c : all) {
    if (c.alive && c.xy.x == pv.x && c.xy.y == pv.y) {
      return c;
    }
  }
  return null;
}

//weighs all moves within the posMoves
void weighMoves(Chess[] all, int fsTimes) {
  weight.clear();

  for (PVector pv : posMoves) {
    weight.add(weighMove(pv, all, fsTimes));
  }

}

//weighs one PVector move given in the parameter
int weighMove(PVector pv, Chess[] all, int fsTimes) {
  PVector temp = xy.copy(); //temp stores the original xy position
  xy = pv.copy(); //change the position to the hypothetical move

  int w = 0; //w stores the weight
  int tempDeath = -1; //id of if someone gets eaten

  //see if it can eat someone in that move
  for (int i = 0; i< all.length; i++) {
    if (i != id) { //make sure we don't calculate itself
      Chess c = all[i]; //take all chess pieces
      if (c.alive && c.xy.x == pv.x && c.xy.y == pv.y) {
        c.alive = false; //if overlapped, eats that piece
        tempDeath = i; //the id of the dead piece
        w += c.worth; //weight increases by the worth of the eaten piece
      }
    }
  }

  //calls foresee recursion to see further moves to calculate into weight
  if(fsTimes > 0){
    fsTimes--;
    w += foresee(all, fsTimes);
  }

  //see if it will be eaten by someone in that move
```

```
   for (Chess c : all) {
    if (c.alive && c.side != side) {
     c.updatePosMoves(c.posMoves, all); //update all other chess pieces' possible move to see if they might
eat it
     for (PVector move : c.posMoves) {
      if (move.x == pv.x && move.y == pv.y) {
       w -= fearOfDeath; //weight decrease by the piece's fear-of-death
      }
     }
    }
   }

   if (tempDeath != -1) all[tempDeath].alive = true; //if someone died temporarily in the calculation, make it
alive again
   xy = temp.copy(); //restore the original xy position of the piece
   return w; //returns the final calculation of the weight
  }

  //weight the current position of the piece to see if it is in danger
  int weighCurrent(Chess[] all) {
   int w = 0;
   for (Chess c : all) {

    if (c.alive && c.side != side) {
     c.updatePosMoves(c.posMoves, all);
     for (PVector move : c.posMoves) {
      if (move.x == xy.x && move.y == xy.y) {
       w -= fearOfDeath;
      }
     }
    }
   }
   return w;
  }

  //visualize the weights of the possible moves of this piece
  void visualizeWeight() {
   for (int i = 0; i<weight.size(); i++) {
    PVector pv = posMoves.get(i);

    PVector loc = getLoc(pv);
    fill(0, 0, 255);
    textSize(s/5);
    text(weight.get(i), loc.x, loc.y);
   }
  }

  //see if the position is occupied and returns an integer of who is eaten
```

```
 //returns -1 if no one is eaten
 int eat(PVector pv, Chess[] all) {
  if (occupied(pv, all) == null) return -1;
  return occupied(pv, all).id;
 }

 //move the chess piece to the possible move that has the highest weight
 //returns the id of the piece it eats if anything is eaten
 int move(Chess[] all) {
  getBestWeight(all, foreseeTimes);
  getBestMove();
  if (bestMove == -1) return -1; //if there is no possible move
  PVector xyChosen = posMoves.get(bestMove);
  int eaten =  eat(xyChosen, all);
  targetxy = xyChosen.copy(); //updates its targetxy to the best move
  return eaten;
 }

 //takes a look at all the possible moves and returns the index of the move with the highest weight
 int getBestMove() {
  ArrayList<Integer> maxMove = new ArrayList<Integer>(); //stores all the moves that have the highest weight
  for (int i = 0; i<weight.size(); i++) {
   int w = weight.get(i);
   if (w == bestWeight) {
    maxMove.add(i); //add move to array if its weight equals the maximum
   }
  }
  if (maxMove.size() > 0) {
   bestMove = maxMove.get((int)random(maxMove.size())); //randomly chooses one of the moves with
highest weight
  } else {
   bestMove = -1; //returns -1 if no move is possible aka all pieces are dead
  }
  return bestMove;
 }

 //returns the value of the highest weight among the possible moves
 int getBestWeight(Chess[] all, int fsTimes) {
  updatePosMoves(posMoves, all);
  weighMoves(all, fsTimes);
  int max = -10000; //max is the highest score among the possible moves
  for (int i = 0; i<weight.size(); i++) {
   int w = weight.get(i);
   if (w>max) {
    max = w;
   }
  }
  bestWeight = max;
```

```
  return max;
 }

 //returns true if the chess can move
 boolean hasMove(){
  return (bestWeight != -10000);
 }

 //display its png icon (for the input textboxes)
 void displayIcon(int scale, int x, int y) {
  if (side == 0)stroke(0);
  if (side == 1)stroke(200, 0, 0);
  fill(255, 253, 250);
  strokeWeight(2);
  ellipse(x, y, scale*0.8, scale*0.8);
  imageMode(CENTER);
  image(img, x, y, scale*0.55, scale*0.55);
 }
}
```

TAB: Gen

```
/*
Chess piece: General
move by: vertically and horizontally, 1 unit
eat by: same as move
limit: stay within the box

*/

class Gen extends Chess {
 Gen(int id_, int s_, int side_, PVector xy_) {
  super(id_, s_, side_, xy_);
 }
 void updatePosMoves(ArrayList<PVector> array, Chess all[]) {
  array.clear();
  PVector step = new PVector(1, 0);

  //loop 4 times for all directions
  for (int i = 0; i<4; i++) {
   step.rotate(PI/2);
   PVector pv = new PVector(xy.x, xy.y);
   pv.add(step);
   pv.x = Math.round(pv.x);
   pv.y = Math.round(pv.y);

   //make sure general is within its own box
```

```
    if (side == 0) {
      if (pv.x >=3 && pv.x <=5 && pv.y >=0 && pv.y <=2) {
        test(array, pv, all);
      }
    } else {
      if (pv.x >=3 && pv.x <=5 && pv.y >=7 && pv.y <=9) {
        test(array, pv, all);
      }
    }
   }

 }
}
```

TAB: Cha

```
/*
Chess piece: Chariot
move by: vertically and horizontally, as many units
eat by: same as move

*/

class Cha extends Chess {

 Cha(int id_, int s_, int side_, PVector xy_) {
  super(id_, s_, side_, xy_);
 }

 void updatePosMoves(ArrayList<PVector> array, Chess[] all) {
  array.clear();

  //loop 4 times for all directions
  PVector step = new PVector(1, 0);
  for (int i = 0; i<4; i++) {
    step.rotate(PI/2);
    PVector pv = new PVector(xy.x, xy.y);
    boolean overlap = false;

    //if the chariot has arrived at overlapping another piece, stop loop
    while (!overlap) {
      pv.add(step); //move a step forward
      pv.x = Math.round(pv.x);
      pv.y = Math.round(pv.y);
      if (pv.x < 0 || pv.x >=9 || pv.y <0 || pv.y >=10) {
        break; //if outside borders of the board, break loop
      }
```

```
      overlap = test(array, pv.copy(), all); //test possible move and if it is overlapping another piece
     }
    }

  }
}
```

TAB: Hor

```
/*
Chess piece: Horse
move by: diagonally, 2x1 units
eat by: same as move
trip: cannot move to 2x1 or 2x-1 if there is a piece at 1x0

*/

class Hor extends Chess {
  Hor(int id_, int s_, int side_, PVector xy_) {
    super(id_, s_, side_, xy_);
  }
  void updatePosMoves(ArrayList<PVector> array, Chess[] all) {
    array.clear();
    PVector[] step = new PVector[2]; //the two moves that horse can move, in four directions, total 8 possible
moves
    step[0] = new PVector(2, 1);
    step[1] = new PVector(2, -1);
    PVector trip = new PVector(1, 0); //the location horse must make sure is empty, so it will not trip

    //loop 4 times for all directions
    for (int i = 0; i<4; i++) {
      trip.rotate(PI/2);

      //get both steps
      for (int j = 0; j<2; j++) {
        step[j].rotate(PI/2);
        PVector pv = new PVector(xy.x, xy.y);
        pv.add(trip);
        pv.x = Math.round(pv.x);
        pv.y = Math.round(pv.y);
        if (occupied(pv, all) == null) { //make sure trip location is empty
          pv = new PVector(xy.x, xy.y);
          pv.add(step[j]);
          pv.x = Math.round(pv.x);
          pv.y = Math.round(pv.y);

          //make sure move is within borders of the board
```

```
      if (pv.x >=0 && pv.x <9 && pv.y >=0 && pv.y <10) {
        test(array, pv, all);
      }
     }
    }
   }

 }
}
```

TAB: Can

```
/*
Chess piece: Cannon
move by: vertically and horizontally, as many units
eat by: vertically and horizontally, jump over one piece

*/

class Can extends Chess {

 Can(int id_, int s_, int side_, PVector xy_) {
  super(id_, s_, side_, xy_);
 }

 void updatePosMoves(ArrayList<PVector> array, Chess[] all) {
  array.clear();

  //Loop 4 times for all directions
  PVector step = new PVector(1, 0);
  for (int i = 0; i<4; i++) {
   step.rotate(PI/2);
   PVector pv = new PVector(xy.x, xy.y);
   boolean barrier = false; //whether there is a piece that cannon will jump over
   while (true) {
    pv.add(step); //move one step forward in the direction
    pv.x = Math.round(pv.x);
    pv.y = Math.round(pv.y);

    if (pv.x < 0 || pv.x >=9 || pv.y <0 || pv.y >=10) {
     break; //if move is outside borders of the board, break loop
    }
    if (!barrier) {
     barrier = (occupied(pv, all) != null); //if the location is occupied by another piece, there is a barrier
     if (!barrier) test(array, pv.copy(), all); //if no barrier, the location is a possible move
    }
```

```
      //if there is already a barrier piece
      else {
        if (occupied(pv, all) != null) { //if the piece is occupied, see if it is a move for eating
          test(array, pv.copy(), all);
          break;
        }
      }
    }
   }

 }
}
```

TAB: Adv

```
/*
Chess piece: Advisor
 move by: diagonally 1x1 unit
 eat by: same as move
 limit: stay within the box

 */

class Adv extends Chess {
  Adv(int id_, int s_, int side_, PVector xy_) {
    super(id_, s_, side_, xy_);
  }

  void updatePosMoves(ArrayList<PVector> array, Chess all[]) {
    array.clear();
    PVector step = new PVector(1, 1); //moves by 1x1 unit

    //Loop 4 times for all directions
    for (int i = 0; i<4; i++) {
      step.rotate(PI/2);
      PVector pv = new PVector(xy.x, xy.y);
      pv.add(step);
      pv.x = Math.round(pv.x);
      pv.y = Math.round(pv.y);

      //test if the possible moves are within bounds of the box
      if (side == 0) {
        if (pv.x >=3 && pv.x <=5 && pv.y >=0 && pv.y <=2) {
          test(array, pv, all);
        }
      } else {
        if (pv.x >=3 && pv.x <=5 && pv.y >=7 && pv.y <=9) {
```

```
        test(array, pv, all);
       }
      }
     }

 }
}
```

TAB: Ele

```
/*
Chess piece: Elephant
move by: diagonally, 2x2 units
eat by: same as move
limit: stay within half-court;
trip: cannot move to 2x2 if there is a piece at 1x1

*/

class Ele extends Chess {
 Ele(int id_, int s_, int side_, PVector xy_) {
  super(id_, s_, side_, xy_);
 }

 void updatePosMoves(ArrayList<PVector> array, Chess all[]) {
  array.clear();
  PVector step = new PVector(2, 2); //the elephant move
  PVector trip = new PVector(1, 1); //the location where the elephant must make sure is empty, so it will not
trip

  //loop 4 times for all directions
  for (int i = 0; i<4; i++) {
   trip.rotate(PI/2);
   step.rotate(PI/2);
   PVector pv = new PVector(xy.x, xy.y);
   pv.add(trip);
   pv.x = Math.round(pv.x);
   pv.y = Math.round(pv.y);

   if (occupied(pv, all) == null) { //make sure trip location is empty
    pv = xy.copy();
    pv.add(step);
    pv.x = Math.round(pv.x);
    pv.y = Math.round(pv.y);

    //make sure elephant is within its own half-court borders
    if (side == 0) {
```

```
      if (pv.x >=0 && pv.x <9 && pv.y >=0 && pv.y <=4) {
       test(array, pv, all);
      }
     } else {
      if (pv.x >=0 && pv.x <9 && pv.y >=5 && pv.y <=9) {
       test(array, pv, all);
      }
     }
    }
   }

 }
}
```

TAB: Sol

```
/*
Chess piece: Sol
move by: vertically and horizontally, 1 unit, only forward never backward
eat by: same as move
limit: can only move horizontally once it passes half-court

*/

class Sol extends Chess {
 Sol(int id_, int s_, int side_, PVector xy_) {
  super(id_, s_, side_, xy_);
 }

 void updatePosMoves(ArrayList<PVector> array, Chess all[]) {
  array.clear();
  ArrayList<PVector> step = new ArrayList<PVector>();
  boolean passed = false; //whether it has passed half-court
  PVector pv = new PVector(xy.x, xy.y);

  //different forward vectors, and different half-court borders for different sides
  if (side == 0) {
   step.add(new PVector(0, 1));
   if (pv.y >= 5) passed = true;
  } else {
   step.add(new PVector(0, -1));
   if (pv.y <= 4) passed = true;
  }

  //if passed half-court, add the horizontal moves
  if (passed) {
   step.add(new PVector(1, 0));
```

```
    step.add(new PVector(-1, 0));
   }

   //test all possible moves
   for (int i = 0; i< step.size(); i++) {
    pv = new PVector(xy.x, xy.y);
    pv.add(step.get(i));
    pv.x = Math.round(pv.x);
    pv.y = Math.round(pv.y);
    //check if within borders of the board
    if (pv.x >=0 && pv.x <9 && pv.y >=0 && pv.y <10) {
     test(array, pv, all);
    }
   }

  }
}
```

Work of Chan Hiu Yan
Student at School of Creative Media,
City University of Hong Kong

For the course SM3803 Generative Coding Studio
Student ID: 55131678

December 2019