The Good Famous Games Group

Team Members

Name	Email	Discord Handle	Strengths	Fun Fact
Huynh Nguyen	hngu0112@student.monash.edu	Nera#4749	Fullstack, confident with using Java	Own a part of Tesla shares
Luke Phillips	lphi0011@student.monash.edu	Lionsluke#4181	UI/UX design	Won a B&F in Footy
Yu Xuan Yio	yyio0001@student.monash.edu	roasty #6664	Backend, Usability in mobile applications.	Survived cancer in 2017
Zareef Masud	zmas0005@student.monash.edu	Zareef#9118	Design	Used to play cricket

Team Schedule

Team Meeting Availability

	MON	TUE	WED	THU	FRI	SAT	SUN
Huynh Nguyen	NONE	AFTER 4PM	AFTER 6PM	BEFORE 2PM	WHOLE DAY	AFTER 4PM	AFTER 4PM
Luke Phillips	BEFORE 4PM	NONE	AFTER 4PM	AFTER 10AM	NONE	NONE	AFTER 7PM
Yu Xuan Yio	Before 2pm & After 8pm	After 10am	After 5pm	None	Before 12pm	Anytime (Notify 1 day in advance)	Anytime (Notify 1 day in advance)
Zareef Masud	Null	3pm-6pm	5pm-6pm	4pm-6pm	3pm-6pm	3-pm-6pm	3pm-6pm

Team Scheduled Weekly Meeting

After a team discussion, we came to a consensus that Friday 8 am would be the best time/day for us to meet as a team.

However, this scheduled meeting is subject to change at any time. The change in the team's scheduled weekly meeting may be because of several reasons. For example, A team member falling ill. In this case, the scheduled meeting will be postponed unless the content of the meeting is of urgent concern. This is why we listed our availability throughout the week above, in case of a need to reschedule.

Communication

The team will communicate primarily through Discord. Instant messaging is preferred as it is 'asynchronous' and all members don't need to be present at the same time for communication.

The secondary communication method would be via email.

Work Distribution

Work distribution will be done during our scheduled weekly meetings or before/after workshops. If a team member has a certain preference for the work they want to do, they must voice out so that the piece of work can be their main responsibility. If more than one member wishes to work on a certain piece of work, they can work together. However, they would also have to take up another responsibility for another piece of work so that the work distribution is fair.

All work done should be appropriately logged in the wiki page found in the GitLab repository. If a member is met with setbacks, the team shall collectively help complete the work.

In the case of Sprint 1, we have decided that everyone should contribute equally to the user stories, domain model and justifications, and wireframes. As a result, we want to have an equal understanding of the baseline which will help each member of the team to their personal task later.

Technology Stack

Integrated Development Environment

The team has decided that we will be using IntelliJ IDEA as our IDE. IntelliJ is an IDE built for Java (JDK version 15.0). It is very well supported and very user friendly. Since all members have experience with it, it will be used as the IDE.

Version Control System

The version control system is the Monash GitLab server. We have agreed on a simple git policy that we are going to follow. The main project will be stored in the main branch. Whenever someone wishes to add a feature, a branch will be created for that feature. When the feature is deemed complete, teammates will review the feature before approving the merge to main.

In order to avoid conflicts in git, we will try to not have two people working on the same feature at once. However, this is probably impossible, meaning conflicts are bound to

happen. Hence, whenever a merge conflict happens, it is of utmost importance that the work in the main branch is preserved and not overridden. The person trying to merge will need to try and resolve the conflict before trying to merge again.

Technical Stack

The team evaluated the project and found that the common languages that could be used are Java, Python, C++, C#, and TypeScript.

Some members preferred not to use Python because of past unfavourable experiences with Python GUI packages. C++ is a lower level language than the others and programming with it may take more time and be more error prone (e.g., faults with pointers). GUI development with C++ is also cumbersome as low level system calls need to be used which is not portable.

Although some members may have used C#, the others have no experience with C#. Taking into account the short timeline that we have to complete the project, the general preference is to use Java as all members have past experience with it. Therefore, Java will be used as the programming language for this project. There are several Java frameworks for building GUIs. Swing is an older and more complex one. There is also Java AWT which is simpler and some members have used it before. Since some members have used the Java AWT framework before it will be used in this project. For members that have never used Java AWT before, spikes will be written for them and they will need to complete the spike as soon as possible.

User Stories

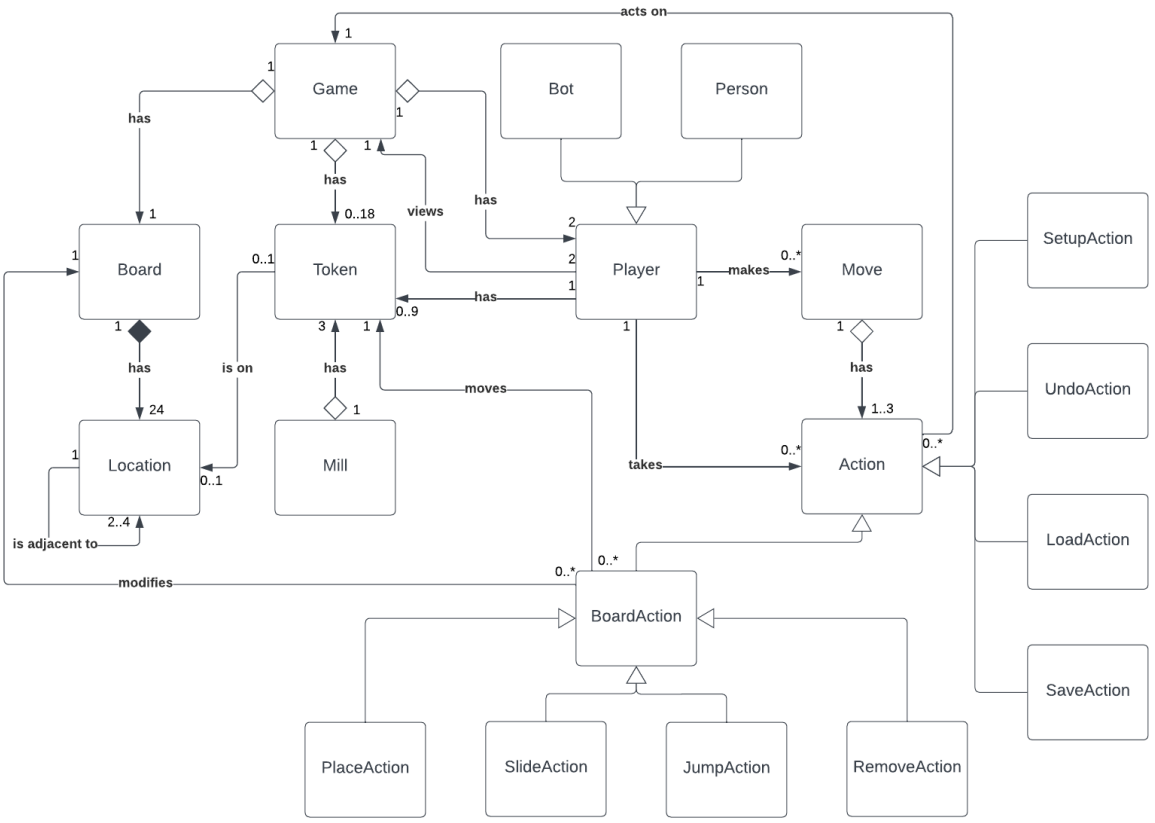
Number	Story	Functionality
1	As a player, I want to start a new game, so that I don't have to continue an old game.	Basic
2	As a player, I want to exit the current game, so that I don't have to keep playing a game.	Basic
3	As a token, I want to be placed on the board from the player, so that I can join the game with another token.	Basic
4	As a token, if I want to be able to form a mill to be able to remove one of the opponent's tokens, so that I can help the player to progress in the game.	Basic
5	As a board, I want to be able to allow the tokens to slide to empty adjacent positions along the grid lines once the player doesn't have more tokens in their token bank, so that the player is able to form a mill.	Basic

6	As a token, I want to be able to jump to any empty position when I am one in three tokens left of my player, to compensate for when my player is about to lose.	Basic
7	As a player, I want to pause the game at any time, so I can interrupt the game if needed.	Basic
8	As a player, I want to alternate turns with my opponent, so that we both have a fair chance at playing the game.	Basic
9	As a game, I want to be able to display the entire game board, what tokens are yet to be placed, and where the player and opponent's tokens are, so that I can become a good famous game.	Basic
10	As a player, I want to see how many tokens I have left to place, so I can plan my next moves.	Basic
11	As a game, I want to allow people to play a game against a human opponent using the same computer, so it will help people to have more fun when playing together.	Basic
12	As a game, I want to allow the user to choose which opponent's token to remove, so that the game has more strategy.	Basic
13	As a game, I want to be able to automatically declare a winner, so I can announce that the game is finished.	Basic
14	As a game, I want to have some rules to be enforced for both players, so that the game is fair.	Basic
15	As a player, I want to save an unfinished game, so that I can continue playing it later.	Advanced (b)
16	As a player, I want to resume a saved game, so that I can finish playing a game I started earlier.	Advanced (b)
17	As a player, I want to view all saved games, so I can choose which game to continue.	Advanced (b)
18	As a player, I want to undo my last move, so that I can change my move.	Advanced (b)
19	As a player, I want to undo multiple moves, so that I can re-evaluate my strategy.	Advanced (b)
20	As a player, I want to delete selected saved games, so that I don't have to keep old games.	Advanced (b)
21	As a game, I want to allow the player to play against the computer, so that the player doesn’t need another person to play.	Advanced (c)
22	As a player, I want to alternate moves with the computer, so that each game is fair.	Advanced (c)
23	As a bot, I want to only make valid moves, so I can follow the rules of the game.	Advanced (c)
24	As a bot, I want to make random moves, so that my play is unpredictable and I can outplay my opponent.	Advanced (c)
25	As a game, I want to provide the player with a winning message when they win, so I can congratulate them.	Basic

26	As a game, I want to show an encouragement message for the player when they lose, so I can motivate them to play the game again	Basic
27	As a board, I want to show which turn is it, so I can help the player to when it is their move	Basic
28	As a player, I want to be able to surrender, so I can end the game if I want	Basic
29	As a game, I want to allow the match to be drawn if both player agree, so that give the best result to both player if needed	Advanced (c)
30	As a token, I want to be different to the opponent token, so my player knows I am their token.	Basic

Domain Model

Link to domain model (Login Required):
https://lucid.app/lucidchart/32240938-372e-4658-b852-6a8543154206/edit?viewport_loc=-362%2C-69%2C2924%2C1627%2CKR~K4MjVOsff&invitationId=inv_51bd236e-e954-4efe-b1d8-1b7ff8091211



Rationales for Domain Entities and Relationships

Defining the Domain Entities

Playing a game of Nine Men's Morris requires a board, two players, and some tokens. Therefore, board, player, and token are fundamental domain entities. The game itself could also be a domain entity. As part of one of the chosen advanced requirements, a player should be able to save and load a game. This reinforces that the game should have its own entity, and it has been included. Another part of the same advanced requirement is that the player should be able to undo moves, so a move could be another domain entity.

In the basic gameplay, the players alternate making moves. At the start of the game, the players can place tokens on empty board locations. So location is another entity. The game progresses by players forming mills with their tokens, so a mill is another entity.

If a player gets a mill then they can remove an opponent's token which isn't part of a mill. This means that on a player's single move they can move (or place) one of their own tokens and then potentially remove an opponent's token as well. These smaller 'moves' can be represented by an 'action' entity, which represents something that a player does. Since a move now contains many actions, the undo operation can either undo single moves or single actions. The undoing of single actions may be more useful to the players, as this will allow choosing a different token to remove from the opponent without having to remake the mill which allowed this. However, both features could be included.

Another option is that players take actions instead of moves, and the move entity is removed. This offers less flexibility since only individual actions can be undone instead of moves. This also doesn't show that a turn could contain many actions. On the plus side it is simpler for the player as they only take actions and don't worry about moves and actions. If only actions are done and undone then a move only serves to group actions. However, an important part of the problem description is that the players alternate moves, and modelling only actions doesn't describe this. So, this alternative won't be taken and a move will still be modelled in order to provide a clearer description of the gameplay.

Since an action models something that a player does, the loading and saving of games and undoing of actions on the board could also be modelled as actions themselves. A distinction needs to be made between actions which affect the board and can be undone, such as moving a token, and actions outside the board which cannot typically be undone, such as saving the game. The idea of undoing moves in the real world usually only applies to moves on the board as the token arrangement can be reversed. If a player takes an action outside the board such as saving a game, they can't undo it and instead have to take an additional

action to discard the save data. A ‘board action’ will be introduced to represent actions which modify the token arrangement on the board and can be undone.

Another important specialisation of the action entity is the action of setting up a new game. To play a new game in the real world the board must be laid out and the players must be assigned tokens. This should also be modelled as a ‘setup action’ since this is something that a player must do to play the game.

Within the board, a player may place a token, slide a token, jump a token, or remove an opponent’s token. All of these are specialisations of the board action since they affect the token arrangement on the board and can be undone.

As part of another advanced requirement, the player should be able to play the game against the computer. Although the ‘bot’ will imitate the player entity, it will be modelled as another entity to clearly show that it is a computer and not a standard human player. To further clarify that the bot and human player are distinct, a ‘person’ entity is added which is another specialisation of the player. Now the player is just some ‘entity’ that plays the game, and it could be a computer or a person.

There is also the option to include entities describing the game state and history of states. Although this seems like it would be useful in describing how the actions affect the game, it is not part of the problem description and isn’t required to model the gameplay. It is more an implementation description and shouldn’t be part of the domain model. To play the game, the players use the board and tokens and take actions which affect them directly. An intermediate state or history only convolutes the diagram and doesn’t really help to understand the game. Therefore, a state or history entity won’t be included.

Another alternative with the tokens is to include a token bank entity to model the tokens which a player has but aren’t yet on the board. Since the player interacts directly with the tokens in the problem description, this intermediate entity isn’t needed and makes the model more complex. A simpler way to model these tokens is to say that a token need not be on a location. This is why the token bank entity won’t be included in the domain model.

Defining the Domain Relationships

To play a game requires one board, two players, and eighteen tokens, so appropriate relationships are formed with these entities. These relationships are aggregations since the game can’t exist without these entities, but the entities can exist without being in a game.

A board contains multiple locations through composition, since a location can’t exist outside the board. Adjacent locations are also related to each other. The corner locations have two neighbours, the edge locations have three neighbours, and centre locations have four neighbours. The locations may have a token on them, and also the tokens may not be on a location as mentioned before. An association will be used instead of aggregation since the tokens aren’t part of the location.

The players have a variable number of tokens depending on how many have been removed by the opponent. The tokens also aren't part of the player, so an association is used instead of aggregation.

A mill could contain tokens through aggregation or a token could be part of a mill through association. The aggregation will be used to clearly show that a mill can't exist without the correct token arrangement.

A move contains one or many actions, and the actions make up a move, so an aggregation will be used instead of an associated. Aggregation is used instead of composition as this allows the possibility of players taking actions outside their turn. For example, if a player is able to save the game when it's the opponent's turn, then they can take an action outside of their own move. This is an assumption which increases the flexibility of the gameplay. The associations between player and action and player and move are also shown. Also note that a move may have one action in the usual case but a maximum of three actions if two mills are formed in one go and two tokens can be removed from the opponent.

The specific actions a player may take are specialisations of the action entity, but the undoable actions are further specialisations of the board action entity. The person and bot are also specialisations of the player.

The board actions operate on and modify the board, which will be shown by association. All board actions move a token in some way, so they also need to move a token. This will be shown with an association from the generalised board action instead of multiple associations from each specific action. Any board actions added in the future should also move at least one token, since by definition a board action modifies the token arrangement. Currently only one token is moved by a single board action, but this could be changed later if new actions are introduced which move multiple tokens at a time. Zero-to-many is the multiplicity on the board action side in both relationships since one board or token may be adjusted by many successive board actions.

The save and load actions will capture and restore the game and all its encapsulated information for future use. The undo action will need to know about the game and its last move or action in order to undo them. The setup action will also need to set up a particular game and will need to have a relationship with it. Since all these actions require a relationship with the game, the base action itself can be associated with a game. This makes sense since all actions occur in the context of a single game. This doesn't affect the board actions since they also occur in the context of a game. Therefore, the base action will be associated with a particular game.

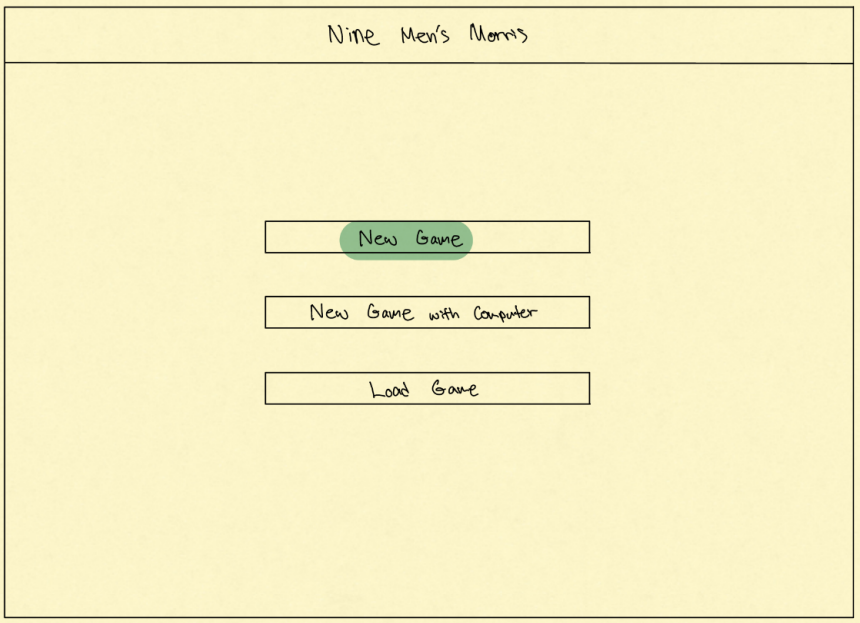
In order to play the game, the players need to view the state of the game in order to strategize and decide what action to take next. They need to look at the board, see the token arrangement, and know how many tokens the opponent has left. All of this information is encapsulated in the game entity. Therefore, the player needs an association with the game in order to view it and take informed actions.

Assumptions

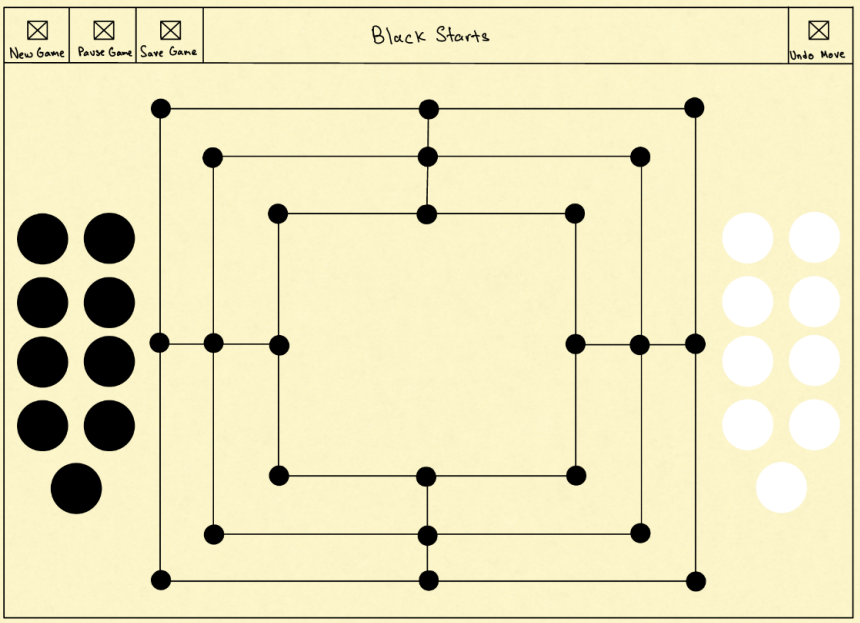
The following assumptions have been made about the gameplay and game rules.

- Saving the game, loading the game, and undoing a move in the game all cannot be undone explicitly.
- Either player may undo the last move regardless of who's turn it is or who made the last move.
- Either player may save the game, load a saved game, or start a new game regardless of whose turn it is.
- If a player makes two mills in one move, they may remove two of the opponent's tokens which aren't in a mill.

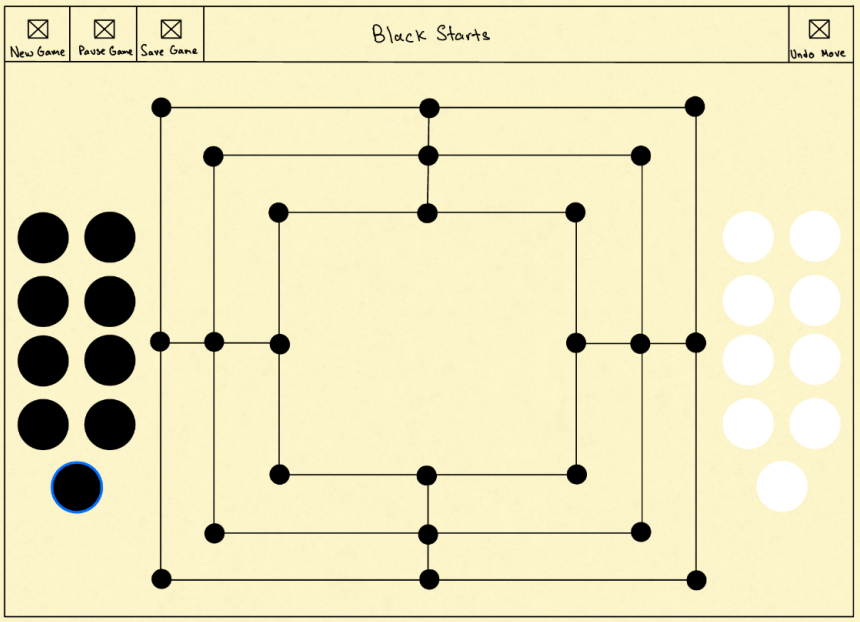
Basic User Interface Design



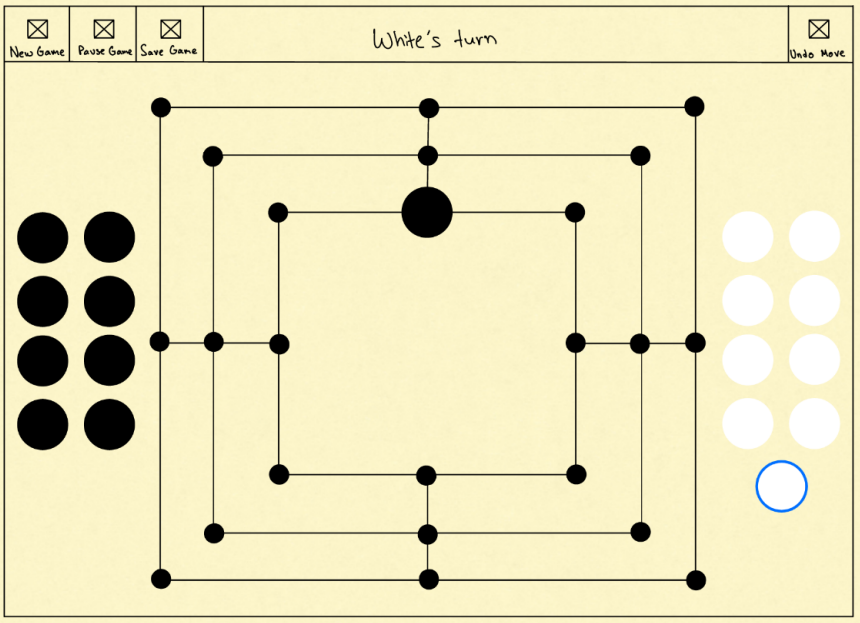
Player select new game



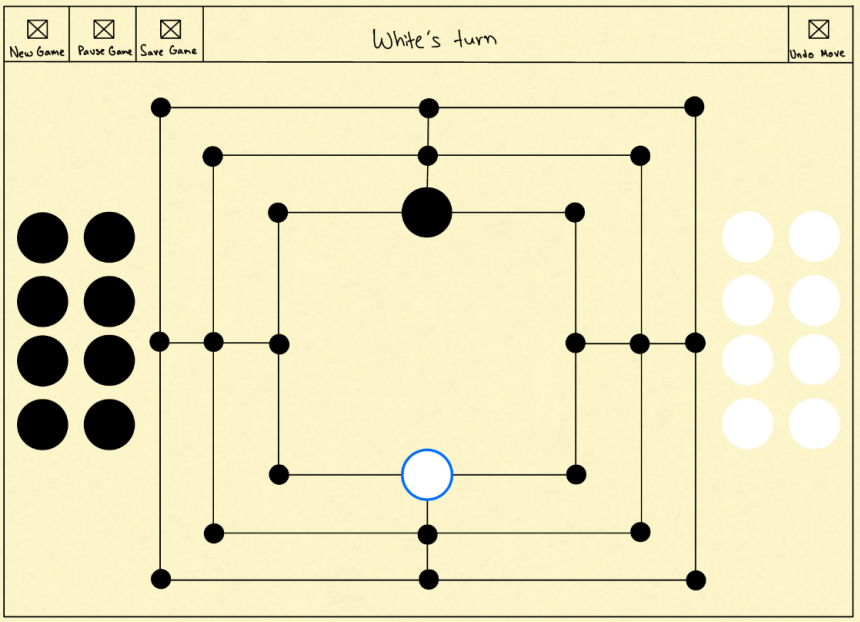
Game starts



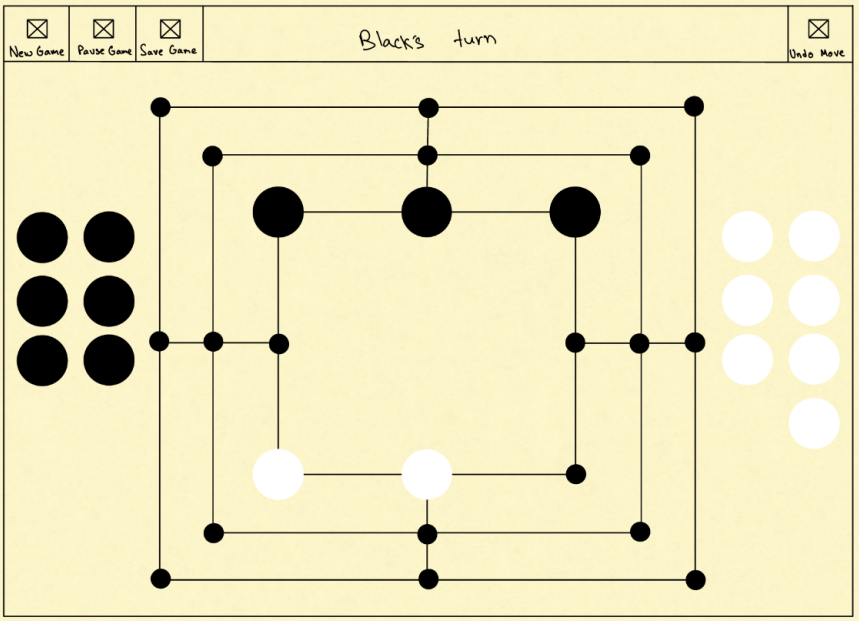
Player selects piece to be placed



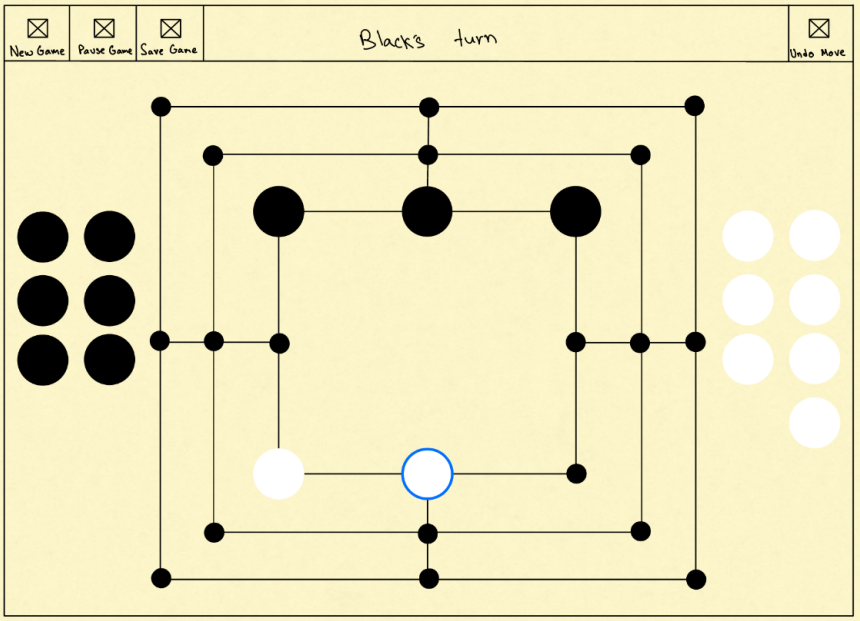
Black places piece, white's turn to act and select piece to place.



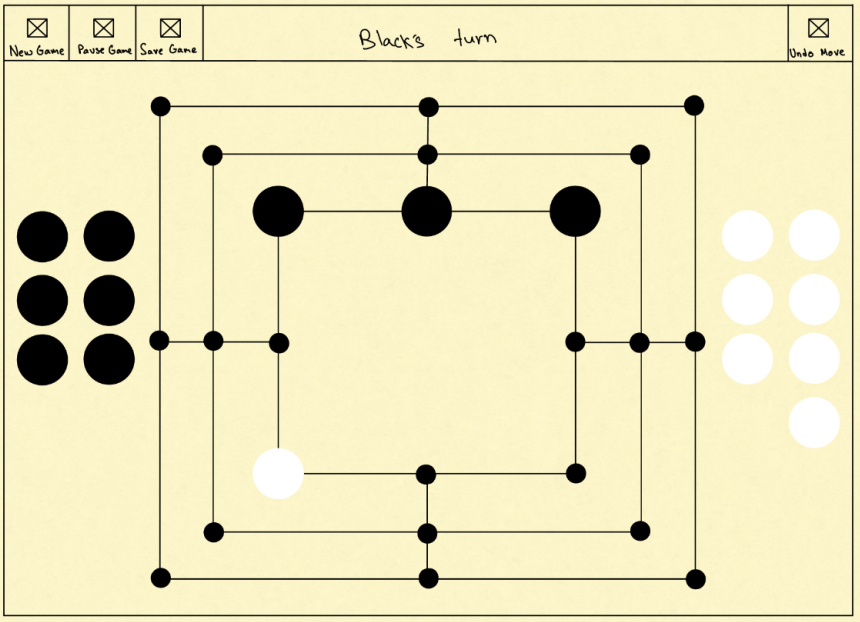
White places piece



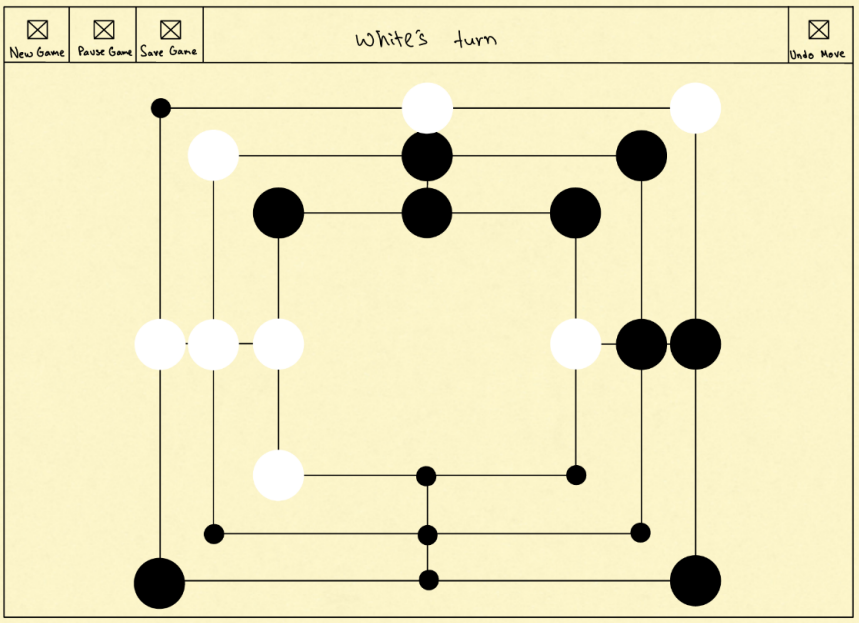
Black has three in a row



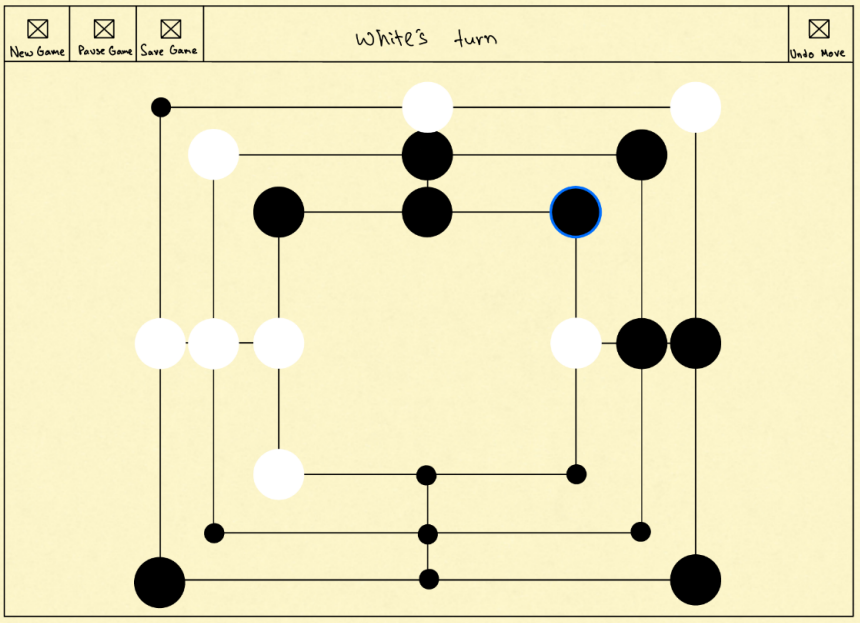
Black select one of white's piece to be removed



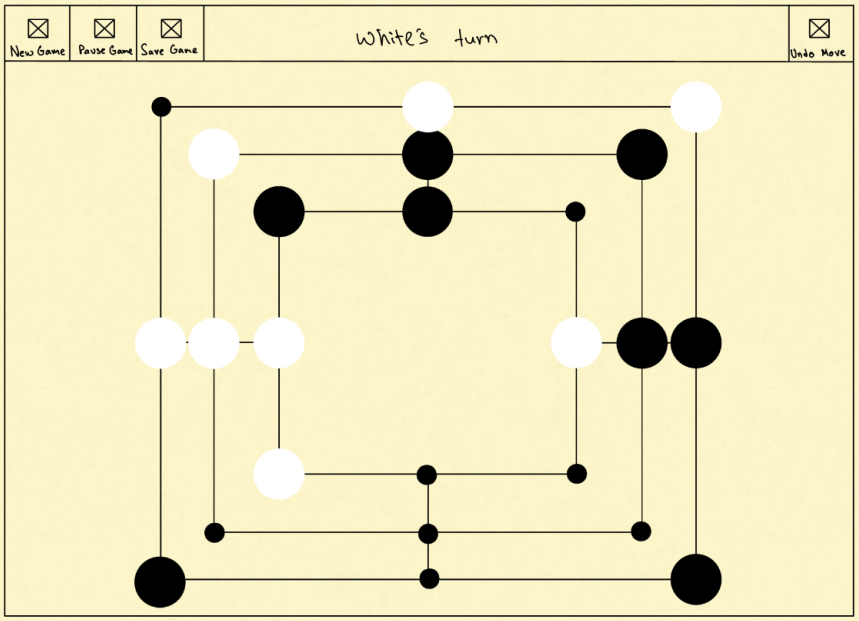
White's piece removed



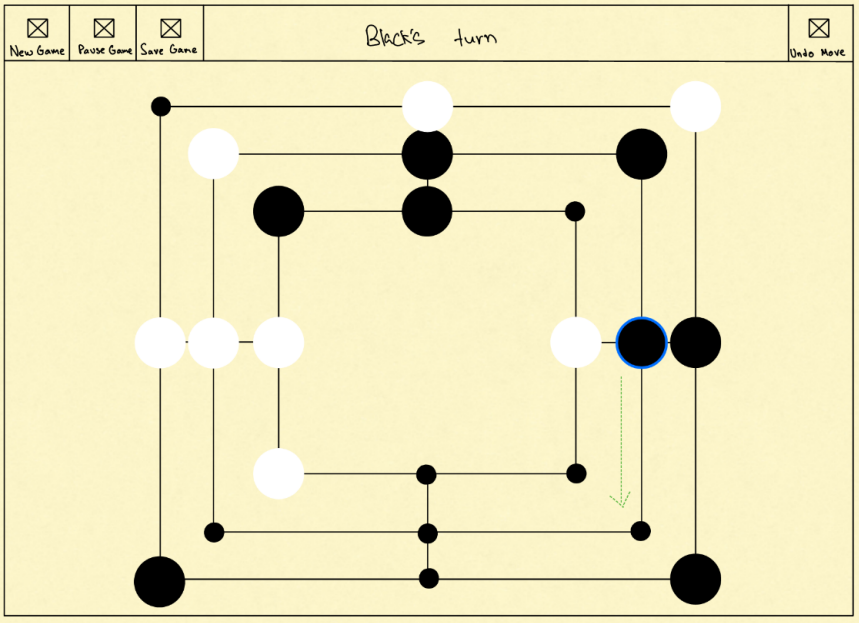
All pieces placed, white has 3 in a row



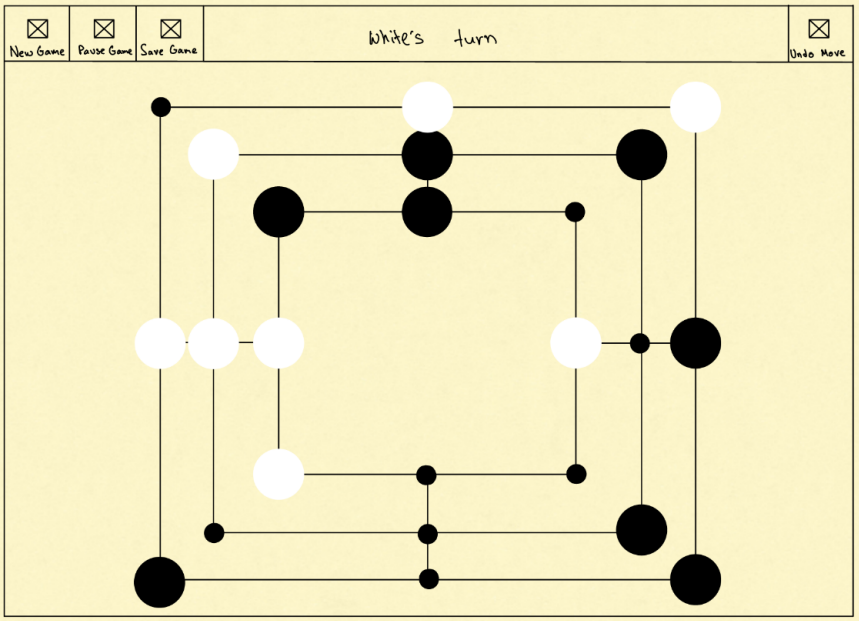
White select one of black's piece to be removed



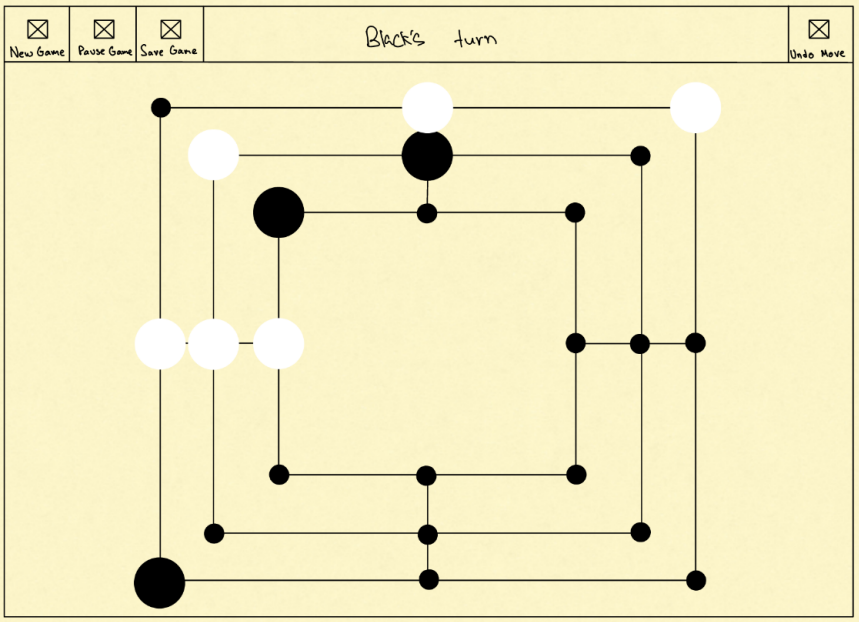
Black's piece removed



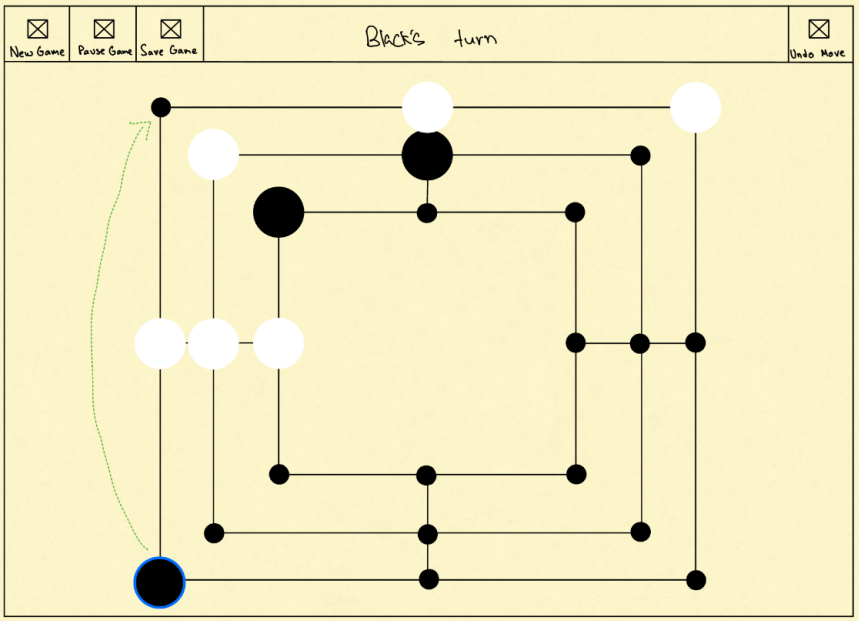
Black selects piece to move downward



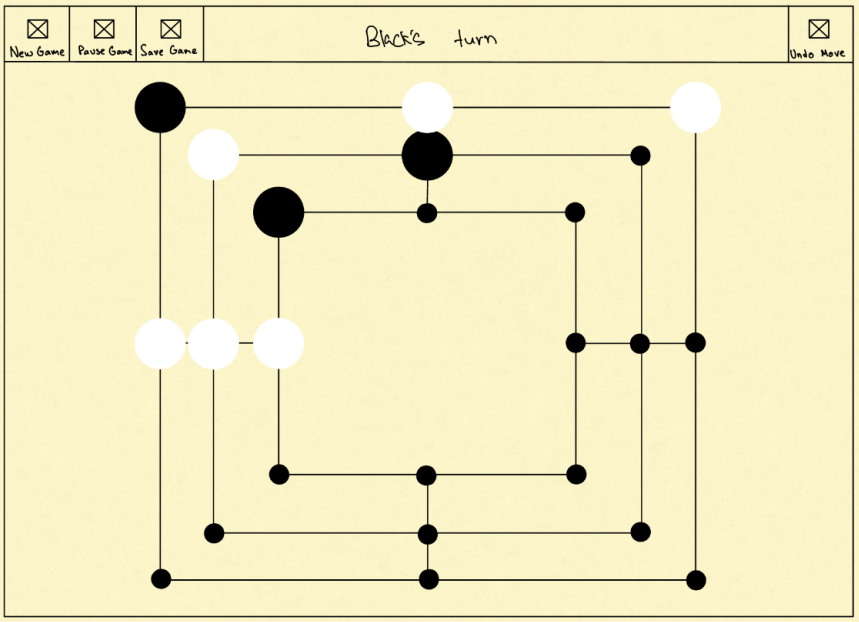
Piece successfully moved, white's turn



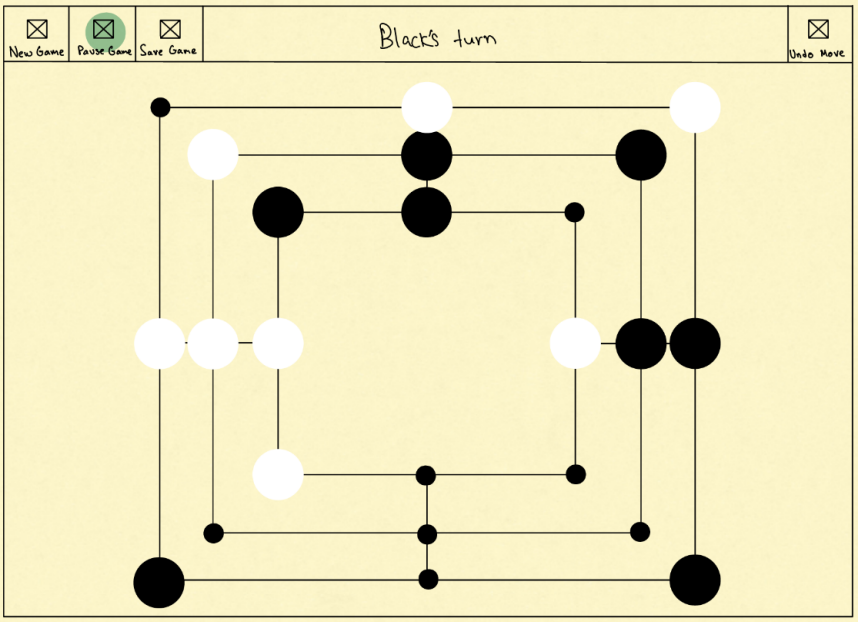
Black down to 3 pieces. 'Flying' enabled



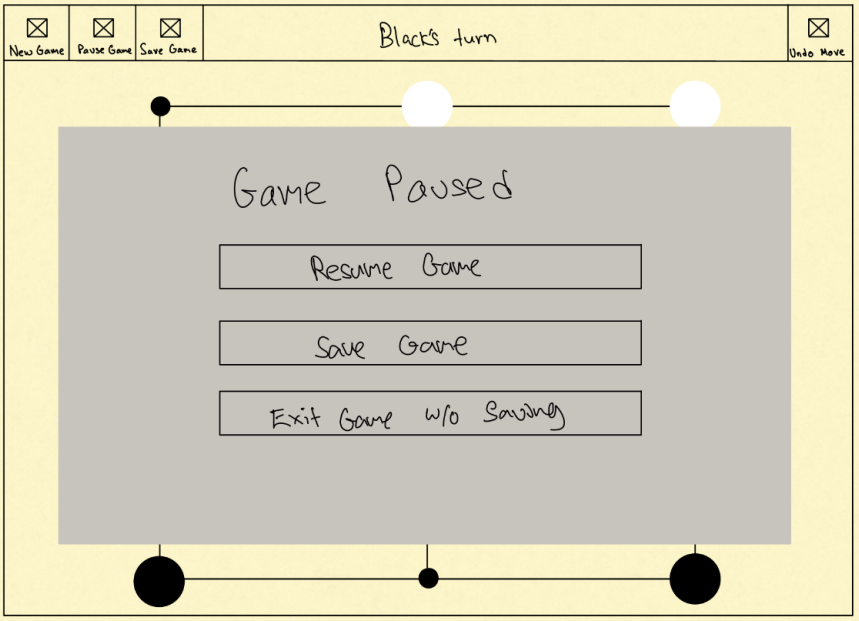
Black selects pieces and chooses to fly upwards as shown in green dotted arrow



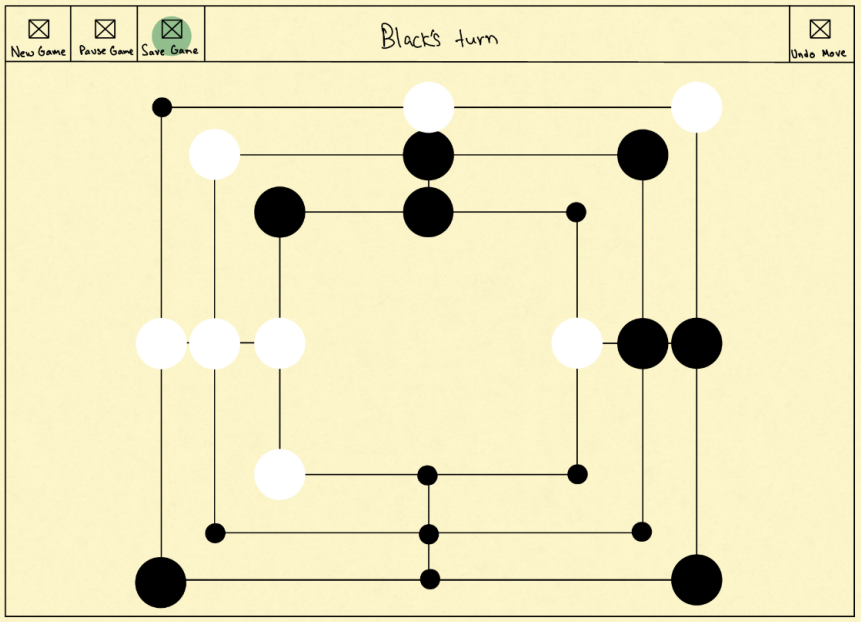
Final board after 'flying'



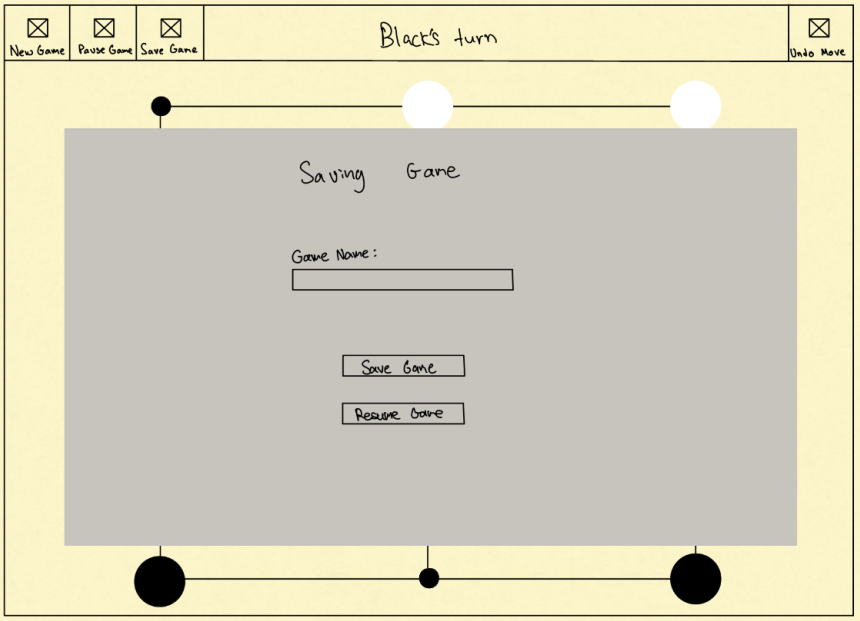
Player selects pause game



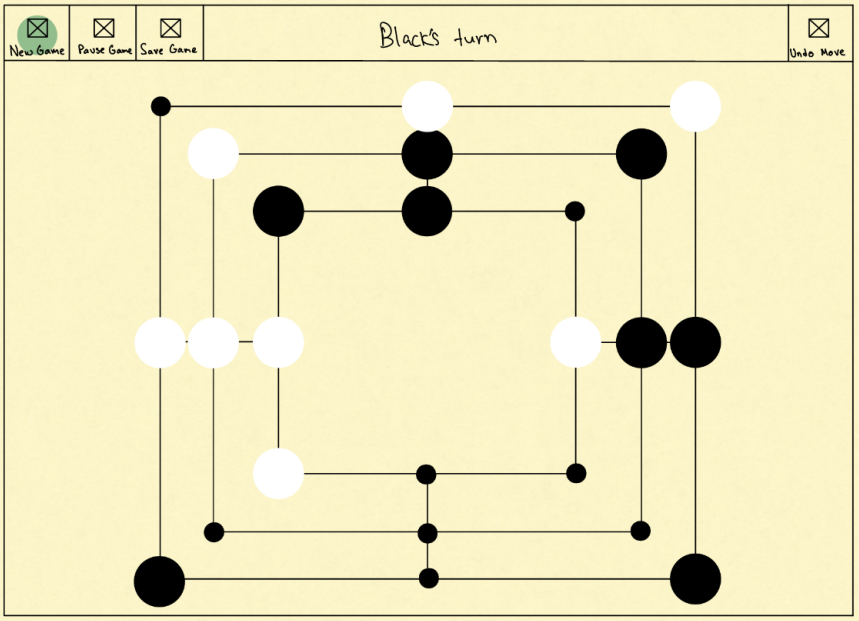
Game paused screen



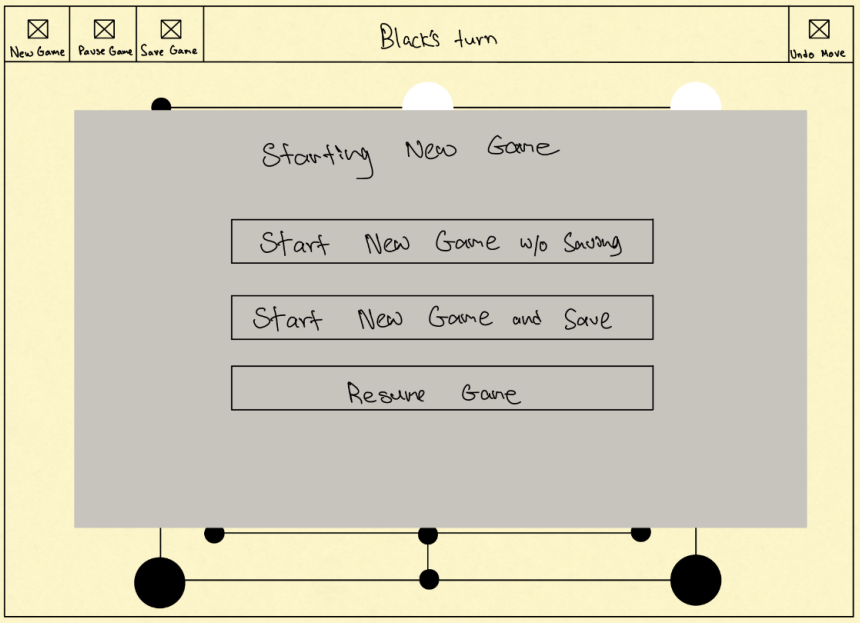
Player selects save game



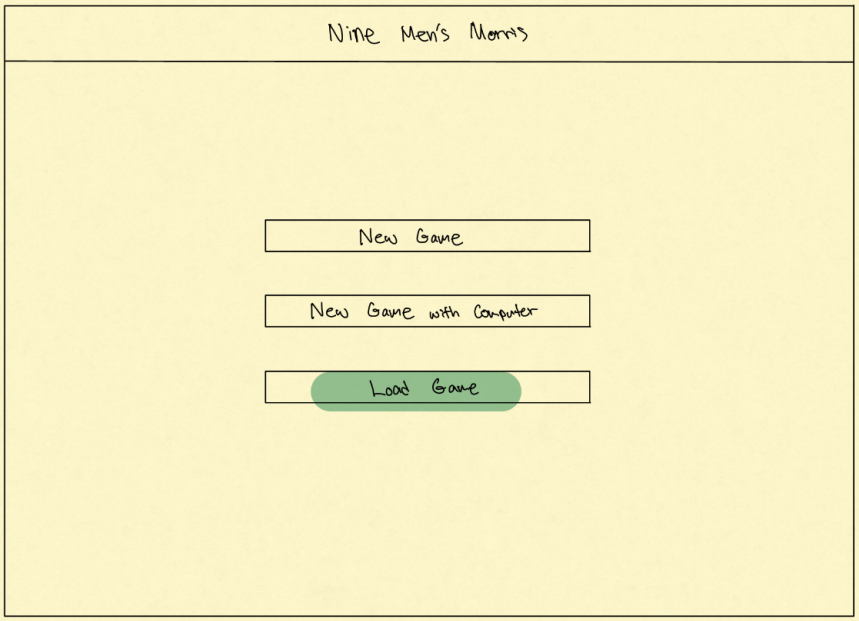
Save game screen



Player selects new game

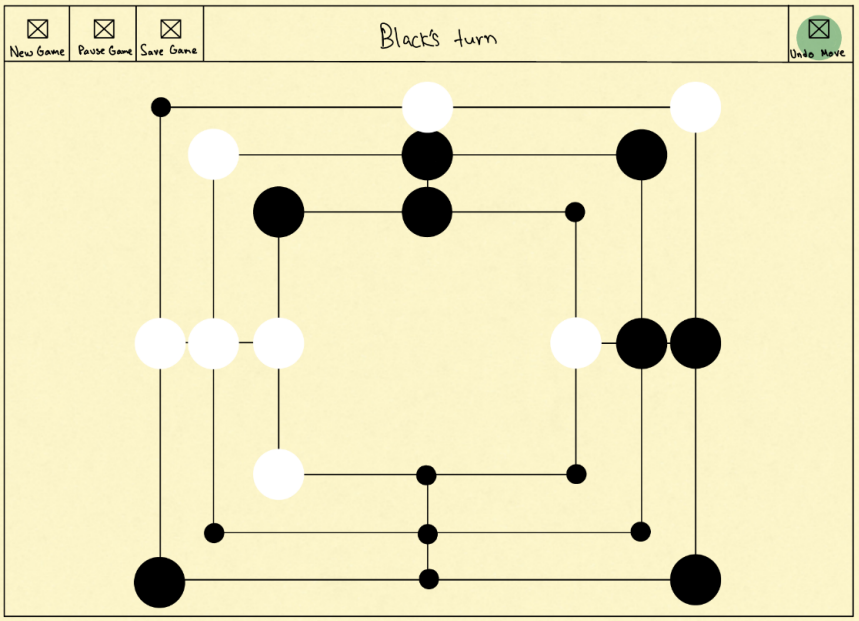


New game screen

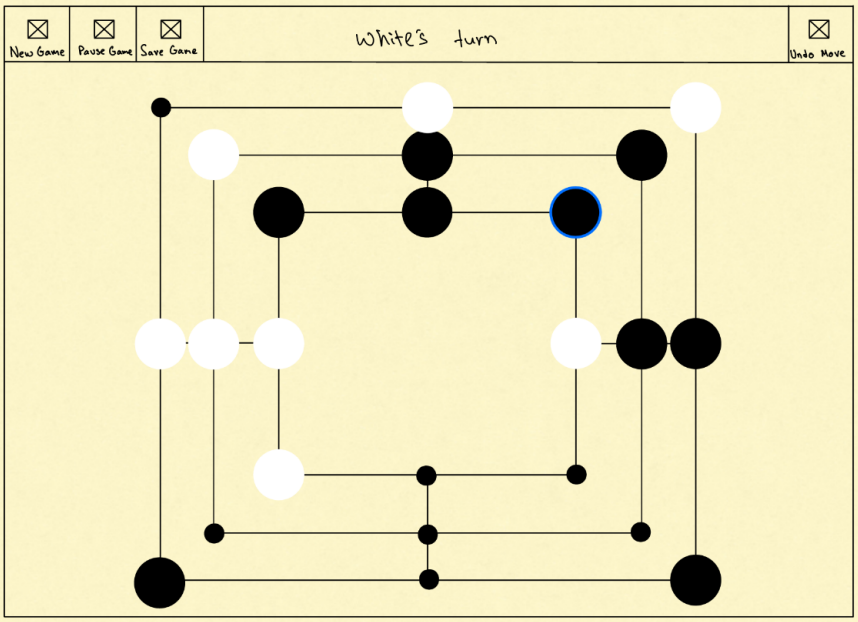


User selects load game

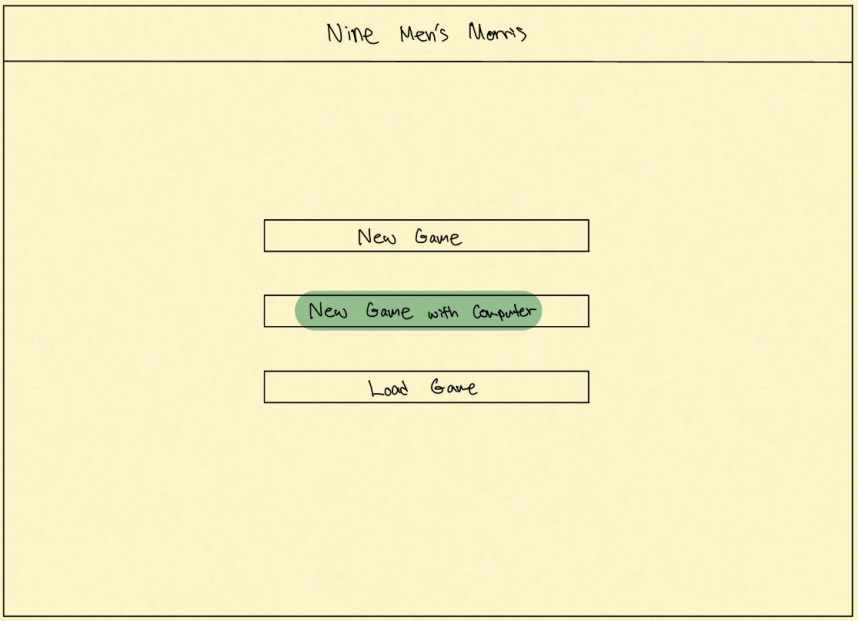
Load game screen, user can select a text file to be loaded

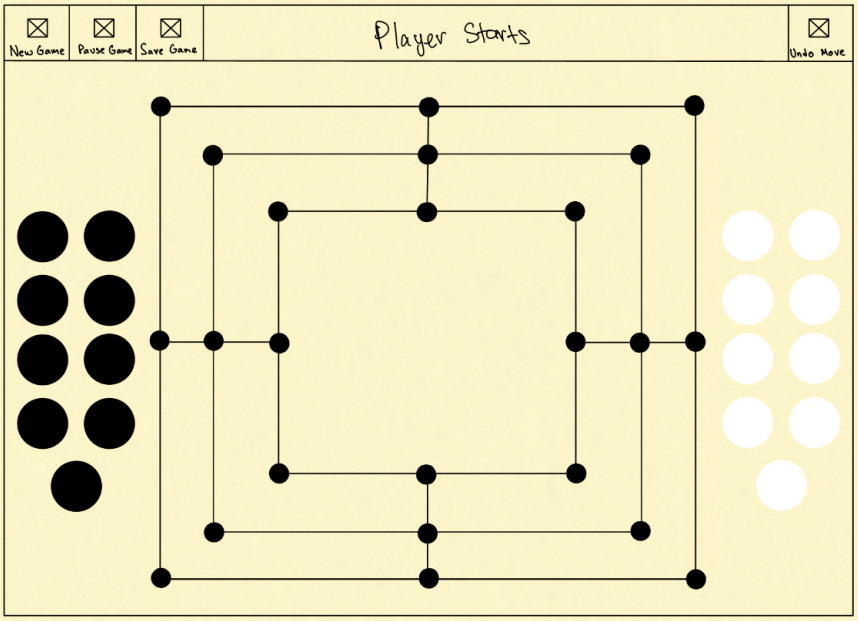


Black selects undo



Turn undone, white's turn





Similar to a human game, except label on top states player/ computer instead of black/ white