GAMESCRAFTERS 2006

ABALONE

DANIEL WEI (DEV)

JERRY HONG (DEV)

MICHAEL MOTTMAN (DEV)

MELINDA FRANCO (DEV)

DOCUMENTATION FOR DEVELOPERS

Version History

Date	Version	Notes
2006.12.6	1.0	First version created
2006.12.18	1.1	Minor fixes

TABLE OF CONTENTS:

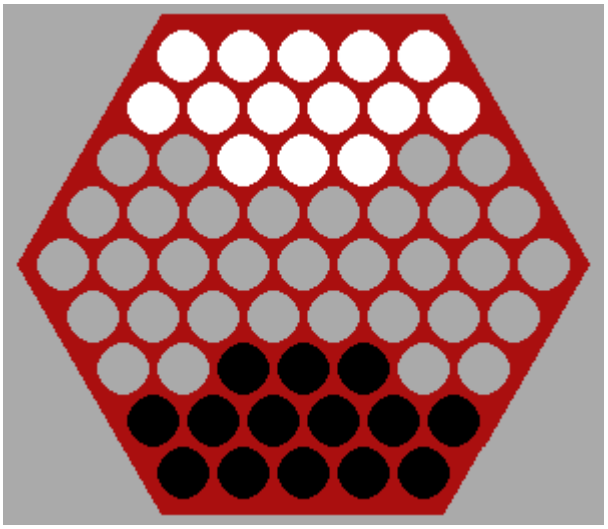
1. Game History

Abalone is a thinking board game that was originated invented in 1987 by Laurent Levi and Michael Lalet. In 1998, it was ranked "Game of the decade" at the "Festival International des jeux."

2. BOARD DESIGN

PART A: BASICS

The board for abalone is a hexagon where there are 5 slots in each slide. An example is shown below:



Picture taken from: http://en.wikipedia.org/wiki/Image:Abalone_start.png

Figure 2-1

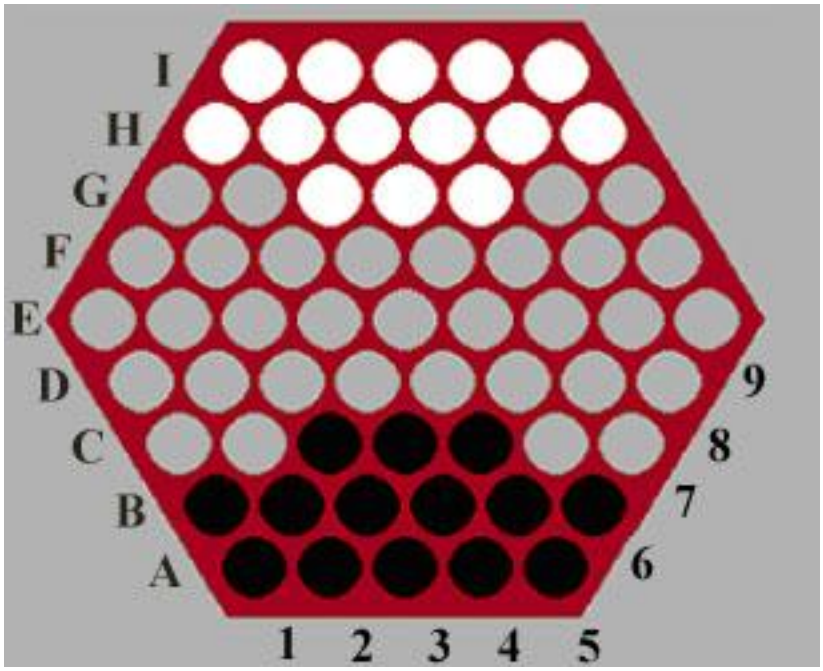
We will from now on call this the 5 X 5 board. As one can see, in a normal abalone game, each player starts off with 14 pieces, and there are a total of 61 spots on the board. Although all abalone games should have a 5 X 5 board, other variations do exists for a smaller size board. Note that there is no variation for a bigger size board. In order to be more efficient, we will show in table 2-1 below how to determine the number of pieces on an N X N board, where $N \leq 5$.

Size N	# of pieces for each player
2	2
3	6
$4 \leq N \leq 5$	$5 + (N-2)^2$

Table 2-1

PART B: POSITION REPRESENTATION

The hexagonal shape of the board creates difficulties in representing the many varies of moves valid in Abalone. The method implemented in this game is not the sole representation of moves, but was selected because of consideration in the ease of the user tying in the moves and the convention followed by most Abalone societies. Consider the following diagram:



Adapted from: http://en.wikipedia.org/wiki/Image:Abalone_start.png

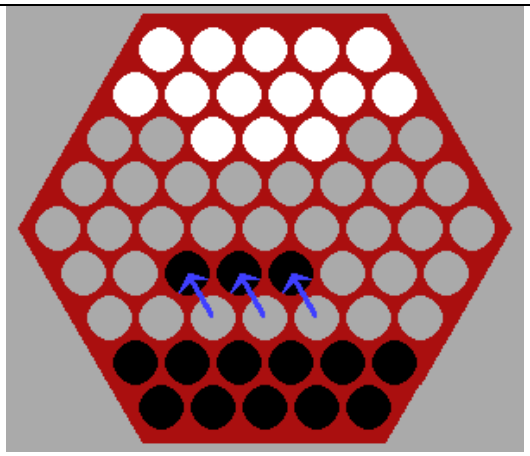
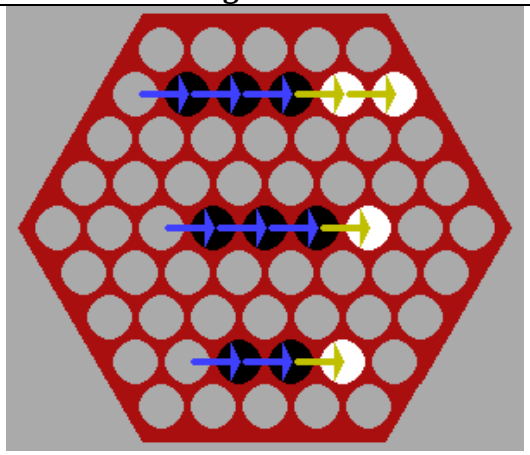
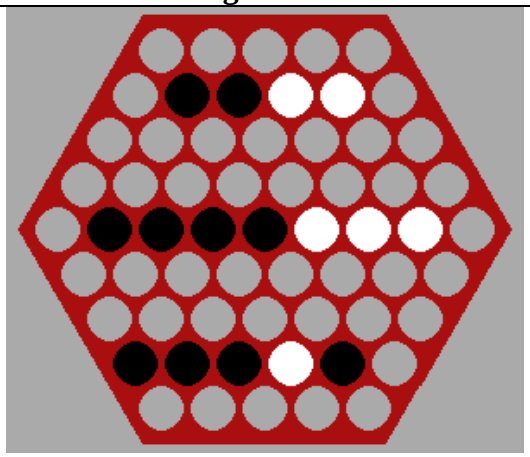
Figure 2-2

Each row is designated by a letter, and each diagonal from bottom right to top left is designated by a number. From this point forward in the text the diagonals will be called "columns" for clarity. The first position (A1) is always the bottommost left position. The rows increment up the board while the columns increment to the right.

Each piece has six cardinal directions in Abalone: West (W), Northwest (NW), Southwest (SW), East (E), Northeast (NE), Southeast (SE).

3. GAME RULES

The objective of the game is to push opposing marbles off the edge of the board. There are several moves that a player can perform to achieve victory.

1 A player can choose to move a line of one, two, or three marbles one space. This is either known as the broadside move or an inline move. See Figure 3-1.	 Picture taken from: http://en.wikipedia.org/wiki/Image:Abalone_broadside.png **Figure 3-1**
2 If one player has more pieces in a line than the opposing character (up to 3), he/she can choose to push the other characters pieces. This is demonstrated in Figure 3-2.	 Picture taken from: http://en.wikipedia.org/wiki/Image:Abalone_push_after.png **Figure 3-2**
3 Here are some examples where the black pieces cannot push the white pieces. See Figure 3.3.	 Picture taken from: http://en.wikipedia.org/wiki/Image:Abalone_push_impossible.png **Figure 3-3**

4. VARIABLES / STRUCTURES

```
#define NULLSLOT 99 //Slot that is not defined on the board or
not used in the move representation
#define MAXN 3 //the maximum n that the solver will solve for
(used only when tiers off)
int N = 2; //board size (defined by length of one side)
int MISERE = 0; // misere off by default
int NSS = 0; // side steps (broadside/inline moves) off by
default
int XHITKILLS = 1; // number of kills needed to win game (1 by
default)
int PIECES = 2; //number of pieces initially per player
int BOARDSIZE; //total slots on board
char *gBoard; // pointer to the global board string (contains
x's, o's, and *'s (empty slots))

struct row {
  int size;
  int start_slot;
}; //basic row structure that contains the size of the row and
first slot integer of the row

struct row **rows; //two dimensional row structure representing
the board
```

5. LIST OF OPTIONS

- Change the value (N) or the size of the hexagon board. The default size is 2
- Toggle from Standard to Misere
- Toggle from Side Steps Allowed to No Side Steps
- Change initial # of pieces
    - Note that this will dynamically update its maximum value depending on the size of N
- Change the # of pieces that must be captured to win

6. MOVE REPRESENTATION (MOVE HASH)

The move hash is represented by a six $digit_{10}$ signed integer with the rightmost two digits representing the first slot, the middle two digits representing the second slot, and the leftmost two digits the third slot. The integer is then added by the direction and multiplied by the sign of the direction.

```
Move x = (sign of direction) * (abs(direction) + (firstSlot *
10) + (secondSlot * 1000) + (thirdSlot * 100000))
```

`MoveHash`
Input: int, int, int, int
Output: int
Takes in the three slot positions (the second and/or third one may be NULLSLOT) and direction integer and outputs the move representation in an integer.

`PrintMove`
Input: int
Output: void
Takes in hashed move integer and outputs the move in a nice formatted string through the use of the function `MoveToString`.

7. POSITION HASH

Uses generic hash. Please look at generic hash documentation for more information.

8. PARSING OF INPUT STRING

A move is designated by listing all of the current affected player's pieces (up to three) followed by the direction of motion. The order of the pieces does not matter as it is handled by the `DoMove` function. There must be a single space separating the each piece and also a single space between the last piece and direction.

Ex. `A1 NE` (A1 to B2)
`B3 B4 B5 NW` (B3 B4 B5 to C2 C3 C4)

Internally, the slots are represented by a two digit integer starting from 1 at the topmost left position (In the 5 by 5 case I5 will be 1) and continuing on to the right of the same row and then down to the next row and so on.

`ValidTextInput`
Input: String
Output: Boolean
This function checks to see if the input string follows the convention mention above. It calls `ValidTextInputHelper` which in turn calls `ValidCoordinate` to determine is the positions are valid. Also, it checks to see if the direction is one of the valid six.

`ConvertTextInputToMove`
Input: String
Output: Move
This function takes in checked string and parses it accordingly. Slot positions are converted from the letter-number representation to a two digit integer representation that is used by the game module. The pieces (up to three) are filled with the respective slot. If less than 3 pieces are moved, the remaining ones are set to `NULLSLOT`. Implicit moves (moving one's own pieces in an inline move) is handled with a `for` loop. The following list details how each direction is represented in the game module: W = -1; E = 1; NW = -2; NE = -3; SW = 3; SE = 2.

9. GENERATE MOVES

`GenerateMoves` determines the Movelist in the following logical order:

- Looks at positive direction moves (move towards the bottom right)
  - Checks for single piece moves (no pushed pieces by rules)
  - Checks for double piece moves
    - No pushed pieces
    - One pushed piece
  - Checks for triple piece moves
    - No pushed pieces
    - One pushed piece
    - Two pushed pieces
- Repeat above for negative direction moves(move towards the top left)

10. PRIMITIVES

Primitive positions are determined by counting up the number of x and o pieces on the board and determining if either player has reached the XHITKILLS. If the difference of either player's total pieces from the initial number of pieces for each player is equal to XHITKILLS the position is primitive.

```
If(((InitialPieces - Current#Xs ) || (InitialPieces -
Current#Os)) == XHITKILLS))
          PrimitiveReached
```

`Game_over`

Input: char[]
Output: Boolean
Helper function to Primitive that determines if a given position is a primitive or not through tallying up each player's pieces.

11. INTERNAL FUNCTIONS

`destination`

Input: int, int
Output: int
Takes in two ints representing current slot and relative direction of the destination and returns the destination slot. It checks to see if destination is valid (slots outside board return NULLSLOT).

`member`
Input: int, int[]
Output: Boolean
Debugging function that is not used in the actual game. Determines if a given slot position is in a given row.

`makerow`
Input: int, int
Output: struct row
Mallocs a new row given the length and beginning slot of the row.

`ValidCoordinate`
Input: int, int
Output: Boolean
Takes in a (x,y) coordinate and determines if it is a valid slot in the board.

12. DESIGN DECISIONS

The board information is represented in a hybrid of two structures, a character array (String) of x's, o's and *'s. The string structure contains information of what type of piece is contained at each slot. This structure is used because it easier to interface with the Gamesman core and also for counting/printing the pieces.

A second structure (the rows**) is created to facilitate the grid nature of the game. Without this structure, the string interface will alone result in massive amounts of unneeded complex math manipulation to check/move pieces around the board. This is due to the hexagonal nature of Abalone. The rows** structure counteract this dilemma by breaking up the board into rows that contain information about its on location in the general board (start slot) and information in how to retrieve an individual slot in the row (size).

13. TERIFYING DESCRIPTION

Tiers are represented by the number of x's and the number of o's on the board. While this creates unsatisfactory large tiers, it is unavoidable due to the loopy nature of the game. Pieces can move back to their original position as long as they are not rolled of the board.

Tier numbers are defined by the following equation:
TierNumber = #Xs * (TotalPieces+1) + #Os

14. BUGS / KNOWN ISSUES

- Board does not solve past 3 by 3 with tiers turned off.
- Board cannot generate past a 3 by 3 board due to the limitations of generic hash