Reversi Game Development

Public Types | Public Member Functions | Private Types | Private Member Functions | Private Attributes | List of all members

ReversiEngine Class Reference

Engine to enforce the rules for a 2-player Reversi/Othello game. More...

`#include <ReversiEngine.h>`

Public Types

enum	**GAME_STATUS_TYPE** { **GAME_OVER** =0, **NEXT_PLAYER**, **INVALID_MOVE**, **INVALID_COORD**, **VALID_MOVE**, **OPP_FINISHED** } Enum to indicate game status given the last applied move. More...
typedef unsigned int	**player_id_t** Type def for the player id.
typedef unsigned int	**board_dim_t** Type def for the game "board".
typedef unsigned int	**player_score_t** Type def for the player score.

Public Member Functions

	ReversiEngine (const **board_dim_t** boardDim=**MIN_BOARD_DIM**) Class constructor; builds the engine given the board size. More...
	~ReversiEngine () Class desctructor; frees memory allocated to game board.
board_dim_t	**getBoardDim** () const Returns the board dimension.
player_score_t	**getPlayerScore** (const **player_id_t** playerId) const Returns the player's score. More...
player_id_t	**getCurrentPlayerId** () const Returns the current player id. More...
player_id_t	**getOpponentId** (const **player_id_t** playerId) const Returns the opponent id. More...

void	**resizeBoard** (const **board_dim_t** newBoardDim) Resizes the board. More...
void	**restartGame** () Restarts the game.
void	**displayBoard** () const Displays the board on the console.
const **board_dim_t** *const	**getBoard** () Returns the board.
GAME_STATUS_TYPE	**applyMove** (const **board_dim_t** xcoord, const **board_dim_t** ycoord) Applies the player's move. More...

Private Types

enum	**DEFAULT_VALUES** { **MIN_BOARD_DIM** = 4, **MAX_NUM_PLAYERS** = 2, **FREE_SPACE_ID** = 0 } Enum to manage default values. More...

Private Member Functions

void	**initBoard** () Initializes the board game for two players. More...
void	**gotoNextPlayer** () Change turns with the other player.
void	**updateScores** () Updates the scores for all players.
bool	**doesMoveExist** (const **player_id_t** playerId) Determines if move exists for player. More...
GAME_STATUS_TYPE	**isMoveValid** (const **board_dim_t** xcoord, const **board_dim_t** ycoord, const **player_id_t** playerId, const bool **applyMove**=false) Indicates if player's move is valid. More...

Private Attributes

board_dim_t	**_boardDim** Dimension of square board game board.
board_dim_t *	**_gameBoard** Game board with discs.
player_id_t	**_playerIds** [**MAX_NUM_PLAYERS**] Array of player identifiers.
player_score_t	**_playerScores** [**MAX_NUM_PLAYERS**] Array of player scores.
player_id_t	**_currentPlayerId** Identifier for the current player.
bool	**_isGameInPlay**

Indicates whether game is in play or not.

Detailed Description

Engine to enforce the rules for a 2-player Reversi/Othello game.

Definition at line 13 of file ReversiEngine.h.

Member Enumeration Documentation

enum ReversiEngine::DEFAULT_VALUES

private

Enum to manage default values.

Enumerator	
MIN_BOARD_DIM	enumeration for the minimum board dimension
MAX_NUM_PLAYERS	enumeration for the maximum number of players
FREE_SPACE_ID	enumeration for the free space identifier

Definition at line 115 of file ReversiEngine.h.

enum ReversiEngine::GAME_STATUS_TYPE

Enum to indicate game status given the last applied move.

Enumerator	
GAME_OVER	enumeration for game over indicator
NEXT_PLAYER	enumeration for next player's turn
INVALID_MOVE	enumeration indicating invalid move
INVALID_COORD	enumeration indicating invalid coordinate
VALID_MOVE	enumeration indicating valid move
OPP_FINISHED	enumeration indicating opponent is finished

Definition at line 25 of file ReversiEngine.h.

Constructor & Destructor Documentation

ReversiEngine::ReversiEngine (const board_dim_t boardDim = MIN_BOARD_DIM)

Class constructor; builds the engine given the board size.

Class constructor to initialize the Reversi engine.

Parameters

boardDim Optional dimension of board game board.

Note

boardDim must be even and at least four.

Engine to enforce the rules for a 2-player Reversi/Othello game.

2014-11-04 Geoff Hayes Initial Release.

Definition at line **12** of file **ReversiEngine.cpp**.

Member Function Documentation

ReversiEngine::GAME_STATUS_TYPE ReversiEngine::applyMove (const board_dim_t xcoord, const board_dim_t ycoord)

Applies the player's move.

Applies a move for the current player.

Parameters

xcoord The x coordinate on the gaming board.

ycoord The y coordinate on the gaming board.

Returns

The status of the game given that move.

Note

The x and y coordinates are one-based with coordinate (1,1) the top-left square of the board, (1,_boardDim) the top-right square of the board, (_boardDim,1) the bottom-left square of the board, and (_boardDim,_boardDim) the bottom-right square of the board.

Definition at line **152** of file **ReversiEngine.cpp**.

bool ReversiEngine::doesMoveExist (const player_id_t playerId) private

Determines if move exists for player.

Determines if at least one valid move exists for the player.

Parameters

playerId The player identifier.

Return values

true if at least one move exists.

false if no moves exist.

Definition at line 363 of file ReversiEngine.cpp.

player_id_t ReversiEngine::getCurrentPlayerId ( ) const inline

Returns the current player id.

Returns the current player identifier.

Definition at line 65 of file ReversiEngine.h.

ReversiEngine::player_id_t ReversiEngine::getOpponentId (const player_id_t playerId) const

Returns the opponent id.

Gets the opponent (other player) identifier given the specified player.

Definition at line 139 of file ReversiEngine.cpp.

ReversiEngine::player_score_t ReversiEngine::getPlayerScore (const player_id_t playerId) const

Returns the player's score.

Returns the score given the player id.

Parameters

playerId The identifier for the player.

Note

playerId is assumed to be positive (>0).

Definition at line 58 of file ReversiEngine.cpp.

void ReversiEngine::initBoard ( )

private

Initializes the board game for two players.

Initializes the board game board for two players, with the four centre squares set with two discs for each player.

Definition at line 40 of file ReversiEngine.cpp.

ReversiEngine::GAME_STATUS_TYPE ReversiEngine::isMoveValid (const board_dim_t xcoord, const board_dim_t ycoord, const player_id_t playerId, const bool applyMove = false)

private

Indicates if player's move is valid.

Indicates if the move is valid or not for the specified player. Applies move if optional flag is set.

Parameters

xcoord	The x coordinate on the gaming board.
ycoord	The y coordinate on the gaming board.
playerId	The player identifier.
applyMove	Optional indicator to apply move the move.

Returns

The status of the game given that move.

Note

The x and y coordinates are one-based with coordinate (1,1) the top-left square of the board, (1,_boardDim) the top-right square of the board, (_boardDim,1) the bottom-left square of the board, and (_boardDim,_boardDim) the bottom-right square of the board.

Definition at line 215 of file ReversiEngine.cpp.

void ReversiEngine::resizeBoard (const board_dim_t newBoardDim)

Resizes the board.

Resizes the board.

Parameters

newBoardDim The new board dimension.

Warning

If a current game is in progress, it will be lost and a new one started.

Definition at line **92** of file **ReversiEngine.cpp**.

The documentation for this class was generated from the following files:

- /Users/geoff/Development/github_repos/madmath-games/reversi/**ReversiEngine.h**
- /Users/geoff/Development/github_repos/madmath-games/reversi/**ReversiEngine.cpp**

Reversi Game Development

ReversiEngine.h

Go to the documentation of this file.

```
/*! \file ReversiEngine.h
    \brief The Reversi Engine class.

    Engine to enforce the rules for a 2-player Reversi/Othello game.

    2014-11-04   Geoff Hayes     Initial Release.
*/

#ifndef REVERSI_ENGINE_H_
#define REVERSI_ENGINE_H_

//! Engine to enforce the rules for a 2-player Reversi/Othello game.
class ReversiEngine
{
    public:

        //! Type def for the player id.
        typedef unsigned int player_id_t;
        //! Type def for the game "board".
        typedef unsigned int board_dim_t;
        //! Type def for the player score.
        typedef unsigned int player_score_t;

        //! Enum to indicate game status given the last applied move.
        enum GAME_STATUS_TYPE
        {
            GAME_OVER=0,    /**< enumeration for game over indicator */
            NEXT_PLAYER,    /**< enumeration for next player's turn  */
            INVALID_MOVE,   /**< enumeration indicating invalid move */
            INVALID_COORD, /**< enumeration indicating invalid coordinate */
            VALID_MOVE,     /**< enumeration indicating valid move */
            OPP_FINISHED    /**< enumeration indicating opponent is finished */
        };

        //! Class constructor; builds the engine given the board size.
        /**
         * Class constructor to initialize the Reversi engine.
         *
         * @param   boardDim   Optional dimension of board game board.
         *
         * @note    boardDim must be even and at least four.
         */
        ReversiEngine(const board_dim_t boardDim = MIN_BOARD_DIM);

        //! Class desctructor; frees memory allocated to game board.
        ~ReversiEngine();

        //! Returns the board dimension.
        board_dim_t getBoardDim() const {return _boardDim;}

        //! Returns the player's score.
        /**
         * Returns the score given the player id.
         *
         * @param   playerId   The identifier for the player.
```

```
56          *
57          * @note    playerId is assumed to be positive (>0).
58          */
59         player_score_t getPlayerScore(const player_id_t playerId) const;
60
61         //! Returns the current player id.
62         /**
63          * Returns the current player identifier.
64          */
65         player_id_t getCurrentPlayerId() const {return _currentPlayerId;}
66
67         //! Returns the opponent id.
68         /**
69          * Gets the opponent (other player) identifier given the specified
70          * player.
71          */
72         player_id_t getOpponentId(const player_id_t playerId) const;
73
74         //! Resizes the board.
75         /**
76          * Resizes the board.
77          *
78          * @param   newBoardDim   The new board dimension.
79          *
80          * @warning If a current game is in progress, it will be lost and
81          *          a new one started.
82          */
83         void resizeBoard(const board_dim_t newBoardDim);
84
85         //! Restarts the game.
86         void restartGame();
87
88         //! Displays the board on the console.
89         void displayBoard() const;
90
91         //! Returns the board.
92         const board_dim_t* const getBoard(){return _gameBoard;};
93
94         //! Applies the player's move.
95         /**
96          * Applies a move for the current player.
97          *
98          * @param   xcoord        The x coordinate on the gaming board.
99          * @param   ycoord        The y coordinate on the gaming board.
100         *
101         * @return  The status of the game given that move.
102         *
103         * @note    The x and y coordinates are one-based with coordinate
104         *          (1,1) the top-left square of the board, (1,_boardDim)
105         *          the top-right square of the board, (_boardDim,1) the
106         *          bottom-left square of the board, and (_boardDim,_boardDim)
107         *          the bottom-right square of the board.
108         */
109        GAME_STATUS_TYPE applyMove(const board_dim_t xcoord,
110                                   const board_dim_t ycoord);
111
112    private:
113
114        //! Enum to manage default values.
115        enum DEFAULT_VALUES
116        {
117            MIN_BOARD_DIM    = 4, /**< enumeration for the minimum board
    dimension */
118            MAX_NUM_PLAYERS = 2,  /**< enumeration for the maximum number of
    players */
119            FREE_SPACE_ID   = 0   /**< enumeration for the free space
    identifier */
120        };
121
122        //! Initializes the board game for two players.
123        /**
124         * Initializes the board game board for two players, with the four
125         * centre squares set with two discs for each player.
126         */
127        void initBoard();
128
129        //! Change turns with the other player.
130        void gotoNextPlayer();
```

```
131
132         //! Updates the scores for all players.
133         void updateScores();
134
135         //! Determines if move exists for player.
136         /**
137          * Determines if at least one valid move exists for the player.
138          *
139          * @param   playerId   The player identifier.
140          *
141          * @retval  true if at least one move exists.
142          * @retval  false if no moves exist.
143          */
144         bool doesMoveExist(const player_id_t playerId);
145
146         //! Indicates if player's move is valid.
147         /**
148          * Indicates if the move is valid or not for the specified player.
149          * Applies move if optional flag is set.
150          *
151          * @param   xcoord        The x coordinate on the gaming board.
152          * @param   ycoord        The y coordinate on the gaming board.
153          * @param   playerId      The player identifier.
154          * @param   applyMove     Optional indicator to apply move the move.
155          *
156          * @return  The status of the game given that move.
157          *
158          * @note    The x and y coordinates are one-based with coordinate
159          *          (1,1) the top-left square of the board, (1,_boardDim)
160          *          the top-right square of the board, (_boardDim,1) the
161          *          bottom-left square of the board, and (_boardDim,_boardDim)
162          *          the bottom-right square of the board.
163          */
164         GAME_STATUS_TYPE isMoveValid(const board_dim_t xcoord,
165                                      const board_dim_t ycoord,
166                                      const player_id_t playerId,
167                                      const bool applyMove = false);
168
169         //! Dimension of square board game board.
170         board_dim_t _boardDim;
171
172         //! Game board with discs.
173         board_dim_t* _gameBoard;
174
175         //! Array of player identifiers.
176         player_id_t _playerIds[MAX_NUM_PLAYERS];
177
178         //! Array of player scores.
179         player_score_t _playerScores[MAX_NUM_PLAYERS];
180
181         //! Identifier for the current player.
182         player_id_t _currentPlayerId;
183
184         //! Indicates whether game is in play or not.
185         bool _isGameInPlay;
186 };
187
188 #endif /* REVERSI_ENGINE_H_ */
```

Reversi Game Development

ReversiEngine.cpp

```
 1  /**
 2   * Engine to enforce the rules for a 2-player Reversi/Othello game.
 3   *
 4   * 2014-11-04   Geoff Hayes     Initial Release.
 5   *
 6   */
 7  #include "ReversiEngine.h"
 8  #include <cmath>
 9  #include <cstring>
10  #include <iostream>
11
12  ReversiEngine::ReversiEngine(const board_dim_t boardDim) :
13      _boardDim(0),
14      _gameBoard(0),
15      _currentPlayerId(0),
16      _isGameInPlay(true)
17  {
18      // initialize the player ids
19      for (int k=0;k<MAX_NUM_PLAYERS;++k)
20      {
21          _playerIds[k] = k+1;
22      }
23
24      // initialize the player scores
25      memset(_playerScores,0,sizeof(player_score_t)*MAX_NUM_PLAYERS);
26
27      // size the board and initialize the board
28      resizeBoard(boardDim);
29  }
30
31  ReversiEngine::~ReversiEngine()
32  {
33      if (_gameBoard)
34      {
35          delete [] _gameBoard;
36          _gameBoard = 0;
37      }
38  }
39
40  void ReversiEngine::initBoard()
41  {
42      // determine the origin, subtracting one since arrays are zero based
43      const board_dim_t xOrigin = _boardDim/2-1;
44
45      memset(_gameBoard,0,sizeof(board_dim_t)*_boardDim*_boardDim);
46
47      _gameBoard[xOrigin*_boardDim+xOrigin]       = _playerIds[1];
48      _gameBoard[xOrigin*_boardDim+xOrigin+1]     = _playerIds[0];
49      _gameBoard[(xOrigin+1)*_boardDim+xOrigin]   = _playerIds[0];
50      _gameBoard[(xOrigin+1)*_boardDim+xOrigin+1] = _playerIds[1];
51
52      // update the player scores
53      updateScores();
54
55      _currentPlayerId = 1;
56  }
57
```

```
58 ReversiEngine::player_score_t ReversiEngine::getPlayerScore(const player_id_t
   playerId) const
59 {
60     player_score_t score = 0;
61
62     if (playerId>=1 && playerId<=MAX_NUM_PLAYERS)
63     {
64         score = _playerScores[playerId-1];
65     }
66
67     return score;
68 }
69
70 void ReversiEngine::displayBoard() const
71 {
72    for (int u=0;u<_boardDim;++u)
73    {
74        for (int v=0;v<_boardDim;++v)
75        {
76            std::cout << _gameBoard[u*_boardDim+v] << "\t";
77        }
78        std::cout << std::endl;
79    }
80 }
81
82 void ReversiEngine::gotoNextPlayer()
83 {
84     _currentPlayerId++;
85
86     if (_currentPlayerId>MAX_NUM_PLAYERS)
87     {
88         _currentPlayerId = 1;
89     }
90 }
91
92 void ReversiEngine::resizeBoard(const board_dim_t newBoardDim)
93 {
94     const board_dim_t oldBoardDim = _boardDim;
95
96     _boardDim = newBoardDim;
97
98     // validate the board size
99     if (_boardDim%2!=0)
100    {
101        _boardDim += 1;
102    }
103
104    if (_boardDim<MIN_BOARD_DIM)
105    {
106        _boardDim = MIN_BOARD_DIM;
107    }
108
109    // allocate memory for the board (if necessary)
110    if (_gameBoard && oldBoardDim!=_boardDim)
111    {
112        delete [] _gameBoard;
113        _gameBoard = 0;
114    }
115    else if (!_gameBoard)
116    {
117        _gameBoard = new board_dim_t[_boardDim*_boardDim];
118    }
119
120    initBoard();
121 }
122
123 void ReversiEngine::updateScores()
124 {
125     board_dim_t const*  pGameBoard = _gameBoard;
126
127     memset(_playerScores,0,sizeof(player_score_t)*MAX_NUM_PLAYERS);
128
129     for (int k=0;k<_boardDim*_boardDim;++k)
130     {
131         const board_dim_t id = *pGameBoard++;
132         if (id>0 && id<=MAX_NUM_PLAYERS)
133         {
134             _playerScores[id-1]++;
```

```
135         }
136     }
137 }
138
139 ReversiEngine::player_id_t ReversiEngine::getOpponentId(
140         const player_id_t playerId) const
141 {
142     player_id_t opponentId = playerId+1;
143
144     if (opponentId>MAX_NUM_PLAYERS)
145     {
146         opponentId = 1;
147     }
148
149     return opponentId;
150 }
151
152 ReversiEngine::GAME_STATUS_TYPE ReversiEngine::applyMove(
153         const board_dim_t xcoord,
154         const board_dim_t ycoord)
155 {
156     ReversiEngine::GAME_STATUS_TYPE status = INVALID_MOVE;
157
158     if (_isGameInPlay)
159     {
160         status = isMoveValid(xcoord,ycoord,_currentPlayerId,true);
161
162         switch (status)
163         {
164             case VALID_MOVE:
165             {
166                 // update the scores
167                 updateScores();
168
169                 // do valid moves exists for opponent?
170                 const player_id_t opponentId = getOpponentId(_currentPlayerId);
171                 if (doesMoveExist(opponentId))
172                 {
173                     // allow the opponent to play
174                     gotoNextPlayer();
175                     status = NEXT_PLAYER;
176                 }
177                 else
178                 {
179                     // opponent can't play, but can current player?
180                     if (doesMoveExist(_currentPlayerId))
181                     {
182                         status = OPP_FINISHED;
183                     }
184                     else
185                     {
186                         // game over!
187                         status = GAME_OVER;
188                         _isGameInPlay = false;
189                     }
190                 }
191                 break;
192             }
193             case INVALID_MOVE:
194             case INVALID_COORD:
195             {
196                 // intentionally left blank
197                 break;
198             }
199             default:
200             {
201                 // intentionally left blank
202             }
203         }
204     }
205
206     return status;
207 }
208
209 void ReversiEngine::restartGame()
210 {
211     resizeBoard(_boardDim);
212     _isGameInPlay = true;
```

```
213 }
214
215 ReversiEngine::GAME_STATUS_TYPE ReversiEngine::isMoveValid(
216         const board_dim_t xcoord,
217         const board_dim_t ycoord,
218         const player_id_t playerId,
219         const bool applyMove)
220 {
221     ReversiEngine::GAME_STATUS_TYPE status = INVALID_MOVE;
222
223     bool foundMove = false;
224
225     // ensure that x and y are within the gaming area
226     if (xcoord<1 || xcoord>_boardDim || ycoord<1 || ycoord>_boardDim)
227     {
228         status = ReversiEngine::INVALID_COORD;
229     }
230
231     // ensure that the selected square is empty
232     else if (_gameBoard[(xcoord-1)*_boardDim + (ycoord-1)] != FREE_SPACE_ID)
233     {
234         status = ReversiEngine::INVALID_MOVE;
235     }
236
237     // ensure that there is a "string" of uninterrupted opponent
238     // discs terminated with a player disc
239     else
240     {
241         const player_id_t opponentId = getOpponentId(playerId);
242
243         // consider all neighbouring squares surrounding the square that
244         // the user has applied his/her move to
245         //
246         // as there are eight neighbours, we look at all combinations of
247         // left or right with above or below
248         for (int m=-1;m<=1;++m)
249         {
250              const int xcoordP = static_cast<int>(xcoord) + m;
251
252              // skip if new coordinate is at edge
253              if (xcoordP<1 || xcoordP>_boardDim)
254              {
255                 continue;
256              }
257
258              for (int n=-1;n<=1;++n)
259              {
260                 const int ycoordP = static_cast<int>(ycoord) + n;
261
262                 // skip if new coordinate is at edge
263                 if (ycoordP<1 || ycoordP>_boardDim)
264                 {
265                     continue;
266                 }
267
268                 if (_gameBoard[(xcoordP-1)*_boardDim + (ycoordP-
1)]==opponentId)
269                 {
270                     // neighbouring square is that for an opponent
271
272                     // if we continue with the same +m,+n step sizes, is
273                     // the end point the player's id?
274                     int xcoordPP = xcoordP;
275                     int ycoordPP = ycoordP;
276
277                     // count the number of opponent squares that we are
278                     // stepping over (and that would be flipped if a successful
279                     // move has been found)
280                     int numOppSqrs = 1;
281
282                     // there will be a maximum of _boardDim steps
283                     for (int steps=0;steps<_boardDim;++steps)
284                     {
285                         xcoordPP = xcoordPP+m;
286                         ycoordPP = ycoordPP+n;
287
288                         if (xcoordPP<1 || xcoordPP>_boardDim ||
289                             ycoordPP<1 || ycoordPP>_boardDim)
```

```
290                             {
291                                 // hit edge of game board, so exit
292                                 break;
293                             }
294                             else if (_gameBoard[
295                                 (xcoordPP-1)*_boardDim + (ycoordPP-1)] ==
296                                     FREE_SPACE_ID)
297                             {
298                                 // hit free space so invalid
299                                 break;
300                             }
301                             else if (_gameBoard[
302                                 (xcoordPP-1)*_boardDim + (ycoordPP-1)] ==
303                                     playerId)
304                             {
305                                 // a valid move has been found
306                                 foundMove = true;
307 
308                                 // exit if we are checking for a valid move
309                                 if (!applyMove)
310                                 {
311                                     break;
312                                 }
313 
314                                 // move is allowed, so apply it working backward
315                                 // until we reach the original position, marking
316                                 // opponent squares as the current player ones
317                                 // (we add one to numOppSqrs to capture the square
318                                 // that the user has selected (stating point))
319                                 for (int k=1;k<=numOppSqrs+1;++k)
320                                 {
321                                     xcoordPP = xcoordPP-m;
322                                     ycoordPP = ycoordPP-n;
323 
324                                     _gameBoard[
325                                        (xcoordPP-1)*_boardDim + (ycoordPP-1)] =
326                                             playerId;
327                                 }
328 
329                                 // exit
330                                 break;
331                             }
332                             else
333                             {
334                                 // neighbouring square is that of opponent
335                                 numOppSqrs++;
336                             }
337                         }
338                     }
339 
340                     // exit early if a move has been found and we are checking only
341                     if (foundMove && !applyMove)
342                     {
343                         break;
344                     }
345                 }
346 
347                  // exit early if a move has been found and we are checking only
348                 if (foundMove && !applyMove)
349                 {
350                     break;
351                 }
352             }
353         }
354 
355         if (foundMove)
356         {
357             status = VALID_MOVE;
358         }
359 
360         return status;
361     }
362 
363     bool ReversiEngine::doesMoveExist(const player_id_t playerId)
364     {
365         bool status = false;
366 
367         // iterate over all possible moves
```

```
368     for (board_dim_t m=1;m<=_boardDim;++m)
369     {
370         for (board_dim_t n=1;n<=_boardDim;++n)
371         {
372             status = (isMoveValid(m,n,playerId)==VALID_MOVE);
373             if (status)
374             {
375                 break;
376             }
377         }
378         if (status)
379         {
380             break;
381         }
382     }
383 
384     return status;
385 }
```