Abstract: Team Awesomeness set out to create a fully functional clone of the classic board game Reversi (Othello©) using only the terminal interface provided by Ncurses. Our overarching goals were to completely implement the basic game architecture in a robust fashion, including a functional, multilevel AI and a powerful undo function, to produce highly-structured and fault-tolerant code, and to demonstrate some of the hidden power of the Ncurses library.
Background Information: The Game of Reversi

The game of Reversi was invented in the later 1800's by two Englishmen, Lewis Waterman and John Mollett.¹ The game is remarkable both for its simplicity of play and for the complexity of its outcomes. However, despite this complexity, during the 20th century Reversi became one of a growing class of board games considered "solved" by modern computers.

The game is played on an eight-by-eight grid of squares. Initially the four center squares are occupied, two by black tokens and two by white tokens. The black player always moves first. The object is to place tokens on the board such that one or more opponent's tokens are trapped between two of the player's own. When tokens become trapped they are flipped to the player's color. Play continues as long as at least one player has a move, and players forfeit moves when they do not have any valid board positions. Game play ends when there are no more moves, whereupon the player with the highest score wins (or a tie results).²

Project Motivations, Goals, and Philosophy

Project Ollehto had a number of overarching philosophical goals in mind when development began. To a great extent the motivations and goals had a larger impact in determining the outcome and shape of the project than any other single aspect, including the nature of Reversi itself.

Highly-structured code, modular in form and object-oriented in function, is in general much easier to maintain and develop than its strictly functional counterpart. One of the Team's primary goals entering into the project was to create code that was as structured—according to the general principles of object-oriented design—as possible. Secondarily, but intimately related to this goal, the Team wished to create objects that would, as much as possible, protect their own member variables from corruption and simplify code changes. Thus numerous abstraction barriers are employed across multiple objects with the goal of making logical errors difficult to produce, resulting in the final API being quite extensive.

The choice of Ncurses as the primary game interface was, in part, motivated by dissatisfaction with other available interface design libraries, and by our group’s strong familiarity with the Ncurses library. We regarded Ncurses not only as a beneficial addition to our game (being somewhat attached to terminal and text-based applications) but also as a chance to challenge ourselves to stretch the library to its functional limit.

¹ http://yahooligans.yahoo.com/content/games/rules/reversi/reversihistory.html
Program Functionality (Interface)

Ollehto is a console application, designed to operate on the Linux platform in conjunction with the Ncurses library, and as such it targets users who, to some degree, are comfortable with console applications. This being said, one of the primary interface design goals of our project was to minimize the extent to which the user has to deal with the text-based quirks of most console application. The primary program interface—that is, the game board and its related effects—is mouse-driven (the most natural computer interface for board games), while the options portion of the program is menu-driven.

Of special interest within Ollehto is the opening cinematic sequence, comprised of various ASCII art images set to a timer. If nothing else this proves that Ncurses is as capable of eye-candy, in its own special way, as any traditional GUI development library.

The main game screen is comprised of an eight by eight grid, the colors of which (along with the various menus which precede it) can be set to one of a number of predefined schemes. There is a scoreboard in the upper right-hand corner of the screen, and (if at least one human player is present in the game) an undo button in the lower right-hand corner. The undo button allows for full backtracking through all previous game states up to the beginning of the game. It only removes human moves (and any intervening computer moves), since several of the AI algorithms are fairly deterministic, and removing computer moves independently seemed unhelpful. When the game is completed the winning player is announced, and the board is rearranged to highlight the extent of the win.

Configurable game options include the state of each player (that is whether the player is human or computer, any combinations of the two being possible), the level of each computer player’s AI (from one to five), and the color scheme of the game menus and the main game interface.

Program Functionality (Internals)

Originally the Team wished to have a fully encapsulated, modular Ncurses "object" as part of its class hierarchy. As development was underway it became increasingly apparent that this would not be possible, given the nature of Ncurses. Thus Ncurses took on its more traditional role as a program "wrapper." All incoming human interactions and outgoing display requests pass through Ncurses, which acts as the principal program driver, controlling interrupts and triggering events.

Within Ncurses sit two fully enclosed objects: the Game object, which maintains game state and communicates with Ncurses, and the History object, which records moves, and which is used by Ncurses to implement the undo function. The Game object contains the game state variables (the Board, the AIs, the player settings, etc.) and a few key functions to allow the game to be reset from a History object and to take a turn.
When a turn is taken it is not known whether the game is over or not, nor which color will be taking a turn. Therefore the Game object must first build a list of possible moves. It reads the board square by square and locates all squares adjacent to an opponent's token for each color. These locations are then pushed onto a list. (The choice of a list was quite arbitrary, as the desired functionality was quick insert time and linear access, so any list-like data structure would have sufficed.)

Once the potential move list is built, it is traversed and each element is tested using the Board's isValid() function. If a potential move is valid it is inserted into a set. (Initially invalid moves were going to be deleted from the list, but it is impossible to traverse a list and change it at the same time.) The choice of a set was motivated by the way in which the possible move list is used: human moves are membership tested against it to determine whether or not they are valid (thus fast access is required), and duplicates are not allowed (so that accurate counts may be taken). The sizes of the resulting sets determine which player takes a turn, and whether or not the game is over.

The Board class is the next most important program structure. The Board is very carefully designed not to allow the possibly of invalid moves, and its API is quite detailed. Internally it simply wraps around a vector of vectors, which represents the two-dimensional board. The choice of this data structure was motivated by the need for fast access time, constant size, and determinate data locations (i.e. using integer index values). Plus the vector structure, being a simple array, lends itself conceptually to representing the game board.

The History class, like the board, was designed with interface in mind. It is a wrapper around a deque object. Elements are inserted and removed from the rear of the queue as moves are taken or undone. When a History object is used to instantiate a Board to a given game state, the history object is read form front to back. Thus two kinds of behaviors are required for the History class to function properly: rear inserts and deletes and forward iteration. Both deques and lists provide this functionality; the deque was adopted because lists have additional functionality (mid-list access, insert, and deletion) that is not desired from our History class.

The final structure of note is the AI class, which is an object wrapper around a number of black-box algorithms, described in a moment. Internally it uses vectors to calculate its own move lists, taking advantage of their constant access time and numeric indexing properties (which are required by some of the algorithms) to calculate computer moves.

It should also be noted that an overarching error handling strategy is used throughout the program. A hierarchy of self-identifying C++ exceptions was defined early on during the development of the game, and they are thrown whenever a specific class encounters an irrecoverable error. All of the errors are caught centrally by Ncurses and dumped to the screen. This technique greatly aided in the debugging process.
Basic AI Strategies

There are five different but generally related strategies that the Ollehto AI employs to calculate computer moves. Before any move calculations begin the AI first computes a list of possible moves using the standard interface of the Board class as described in the API. Then, based on the strength level it was passed at construction time, it uses one of the following algorithms. It is of interest to note that the first four AI levels correspond to the natural progression of a new player learning Reversi strategy:

Level 1: Randomly Selected Move
This is the simplest algorithm and the most natural given that the AI has already calculated an initial list of possible moves. The AI simply selects one the possible moves at random and returns it to the Game class.

Level 2: Greedy Algorithm
The second strategy follows naturally from the first. The primary objective of Ollehto is for a player to maximize the number of moves he or she has on the game board, thus a greedy algorithm is employed which chooses the move that maximizes the number of tokens the player will capture and this move is returned to the Game class.

Level 3: Weighted Greedy Algorithm
The simple greedy algorithm (as a hypothetical player learning Reversi will soon discover) does not work well in many cases. There are moves that are naturally more favorable than others: namely edges and corners. Correspondingly, the third AI algorithm assigns positive bonuses to moves that seize either corner or edge locations, these being in general the best locations on the board.

Level 4: Complex Weighted Greedy Algorithm
The fourth AI algorithm is an expansion of the third. The addition takes into account the fact that not all favorable locations are equally favorable (for example edge locations are more valuable the closer they are to the corners) and the fact that some locations are actually poor choices (such as locations immediately adjacent to corners). The algorithm weights these additional factors accordingly and makes its decision greedily based on overall score.

Level 5: Deep Blue Jr. (Mix-Max Depth Search)
The final algorithm takes all of the weighting mechanisms of the fourth algorithm into account and applies them to minimizing the effectives of the opponent’s next move. To do this the algorithm must calculate all the possible moves one level beyond each of its own valid moves. After it has calculated these moves it then chooses its own move based on the one that leads to the opponent’s minimum score. This technique is known as min-max searching.³

³ http://www.generation5.org/content/2002/game02.asp
⁴ http://www.site-constructor.com/othello/
Performance Issues

Performance was not generally of great concern during this project, with this exception of AI development, and in this case performance is more related to algorithmic efficiency than the choice of any one data structure. Overall the program performs very well; there is no noticeable lag, and a complete game can be played between two computer opponents of the highest level in less than one half of one second, including display updates.\(^5\) This, to us, is more than acceptable.

There are two locations in the game where other classes of data structures could have been employed that may have had an impact on the performance of the game: the Board class (where a map may have been used instead of a vector) and the AI (similarly). Generally maps are employed in cases where duplicate entries must be avoided, non-numeric key values are desired, or space is of primary concern. Since the Reversi game board is only 64 squares total, and our board coordinates are numeric, none of these situations applied. Additionally, vectors have an \(O(c)\) access time, whereas maps have an \(O(\log n)\) access time (since they use trees as the underlying data structure). Therefore, we can comfortably say using maps would not have improved any of our relevant performance metrics. The same is true of the AI class, where speed is much more important than spatial efficiently.

Conclusions

In every aspect Ollehto can be considered a success. All of the Team’s key objectives were met: we used Ncurses to present an attractive interface, implemented a highly-structured, and to our knowledge, bug-free game implementation that takes advantage of the methodology of object-oriented programming. Our AI algorithms run very quickly, and the highest-level AI can only be beaten by our Team Reversi guru (so far). The project was coordinated smoothly, and delivered an impressive product on schedule. In the process of developing Ollehto we learned a great deal about the Ncurses library, the principles of object-oriented development, the underlying data structures that power the game, and how to organize our efforts as a group.

\(^5\) To allow human players to see the result of their move before the computer’s move is resolved, a delay was added between the displaying the human’s move and the computer’s.
Biographies

Travis Clark is a Computer Scientist (Computer Engineering convert) hailing from Salina "Middle-of-Nowhere" KS. His deep love of text-based program interfaces led him to learn the ways of Ncurses and the expanded interface horizons it opened to him. He currently resides in the Fitzpatrick Engineering Computing Facility.

Patrick Davis is a junior Computer Science major who lives in St. Edward's Hall. He is on Notre Dame's #21-ranked varsity swimming team, specializing in distance freestyle events. Interests include reading existential philosophy, knowing the movie "Best In Show" by heart, and studying the intricate minutiae of NCAA compliance regulations.

Lance Gallop is a fifth-year senior majoring in Computer Science, Philosophy, and Theology. In his spare time he writes beatnik poetry, listens to Icelandic music, and rants about Ted Danson, who really did deserve an Oscar for the made-for-TV version of *Gulliver's Travels*. It was a magnificent performance.

Matt Tanner is your average junior CS student. In addition to drawing pictures, working on the GUI and testing the project, he also wastes away his time in front of a television, dominating every video game to come across his path. Matt Tanner: CS student, project designer, video game legacy.