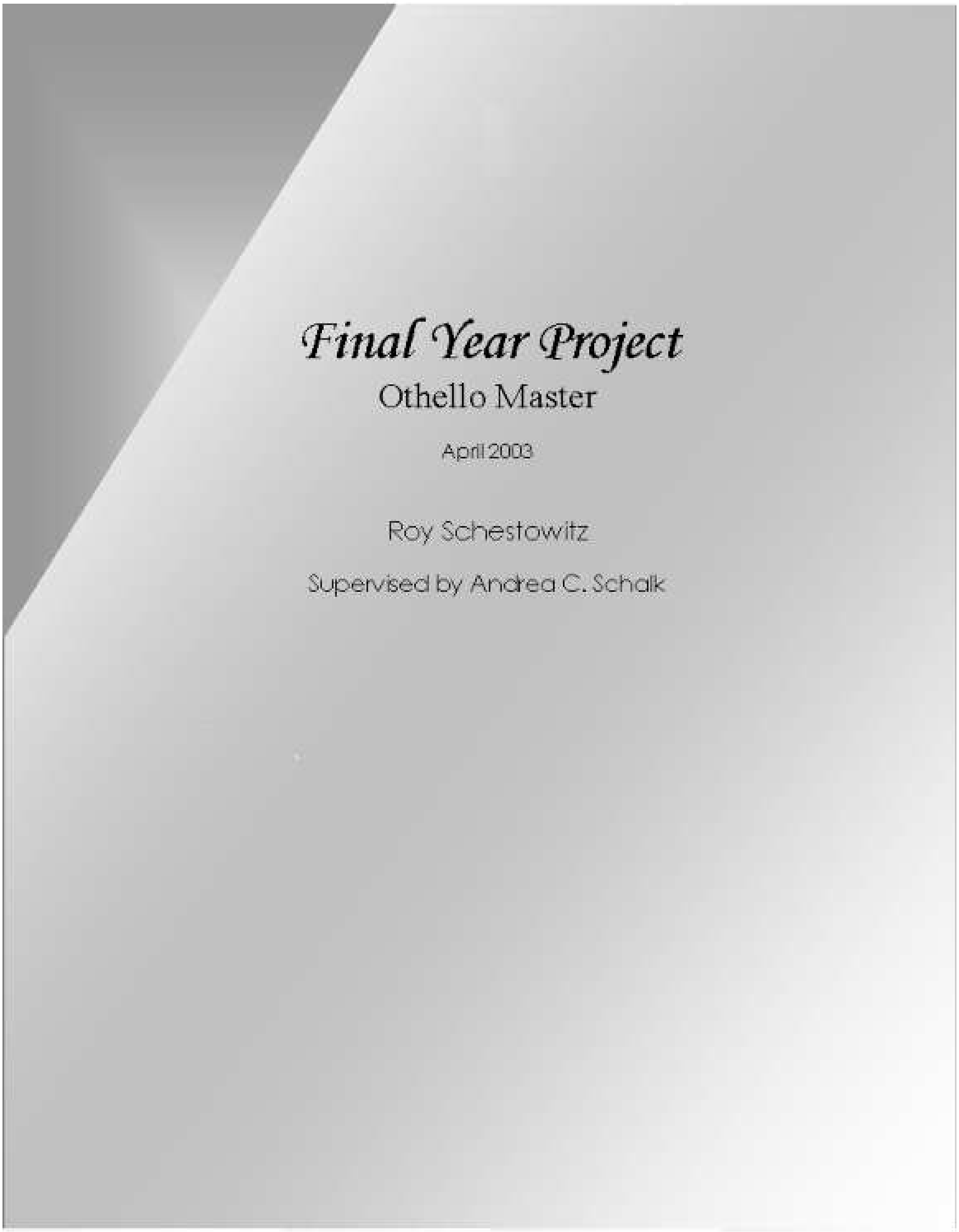


Final Year Project

Othello Master

April 2003

Roy Schestowitz

Supervised by Andrea C. Schalk

Final Project Report - Othello Master

Author: Roy Schestowitz / Supervisor: Dr. Andrea Schalk

April 24th, 2003

Abstract

Othello Master (project number 500) utilises the principles of game theory and implements them in an imperative programming language. Its main strength is its performance against other Othello-playing applications and it is can be distinguished from typical such applications in the following aspects.

The package is built to be used as a powerful tool that can produce and analyse occurrences of interest in the games played or simulated. It can play a series of games independently and summarise the results of these in files. Amongst some of the more interesting features is the inclusion of opening libraries, customised move computation and different difficulty levels, each of which corresponds to a different approach of generating a valid move in the game.

This report presents the development life-cycle of the project and derives further conclusions and contributions with respect to the initial plans.

Keywords: Game Theory, minimax, Othello, Reversi, simulation, OpenGL.

Contents

1 Introduction

Othello Master is an implementation of the game Othello (also known as Reversi) using GLUT[1]. The application will rely on principles of Game Theory, namely alpha-beta search and the minimax algorithm which will be explained later. Using these principles, the application is able to play the game at a very high level of competence, as well as provide a generic platform for games to be played on.

Game-playing programs have existed from the early years of computer science and have improved in their ability very rapidly. Game Theory was developed and established in the 1940's, making a contribution to the world of mathematics. Some of the more prominent impacts that the topic has had is reflected in the field of economy where prediction of trends and human behaviour is fundamental[2]. Programs that play well, on the other hand, served people's curiosity and set new challenges in the scene of world-class Chess[3]. Artificial Intelligence is now said to be tightly associated with such programs that can "think". Computers, however, use brute-force to simulate human thinking, whereas real neural functions use associations, visual memory, etc. It would be safe to say that computers have not yet reached this goal of being able to think. Brute-force may work for a game of Chess, but see Go for the very contrary.

Othello can be considered an instance of the games for which brute-force is of real use, however, as explained later, there is a snag to this. *Othello Master* will make use of the classical approaches to carry out its process of playing. It will traverse the game tree (defined later) and inspect various properties systematically. There is no real thinking involved and the only rational decisions made are in the mind of the programmer.

Purpose

The project will attempt to go beyond the basic task of playing well. Collecting some results and bringing about some conclusions regarding Othello and game-playing programs is set as an objective. Some of the data and lessons learned will be recorded and the results summarised and made available on-line.

It is by no means the case that this is a first attempt or a genuine implementation of a game-playing engine. In fact, game theory and its principles are heavily used not only in the domain of board games, but also in a great number of predictive systems that are a common research and development area.

Aims

While the formally set goals were specified in the Project Proposal and Plan, it is of more use generalising the expectations and rephrasing them with respect to the up-to-date application state. Confer Appendix B to view the initial goals.

The main aim of any game-playing program is, first and foremost, to play well. To expand this proposition, let us say that such a program needs to be able to determine winning strategies (see later) and react within a reasonable

---

[1] Provides scene rendering and user interaction facilities.

[2] The importance of Game Theory was prominent enough to grant John Nash, who can be considered to be the father of the theory, a Nobel prize for Economy.

[3] According to a face-off that took place in January 2003, even the top Grandmaster, Gary Kasparov, is now unable to defeat IBM's Deep Blue overall.

amount of time[4]. This problem and the solutions we have will be further explained later.

The second and very important aim is to convert the program into a flexible tool that allows its clients to investigate the behaviour of the different algorithms and collect results from a series of games rather than iterating manually.

By present day, not only were all requirements and aims realised, but some extensions were put in place too. Such extension can be sub-categorised into these which sit on top of existing element and these which are rather independent and offer extra functionality to *Othello Master* as a demonstration and analysis tool.

Milestones

The detailed milestones are important for the understanding of the development process, which is described later in this document. Please confer Appendix I for better familiarity with what was to be achieved.

Achievements

The contributions of such a project to future projects or derivation of conclusions are mainly ones that root in the experience acquired and the results collected from the simulation phase (see the section on Simulation). There are different ways of looking at these contributions and breaking them down into distinct parts. The following suggests one way of describing the contributions that have been made.

Firstly, the project is intended to observe the effectiveness of a combined series of tests in the evaluation function (as defined later). The evaluation function is broken down into numerous steps and each of these steps (or a combination of several of them) may be reasoned about by running the application to their exclusion[5]. In other words, it may be discovered that the program will suffer due to the weaknesses in the evaluation function. These weaknesses will be assumed to incur due to an exclusion of one or more evaluation steps, e.g. score evaluation, as seen later.

Secondly, assignment of values in the process of evaluation is of great importance. One of the hardest tasks when constructing a game-playing program is finding the appropriate values to be assigned to game states. This can make the most significant impact on the moves that are believed to be advantageous and are therefore chosen. Needless to mention, these values are game-specific, but nevertheless, they can be generalised for any application that handles this game. The choice of the values usually involves a lot of testing and fine-tuning. By recording the values that were claimed to be reasonably advantageous, we can provide guidance for the next person who wishes to design an Othello-playing engine.

More on the contributions from a prospective point of view will be discussed in section 8. Results and conclusions will also summarise the contributions which were practically made.

---

[4] In a large game as the one we are concerned with, the task of plausibly analysing the whole depth of the game is in the complexity class EXPTIME (includes games like Chess and is the superset of all other complexity classes) and is therefore not quite feasible in reality. Solving any such game (or determining its value as explained later) requires a vast amount of calculation which is far from the reach of any contemporary powerful machine.

[5] The elimination of some of these tests will appear to have different outcomes depending on the opponent.

Report Overview

This document will explain some of the fundamental principles that are essential for the construction of the application and will go on to the description of the development process[6]. An Analysis follows and a summary, most of which will comprise conclusions and suggestions, will be the closing segment. For completeness and intrinsic knowledge, project documentation, code and peripheral information will be appended.

Figure 1 below presents an instance of the application being run. The game board is embedded in a much larger scene[7] and the view is dependent on the user's choice. Necessary figures and more interactive interface are presented once an overview on the game board is triggered to be activated. Such overview is also invoked once the mouse cursor has moved.

Figure 1: *Othello Master* instance

Acknowledgments

My thanks to my project supervisor and Third Year tutor, Dr. Andrea Schalk, for offering advice and supporting my decisions from the very start. Dr. Graham Gough has provided a simple hashing implementation which was an essential starting point for the opening libraries implementation and Nate Robins is responsible for much of GLUT, without which the project would not have exhibited some of the more startling graphical features.

The Web space from which releases, information and documentation are available is hosted on the domain owned by Daniel Sorogon. The project would have primarily comprised the local source code and documentation if it were not for the existense of the address below:

http://www.danielsorogon.com/Webmaster/Projects/OM.html

Some of my essential knowledge of how to organise, manage and document source code would have lacked precision if it were not for the advice and guidance from Dr. Jim Garside throughout summer 2002[8]. Furthermore,

---

[6] The conventional software development process along with some other unique parts constitute the technical aspect of the project.

[7] A great amount of time and effort was put into studying techniques such as texture mapping to construct the whole scene and the very small details within it.

[8] A different project, Komodo Manchester Debugger, involved the development of a much larger package at the time.

command-line processing techniques were acquired from Charlie Brej, a current MPhil student, at that same period of time.

Last but by no means least, I would like to thank my employers for showing full understanding and allowing me to labour on my project since October 2002.

2 Background Theory

2.1 Othello

The game Othello is considered to be a game that requires comprehensive experience to be mastered. It is also said that Othello "*takes a minute to learn and a lifetime to master*". What makes it particularly interesting is the difficulty in telling which player is in a point of advantage until the late stages of the game. This is, in principle, where lack of skill and experience take their toll.

The following is a brief summary of the rules:

> At the beginning of the game, four stones[9] are already placed at the centre of an 8x8 standard game board. Two stones of each one of the players are placed diagonally and, by convention, the player to make the first move is of white colour, whereas the other is of black colour. The stones are all two-sided and flipping them changes their colour to the opponent's colour. A legal move is such that it reverses one or more stones of the opponent's colour. To reverse a stone, a player places one of his/her stones in such a way so that it surrounds a sequence of one or more of the opponent's stones. Such a sequence of opponent's stones must be ending in a board slot that is occupied by the reversing player's stone. All straight lines are applicable in such a reversal: horizontal, vertical or diagonal. If no reversal is possible, the turn is passed to the opponent. The game is finished when neither player has any legal moves left. Usually, by this point the board is completely full. Whoever has the most stones placed on the board at that point wins the game.

The full and elaborated rules are available at:

> **http://home.nc.rr.com/othello/rules/**

2.2 Game Trees

A game where decisions are made by a set of players can be described as a tree. At the start, there is one single state which is the opening state; no choice has yet been made by this point. That state corresponds to the initial board layout and in the case of Othello, the state is one where 4 stones are placed in the centre of the board. As the game progresses, a set of players can pull the state of the board in different directions, meaning that the state of the board at any point will depend on which *paths* in the tree have been picked up by the players in the game. In most board games, any state of the board is changed upon a placement or displacement of a game piece. A

---

[9] The notion of a stone is analogous to one of a game piece in our circumstance.

node in the tree, depicted below as a circle, presents one distinct board state that can be reached in one distinct way from the root. In other words, only one path can lead to a given node, hence forming a tree and not a graph. The nodes are the decision point in the game and combined nodes, connected by the relevant edges, form a path.

The game is played until a certain depth is reached. In the game of Othello that depth is not a fixed number[10].

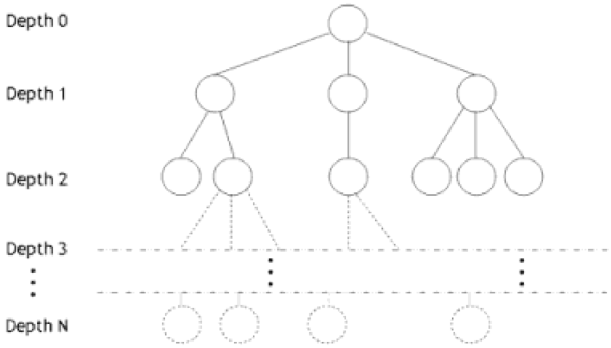


Figure 2: Game tree

As explained before, most large games such as Othello are too complex to be fully explored and hence solved[11]. With respect to game trees, what this means is that the game tree of the game Othello is too large to be fully traversed.

2.3 Strategies

A strategy defines a choice of a path in the game tree. It can be seen as the tendency adopted by one player, e.g. choosing to stand when having a sum of 20 in Blackjack. The focus of game-playing programs development is the discovery of good strategies.

2.4 Evaluation Function

For the time being, let us just think of this as any function that takes a state of a game as an input and returns some value that has been assigned to it.

In order to look ahead in the game, one needs to choose an algorithm that will account for the usefulness of a move from the perspective of each player in the game. Another way of expressing this notion of usefulness is to talk about the *state evaluation* [12]of a move. A simple example would be to describe a winning move as one that carries a high value.

---

[10] Up to 60 moves can be played, but the game may end early if a player is left with no stones on the board. That, in fact, is a state of deadly embrace, frequently known as a deadlock.

[11] Solving a games means that the final outcome of the game (in Othello this can be a win, a loss or a draw) is **known** rather than speculated. Having solved a game, a choice of a good path in the tree is trivial.

[12] The term evaluation, in simpler terms, is a perceived assignment of *value* that can be desirable or undesirable.

In order to find out what the usefulness is, we define an *evaluation function* which, given a state in the game, will return some information regarding that state. See the section named Game Engine to get familiar with the evaluation function that is used in *Othello Master*.

The ideal would be to use a very powerful computer. In such a case evaluation functions are not necessary. Instead, searching the trees to leaf level reveals who can force a win and by which positions (or strategies) this can be achieved. In some games, no such strategies exist, e.g. Paper-Stone-Scissors. Since in most games that we are concerned with searching to this level is infeasible, limiting the search depth is a reasonable solution. The evaluation function then provides a value for a position instead.

2.5 The minimax algorithm

The following explanation is phrased in simple and general terms that are intended for readers with a fair knowledge in programming, but none in the field of Game Theory. To gain better knowledge that is expressed in mathematical and technical terms, please consult the corresponding references.

The principles described here and onwards are strictly concerned with games where 2 players are involved and the information is complete [13]. This means that the game position is always clearly known to both players. Also, due to the existence of only 2 players, from one player's point of view, any move made by his/her opponent has a direct influence on that one player[14].

In a game tree, we can incorporate the results obtained from the evaluation function. In other words, we can assign to each node of the tree the evaluation's returned value once it has been calculated. Since these game trees are usually enormous in size, the traversal to is limited to a certain depth. Trees generated can be depicted similarly to the one in Figure 3 below

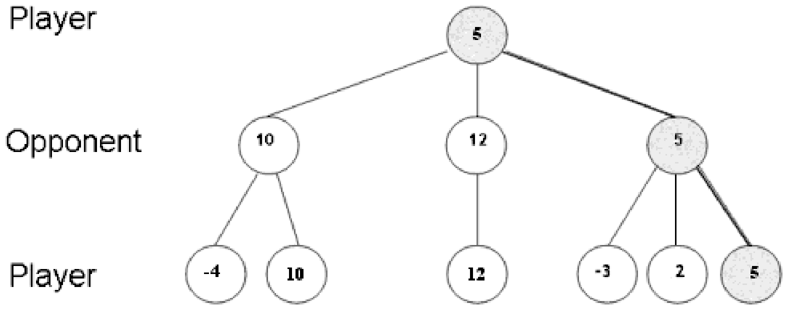


Figure 3: Minimax algorithm

Knowing that each of the players wishes to maximise his/her pay-off at any given opportunity, we can predict the path that will be followed down the tree. In Figure 3, it is the case that high values are advantageous to Player 1, whereas Player 2 is after small values[15]. That means that Player 1 will strive to make a maximal choice, whereas

---

[13] Complete information implies no elements of chance, e.g. dice being thrown and no missing knowledge such as a set of cards that the opponent holds away from sight.

[14] In fact, in *Othello Master* the two are complementary so a good move made by Player 1 always disadvantages Player 2 and vice versa.

[15] It is worth mentioning that this choice is arbitrary. We could default to high values being beneficial to Player 2.

Player 2 will settle for the minimal choice. Given these concepts, we finally see why the path picked is sensible. We choose the maximum, minimum, again maximum and so forth at each level of the tree. The name *minimax* has been extracted from this very intriguing behavior. For a step-by-step illustration of the minimax algorithms, see Appendix H.

A substantial step towards our goal has been taken once the minimax algorithm has been realised. Various intricate extensions make it even more efficient and sophisticated, but the only one worth mentioning is alpha-beta pruning which helps minimax save computational effort.

2.6 Alpha-beta pruning

Alpha-beta pruning relies on the algorithm presented above. It is an extension that allows the process carried out by the minimax algorithms to be carried more wisely and explore more important paths in the game tree. It is based on the idea that branches in the game tree should no longer be explored if they offer a solution that is no better than what has already been found. The *pruning* of the tree means that we can leave out parts of the tree without needing to explore them. They simply do not offer any better choices than the ones already discovered. In order to allow for pruning to take place, the values of choices already explored need to be recorded as a range between two numbers. Alpha and beta were the original names for the variables holding these values and defining that range. The range indicates which values are still sought and it is modified as the tree is traversed. Making good traversal order allows for more pruning, but is a very difficult task. See the references for more details on the issues and implementation of alpha-beta pruning.

2.7 Game Value

The game value indicates what the pay-off for each player is when the game is orchestrated by optimal strategies. It can indicate if one player can ensure at most a win, a draw or a loss. Within the population of *large games*, we usually do not know which of these categories a game falls into because the game tree is extraordinarily wide. Othello is one of these games.

3 Design

Design of the application was broken down into two distinct stages. Firstly, there was a concern for the generic structure of the system and the interaction inside that system. Division into several logical units should allow clearer distinction between different functions and different global variables.

Once a reasonably efficient and realistic picture has been made concrete, specification of the operations and their algorithmic structure, expressed as pseudo-code, can finally be put together.

3.1 Higher-level Design

The system has been divided into the following units, where each unit corresponds to a C source file and its header. An alternative way to look at these elements would involve the notion of compilation units.

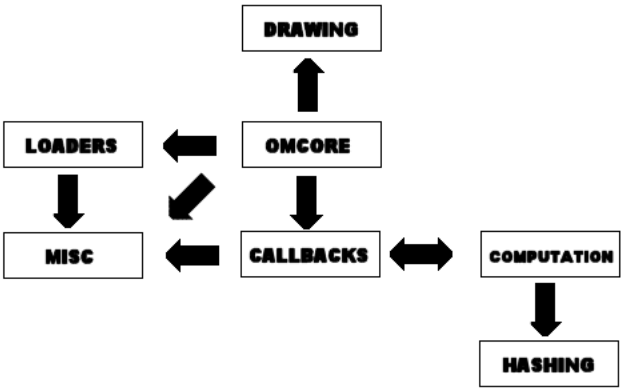


Figure 4: Structure of *Othello Master*

The seven different units, few of which have strong dependencies upon others, can be expressed as follows

- Omcore

  Uses: Loaders, Drawing, Callbacks, Misc.

  Purpose: Providing the main procedures and application calls. This includes processing command-line arguments, invoking OpenGL, initialising the application state, assigning callback functions and constructing the game menus.

- Callbacks

  Uses: Computation, Misc.

  Purpose: Specifying actions to be taken once menu items are selected, scheduled tasks need to be invoked, keyboard or mouse events occur or a valid move in the game needs to be computed. Functions within this unit are not called explicitly. They are rather a result of service registration. Such registration is carried out in Omcore.

- Computation

  Uses: Misc, Hashing.

  Purpose: Encapsulating the functions that will be used to analyse board states and generate moves on behalf of the non-human player/s. Generation of such moves will include move databases, opening libraries, definition of pre-processed factorisation values and randomisation. These techniques and their implementation will be discussed in greater depth later.

- Drawing

  Uses: None.

  Purpose: Drawing all the scene elements. This unit will be called perpetually to accommodate the application frame with the appropriate objects, the most important of which being the board. This unit will create objects in 3D space as well as orthogonally, namely the text and annotation that need to be displayed on the screen. This unit is further sub-divided into the code which handles orthogonal rendering and the code which handles perspective rendering. It contains many nested functions to make the rendering more readable and extensible.

- Hashing

  Uses: External libraries only.

  Purpose: Allowing for opening libraries to be stored and retrieved from very quickly[16]. It is an ad-hoc facility that comprises the functions which are invoked by the program only. Small redundancy only exists where the program's extensions may require it. This is the only unit that includes segments of code (approximately 200 lines) which were originally exported (see Acknowledgments).

- Misc

  Uses: None.

  Purpose: Gathering a collection of Othello-specific functions that are vital for the application. Such functions would be of little use had the application been converted into one that plays Chess or Checkers amongst other games. That is the main reason for the separation of this somewhat mixed unit.

- Loaders

  Uses: Misc.

  Purpose: Providing all the game-specific I/O functions. Originally, this unit comprised some particular file loading functions. As the application expanded, more file-handling procedures were assigned to this unit.

3.2 Lower-level Design

This stage comprised composition of functions in pseudo-code and natural language. The identified functions, which would ultimately allow a game of Othello to run and for the computer to generate a move, were now put within individual files. These segments of expressions and algorithmic concepts were to be later translated into C code.

A full pseudo-code example from the project is available under Appendix G.

4 Implementation

Overview

Coding of the required application was in the imperative language C. Object-oriented languages were not the most convenient choice due to the nature of the given OpenGL libraries. The Makefile was constructed by Mr. Toby Howard and was made available to any departmental client of the OpenGL libraries. The destination platform was Linux. Attempted porting to Windows was obstructed due to the absence of the corresponding OpenGL libraries in the department.

Some specific aspects of the implementation are worth mentioning more than others. While some of the functionality coded may be either trivial or too general to be argued about, there are certain points that not only are less trivial, but also provide a good overview on the operation of this particular game-playing program. It would therefore be wiser choice to focus on the operations and procedures associated with the game engine. Also, some of the

---

[16] Time complexity is O(1) instead of O(n) if a linear list of library entries is used.

related key issues that need to be addressed in order to support this engine will be covered. This section illustrates some of the implementation issues of the supporting components and the subsequent section explains some of the playing-engine components and properties. Inevitably, summarising the full structure and game features of the package is beyond the scope of these report. In particular, some of the low and intermediate level implementation issues will be left out. Instead, assorted issues will be specified and analysed. A full picture can only be obtained by reading through the enclosed source code.

4.1 Statistics

One of the most important aspects of the implementation was to allow for logging of automated games[17]. It is essential that the program can be easily invoked and customised not only via the menu entries, but also from a shell using parameter passing. To invoke *Othello Master* as required, batch files were used. The batch files conduct a series of specific games and the results of these games are summarised in one single file (or more where appropriate). In some cases, logs are generated for each individual game as well. A segment of a sample batch file is available in Appendix C.

Summary for a series of games is assembled in what is called a *report*. A report holds information on the results of one or more games and, as each further game is played, its result is appended to the given report file. For quick analysis, the score gap is recorded at the rightmost column and that alone should suffice when gathering and analysing the result as the section on simulation illustrates. A truncated report file is available in Appendix D.

For various purposes, such as archiving games and reviewing them, an option for maintaining log files has been included. The users can invoke log file activation at any point throughout the game using the appropriate menu entry or request the application to retain a log under a specific file name (see on-line manual). A game can be fully reconstructed using a log file and, on top of that, it offers some presentation means that ease the understanding of the game. A partial log file sample has been put in Appendix E.

4.2 State-based Facilities

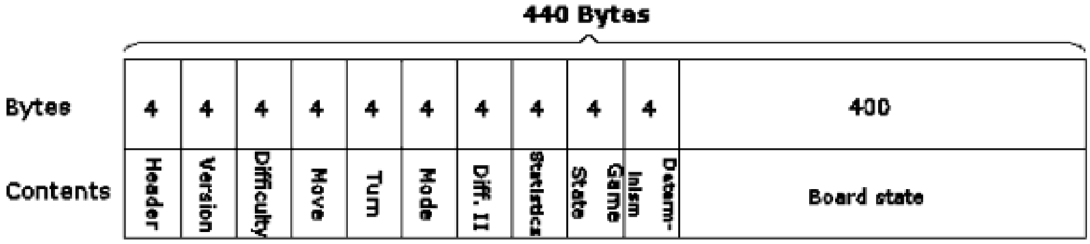
Representation of a game state is necessary for various functionalities. When talking about a complete representation of some point in the game, we are concerned with the ability to store some relevant objects (or in our case, global and local variables) so that we can retrieve them later. The importance of such retrieval is that it allows us to alter the state of the game and reproduce an older one that is, in fact, also a valid one.

Storage method of such objects may vary in practice. One possibility is storing a program state in physical files or some alternative form of I/O device. The other possibility is to store it in some newly allocated memory where the obvious problem is volatility. We may not be able to regain access to that program state data once the program's run has ended.

In *Othello Master*, two fundamental functionalities use the above principles. The first of these is the save/load mechanism and the second one is the undo function which is mainly an extension of the first one. Given in Figure 5 is the size and structure of a data object representing a game state.

---

[17] Games where no human player is involved. This means that no outside stimuli are required.



440 Bytes

Bytes	4	4	4	4	4	4	4	4	4	4	400
Contents	Header	Version	Difficulty	Move	Turn	Mode	Diff. II	Statistics	Game State	Determinism	Board state

Figure 5: State object layout

it can be seen that much of the space is used to store the state of the game board. A few slots are used to store information that is related to the game itself, whereas the rest retain some information about the program and the mode that it is in. For example, <turn> indicates whose turn it is to be taken in the game, yet <game mode> clearly refers to information that is irrelevant to it.

Load and save

To simplify and hide I/O operations and low-level procedures from the user, an intermediate level of interaction, that is, an interface has been incorporated into the application. The user is given 10 statically fixed slots into which games can be saved and loaded from. These slots are in fact an abstraction of fixed filenames that uniquely identify a slot. Lower-level functionality is still available, just in case the user may wish to use it. It is the case that the user can manage to load any file in command-line mode, given that the file contents strictly corresponds to the above file structure.

While a facility such as this is highly common in any commercial game, it is of very little use to a user or a client of *Othello Master* that wishes to handle it as a statistics gathering tool (see more later). Nevertheless, the state-based facilities do not neglect the important statistics generation facilities. The flexibility of use, as it is described above, would expand the scope of operation, e.g. staged simulation or mid-game simulation[18].

Undo

Implementation of undo was made simple due to the existence of the above functionality. As the program runs and the game progresses, a trail of packets of the above format are retained and overwritten repeatedly. A user's request for *undo* will simply require a transition into a previously saved state. An *undo stack*[19] of size N would require proportionally more space, namely N times the current space to the benefit of several consecutive undo calls. This is one of the possible extensions to be considered where necessary. *Redo*[20] is another extension to bear in mind, but it appears somewhat futile or redundant in any game-playing program.

---

[18] An instance of this would be a simulation that spans several consecutive days or one that aims to analyse the behavior taking place only at the end of the game.

[19] One that saves multiple states and stores them in a stack-based data-structure.

[20] *Undo* refers to aborting a previous action, whereas *redo* reverts this abortion.

4.3 Board Editing

As in most advanced game-playing applications, the ability to manually alter the layout of the board has been fully implemented. The inclusion of such a feature caters for the flexibility that is needed if, let us say, one player wishes to give the opponent a corner stone. Such an action often takes place in Othello games if one player is significantly more experienced than another.

Figure 6 illustrates the scenario above as viewed in *Othello Master*

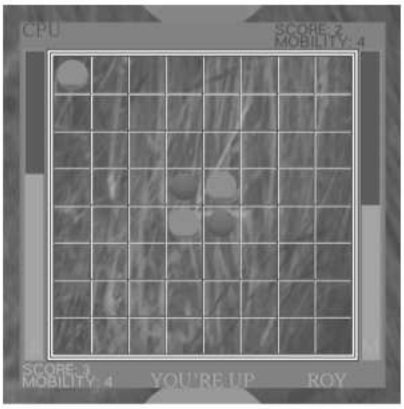


Figure 6: Board editing

The procedure of editing the game board is quite simple. The user needs to enable the mode of editing, perform all the required modifications i.e. place or remove stones and then call for the game to resume.

4.4 Opening Libraries

Opening libraries are used to save computation effort while the game is played. Instead of generating a move in the usual way, we can simply use a size limited library that indicates which moves to take at some distinct points in the game. The name *opening* library is due to the limitation of the amount of information that we can store in a library as such. In the game Othello there will be nearly 3^{64} possible board states[21]. Opening libraries are based on off-line computation of moves that are conventionally preferable. The implementation of these in *Othello Master* is as follows:

> *Othello Master* extracts libraries which reside in files and interprets them as binary inputs. Once the opening libraries are enabled, the contents of the library chosen is fed into the newly allocated hash table. When the hash table is fully initialised, the program will constantly test for the existence of a board state in the hash table. In other words, it will be in search of a matching entry every time a move is to be computed. At a later point in the game, opening libraries are automatically disabled[22]. If a match is found, the move will be carried out according to the entry referred to from the hash entry. If not, a move will be computed in the normal way, i.e. traversal and evaluation.

---

[21] That, of course, is a rough estimate taking into account all board layouts for 64 slots and 3 slot states, namely empty, White or Black.

[22] There is no point in searching for entries inside the hash table once the game has reached a high depth. Common scenarios usually occur at the very beginning, before the tree becomes wide.

The following diagram illustrates this graphically

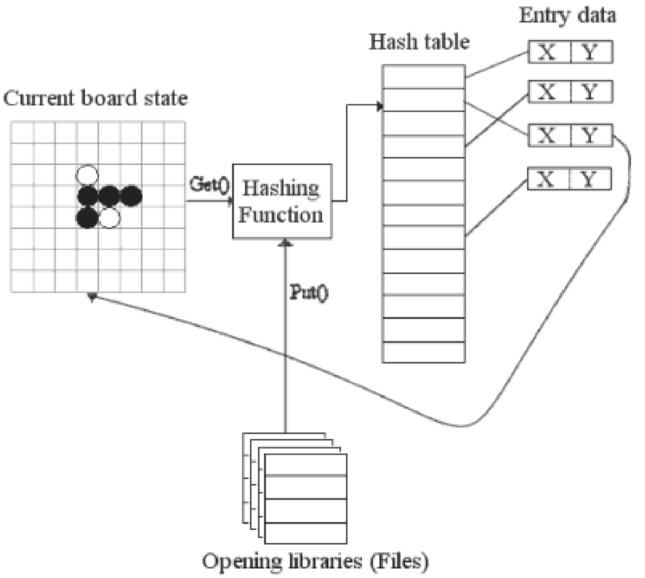


Figure 7: Opening libraries

In the above, the file objects are binary files that hold the libraries statically and can be changed to accommodate for alternative strategies[23]. Also, multiple libraries can be used in conjunction with one another, thus broadening the coverage and significance of the hash table. The hash table itself is defined to be of a statically defined size and uses open hashing (also known as separate chaining or bucket hashing). A more common use of hash tables is for retrieval of values for some states, especially where time is crucial and great depths need to be covered (c/f Chess). Each of the entries in the hash table is a pointer to an independent data field whose data comprises the coordinates indicating the most desirable move. These coordinate are passed on to update the state of the game board; this makes the (finite) loop much more evident.

4.5 Version Control

Othello Master 0.7.5 is the most recent fully-tested version. The project commenced with the creation of prototype versions and moved on to being classified by 3 separate figures, each of which has an intrinsic meaning. Older versions were retained and backed-up safely and simpler version of the project are still available which are more simplistic implementations. Management of versions was kept simple as this was a one-person project and no concurrent versioning system was in need of. Roughly 30 working revisions were kept safe during the eight

---

[23] Some game-playing engines exploit mobility quite heavily. To know more about such strategies, find out about the Othello engine called PilotH.

months of heavy development. To get a vague picture of progression, a sample change log can be viewed in Appendix F. Distinction between the different vesion is apparent in the source code and in a key file named <version>.

5 Game Engine

5.1 Computation

The term 'computation' within *Othello Master* refers to the generation of a move carried out as a result of the CPU taking several evaluation tests and then making a stone placement. The invocation of the **cpu_move()** function, which is the main move computation routine, can be carried out by either of the two players in version 0.5.6 and later. It only requests that a move, regardless of what type of move, *can* be made. This is always assured by a call to **check_deadlock()**[24] before the function call to **cpu_move()**.

The computation process comprises 3 distinct steps:

1. Initialisation[25]
2. Moves evaluation
3. Stone placement

Vis-a-vis, on top of these there is an *irrelevant* step that records the move made in a log file where applicable. More detailed description of the above steps follows.

5.1.1 Initialisation

Initialisation starts when the function's input player, for which the computation is requested, is analysed to make the function symmetric. The function records and sets up the whole procedure to later on deal with only the correct player.

The current applicable level of difficulty is analysed in order to allow the appropriate algorithm to be picked up. For more detail on levels of difficulty see the later section named Levels.

In difficulties other than 'Beginner', a randomisation process is then used to make the board pointer[26] cover all squares either horizontally first or vertically first without the user being able to know.

---

[24] This was chosen to be the function that will check if a game has reached a state of deadlock and react sensibly.

[25] Responsible for preparing all the variables for the subsequent moves evaluation and performing a few routine checks. This stage is more comprehensive than one might anticipate.

[26] Scanning of the board is performed serially and a board pointer is the current board slot being scanned.

5.1.2 Moves Evaluation

Moves[27] evaluation is carried out for 'Novice','Expert','Pre-master' and 'Master' levels. For each of these levels, there are two factors that determine the move that will be made, i.e. the apparently best move.

The first of these is the moves evaluated. It is important to bear in mind that there is an enormous number of moves to be considered if we look ahead to some considerable depth. What is tested and evaluated is always (to exclude the point where end of game is foreseen) a sample of moves which is believed to be a relevant one. Attempting to cover all possible choices to the point where the game is completed can be very cumbersome. In a game like Othello where up to 60 moves can be made in total and at a given point some 5 moves are typically available, we could hypothetically evaluate 5^{60} states. As this contradicts the goal of playing reasonably fast, we must use heuristics to evaluate a good sample of states only.

The second of these factors is the evaluation of a given board state. For more details on evaluation, see the later subsection.

5.1.3 Stone Placement

After all the necessary evaluations have been completed, the position of the stone to be placed is known. It is then the time to place the stone, flip the appropriate stones as a result of the placement, obtain the new score and mobility values, check if a deadlock has occurred and then pass the turn to the opponent.

5.2 Levels

Different levels of difficulty in *Othello Master* were implemented as different approaches of computing a move. The idea behind this choice was to allow the comparison of the performance (within this one application or outside it) of different algorithms. Comparing the performance of two so-called depth search engines with a different depth value could be a predictable and dull process. This application mainly attempts to show the advantages some approach has over another, as well as dealing with different depths and various evaluation methods.

5.2.1 Beginner

This difficulty is a very basic one and it uses a very simple algorithm. When set to be deterministic, is it meant to place a stone in the first available and legal block on the board. It can search the board either from the left to the right or from the top to the bottom when searching for such a block (so this depends on the direction of the board pointer, as explained above). When set to be non-deterministic, it attempts to place a stone in a random block, provided that the placement is a legal one.

5.2.2 Novice

Uses the simplest form of evaluation. For each possible placement, it inspects the value assigned by the evaluation function to the board. It will make a temporary placement of a stone for each of the placements available and find

---

[27] Plural form as many are evaluated, but only one is eventually chosen.

out which is believed to be the best one. Again, it can search the board either from the left to the right or from the top to the bottom when searching for the best placement. By doing so, it may make different moves when two placements produce a board to which the same value has been assigned by the evaluation function. When the computation is set to be non-deterministic, the behaviour is similar, but is affected by an offset which is described in the part titled Randomisation Keys.

5.2.3 Expert

Looks ahead 3 moves. It assumes that the opponent makes his/her best move in the single turn that is speculated here. For each of the available placement, a temporary board structure will hold the board state after the placement has been made, as well as two more placements that are believed to be ideal in the short term (it does not matter if the opponent did his/her worse as we have accounted for the worst case). Non-determinism and the form of search for available placements on the board is performed in the same way as described for ’Novice’.

Originally, a look-ahead of 5 and 7 moves was set for ’Expert’, but it did not perform as well as the current one of 3. For explanation of this, see the later section which analyses this behaviour.

5.2.4 Pre-Master

Looks ahead 14 moves. Works in the the same manner as ’Expert’ and is designated to demonstrate the influence of over-speculation.

Over-speculation is the case in which the game state that is accounted for is too far-fetched. Predicting 7 moves for each player in a game of Othello is without a doubt a bad idea. Trends in the game change much more rapidly than throughout nearly a quarter of the total game. The evaluation function will normally work its tests on a board that is inadequate in the sense that it only has a mere reflection on the provision of moves.

Choosing a depth in the range of 3 to 14 would have perhaps be more fruitful, but as mentioned above, such depths have been tested at early stages of the game-engine design and have proven to be weaker than that of depth 3, yet stronger than that of 14. It was therefore decided to stick with the depth of 3 and to assign it to the ’Expert’ level.

The above is to show us that proficiency is not proportional to the depth of exploration, but is rather about intelligent evaluation of some future game state. This exception is the end of an Othello game which means that a modified algorithm could benefit from descending to different depths at different points in the game. How to discover the right moments to explore the game tree further is the another big issue.

5.2.5 Master

Performs a full width search for the opponent. It takes into account all possible moves that the opponent can make and accumulates them sensibly. It is in some sense an enhancement of ’Expert’ (hence depth 3 at present state) where only one move that the opponent can make was taken into account.

The accumulation of the values assigned to each possible move that the opponent makes is controlled by preprocessor definitions. The moves for which the consequences seem preferable are assigned larger coefficients than those for which the evaluation function returns a small value, i.e. the consequences seem unwanted.

The following figure shows how the accumulation is performed when the significance of the Nth best move is factorised by 10^{N-1}

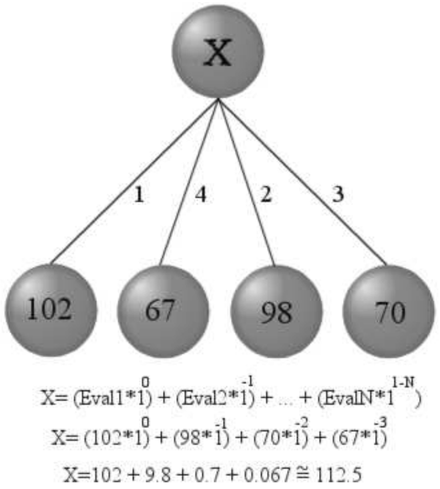


Figure 8: Master evaluation formula

The numbers on the edges indicate how good the node they lead to is and the calculation carried out illustrates what values are cascading up the tree.

5.3 Evaluation

Evaluation is a process comprising of several individual steps, each of which has its partial impact on the main evaluation. An accumulator is used to combine all of these separate steps and hold a final value.

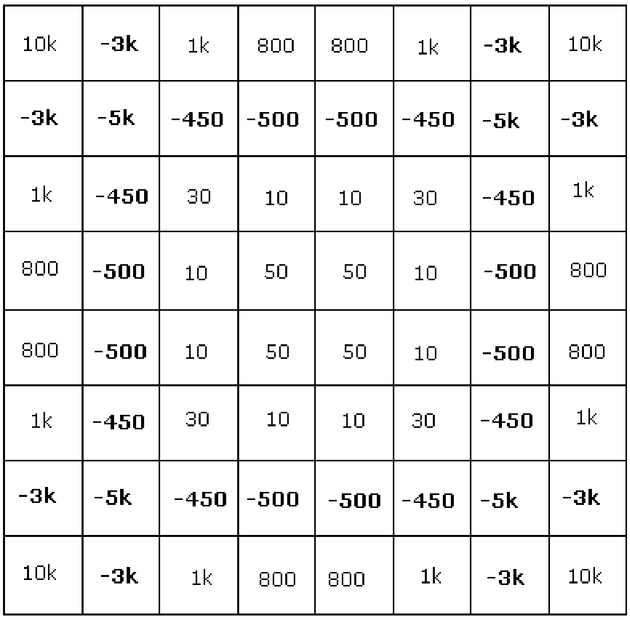
Each of these steps is controlled by pre-defined settings, as well as a run-time menu labeled 'customised computation' and command-line options.

The steps are as follows:

1. **Board Positions:**

   A value is assigned to each stone according to its position. The value of slots all on the board is associated with a pre-defined signed integer and that can be fine-tuned between each run to optimise the evaluation's faithfulness. By doing so, real improvements in the performance of the computation can be observed.

   The following figure presents the values that were made permanent in the header file once the fine-tuning process had been completed. The value in each of the board slots indicates the significance of holding that position.



10k	-3k	1k	800	800	1k	-3k	10k
-3k	-5k	-450	-500	-500	-450	-5k	-3k
1k	-450	30	10	10	30	-450	1k
800	-500	10	50	50	10	-500	800
800	-500	10	50	50	10	-500	800
1k	-450	30	10	10	30	-450	1k
-3k	-5k	-450	-500	-500	-450	-5k	-3k
10k	-3k	1k	800	800	1k	-3k	10k

Figure 9: Score accountability map

An important fact to point out is that throughout this stage, opponent stones placed in a specific positions on the board will increase the opponent's perceived evaluation. Therefore, opponent stones will be accountable in the sense that their occupation of a board position may decrease or increase the evaluation in a complementary manner, e.g. Player 1's evaluation will decrease once Player 2 has captured a corner.

2. **Mobility accountability**

   Leaving the opponent with a few legal moves to make leads to big advantage in Othello. This step takes into account the mobility of oneself and his/her opponent. The weight of this process is also pre-defined and should be set high prior to games against stronger Othello-playing engines, which exploit mobility to ensure a win. Mobility is discovered by searching for all possible moves. This is a rather expensive process, which justifies minimisation of the number of calls to it. Methodologies for speeding up this process (and revealing the minimal number of calls required for this procedure) should be left to the more detailed literature. Such methodologies typically bog down to bit-wise operations and implicit board representation.

3. **Score accountability**

   This refers to the number of stones each side has got placed on the board. Having many stones at the beginning of the game proves to be disadvantageous, whereas it is a big advantage towards the end of the game, bearing in mind that the winner is determined by the final number of stones. The score of both sides is taken into account, as well as the *move count* (the move count indicates how far the game has gone).

   In practice, this was implemented in *Othello Master* as follows:

   Stones are being counted for each one of the two player and the gap in score is then retained. That gap is then used in an equation where a factor of this component's significance is present, as well as the move count for the game. As the game progresses and the board is occupied by more and more stones, this step will have an linearly increasing impact on the overall evaluation. This satisfies the above arguments.

4. **Line accountability**

   Capturing a series of horizontal, vertical or diagonal stones leads to advantage in Othello since these are hard to be recaptured by the opponent. An incomplete series of stones where only one side or edge stone is missing is also a powerful element. Both of these are taken into consideration to form an assessment of structure. The weights of different forms of lines are all controlled by pre-defined values. Lines which are adjacent to the side of the board prove to be more useful than lines stretching across the centre. Straight lines are also defined to be more powerful than diagonal lines. Most importantly, complete lines (sequences of stones) are more desirable than incomplete ones since the latter can still be fully recaptured by the opponent.

   The following figures illustrates the division of lines and their corresponding value that is later multiplied by the weight of this whole step. Firstly, here is the break-down of the diagonal and vertical complete lines that can possibly form a sequence on the game board

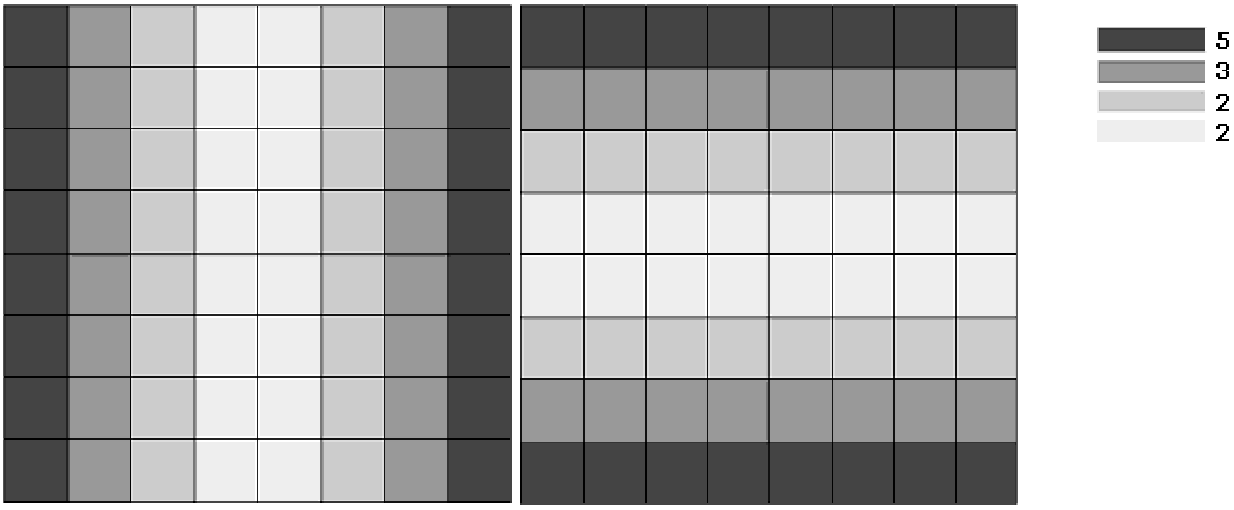


Figure 10: Straight lines accountability

Secondly, for diagonal layout of lines, here is a simple break-down that covers all the cases for complete lines formation

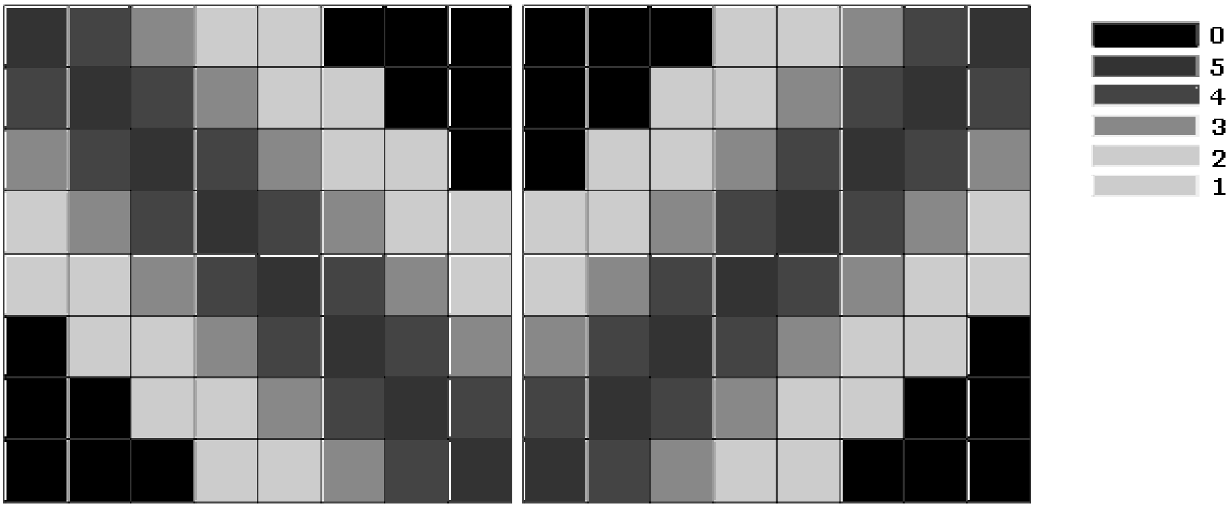


Figure 11: Diagonal lines accountability

Incomplete lines, i.e. ones that lack a single corner or side stone for completeness, are not illustrated in the figures. There are 8 cases which resemble the above figures and nearly parallelise them. For horizontal lines, there would be the case where a the leftmost stone is missing and the rightmost stone is missing. Similarly, for vertical complete lines and for each of the 8 possible lines there would be a case where the top stone is missing and a case where the bottom stone is missing. There would be 18 such cases for diagonal lines, 9 for each direction. To calculate the total number of distinct cases of incomplete lines, let us take symmetry arguments into account and say that vertical and horizontal layouts are analogous (board rotation satisfies this argument). The same argument should apply to diagonal lines as we can flip or mirror the layout of the board. There are two types of incomplete lines. These depend on the missing stone and will total at 8x2 for each of the two straight lines cases, i.e. 8 x 2 x 2 for all straight lines. For diagonal lines, the same arguments apply but there are 9 cases of diagonal lines, hence totaling at 9 x 2 x 2. Overall, we have 8 x 2 x 2 + 9 x 2 x 2 = 60 distinguishable incomplete lines. Their final values can be derived from the following table

Type	Position	Type Factor	Value	Total
Straight	Edge	5	4	20
Straight	Near Edge	5	3	15
Straight	Near Middle	5	2	10
Straight	Middle	5	2	10
Diagonal	Middle	4	5	20
Diagonal	Near Middle	4	4	16
Diagonal	Between Middle And Corner	4	3	12
Diagonal	Near Corner	4	2	8
Diagonal	Corner	4	1	4

Figure 12: Diagonal lines values

For full and precise definitions, confer computation.h in the auxiliary appendices.

5.4 Randomisation Keys

In certain points in the stage of simulation (see later) and as soon as we have seen the deterministic behaviour of all possible games, we should be interested in a more advanced behaviour that cannot be predicted or repeated continuously. Nonetheless, we want to hold on to the strength of the game-playing engine. Once non-determinism is enabled (or determinism disabled) in the program, each evaluation is subjected to an offset which is a random number within a given range. Normally this offset is pre-defined as a relatively low value and it expresses the offset in percentages, e.g. 5 implies that the computation is susceptible to a 5% offset at most. This allows the evaluations to return a different value each time and the behaviour of the game-playing engine can, thereafter, be different in each run, resulting in different board states and different game outcomes.

6 Results

6.1 Performance

Othello Master has been put in involvement within a small conducted tournament in which it played against 5 other freeware Othello-playing applications. Its performance was always better when the level of difficulty set for the its engine was Expert. This means that a more complex and more sophisticated approach did not necessarily surpass its classical predecessor. A sensible explanation for that would be the relative simplicity of the game Othello, the other possibility being that some approaches were not fine-tuned sufficiently. It would be valuable to also point out that the advanced concepts used in the difficulty 'Master' were not adopted from any existing game-playing program, but were a product of own intuition.

The following are the results of *Othello Master* playing against other applications. A total of 3 games for each combination of rivals took place and the figure indicates the number of times *Othello Master* won these 3 games.

Application/Othello Master	Othello Master - Expert	Othello Master - Master
Lagno	3	2
Palm Othello	3	1
Reversi for Windows	3	3
Qthello	2	2
Othello	3	3

Figure 13: Comparative tournament results

The sources for each of these applications are specified below:

- Lagno - *An Othello-playing program that is available on the departmental Linux machines.*
- Palm Othello - *available from www.freewarepalm.com*
- Qthello - *available from Cnet.com/download.com*
- Othello for Linux - *available from Cnet.com/download.com*
- Reversi for Windows - *available from Cnet.com/download.com*

6.2 Algorithm Analysis

An explanation for the above results is required. It has been claimed that the third most complex algorithm performs the best of all and to argue that this is a sensible outcome, we must look in greater depth into the behaviour of each of the more interesting difficulty levels. Arguments on the weakness of 'Pre-Master' have been stated at an earlier stage, but the following table should further clarify the problems and strengths of each approach (or level of difficulty)

	Expert	Pre-Master	Master
Pros	Simple Quick	Looks deep into game tree Simple	Efficient Good coverage for the unpredictable Fast
Cons	no account of opponent making choice that is undesirable [28]	Same as for Expert Heavy and slow Over-speculative	Accounts for many moves Higher memory consumption

Figure 14: Analysis summary

Look-ahead of 14 moves, as explained in brief in the section on levels, does not produce satisfactory results. Most people would consider that very baffling, but careful consideration of the game Othello should clear the doubts. This issue is closely related to the following observation that demands an answer. Why did a depth of just 3 suffice in our traversal of the game tree? The answer is closely related to the game in question. In the game Othello, as oppose to games like Checkers or Chess, a good move is less than obvious or foreseen. The game depends largely on patterns. A look-ahead of 6 or 8 moves, for example, will not reveal a key event such as a capturing of a piece or a check. When the program under consideration looked ahead to a level somewhere in between 3 and 14, no account was taken for key events. Very few of them even existed, except for corner stones occupation and low mobility alerts.

This does not yet fully explain why the simpler approach titled 'Expert' surpassed the more complex 'Master'. A sensible hypothesis may be that alternative (and worse) moves did not occur in practice and therefore did not aid the evaluation. As a matter of fact, they appeared to have made more harm than good.

7 Simulation

Othello Master was built to produce statistical results, allowing analyses to be carried out by the user more efficiently. This comprises and relies on the following components:

- Log generation
- Statistical gathering[29]
- Batch files

---

[29] Collective logs put in a single file to be used for analysis and statistics.

- Random seeds[30]
- Auto-exit

Given the above, it is possible to run the program repeatedly and record the information that is of use in the subsequent analysis. The program is invoked with different levels of difficulty being involved and allows the user to see the results produced. Since a level of difficulty corresponds to an algorithmic behaviour, we can gradually find out how different algorithms perform against one another, or even more interestingly, within some given population of algorithms (confer the field of game models and evolution). What we are expecting to see is that a complex algorithm that consumes more resources (memory space and CPU power) will obtain results that are superior to these of simpler algorithms. The results of a sequence of games are put together in a single file. An example of such file is available in Appendix D and will be explained later on. Once we have gathered reports for all possible face-offs[31] , we can derive the conclusions from a graphical representation. The following presents the results of running the program for all possible combinations of the five levels

White/Black	Beginner	Novice	Expert	Pre-Master	Master
Beginner	6.5	2.5	1	4.5	1
Novice	12	8.5	3	4.5	4.5
Expert	12.5	14	12	13.5	5.5
Pre-Master	12.5	5.5	4	5	4.5
Master	13	6.5	4	6.5	4.5

Figure 15: Non-deterministic simulation results

In the graph below, each lines represents one level whose performance can be derived by the Y axis value

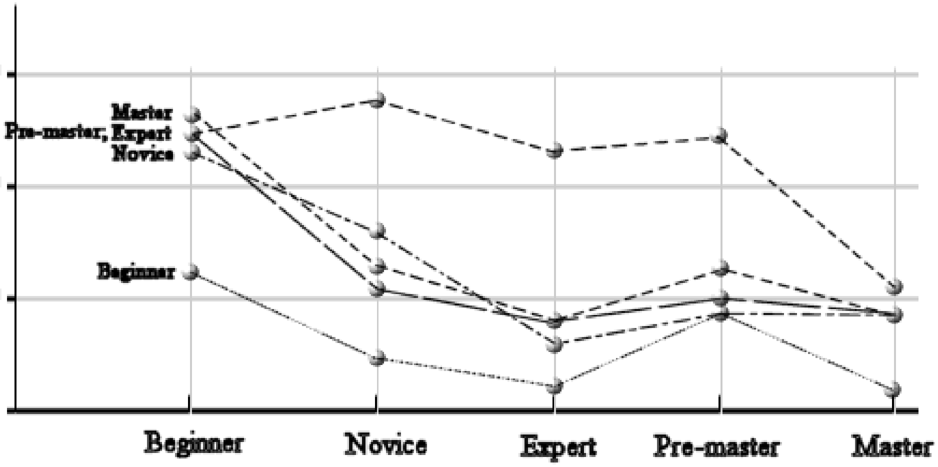


Figure 16: Non-deterministic simulation graph

---

[30] Used to abstain program determinism.

[31] There will be N + (N-1) + (N-2) + ... + 0 face-offs assuming an existence of N separate levels of difficulty.

The following presents the score difference in a game that is played deterministically i.e. with no random offsets. Positive numbers symbolise games won by White who opens the game, whereas negative (in bold face) indicate a loss.

White/Black	Beginner	Novice	Expert	Pre-Master	Master
Beginner	**-26**	**-12**	**-2**	**-8**	**-4**
Novice	44	**-14**	**-10**	**-4**	**-5**
Expert	34	16	10	**-6**	33
Pre-Master	**24**	**-22**	-12	8	6
Master	20	**-2**	17	16	**-4**

Figure 17: Deterministic simulation results

And the corresponding graph is given below

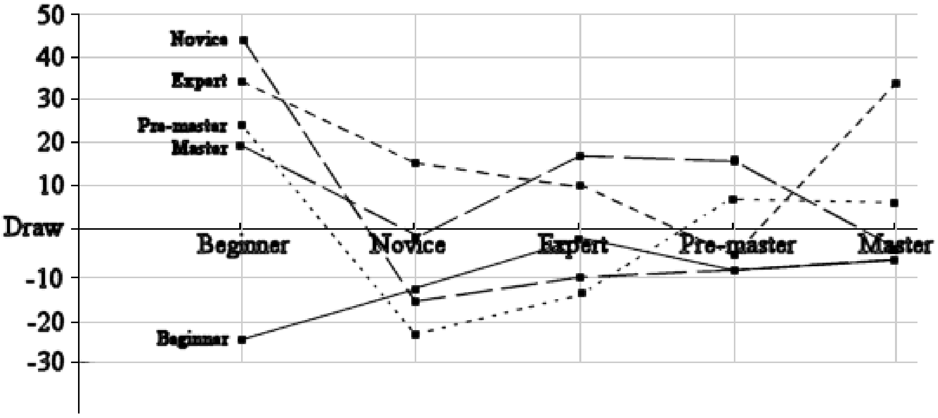


Figure 18: Deterministic simulation graph

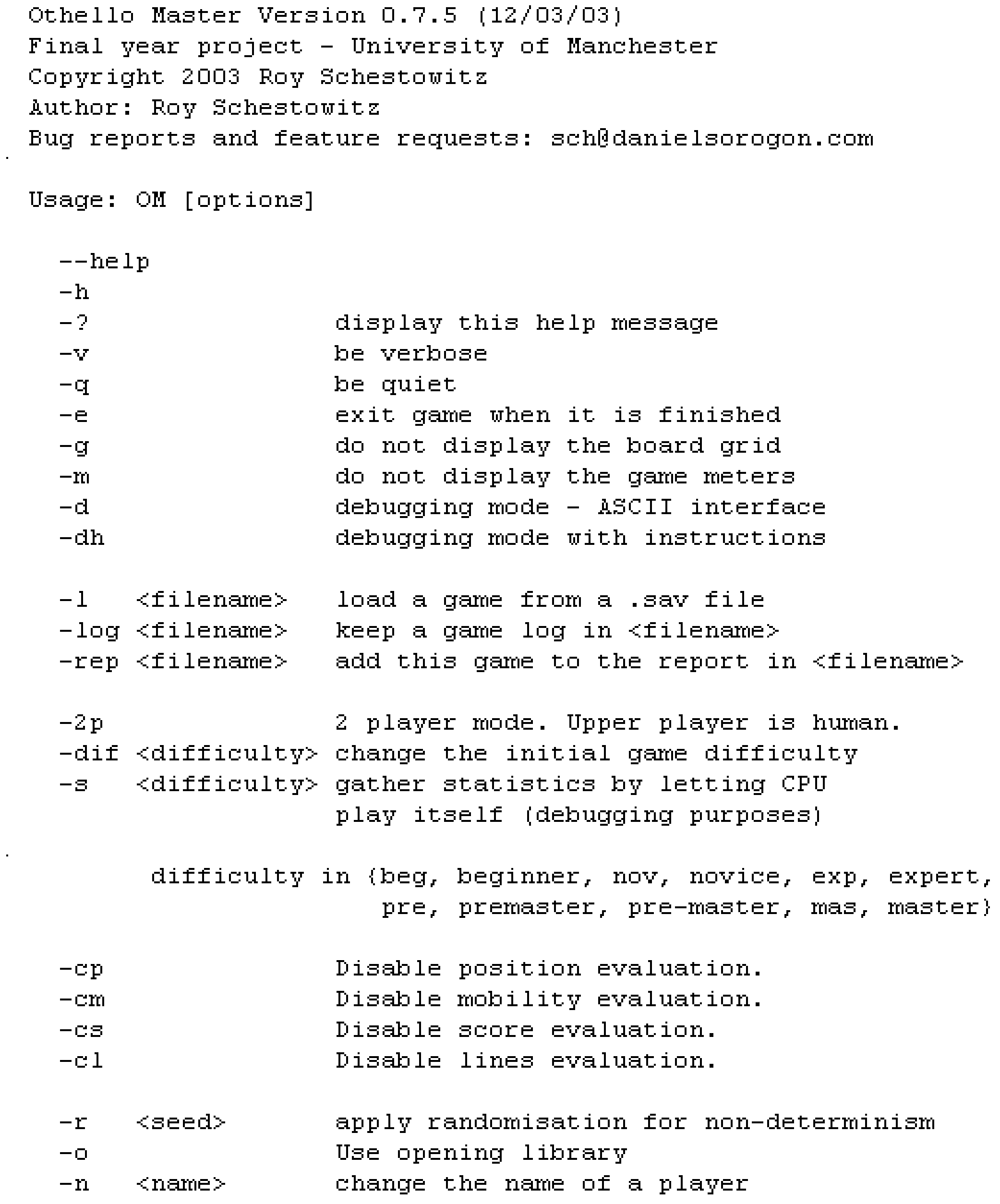
8 Documentation

8.1 User's Manual

Much of the information aimed at the user of Othello Master is available on-line. However, some instructions are embedded in the application as specified therein.

8.1.1 Command-line arguments

Othello Master, being a tool that needs to be invoked and controlled from the console level (bash, for example), offers a very flexible and extensive range of command-line arguments that cover most of the features available from the game menus and controls. The application can work independently for the sole purpose of generating logs and statistics. The following is invoked when the client calls the executable named OM[32] with the help option parameter or provides the executable with an uninterpretable input.



```
Othello Master Version 0.7.5 (12/03/03)
Final year project - University of Manchester
Copyright 2003 Roy Schestowitz
Author: Roy Schestowitz
Bug reports and feature requests: sch@danielsorogon.com

Usage: OM [options]

  --help
  -h
  -?                display this help message
  -v                be verbose
  -q                be quiet
  -e                exit game when it is finished
  -g                do not display the board grid
  -m                do not display the game meters
  -d                debugging mode - ASCII interface
  -dh               debugging mode with instructions

  -l   <filename>   load a game from a .sav file
  -log <filename>   keep a game log in <filename>
  -rep <filename>   add this game to the report in <filename>

  -2p               2 player mode. Upper player is human.
  -dif <difficulty> change the initial game difficulty
  -s   <difficulty> gather statistics by letting CPU
                    play itself (debugging purposes)

        difficulty in {beg, beginner, nov, novice, exp, expert,
                          pre, premaster, pre-master, mas, master}

  -cp               Disable position evaluation.
  -cm               Disable mobility evaluation.
  -cs               Disable score evaluation.
  -cl               Disable lines evaluation.

  -r   <seed>       apply randomisation for non-determinism
  -o                Use opening library
  -n   <name>       change the name of a player
```

Figure 19: Command-line prompt

A few points to be aware of:

- Parts of the above output are determined dynamically rather than statically. Not all parts of the output are embedded within the code, but are rather extracted from definitions, e.g. date and package.

---

[32] As a matter of fact, it is an abbreviation or an alias to the original executable which is omcore.

- The word *in* indicates membership in a set. For instance, the option <-dif master> will do what one would expect – activate the difficulty level 'Master'.

8.1.2 Debugging Mode

For quick and simplistic interaction with the game, an ASCII representation was made available long after the OpenGL-based representation had been completed. The plays are carried out using the keyboard[33] and on-screen instructions may be used to aid the client of this service. The main purpose of this mode was for the developer to test and trace various objects in the program. It can nonetheless be used as an alternative user interface too.

8.1.3 Menu

The following summarises the menu entries and sub-entries. These cover most of the functionality that is available once the game has been invoked, but some functionality is only available via command-line arguments[34].

---

[33] Input is provided as board coordinates, e.g. H5.

[34] The opposite case is also true; some functionality is restricted to game menus only.

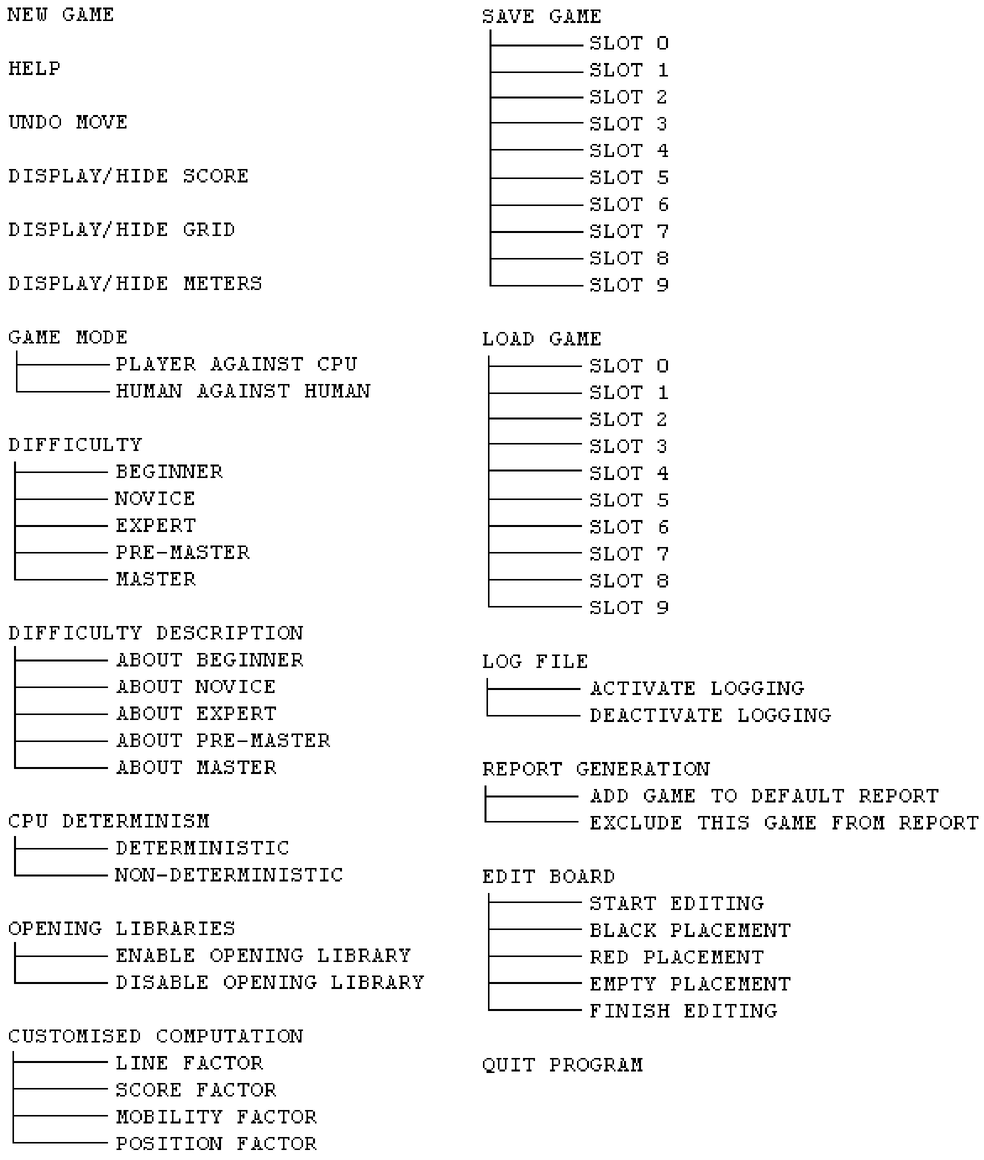


Figure 20: Game menu

8.1.4 Difficulty Description

Descriptions of the underlying algorithms that are playing against the user have been hardwired into Othello Master. These can be invoked from the game menu and target the more advanced user who may be curious to know what is happening deeper inside the machine.

8.1.5 Controls and Help Screen

As in the above, for the more common user, game instructions and controls are built into the OpenGL environment and are available from the game menu.

8.2 Application Documentation

As explained in the Project Proposal and plan (c/f Appendix B), a future programmer that embarks on the process of extending the application may wish to gain familiarity with the code. Amongst the aids for this process are the system's hierarchy and interaction diagram (c/f High-level Design) and some form of functions description and mapping. To provide the latter, a full summary of the functions has been assembled manually (no tool could generate natural language or even extract the interfaces as Javadoc does) and is available under Appendix A.

9 Maintenance and Extensions

Maintenance may rely on the available code and its aforementioned documentation. The code was built reliably enough and remained loosely coupled in order to enable other games to be built upon the existing services. Although a large set of operations was fully implemented and tested, future requests or different application requirements could justify further extensions.

The terms *maintenance* and *extension* are closely related in this context, for after all the most productive type of maintenance would be an extension of the existing application.

For the more curious readers, the functional summary[35] of the C implementation (as presented in Appendix A) is a good point to refer to at this stage. Many of these functions have been sufficiently generalised and parameterised to allow for use in other game-playing programs. Such game-playing programs need not involve the game Othello, but optimally should involve a two-dimensional board. This arbitration is not a trivial one and it would have been even more apparent had the application been implemented in an object-oriented computer language.

10 Conclusions

The project implemented and employed algorithms sufficiently strong to perform convincingly well against other available applications and has managed to analyse its different approaches to conclude which ones surpass all others and why. The project was constructed in accordance with the software engineering life-cycle and possible extensions have been proposed.

The project may contribute and assist others as it lays out successful approaches for artificially playing this game (or possibly any game) as well as bad approaches. There are plenty more simulation tasks that could take place, e.g. ones which exclude some evaluation tests, and without a doubt, this could push the project even further ahead and result in some slight improvements.

---

[35] Includes a general description along with concise explanation of the various inputs and outputs. The source code itself comprises a great deal of comments and annotation that leave little place for confusion.

11 References

11.1 Literature

- Gough, G. D. and D. E. Rydeheard - *Introduction to Algorithms and Data Structures* (2001). Department of Computer Science, University of Manchester.
- Howard, T. L. and Steve S. Pettifer - *CS2072 Computer Graphics* (1999). Department of Computer Science, University of Manchester.
- Howard, T. L., Roger J. Hubbold and Steve S. Pettifer - *CS3071 Advanced Computer Graphics* (2002). Department of Computer Science, University of Manchester.
- Kilgard, Mark J. - *The OpenGL Utility Toolkit (GLUT) Programming Interface, API Version 3* (1996). Silicon Graphics, Inc.
- Latham, John T. - *CS1052 Systems Design and Development* (2000). Department of Computer Science, University of Manchester.
- Schalk, A. C. - *CS 3192 The Theory of Games and Game Models* (2001). Department of Computer Science, University of Manchester.
- Warboys, B. C. and John T. Latham - *CS2341 Software Engineering* (2001). Department of Computer Science, University of Manchester.
- Warboys, B. C. and Nick P. Filer. - *CS2452 Software Engineering II* (2001). Department of Computer Science, University of Manchester.

11.2 On-line Sources

- http://www.danielsorogon.com/Webmaster/Projects/OM.html

  The main page for information, resources and archives for *Othello Master. Last visited May 2003*

- http://home.nc.rr.com/othello/rules/

  The full rules for the game Othello. *Last visited March 2003*

- http://www.mattelothello.com/purpleindex.html.

  A great resource of tips on Othello playing techniques and strategies. *Last visited on January 200*3.

- http://www.dcc.unicamp.br/~stolfi/EXPORT/images/textures/ppm-400x400/

  A graphical library of PPM images to be used as a resource of textures for the OpenGL GUI. *Last visited on February 200*3.

- http://crow.cs.und.ac.za/angus/GameProgramming/OglTextures.html

  One of many OpenGL FAQ sources to be used for scene rendering. *Last visited on November 2002*.

- http://www.nada.kth.se/~gunnar/howto.html

  Notes on searching and position evaluation in Othello. *Last visited on February 200*3.

- http://home.tiscalinet.ch/t_wolf/tw/misc/reversi/html/index.html

  The Anatomy of a Game Program. *Last visited on November 2002*.

- http://vcg.iei.pi.cnr.it/swform.html

  Bump mapping demonstration and illustration program. This complements the graphical aspects of the application. *Last visited on November 2002*.

- http://www.codesampler.com/oglsrc.htm

  More advanced OpenGL examples. *Last visited on January 200*3.

- http://www1.ics.uci.edu/~eppstein/180a/w99.html

  Strategy and board game programming. *Last visited on January 2003*.

- http://www.maths.nott.ac.uk/personal/anw/G13GAM/

  Game theory from Nottingham University. *Last visited on January 2003*.

12 Appendices

12.1 Appendix A

Functions Summary

The following presents the components of the program and the functions that they comprise, along with their description, inputs and outputs (where appropriate).

Callbacks

- *void load (int menuentry)*

  Description: the load game submenu callback function

  Inputs: the callback value

- *void save (int menuentry)*

  Description: the save game submenu callback function

  Inputs: the callback value

- *void determinism_callback (int menuentry)*

  Description: the determinism callback function

- *void edit_board_callback(int menuentry)*

  Description: board editing submenu callback function

  Input: the callback value

- *void customise_callback (int menuentry)*

  Description: customised computation submenu callback function

  Input: the callback value

- *void open_report (char *filename)*

  Description: opens the file where a game report would be appended

- *void open_log_file (char *filename)*

  Description: opens a log file displaying some initial information

  Input: the name of the log file to be created

- *void opening_library_callback (int menuentry)*

Description: the opening moves library submenu callback function

Input: the callback value

- *void report_callback (int menuentry)*

  Description: the report generation submenu callback function

  Input: the callback value

- *void log_file_callback (int menuentry)*

  Description: the log file submenu callback function

  Input: the callback value

- *void difficulty_description_callback (int menuentry)*

  Description: the difficulty description submenu callback function

  Input: the callback value

- *void difficulty_callback (int menuentry)*

  Description: the difficulty submenu callback function

  Input: the callback value

- *void gamemode_callback (int menuentry)*

  Description: the game mode submenu callback function

  Input: the callback value

- *void menu (int menuentry)*

  Description: the menu callback function

  Input: the callback value

- *void mouse (int button, int state, int x_val, int y_val)*

  Description: reacts to mouse events sensibly

- *void idlefun ( )*

  Description: carries out operations when OpenGL is idle

- *void automated_moves (void)*

  Description: gets the CPU to play a move when necessary

- *void display (void)*

  Description: the main display loop. Called from main() to do all the drawing to the frame buffer

- *void reshape (int w, int h)*

  Description: the reshape callback function

  Input: w and h which are the new width and height allocated to the frame by the window manager

- *void mouse_motion (int x, int y)*

  Description: a callback function that is invoked upon an event of a mouse move

- *void keyboard (unsigned char key, int x, int y)*

  Description: the keyboard callback function. Invoked when a key is pressed

Computation

- *board_map make_nth_best_move (board_map inputboard, int color, int priority)*

  Description: makes the Nth best move for color given N

  Input: the board that is dealt with, the color for which the move is carried out and the priority n where 1 is best choice

  Returns: the board after the move was carried out

- *int get_mobility_of_color (board_map inputboard, int color)*

  Description: gets the number of moves available

  Input: the board that is dealt with and the color for which the number is calculated

  Returns: the number of moves available

- *int evaluate_straight_lines_complete (board_map input_board, int color)*

  Description: calculate the value of the complete straight lines on the given board

  Input: the board that is dealt with and the color for which the value is calculated

  Returns: the value of the lines

- *int evaluate_diagonal_lines_complete (board_map input_board, int color)*

  Description: calculate the value of the complete diagonal lines on the given board

  Input: the board that is dealt with and the color for which the value is calculated

  Returns: the value of the lines

- *int evaluate_straight_lines_incomplete (board_map input_board, int color)*

  Description: calculate the value of the incomplete straight lines on the given board

  Input: the board that is dealt with and the color for which the value is calculated

Returns: the value of the lines

- *int evaluate_diagonal_lines_incomplete (board_map input_board, int color)*

  Description: calculate the value of the incomplete diagonal lines on the given board

  Input: the board that is dealt with and the color for which the value is calculated

  Returns: the value of the lines

- *int evaluate_lines (board_map input_board, int color)*

  Description: calculates the value of the line occupancy for color gap in a given board

  Input: the board that is dealt with and the color for which the value is calculated

  Returns: the value required

- *int calculate_mobility_difference_of_board (board_map inputboard, int color)*

  Description: calculates the mobility value gap in a given board

  Input: the board that is dealt with and the color for which the advantage is calculated

  Returns: the mobility gap in board (positive if color's mobility is higher)

- *int calculate_score_difference_of_board (board_map inputboard, int color)*

  Description: calculates the score gap in a board

  Input: the board that is dealt with and the color for which the advantage is calculated

  Returns: the score gap in board (positive if color's score is higher)

- *int find_value_of_position (int i, int j, int color)*

  Description: finds the value of position i,j standing for A-H and 1-8.

  Input: the board coordinates ranging from 1-8 and the current color for randomisation reasons

  Returns: the value of the position inquired

- *int evaluate (int color, board_map input_board)*

  Description: evaluates the current position of the board from the point of view of color and assumes that color is currently up

  Input: the color for which the evaluation is carried out and the state of the board

  Returns: the evaluation value

- *board_map compute_move_for_color (board_map inputboard, int color)*

  Description: computes a complex move for a given colour

  Input: the colour of player whose turn it is, input board

  Returns: the board with the new stone put

- *int random_number (int N)*

  Description: returns a random integer between 0 and N

- *int zero_one_random (void)*

  Description: returns 0 or 1 randomly

- *int load_opening_library_file(char *filename)*

  Description: load an opening library of *Othello Master* from a file given a filename

  Entry: filename is a pointer to the characters of the file to open

  Returns: true if save operation was successful, false otherwise

- *void init_opening_library (void)*

  Description: initialises the opening hashtable library

- *char* find_board_state_ident (void)*

  Description: assigns the distinct identifier to a board state

- *void cpu_move (int color)*

  Description: makes a move on behalf of color using the CPU

Drawing

- *void drawString (void *font, float x, float y, char *str)*

  Description: draws a string in a given (x,y) position on the window

  Entry:

  - *font points to a given font defined by GLUT
  - *str is the pointer to the string to be displayed
  - x is the x-coordinate for the string to be drawn at
  - y is the y-coordinate for the string to be drawn at

  Exit: the pointer at *str is unchanged as well as *font

- *void draw_scene (void)*

  Description: draws the scene in which the game takes place

Hashing

- *void error (char *message)*
- *void fatal_error (char *message)*
- *Index sec_hash (Key_Type key)*

  Description: secondary hash function
- *Table initialize_table (Table_size table_size)*

  Description: initialise a table of given size
- *int find_pos_for (Key_Type key, Table H)*

  Description: find_Pos_For a key in a hash table. Uses linear probing
- *int find (Key_Type key, Table H)*

  Description: see if key is in hash table
- *int getX (Key_Type key, Table H)*

  Description: get X value of table entry
- *int getY (Key_Type key, Table H)*

  Description: get Y value of table entry
- *Table insert (Key_Type key, Table H, int the_X, int the_Y)*

  Description: insert a key in a hash table
- *Table delete (Key_Type key, Table H)*

  Description: delete a key from the hash table
- *Table rehash (Table H)*

  Description: rehashing function
- *void print_table (Table H)*

  Description: prints out the given table
- *static Index hash (Key_Type key, Table_size H_SIZE)*

  Description: load a game of *Othello master* from a file given a filename

  Entry: filename is a pointer to the characters of the file to open

  Returns: true if save operation was successful, false otherwise

Loaders

- *GLubyte * glmReadPPM (char *filename, int *width, int *height)*

  Description: loads a PPM file

  Entry:

  - *filename is a pointer to the string holding a filename to open
  - *width is a pointer to some address where image width will be stored
  - *height is a pointer to some address where image height will be stored

  Exit: width and height are pointers to image dimensions

- *int save_game_to_filename (char *filename)*

  Description: saves a game of *Othello master* on a file

  Entry: filename point to filename string

  Returns: true if save operation was successful, false otherwise

- *int save_game (int slot_number)*

  Description: saves a game of *Othello master* on a slot

  Entry: slot number

  Returns: true if save operation was successful, false otherwise

- *int load_game (int slot_number)*

  Description: load a game of *Othello master* from a file given a slot number

  Entry: slot number

  Returns: true if save operation was successful, false otherwise

- *int load_game_from_filename (char *filename)*

  Description: load a game of *Othello master* from a file given a filename

  Entry: filename is a pointer to the characters of the file to open

  Returns: true if save operation was successful, false otherwise

Misc

- *int reducible (int i, int j, int color, board_map board)*

  Description: determines whether a move/placement is legal for a given colour/side in an i,j coordinate on the board

Inputs: color of the player whose turn it is; the i and j board coordinates which are virtually the X,Y position of the board at which the stone is to be put (progressing left to right, top to bottom); the board to be dealt with

Returns: boolean indicating is the placement is legal or not

- *void calculatescore (void)*

  Description: calculates the current score for red and black

- *board_map reduce (int i, int j, board_map board)*

  Description: a function used to do all the Othello-wise reductions and substitutions in colours of the objects.

  Inputs: the coordinates of the last stone put, according to which the correct reductions can be carried out; the board to be dealt with

  Returns: the new layout of the board

- *void calculate_mobility (void)*

  Description: calculates mobility of both sides

- *void check_deadlock (void)*

  Description: checks if a deadlock has occurred in which case the game has reached an end or turn passed (if mobility of current side is 0)

- *void finishoff (void)*

  Description: called when the game has reached an end and declares the result

- *void file_draw_ascii_board (board_map board, FILE * file)*

  Description: puts an ASCII representation of the board on the log file

  Inputs: the input board and the file pointer

- *void close_log_file (void)*

  Description: closes the log file when a game is finished and displays the game score

- *void add_to_report (void)*

  Description: adds the given game to a report file

Omcore

- *void adjust_look_at_center (void)*

  Description: sets the camera to point to the X,Y,Z origin

- *void draw_objects (void)*

  Description: draws the stoned that are dynamically changing their layout

- *void initboard (void)*

  Description: initialises the board when a new game begins

- *void initmenu (void)*

  Description: sets up the menu and takes care of the entries and their callback value

- *void init (void)*

  Description: initializes the program. This function is called once at bootstrap

- *void draw_ascii_board (board_map board)*

  Description: draws an ASCII representation of the board

  Inputs: the input board

- *void annotate_board (void)*

  Description: annotates the overhead main view

- *void annotate_help (void)*

  Description: annotates the help screen

- *void annotate_difficulty (void)*

  Description: displays the description of the algorithm used

- *void annotate (void)*

  Description: adds text on top of the scene depending on which view is enabled

- *void set_name_of_player (char *name)*

  Description: called when the name of the player is entered and to record that name

  Inputs: the pointer to the characters representing that name

- *void display_debugging_instructions(void)*

Description: Prints some brief debugging instructions

- *void display_help (void)*

  Description: displays the command line option to the user and quits

- *void process_command_line (int argc, char *argv[])*

  Description: a function to process the command line arguments inputs - the arguments and the number of arguments

- *void set_difficulty(char *diff)*

  Description: sets up the difficulty of player 2

- *void set_up_stat_mode_difficulty(char *diff)*

  Description: sets up the difficulty of player 1 when gathering statistics

- *int main (int argc, char **argv)*

  Description: the main function. called when the program is started.

  Inputs: the arguments and the number of arguments from the command line

- *void quit_game (void)*

  Description: the main exit procedure. Close any connection and files here.

12.2 Appendix B

Project Proposal and Plan

The following presents the relevant parts of a past document. That past document briefly outlined the objectives, milestones and characteristics of the program to be created.

Introduction

This project has been set to produce an application which will be titled *Othello Master* due to some visual similarity to an older game called Chess Master. It will require knowledge of game theory and advanced computer graphics. The main research issues to be addressed are:

1. How to represent a game scene in a convincing three-dimensional way. This project also attempts to put considerable effort into advanced graphical interfaces.
2. How to carry out the complicated task of computing a good move in the game Othello.
3. How to utilise different algorithmic approaches to solve problem 2.
4. How to enable the program to play against itself using different approaches and generate statistics for beneficial research[36] or analysis of the program's performance.
5. How to store the state of a program in a reliable and persistent manner, with the possibility of retaining a stack of multiple states.

Othello Master will utilise the basic principles of game theory and implements them in an imperative programming language. It also will work closely with a package called GLUT[37], which will be responsible for the graphical domain of the program.

Background

Othello

Othello is an easy game to learn, but long practice and experience are required to master it.

There are a few extra points that are worth mentioning:

- Only one type of element can be placed on the board. A board slot can be either empty or occupied by a stone of one player.
- Almost any stone on the board can be recaptured by the opponent i.e. the game score is highly volatile.
- Low number of possible moves is a great disadvantage in the game.

---

[36] The main research issue to be addressed in this project shall be the strength of game playing algorithms, derived from the results of games being played.

[37] Provides scene description and user interaction facilities.

Game-Playing Programs

A game-playing program will be required, above all, to generate a valid and good move given any game state. The game itself can be viewed as a form of a tree - one which specifies all possible plays[38] in the game. In practice, it will attempt to find the best path within the game tree. Our application will evaluate a current game state by inspecting various different aspects such as current positioning, prospective positioning, structure formed by stones, etc.

Structure and formation are some very subtle issues that differ significantly from one game to another. In a game like Chess, we might want to take into consideration the structure that a group of pieces form, or the structure of the board slots that are reachable by any of the pieces. In a game like Othello, the shape of an occupied cluster of stones might be an aspect that is worth paying great attention to. Analysis of the formation of stones which were placed on the sides and corners of the board may even be a more fruitful process.

At the design stage, many of these issues must be taken into consideration. The most important aspect of this project will be the strength of the game-playing engine, given that the creation of a strong playing-engine is my personal main ambition.

Game Theory

In order to apply our logical analysis and knowledge of Othello within a programming language and practically be able to generate a good play, we must be aware of some of the basic elements of game theory.

Alpha-beta pruning[39] traverses the game tree and evaluates the state of the game (as seen from one player's point of view) at different positions of the tree using an evaluation function. Since we cannot traverse the whole game tree of an Othello game[40], we can concede certain plays that appear to lead to worse evaluation results.

Much more on the broad topic of game theory and its application within my project will be available in the final project report.

Progress

This section presents the details of the progress made up to this date. The initial plan was to complete the requirement analysis and specifications by this point. Necessary reading, primarily concerning game theory, should have been almost completed by this point too, as well as some reading about the graphical aspects, bearing in mind that the graphical domain can, at this point, be accommodated by some simple I/O facilities such as a terminal/shell under Unix.

It is now well known and defined what is to be achieved and design work can take place shortly. Many of the issues concerning the construction of the game-playing engine have now been resolved too. Literature sources on game theory have provided me with sufficient knowledge about the process that needs to be carried out in order to predict the opponent's moves and the process of evaluation. Furthermore, supplementary notes and on-line sources have given me some better insight into techniques of designing and coding game-playing applications.

---

[38] Sequence of valid moves in the game. This can be more abstractly defined as a path in the game tree.

[39] Basic theoretical principle that allows anticipating moves in a game.

[40] As the traversal depth is incremented, the time complexity shows exponential characteristics.

What is required to be achieved by the system is more precisely recorded in various sketches in my project log book as well as in later section.

Objectives

Main Objectives

The application should:

1. Allow an Othello game to be played in accordance with the predefined rules and deal with all the game exceptions e.g. deadlocks.
2. Make use of alpha-beta pruning and ensure that its function is sufficiently parameterised so that we can control its behaviour rather easily. The result of such parameterisation is high usability and flexibility.
3. Enable the program and the user to control the behaviour of the game-playing engine. Such control should allow the program to play better, worse or even more quickly by fine-tuning and alternating the functions used to compute a given move.
4. Allow the inclusion of extra features such as:

- Load and save game
- Board editing tool with an appropriate interface
- ASCII mode Othello
- Undo function relying on a stack of game states
- View customisation

Secondary Objectives

Convert the application into a package that:

1. Provides support for the standard player vs. game mode and boasts various different levels of strength, corresponding to different approaches and algorithms that are being utilised. The main intention is to have different levels of difficulty, where preferably and quite naturally, the broader[41] algorithms should produce better game performance.
2. Provides multi-player mode, where one human player plays against another.
3. Allows watching the program as it plays against itself on different levels of strength. This will, in principal, allow tracing a game that is being played very quickly and be of assistance in the later debugging process.

---

[41] Breadth typically refers to LOC (Lines Of Code) value, however, larger amounts of code do not imply better performance. Broad algorithms can be defined as those that perform deeper and wider search of the game tree.

4. Records game information by generating a game log. Such a log will specify all placements of stones, display the final board state and, most importantly, record the final score.
5. Generates statistical reports automatically for many consecutive games.

Mandatory Milestones

The following suggests a listing of the features that *must* be generated by the completion of the project:

1. Othello game can be played with the control of two user controlled players.
2. Alpha-beta pruning implemented and the program is able to carry out a legal play.
3. The above simple play is now changed to a sensible and good move and the program is able to compete with other Othello game-playing application.

In order to convert *Othello Master* into a unique package, we may as well wish to include the following as milestones:

1. 'Save game' and 'load game' features are implemented and an 'undo' stack is maintained throughout the application's run.
2. Versatile computation algorithms are devised and the program can play against itself.
3. Logs with a sensibly informative layout are generated for each game.
4. Reports are generated and hold statistical data about multiple games played within the application.
5. The board state can be altered manually using a board editing tool.
6. ASCII interface implemented and is bug free.
7. Customisation of the user interface is available.
8. Computation of a move is controlled by command-line arguments or menu entries within the GUI.

System Development Outline

The system is to be built in several stages, as defined by the software engineering discipline:

Requirement Analysis

At this Stage the requested result is to be discovered and analyzed. It is important that it is ensured that this project will evolve into a non-trivial entity - one which will make this project a 'one of a kind' and, therefore, more interesting to analyze and discuss.

Specification

This phase will produce some clear description of what is to be achieved without having us concerned about how to implement everything.

System Design

We are now looking at how the system will be implemented. This can be logically divided into two different stages:

High-level Design

Also functional design. We need to agree on which modules the system will comprise of and what each one of them is responsible for. Diagrams could, indeed, be useful here as we are interested in some structural representation, which will ease our process of understanding the system.

Low-level Design

We would need to incorporate the above and develop some pseudo-code incrementally. This will be a very time consuming stage where programming skills are essential. Also, efficiency issues should be addressed here, otherwise they will have to wait to a much later stage where the effort spent will be greater.

Design of the algorithms for all the different approaches takes place at this stage. This is a good point to refer to the information sources and reveal some of the different conventional methodologies that are used to solve the problem of computing a good play. A simple and useful starting point would be designing one algorithm that will make an arbitrary placement of a stone (preferably a random one).

Implementation

Using the output of the above stage (diagrams, paper or text) we need to code what was agreed on previously. If C is used, a Makefile needs to be created to link all the different parts of the overall system and the pseudo code needs to be translated into (lower-level) imperative code.

Testing

Stability

Is our program robust enough to allow all games to be ended gracefully without crashing? If exceptions occur, does the system catch them? Does it handle them appropriately? All of these factors should be taken into consideration if we wish to extend the system comfortably and see if it is going through unexpected and undesirable paths.

Performance

Let us check if the system is operating quickly enough. Should we impose some constraints on the time that the CPU can spend computing a move? Should we modify the graphical engine to allow more frames to be generated in a given time?

Strength

It is highly advisable that *Othello Master* is then put against as many other Othello-playing applications as possible. Testing it against other playing engines will give us an indication of how well it plays against other game engines with (possibly) different approaches. Allowing experienced human players to play against *Othello Master* may also prove to be useful since more constructive analysis and feedback can be obtained. The drawback of this idea is the fact that this can be slow and tedious.

Assignment of difficulty titles to the various approaches is left untouched until the simulation phase which will be discussed later.

Documentation

User's Manual

The user will require having a help feature within the program, but is that enough? Perhaps the user may wish to use some non-trivial options too. We can provide some command-line options specification, as well as a user's manual, which will explain how to use the system more productively. A more detailed manual with concise reference to all the features that are available in *Othello Master* will form a part of the final project report. Relevant bits from this report can then be exported and expanded to form an official user's manual.

Programmer's Manual and Documentation

The code which may be very readable to its creator is not always as manageable and understandable to another programmer that wishes to extend, improve or maintain it. Therefore, some more abstract views of the system, along with some description in natural language can save a lot of time and effort. The following are some basic suggestions:

1. It needs to be assured that the code is well documented, preferably with comments where appropriate.
2. A functional and modular summary is necessary to give a system overview. This can also prove to be useful if we want to re-use the code, let us say, for some new chess application. Finding the appropriate functions fairly trivial in this way.
3. A diagram of the system structure would be highly appreciated by the next person to familiarise himself/herself with the code structure. It often proves to be crucial for the understanding of the interactions within it.

4. A report of how the system works as a whole, how to compile it, platform/OS dependent code and debugging tools are a necessity. It is a very common phenomenon for a programmer to find himself/herself lost with a piece of code that is unreadable, unusable or lacks the compilation facilities.

Simulation and Research

This is a phase that is applicable in our project, but not necessarily in other software engineering development processes. We can now utilise batch facilities to have the program play itself. Most importantly, if we have written different game-playing algorithms (corresponding to different approaches), we can now gather some game statistics which will indicate which approach is stronger than another or which seems to have special weaknesses against specific other approaches. if simulation time allows, we can extend this series of tests[42] by parameterisation of the existing algorithms. Fine-tuning of those algorithms is now applicable, but more on this will be discussed in the final project report.

We may have to conduct a more complex analysis that will produce graphs and tables. These will also be valuable for the later project report. Some research can take place on performance and strength of certain Othello-playing algorithms, as the end of this section will discuss in greater detail.

Having performed these analyses, we can now have some vague conclusion as for which approaches are stronger than others. Since the interface to the end-user should not bog down to details of implementation, linking each approach to some word that will indicate a difficulty level, will be a wise step. The allegedly best algorithm that we have put together should, for instance, be titled ’Master’.

To determine which algorithms appear to perform better than others within a game of Othello and to adjust or fine-tune those appropriately, we will certainly have to re-write some code. This will require a long repetitive process of testing and re-implementing the code. In fact, a testing-implementation loop will consume a considerable amount of time at this phase.

As for the aspect of research, we can derive from the above figures and logs (or possibly even charts):

- The time complexity of each approach - by accumulating the duration of each move.
- The strength of each approach - by inspecting the results of a series of games.
- Stability - hopefully, this will not be an issue. If and when bugs occur, the full attention should again be diverted to the previously specified testing phase.

The results of these observations and analyses shall also form a reasonably large section in the final project report. Extending a project which initially focused on construction and development into one of contribution to research has often been encouraged by the project supervisor.

Maintenance and Extension

When the system, as it was initially intended to look like, has become a finished package (as specified within the main objectives) , we might desire to extend it or improve it by adding more features or reviewing the algorithms

---

[42] For N levels of difficulty in our game, N! testing series shall take place.

respectively. This stage is a never-ending process which would be classified as 'just a possibility' at this point. The tasks listed as secondary objectives should be the ones to be taken into consideration here.

Shall any other ideas come up during development, assuming that they are of some importance of use, we may add them to the list of the secondary tasks. It is quite likely that some of these features that are identified later are even more useful than the ones currently suggested. The previous prioritisation should be re-evaluated in such a case too.

However, since the code is flexible enough to allow the game to mutate into another[43] , it is valuable to bear in mind that maintenance and extensions could potentially be carried out by other game programmers. The documentation for this application, as it was specified earlier, should practically serve its purpose here.

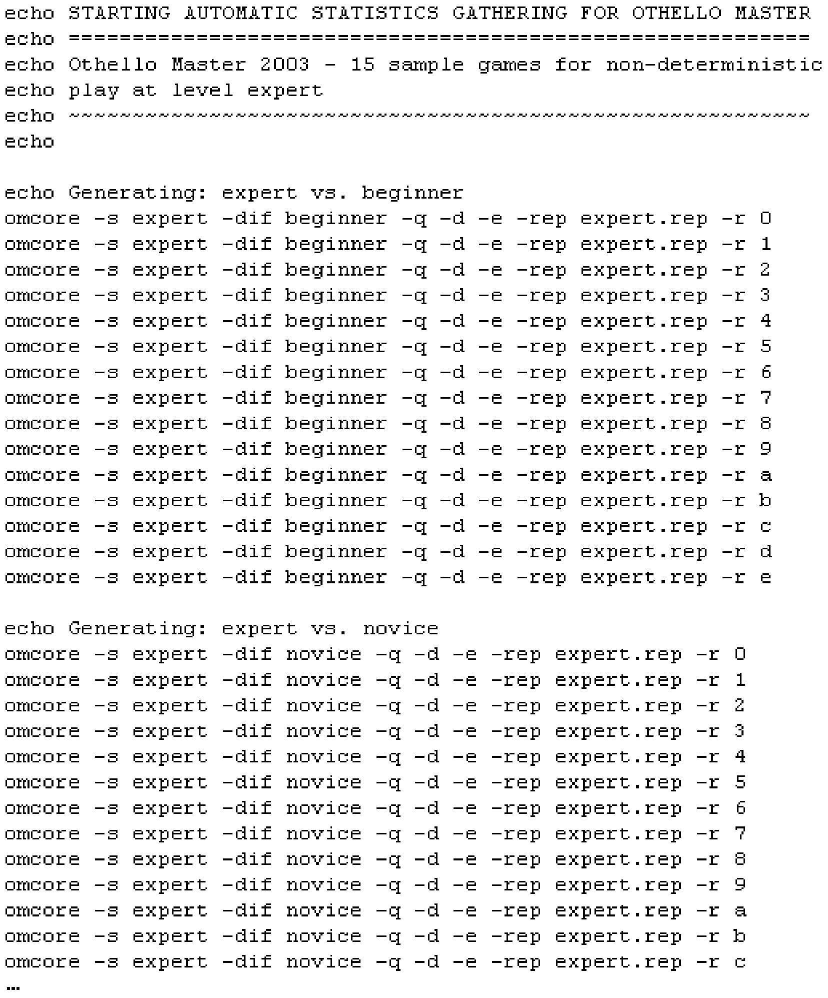
---

[43] For instance, most of the code can be reused for the construction of a Checkers application. To name but very few similarities, the presentation layer is analogous and storage of any game state is identical.

12.3 Appendix C

Batch Files

The following is a segment of a sample batch file that is used to run and record the results of a large number of games. There are several such files and they were invoked for long hours on multiple machines in order to generate some of the figures and charts.



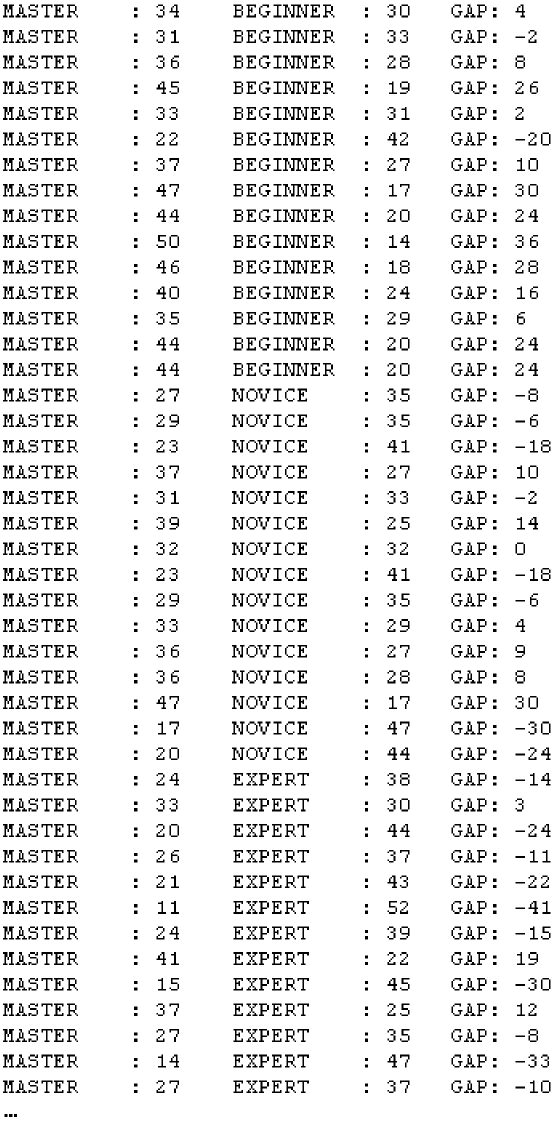
```
echo STARTING AUTOMATIC STATISTICS GATHERING FOR OTHELLO MASTER
echo ===========================================================
echo Othello Master 2003 - 15 sample games for non-deterministic
echo play at level expert
echo ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
echo

echo Generating: expert vs. beginner
omcore -s expert -dif beginner -q -d -e -rep expert.rep -r 0
omcore -s expert -dif beginner -q -d -e -rep expert.rep -r 1
omcore -s expert -dif beginner -q -d -e -rep expert.rep -r 2
omcore -s expert -dif beginner -q -d -e -rep expert.rep -r 3
omcore -s expert -dif beginner -q -d -e -rep expert.rep -r 4
omcore -s expert -dif beginner -q -d -e -rep expert.rep -r 5
omcore -s expert -dif beginner -q -d -e -rep expert.rep -r 6
omcore -s expert -dif beginner -q -d -e -rep expert.rep -r 7
omcore -s expert -dif beginner -q -d -e -rep expert.rep -r 8
omcore -s expert -dif beginner -q -d -e -rep expert.rep -r 9
omcore -s expert -dif beginner -q -d -e -rep expert.rep -r a
omcore -s expert -dif beginner -q -d -e -rep expert.rep -r b
omcore -s expert -dif beginner -q -d -e -rep expert.rep -r c
omcore -s expert -dif beginner -q -d -e -rep expert.rep -r d
omcore -s expert -dif beginner -q -d -e -rep expert.rep -r e

echo Generating: expert vs. novice
omcore -s expert -dif novice -q -d -e -rep expert.rep -r 0
omcore -s expert -dif novice -q -d -e -rep expert.rep -r 1
omcore -s expert -dif novice -q -d -e -rep expert.rep -r 2
omcore -s expert -dif novice -q -d -e -rep expert.rep -r 3
omcore -s expert -dif novice -q -d -e -rep expert.rep -r 4
omcore -s expert -dif novice -q -d -e -rep expert.rep -r 5
omcore -s expert -dif novice -q -d -e -rep expert.rep -r 6
omcore -s expert -dif novice -q -d -e -rep expert.rep -r 7
omcore -s expert -dif novice -q -d -e -rep expert.rep -r 8
omcore -s expert -dif novice -q -d -e -rep expert.rep -r 9
omcore -s expert -dif novice -q -d -e -rep expert.rep -r a
omcore -s expert -dif novice -q -d -e -rep expert.rep -r b
omcore -s expert -dif novice -q -d -e -rep expert.rep -r c
…
```

Figure 21: Sample batch file

12.4 Appendix D

Reports

Below is a sample report file (.rep) that holds results of several games that are played serially and have some mutual property, e.g. games where the starting/opening player is of level 'Beginner'.



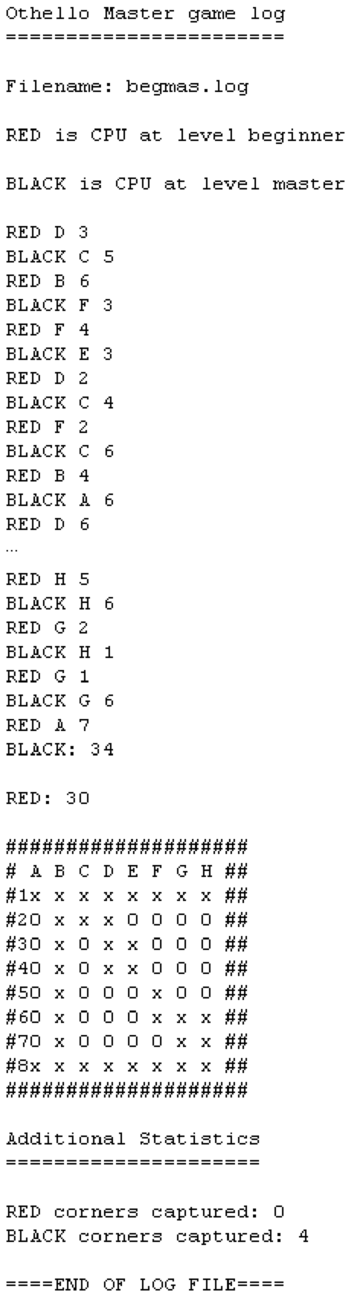
```
MASTER    : 34    BEGINNER  : 30    GAP: 4
MASTER    : 31    BEGINNER  : 33    GAP: -2
MASTER    : 36    BEGINNER  : 28    GAP: 8
MASTER    : 45    BEGINNER  : 19    GAP: 26
MASTER    : 33    BEGINNER  : 31    GAP: 2
MASTER    : 22    BEGINNER  : 42    GAP: -20
MASTER    : 37    BEGINNER  : 27    GAP: 10
MASTER    : 47    BEGINNER  : 17    GAP: 30
MASTER    : 44    BEGINNER  : 20    GAP: 24
MASTER    : 50    BEGINNER  : 14    GAP: 36
MASTER    : 46    BEGINNER  : 18    GAP: 28
MASTER    : 40    BEGINNER  : 24    GAP: 16
MASTER    : 35    BEGINNER  : 29    GAP: 6
MASTER    : 44    BEGINNER  : 20    GAP: 24
MASTER    : 44    BEGINNER  : 20    GAP: 24
MASTER    : 27    NOVICE    : 35    GAP: -8
MASTER    : 29    NOVICE    : 35    GAP: -6
MASTER    : 23    NOVICE    : 41    GAP: -18
MASTER    : 37    NOVICE    : 27    GAP: 10
MASTER    : 31    NOVICE    : 33    GAP: -2
MASTER    : 39    NOVICE    : 25    GAP: 14
MASTER    : 32    NOVICE    : 32    GAP: 0
MASTER    : 23    NOVICE    : 41    GAP: -18
MASTER    : 29    NOVICE    : 35    GAP: -6
MASTER    : 33    NOVICE    : 29    GAP: 4
MASTER    : 36    NOVICE    : 27    GAP: 9
MASTER    : 36    NOVICE    : 28    GAP: 8
MASTER    : 47    NOVICE    : 17    GAP: 30
MASTER    : 17    NOVICE    : 47    GAP: -30
MASTER    : 20    NOVICE    : 44    GAP: -24
MASTER    : 24    EXPERT    : 38    GAP: -14
MASTER    : 33    EXPERT    : 30    GAP: 3
MASTER    : 20    EXPERT    : 44    GAP: -24
MASTER    : 26    EXPERT    : 37    GAP: -11
MASTER    : 21    EXPERT    : 43    GAP: -22
MASTER    : 11    EXPERT    : 52    GAP: -41
MASTER    : 24    EXPERT    : 39    GAP: -15
MASTER    : 41    EXPERT    : 22    GAP: 19
MASTER    : 15    EXPERT    : 45    GAP: -30
MASTER    : 37    EXPERT    : 25    GAP: 12
MASTER    : 27    EXPERT    : 35    GAP: -8
MASTER    : 14    EXPERT    : 47    GAP: -33
MASTER    : 27    EXPERT    : 37    GAP: -10
...
```

Figure 22: Sample report file

12.5 Appendix E

Log Files

This is a partial log file that is constructed on demand or is recorded throughout the statistics gathering phase. It provides data on one individual game.



```
Othello Master game log
=======================

Filename: begmas.log

RED is CPU at level beginner

BLACK is CPU at level master

RED D 3
BLACK C 5
RED B 6
BLACK F 3
RED F 4
BLACK E 3
RED D 2
BLACK C 4
RED F 2
BLACK C 6
RED B 4
BLACK A 6
RED D 6
...
RED H 5
BLACK H 6
RED G 2
BLACK H 1
RED G 1
BLACK G 6
RED A 7
BLACK: 34

RED: 30

####################
# A B C D E F G H ##
#1x x x x x x x x ##
#2O x x x O O O O ##
#3O x O x x O O O ##
#4O x O x x O O O ##
#5O x O O O x O O ##
#6O x O O O x x x ##
#7O x O O O O x x ##
#8x x x x x x x x ##
####################

Additional Statistics
=====================

RED corners captured: 0
BLACK corners captured: 4

====END OF LOG FILE====
```

Figure 23: Sample log file

12.6 Appendix F

Sample Change Log: Version 0.5.4 - Version 0.7.5

The following is the most recent listing of updates and changes made to the program. Such listing is available on the Web site and allows both the programmer and the supervisor to trace progress and changes to the up-to-date release.

- The evaluation function now takes into account the sequences of stones on the board.
- Diagonal sequences are taken care of as well.
- Incomplete sequences are taken into account and are credited accordingly.
- Components within the evaluation function can now be switched on and off very easily.
- New menu entries for customising the computation.
- New difficulty level added.
- Width search is now available.
- A new function to retrieve the Nth best move.
- Getting mobility for a given color in a given board is now now possible in a direct function. Many functions are broken down to several inner functions.
- Difficulty modification in command prompt.
- CPU can play itself.
- CPU can play itself at all available difficulties. Currently 5 difficulties are available, so 25 tests are available.
- Neat organised reports are generated automatically for each game that takes place.
- A batch file omstats.bat was created to automatically generate up-to-date reports to all possible (25) games between CPU and CPU.
- Severe bug with get_Nth_best_move() has been traced and debugged.
- Non-deterministic move algorithms are built up for all difficulty levels so games end differently given random seeding.
- Log files now include a final board state expressed as ASCII.
- Saved game format has been upgraded as well as the appropriate version identifier.
- A report file infrastructure has been established yet currently not anything sophisticated is generated. The report file should be used to gather a large amounts of information condensed to allow game analysis.
- Saved game bug found and resolved.

- Generation reports using batch files to analyze a sample of non-deterministic games. Currently only beginner has been covered.
- Randomisation enhanced to make games even less predictable. The seeding is performed by srandom().
- Internal code change - all callback values are defined as preprocessor values for readability.
- Full and detailed comments in callbacks.c.
- View changes interface to enumeration for readability.
- Full and detailed comments in loaders.c.
- Late file close in loaders.c spotted and corrected.
- Instructions for debugging mode have been added.
- Upgrade the debugging mode interface.
- Annotation of objects in drawing.c - the scene description module.
- Comments added to misc.c to complete code documentation.
- 2 new options added to command line - hiding grid and hiding the game meters.
- omcore.c, the main procedure, is not fully commented.
- Documentation of computation.c
- Command line option for debugging with instructions and without instructions.
- Explanation of difficulty constraints added.
- Detailed documentation of computation.
- Undo implemented, but still not tested.
- Editing tool created to change the layout of the board during the game.
- Visual indication of whether program in editing mode or not added.
- Command line options now include control of computation. This feature is useful for automatic gathering and analysis of the program's behaviour.
- Inefficiency issue in texture processing addressed and now FPS rate has gone up.
- Screen cleared in ASCII mode at every iteration.
- Dynamic game view added.
- Log file includes corners count.
- Report file includes the score gap between the two sides.
- Submenu for algorithm description has been added.
- Algorithm descriptions added.
- Opening libraries added with command-line and menu options.

12.7 Appendix G

Pseudo-code

The following is an excerpt from the program's pseudo-code. Many such algorithmic pieces could be included, but it is the principle that counts and not the quantity.

Let us look at a pseudo-code example that is meant to generate a valid random move in the game. The following assumes some other function have been or will be provided

```
1. Reset placement[44] status
2. While the placement is unset do the following:
   (a) Scan each row in the game board
       i. Scan each board slot in the row
          A. If the slot is occupied, do nothing
          B. Otherwise, check validity of placement
             • If the placement is valid
               – Perform it
               – Set placement status
             • Otherwise, do nothing
3. Do the appropriate flips on the board
4. Check if a deadlock has occurred
```

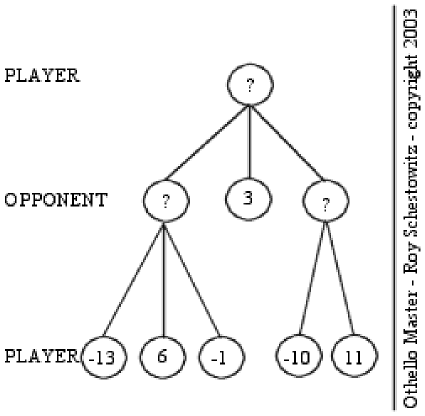
---

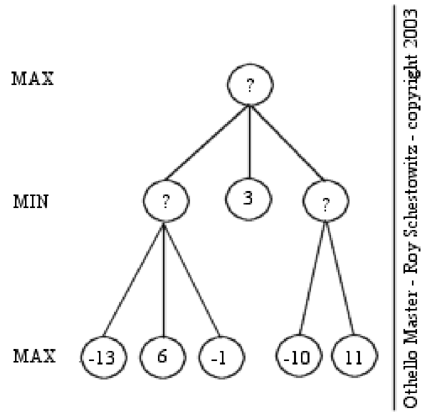
[44] Placement refers to the putting a stone on the board.

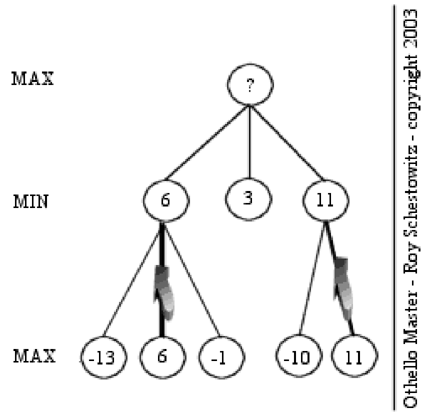
12.8 Appendix H

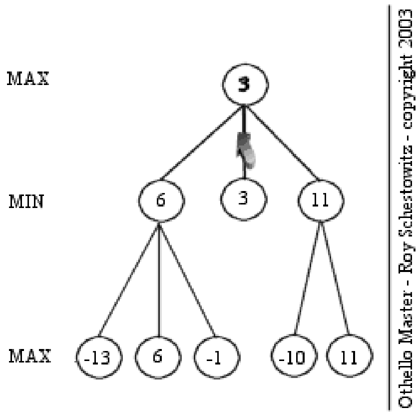
The Minimax Algorithm

This is a more detailed example showing the minimax algorithm working through the tree. The bottom-up traversal is now more apparent. Please note that by most conventions minimum and maximum are carried out in reverse order to the ones below.









Figures 24-27: The minimax algorithm

12.9 Appendix I

Milestones in *Othello Master*

The following is a very detailed listing of the milestones. It is fully separated and independent from the brief outline given in the Project Proposal and Plan. It is recommended, but not essential, that this section should be read prior to the viewing of the sections on Design and Implementation.

One of the more straight-forward parts of approaching the design of the application is the one concerned with the game Othello. Literature on the topic and on-line resources are not difficult to obtain.

The main challenges at that stage are:

1. The sufficient understanding of the user interfacing tool specifications.
2. Implementation of an algorithm that performs well, that is, finding one that is more advanced than the existing ones.
3. Generation of good moves in a sensible period of time.
4. Providing debugging facilities to be used later.
5. I/O operations with issues of robustness addressed.

Building an application which is flexible enough to generate more than just *repetitive* [45] outputs that correspond to the 8 by 8 game board is the more interesting aspect. We are now interested in the inclusion of more than one algorithm that can generate moves. We want to be able to manipulate these algorithms, the values that they accommodate and the way they are invoked by the user of the newly created tool. In summary, here are some of the challenges to be coped with:

1. Design of multiple algorithms that play Othello.
2. Allow for algorithm customisation, i.e. change in the intrinsic behaviour of the algorithms above.
3. External constant definitions[46] for fine-tuning[47] purposes.
4. Modification of the application's invocation methods[48].

Lastly, there is a great amount of work to be completed once the development is put aside. The following lists all in order of precedence.

1. Simulation of *Othello Master* at different modes.

---

[45] A program with no variability will resemble an FSM (Finite State Machine) that perpetually makes its decisions in the exact same manner, hence resulting in identical game results.

[46] Or more precisely, C preprocessor elements that will hold integers. These will give control over the behaviour of the program from outside the its body, that is, from its header files.

[47] This notion will be mentioned more frequently later. At the moment, let us use the analogy of airplanes being adjusted to handle a flights under stormy weather.

[48] As will be explained later, this is vital for batch-mode invocation.

2. Testing for strength by benchmarking against other relevant applications.
3. Analysis of the results obtained in (1) and (2).
4. Composition of documentation documentation for the application and its code in particular.
5. User's manual is to be built into the application's menu and command-line help options. It should be nearly self-explanatory and allow the user to use the tool as effectively as its developer. This task clearly has some implementation bias.