Fast and Furious Game Playing: Monte Carlo Drift

Design report

Prateek Bhatnagar, Gabriel Prevosto,
Benoît Viguier

Supervisors: Nikolaos Parlavantzas, Christian Raymond

02/02/2015

Contents

1 Introduction

The project is called *Fast & Furious Game Playing, Monte Carlo Drift*. Its purpose is to create an Artificial Intelligence able to compete against humans using the *Monte Carlo Tree Search* algorithm.

We have chosen the game Arimaa because it is a two-players strategy board game not solved[1].

A human plays a game by thinking of all possible moves as per one's imagination and then opts for the best amongst them. Before taking ones turn, a player can visualize ones options and predict how an opponent will counteract them. The algorithm will do the same by building a search tree containing the different posibilities. The Minimax algorithm does it by exploring all possibilities, which is heavy. The MCTS algorithm is lighter and converges to the Minimax algorithm, therefore it has been chosen for this project.

This algorithm will be parallelized in order to optimize it in a set of multi-core machines, allowing it to go further into the search tree, thus improving its efficiency.

In this report, the data structures that will be used for the development of the project are discussed. A parallelization strategy will be finalised from the various strategies that have been discussed till now by referring the test results. Various parallelisation frameworks viz. OpenMP and MPI.They will be tested and the most optimal framework for the project will be chosen for implementation of the project. A study for the GUI of the project is done with the help of UML diagrams including the detailed explanation about the various interfaces that will be used.

[1]A game solved is a game where good algorithms are able to find the perfect move in each situation to win, or to draw. For instance, *Tic Tac Toe* or *Draughts* are solved games.

2 Base algorithm

The base algorithm is a ~~simple~~ implementation of the MCTS algorithm. All the functions related to it are included in the mcts namespace. It takes as a parameter the abstract class *TheGame*, the position (*Bitboard*) to start the search with and an object (*MctsArg*) to set the parameter to the MCTS algorithm such as the time to search on, the depth of the tree, the number of simulations to run and the number of nodes to create in the tree. The following Class Diagram1 represent the links between the differents kinds of objects.

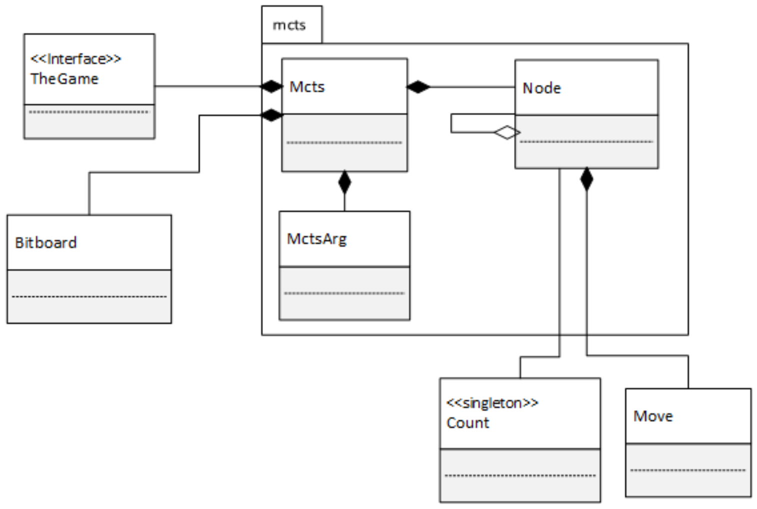


Figure 1: *Class Diagram of the mcts namespace*

The class Node represents the tree, it is stored as an array in the mcts object. A detailed implementation is provided in the Data Structure part.

The singleton *Count* is used for the statistics (such as the number of leaves, nodes created and the number of ~~simultions~~ run). Whilst providing statistics it also makes sure that no memory leaks are happening, monitoring creations and destructions of the main objects.

The following figure2 represents the order of the call for the main function while providing some details about the implementation.

```
Move GetBestMove() {
      explore() {
         While {                                       // loop simulations
            UpdateNode(node);                          // expand the _root
            While {                                    // explore the tree
                  node = select_child_UCT();           // select the node with UCT
                  UpdateNode(node);                    // expand the node
            }
            If node non terminal
                  result = playRandom(node);           // run random simulations
                        update(result);                // feedback the results
            elseif winning move
                  node->forceSetUCT(10);
                  feedbackWinningMove(node);
                        updateLosingParent(node);
                  update(result);
            else
                  update(result);
         }
      }
      _root->select_child_WR()->getMove();
      // chose the child with highest the winrate and return its move
    }
```

Figure 2: *Overview of the implementation of the function GetBestMove()*

explore : start the tree exploration.

UpdateNode : make sure the node has children and update its terminal value if required.

select_ child_ UCT : go through all the children of a node and return the one with the highest result at the UCT function (refer to the *pre-analysis report, part 3.4.4*).

playRandom : start to run the random simulations and return the winner/tie.

update : update the node statistics and feedback the result to its parents.

forceSetUCT : In the event of a winning move, the uct value is set to 10 in order to make sure that each and every further explorations go through that node.

feedbackWinningMove : feedback the winning move to its parent, one does not want to go to that position because it would be a winning strategy for the opponent.

updateLosingParent : in the event of all the siblings being a losing move, update its UCT value to winning strategy in order to propagate the results.

select_ child_ WR : return the child with the highest win rate.

3 Data structure

Given the size of the data our software will be working on, it requires an efficient way of storing them. There are 2 kinds of data, on the first hand the ones the programs works on; on the other hand, the parameters and utilities.

3.1 Representation of the playouts

3.1.1 Bitboards

Boards are stored as bitboard. As the game is played on a 8×8 board, it is convenient to use a 64 bit integer to store the position of the pieces. That way, each and every kind of pieces are stored on the same x64 integer, saving space as opposed to a matrix 8×8 retaining all the informations. Players own rabbits, cats, dogs, horses, camel and elephant; thus using 6 integers for each player do the job. Adding an additionnal bitboard to store the position of every pieces of each players helps to increase the speed of the algorithm by reducing the number of test required to be done during the playout phases. It also allows quick tests and modifications such as bit twiddling given the nature of the data.

3.1.2 Nodes

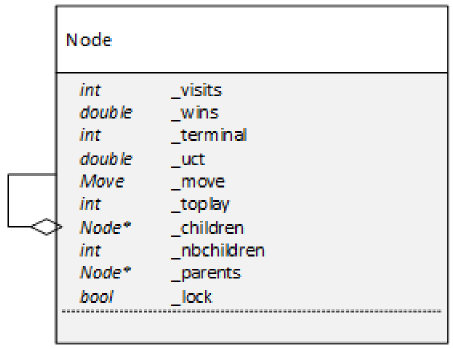


Figure 3: *Details of the data cointained in a node.*

Nodes contain statistics about the previous results, a pointer to their parent, a pointer to the first of their children and the number of children they own. They are stored in an array (*_tree*) set at the begining of the program, thus grouping them on a countinuous memory segment. Therefore the time to gain access to them is decreased.

3.1.3 Prunning

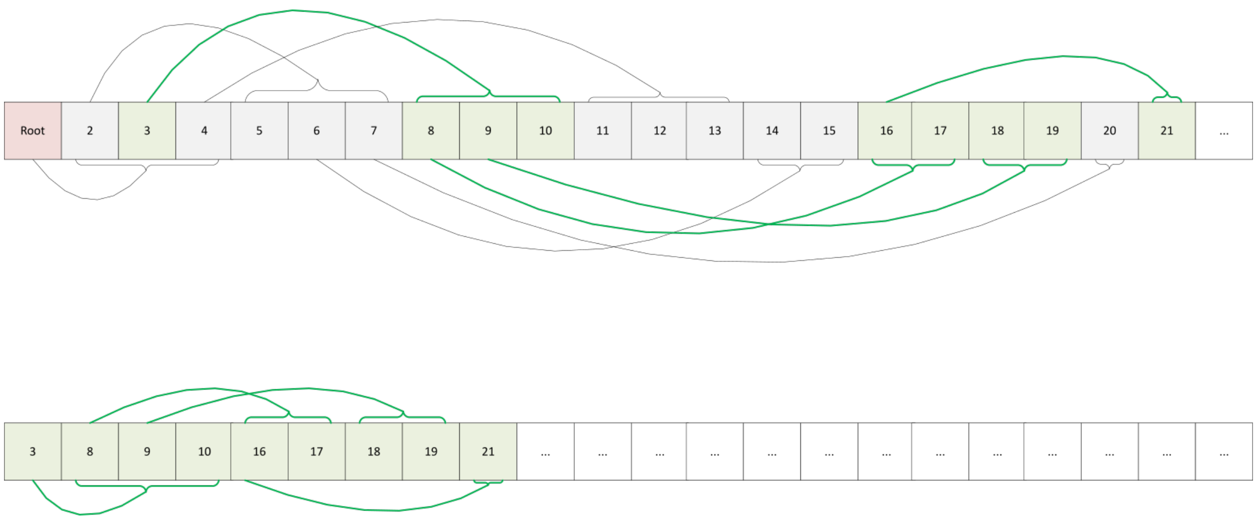


Figure 4: *Prunning of the tree (before and after).*

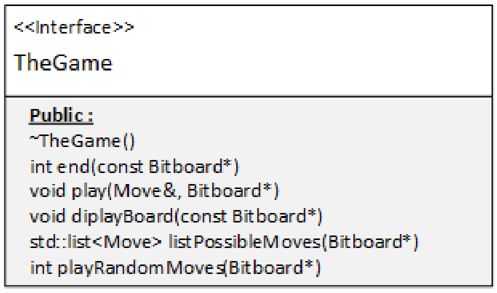
In order to prune the tree, we use the following method : create a copy of the current tree (_*tree*) into a buffer (_*buff*). The root of the buffer tree will be a copy the chosen node. Then the children are saved going down through the branches. The advantage of this method is that you only copy the nodes you want to keep. However the memory used by the buffer needs to be the same as the one of the tree before the prunning. Thus the maximum memory that can be used by the tree (_*tree*) is half the memory used by the program. In order to dermine the number of leaves to be created, the program checks how much memory there is left on the computer and use a fixed percentage of it.

$$N = \frac{R \times 90\%}{2} \tag{1}$$

N = number of leaves.
R = RAM left to use.

We chose to limit the memory used by the tree to 90% of the available memory in order to not overload the RAM and to let some left for other operations such as simulations. It also allows us to make sure that the swap will not be used to store the tree as it impacts heavily on the speed of its exploration.

3.1.4 Game abstraction



```
<<Interface>>
TheGame

Public :
~TheGame()
int end(const Bitboard*)
void play(Move&, Bitboard*)
void diplayBoard(const Bitboard*)
std::list<Move> listPossibleMoves(Bitboard*)
int playRandomMoves(Bitboard*)
```

Figure 5: *Details of methods cointained in the interface TheGame.*

In order to know the moves that can be played and the winning conditions, the mcts algorithm has to have access to a class describing the game. This abstract class is implemented and passed as a parameter at the instantiation of the mcts object. It allows genericity in the algorithm and permits to use it on any board games such as Connect4 or Arimaa. This interface has 4 main methods :

end : check if the board provided is a final state.

play : play a given move on a given board.

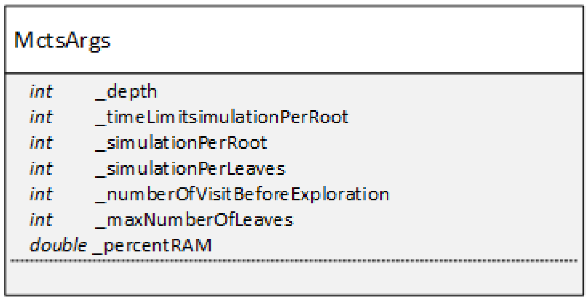
listPossibleMoves : list all the possible moves given a board, this is used when a node is to be expanded.

playRandomMoves : play random moves until a final state is reached and return the winner, this is the random simulation part.

3.2 Parameters and utilities

3.2.1 Parameters

Instead of directly passing all the parameters to the mcts object at its instantiation, we decided to create an object that would provide them, given appropriate inline getters. The point of this is to allow quick modification on the parameters without having to rewrite some parts of the mcts to make sure that each files is updated. The main idea applied here is to separate the data from the algorithm with the same principle as the MVC design pattern.



```
MctsArgs

int    _depth
int    _timeLimitsimulationPerRoot
int    _simulationPerRoot
int    _simulationPerLeaves
int    _numberOfVisitBeforeExploration
int    _maxNumberOfLeaves
double _percentRAM
```

Figure 6: *Details of the parameters of the algorithm.*

3.2.2 Fast log

The MCTS algorithm does not require an exact value of the $ln(x)$ function in order to calculate the UCT value of a node. As numbers are stored in binary on computers, it is more interesting to get their log in base 2 and to divide it by $ln(2)$.

$$ln(x) = \frac{log2(x)}{ln(2)} \tag{2}$$

$$ln(x) = log2(x) \times \frac{1}{ln(2)} \tag{3}$$

$$ln(x) = log2(x) \times 0.69314718f \tag{4}$$

Depending on the main operating system (Windows or Linux), the calculus of $log2(x)$ will differ. On linux, a quadratic approximation is made, for more details, refer to annexes7.3. On windows, *_BitScanReverse64(&y, x)* is slightly faster.

3.2.3 Random numbers : Mersenne Twister

Given the number of playouts to be simulated, the MCTS algorithm requires a fast random number generator. The Mersenne Twister which is implemented in the STL is faster than the basic rand() function. Therefore we decided to use it.

4 Parallelization

4.1 On clusters

The parallelization strategy we have chosen to use on clusters is *Root parallelization*. *Root parallelization* consists in giving the tree that we want to develop to every thread, letting them develop it randomly without any communication with the environment during a certain amount of time, and then merging the results of each tree. It is depicted in figure 7.

This method has the great benefit of minimizing the communication between the machines, as they only communicate at the beginning and at the end of the algorithm, without needing any further synchronization. In order to apply this strategy, we have chosen a master-slave approach, with one master machine collecting the results of every other machine once they are done with their processing.

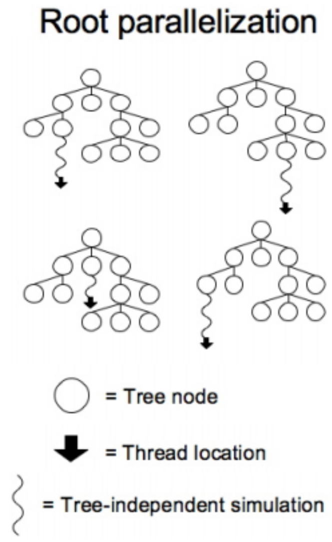


Figure 7: Overview of *Root Parallelization* [?]

The *Message Passing Interface Standard* (*MPI*) is a message passing library standard. We might use it for machine parallelization if the tests give satisfying results. In order to make the different machines communicate, it opens ssh connexions between them. The general structure of a MPI program can be seen in figure 8.

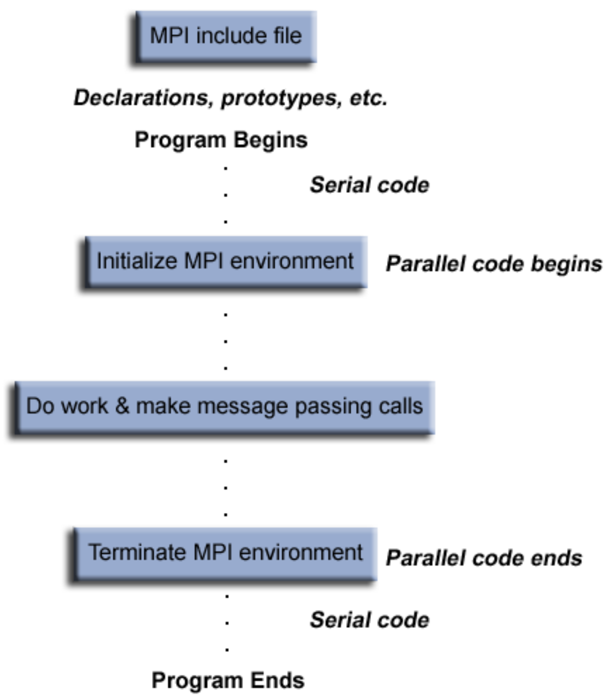
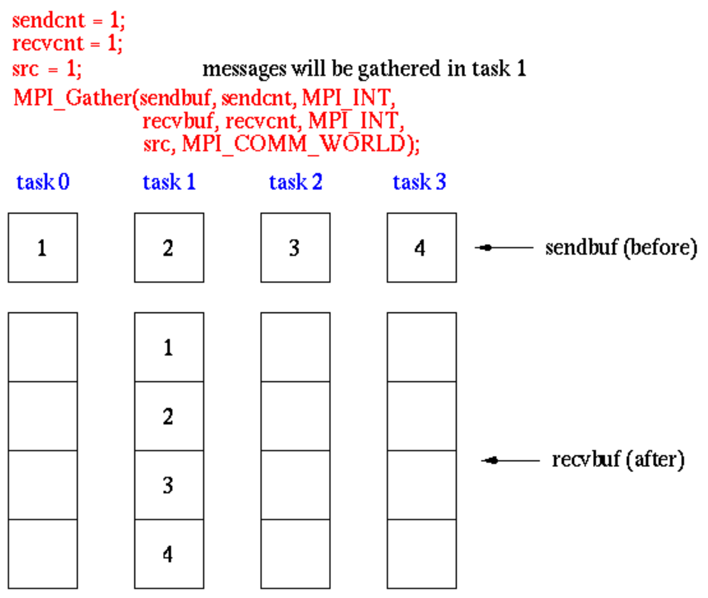

MPI include file
Declarations, prototypes, etc.
Program Begins
Serial code
Initialize MPI environment
Parallel code begins
Do work & make message passing calls
Terminate MPI environment
Parallel code ends
Serial code
Program Ends

Figure 8: General MPI Program Structure. ~~[?]~~

MPI uses objects called communicators and groups to define which collection of processes may communicate with each other. Within a communicator, every process has its own unique, integer identifier assigned by the system when the process initializes. These ranks are used by the programmer to specify the source and destination of messages.

MPI allows for synchronized, blocking and non blocking routines. Synchronized sending request only return after the message has been received, while blocking ones return after it is safe to modify the data in the sending buffer, and non blocking ones return immediately after sending the order, but offer no guarantee that it has already been executed.

Another type of routines handled by MPI is Collective Communication Routines. They allow to make a similar operation on every process of a given communicator. For example, it is possible to spread data from a single process to each other process, or on the contrary to regroup data from every process into a single one (as represented in figure 9. This will be especially useful for the algorithm, as it will require to regroup the results of every process.



```
sendcnt = 1;
recvcnt = 1;
src = 1;                 messages will be gathered in task 1
MPI_Gather(sendbuf, sendcnt, MPI_INT,
           recvbuf, recvcnt, MPI_INT,
           src, MPI_COMM_WORLD);
```

Figure 9: Example of a Collective Communication Routine. ~~[?]~~

4.2 On computers

Given the size of the tree to be explored (at least 400 million nodes), a root parallelisation strategy on computers could not be applied. A 11 million nodes tree require about 2 Go of RAM. Should it be timed by 40 and the number of core, there would be not enough RAM.

	Before	After
Percentage of memory used	39%	93%
Total physical memory	3981 MB	3981 MB
Free physical memory	2411 MB	255 MB

Comparison between the RAM available before and after the creation of a tree (and its buffer) composed by 10 940 207 nodes.

With this in mind, a tree parallelisation strategy has to be implemented on the computer. In this method, multiple threads share the same tree, the exploration is done the same way as it is with one thread.

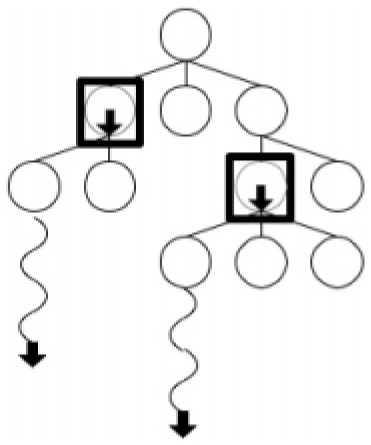
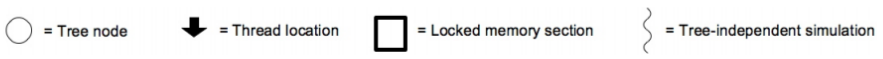


Figure 10: *Tree parallelization*



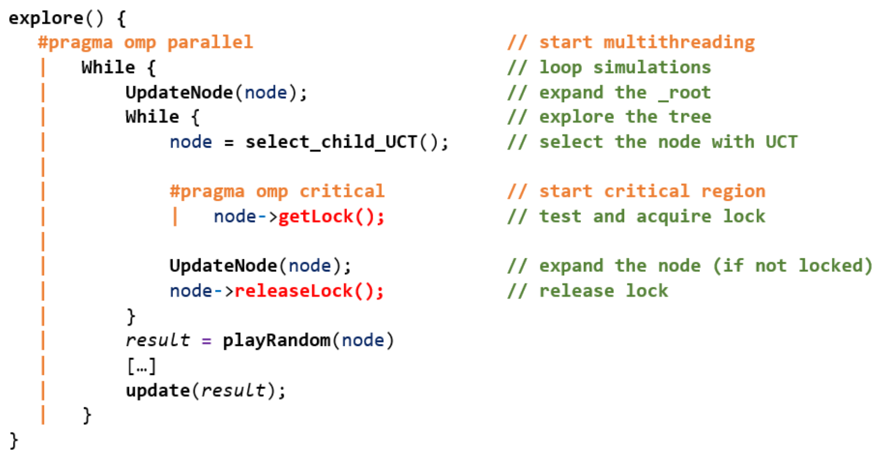
The main problem of this method is that multiple threads can access the same node and corrupt data given by another thread. To avoid such problems, we will be using *virtual loss* and *local mutexes*.

Should a thread go through a node, it automaticaly increases the value of its visit counter. This will virtually decrease the winning rate of the node and therefore decrease its attractiveness.

Each node also has a boolean to serve as a lock in order to prevent concurency. Its access is done in a critical section (atomic capture). Should a thread access a locked node, it will stop exploring and start its simulations.

This method of parallelization can be done by multithreading the *While // loop simulations* descriped in the base algorithm implementation11. Using OpenMP, this is easily implented with a single #pragma statement :
#pragma omp parallel shared(i,timeend)
where *i*, the number of simulation run and *timeend*, the time until simulations are to be run are defined as shared variables.

In order to prevent any conflicts between threads, a small critical section has to control the lock of the nodes. Placing the following statement before *node->getLock()* do the job :
#pragma omp critical



```
explore() {
  #pragma omp parallel                        // start multithreading
  |   While {                                 // loop simulations
  |       UpdateNode(node);                   // expand the _root
  |       While {                             // explore the tree
  |           node = select_child_UCT();      // select the node with UCT
  |
  |           #pragma omp critical            // start critical region
  |           |   node->getLock();            // test and acquire lock
  |
  |           UpdateNode(node);               // expand the node (if not locked)
  |           node->releaseLock();            // release lock
  |       }
  |       result = playRandom(node)
  |       […]
  |       update(result);
  |   }
}
```

Figure 11: *Overview of the implementation of the tree parallelization using OpenMP*

5 The Graphical User Interface

5.1 Overview of the UI

A user interface (*UI*) will help visualizing the game. It will be coded in *C++* and will use the graphic library *SFML 1.6*. Its funcionnalities will include the following:

- 1 Versus 1, player versus AI and AI versus AI
- Mouse and keyboard controls
- Configurable controls and resolution
- Assisted unit placement
- Game saving and loading
- Game replay
- Fullscreen mode
- Graphical mods

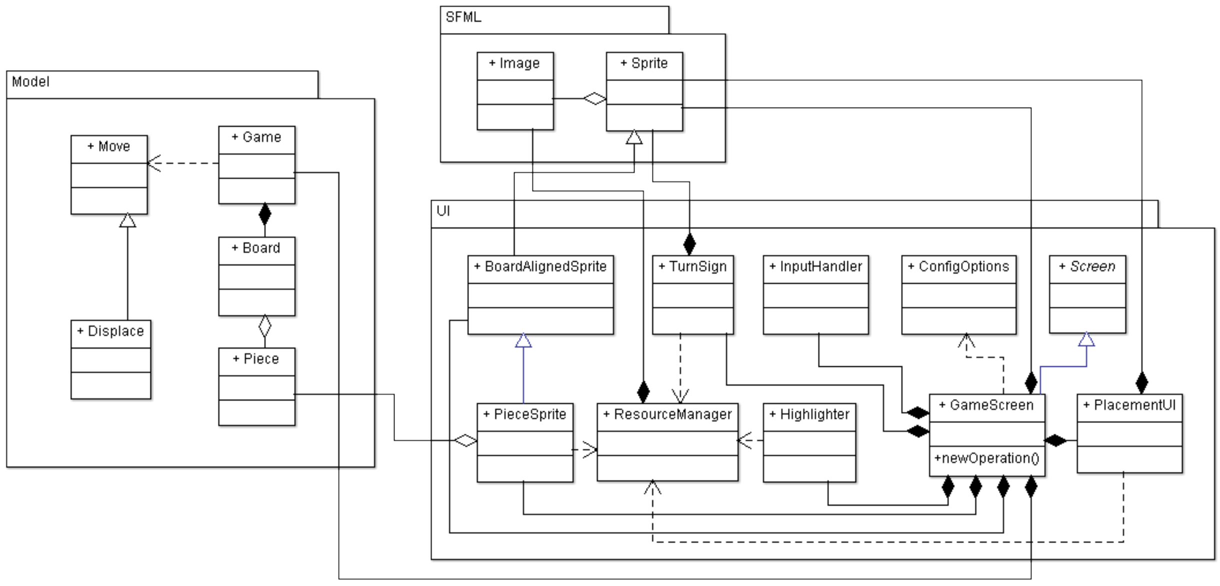


Figure 12: A UML diagram for the UI.

A UML diagram for the UI is presented in figure 12. In this diagram, you can see three modules:

- The model, rogrouping the rules of Arimaa
- The UI itself, handling the inputs and outputs, and interfacing with the Model
- The SFML, a library that loads and displays the assets

5.2 The model

The model handles the rules of the game. A game is modelized by the class *Game*. This class ~~conains~~ a *Board*, itself containing instances of *Piece*. While the *Board* allows any and all modifications to the pieces' placement, the *Game* only allows actions that follow the rules. This is done through the use of the *Move* class, representing a move in the game. Any instance of the *Move* class can be verified by the *Game*, and executed if found valid. The class *Displace* inherits from *Move*, and modelizes pushes and pulls.

5.3 The UI

The core of the UI lies in the class *Screen*. It modelizes a screen that can be displayed and updated at each frame. The *GameScreen* is the main screen of this UI : it displays the game as it is played. Among the other important classes are *ResourceManager*, that uses the *Flyweight* design pattern to manage the assets; then there is *InputHandler*, that associates keyboard inputs with strings representing the different actions, and also *ConfigOptions*, that loads the options contained in a .ini file and relays them to the application. Most of the other classes represent the different graphical elements (pieces, the cursor...) and will not be explained in detail here.

6 Conclusion

The focus of this report is related to designing of the project. The data structures that are used for the development of the project are finalised such as bit board and nodes. An attempt has been made to derive the association between the data structures, interfaces and MCTS using the class diagram.

The parallelization strategy that has been chosen is root- root parallelisation with MPI framework for clusters. The tree parallelization strategy has been chosen for implementation on the machine(computer) with OpenMP framework. The testing of both the strategies have been done successfully.

A graphical user interface is designed and is demonstrated with the help of UML diagram and is divided into three parts as the user interface, the model and the SFML.

The development of this report will serve as a base support for the concrete development of the project.

7 Annexes

7.1 MCTS Class Diagram

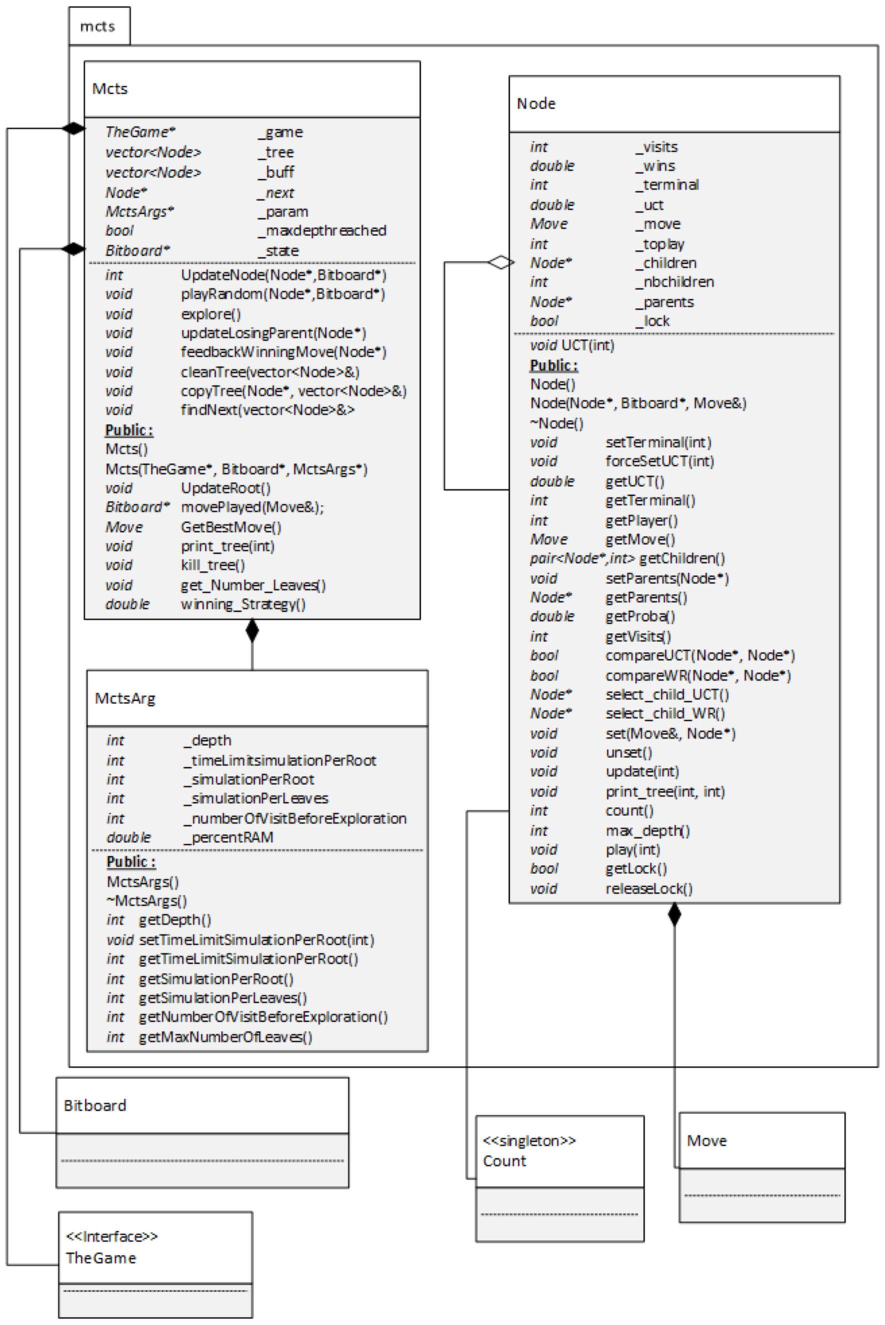


Figure 13: *Details of the mcts namespace.*

7.2 Data Structure Class Diagram

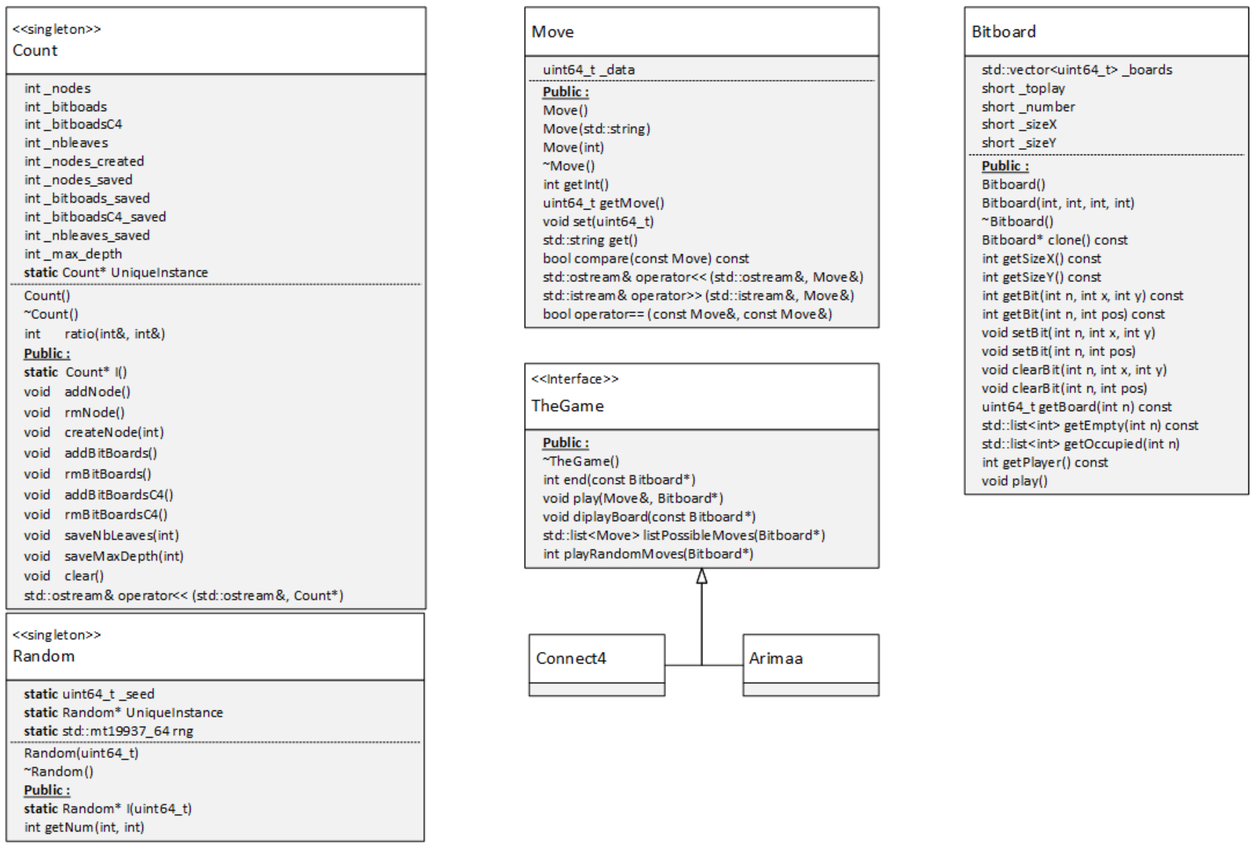


Figure 14: *Details of the DataStructure class diagram.*

7.3 Linux $log2(x)$ Implementation

```
static inline float log2(float val)
{
  int* const   exp_ptr = reinterpret_cast <int *> (&val);
  int          x = *exp_ptr;
  const int    log_2 = ((x >> 23) & 255) - 128;
  x &= ~(255 << 23);
  x += 127 << 23;
  *exp_ptr = x;
  val = ((-1.0f / 3) * val + 2) * val - 2.0f / 3;
  /*
  The line computes 1+log2(m), m ranging from 1 to 2.
  The proposed formula is a 3rd degree polynomial keeping first derivate continuity.
  Higher degree could be used for more accuracy.
  */
  return (val + log_2);
}
```