**Project Report: An Advanced Aeroplane Chess**

**Introduction**

Aeroplane chess is a modern Chinese chess game based on a kind of British chess. Like other chess game, aeroplane chess have been transplanted to computer platform, and we can find many versions of aeroplane chess game online. However, traditional aeroplane chess seems to lose its attraction among teenagers due to the restricted game mode. In this project, our group created an advanced aeroplane chess game. It can not only perform the function of traditional aeroplane chess, but also add new features to make the game totally different. Additionally, this game supports LAN battle, which means friends can play this game online. We hope by introducing this project, our generation can recall the happiness we had once upon a time.

Here is our group members’ name and contribution list:

(此处插入名单和分工表)

**Description of the game**

When you enter the beginning interface of this game, you can start a game, read the help, or exit the game. After you start a game, you can change the game mode (online/offline mode, traditional/advanced mode), numbers and type of players. （这里加图）

Game mode:

There are two game modes: traditional mode and advanced mode. Traditional mode is not different from the most common aeroplane chess game. As for the advanced mode, we add an ability card system to the game, which allow players to use certain ability cards to hinder enemies or buff their own chess to win the game. The mechanism of ability card system is as follows:

1. When players roll the dice to 1 or 6, the player get a chance to draw an ability card instead of moving a chess. One player can only possess at most 1 card at a time. If the player draws a card when having a card already, the original card will be replaced by the new card.
2. One ability card can only be used once, and one player can only use ability card once in one turn. When the player chooses to use ability card, then he or she cannot roll the dice or move chesses anymore.
3. There are four kinds of ability cards, and each of them can be drawn with equal possibility. They are:

* Attack: assign this ability to one OWN chess. Enemy chess less than 5 steps far from this chess will be crashed (send back to airport) unless they are protected.
* Defense: assign this ability to ANY chess. This chess will not be attacked (but still can be rammed) or be interfered in 5 rounds.
* Interfere: assign this ability to ANY chess. The chess cannot move/attack/defense in 4 rounds.
* Eliminate: assign this ability to ANY chess. The buff state (defensed/interfered) of the chess will be eliminated.

**Basic Framework**

There are three main components in this project, which are main interface, AI core and LAN system.

（这里需要图）

* **Main interface**

**Start\_Scene**

**HelloWorldScene**

**plane**

**dice**

**win\_judge**

**End\_Scene**

**card\_generator (Advanced mode)**

**card\_slot (Advanced mode)**

* **AI core**

The inspiration for this AI core comes from a design of Chinese chess AI. This aeroplane chess AI consists of three basic classes: the move generator (movegenerator.h / movegenerator.cpp), the evaluator (evaluator.h / evaluator.cpp) and the search engine (searchengine.h / searchengine.cpp). Move generator is a class which can generate and store all possible moves according to the current game state. Evaluator is a class which can evaluate the value of a specific move; the larger the value, the better the move. Search engine is the main class for this AI core; it combines initialized move generator and evaluator object in order to simulate a real player’s action. In addition, the AI core contains a define.h in order to store some common data structures and macros for the above three classes. Detailed implementation are as follows:

**define.h**

This head file defines some necessary data structures and macros for the AI. For example, the representation of chess ID, chess colors, chessboard coordinates and etc. They are configured as some macros for the convenience of programming.

There are three important data structures: CHESS, CHESSMOVE, and COORDINATE. CHESS is a structure containing the ID (representing the chess uniquely), the color, the coordinate, buff state and buff round left of a chess. CHESSMOVE is a structure containing the ID of the chess being manipulated, the roll point and the information about ability card using. COORDINATE is a structure containing a string representing the region and an integer representing the location. The reason for using this coordinate representation is that the chessboard of aeroplane chess is not full-covered. Some points on the two-dimensional plane is out of the chessboard, and chess will have different behavior in different region of the chessboard. Therefore, I did not choose two-dimensional coordinate in the convenience of computation.

**Move generator**

When creating a move generator, the constructor will set off mode (deciding what roll point can take off), set the game mode (traditional mode or advanced mode), and set random seed. Setting random seed when initializing will let the dice roll result different every time.

The core function for move generator is createPossibleMove(). This method generate currently possible move according to current chessboard, roll point, the color of the player and ability card the player have. Each possible move is constructed into a CHESSMOVE structure and is stored into the moveCount array.

**Evaluator**

Evaluator only have one function evaluate(). The main logic for evaluating is to compare the chessboard before and after a certain move. If the new chessboard results in a favorable situation, the value will be higher. If the new chessboard confronts a bad situation, the value will be lower.

Because of the mechanism of aeroplane chess, the evaluation process is basically a simulation of human player’s thinking mode. First, the evaluator counts benefits of a move. Benefits include bonus from using ability card efficiently, crashing enemy chess, moving own chess ahead, etc. Then the evaluator counts damages of a move. Damages include the threat from enemies and punishment from overlapping. The final value is counted by benefits subtracting damages.

**Search engine**

This is the main part of AI. When initializing, the search engine must point to a move generator object and an evaluator object in order to function. The play() method will let the AI change chessboard intelligently. This method simulates player behavior like this:

1. Copy the chessboard into cur\_Chessboard which is inside the search engine instance;

2. Roll the dice. Then create move with createPossibleMove().

3. If there is only one move, it must be "not possible to move" case. Then do not do any movement. If there is more than one possible move, search a good move and make move with three methods, searchAGoodMove(), makeMove(), useAbility().

4. If game is not over and roll point is six, continue until the third time roll. Else, player end its turn.

5. Copy the modified chessboard in place to the original one.

searchAGoodMove() will evaluate all possible move and choose the move with the maximum value. makeMove() and useAbility()will modify chessboard according to the move AI choose.

* **LAN system**

Implementations on LAN system is a failure. Difficulties that we have encountered and the solutions to them, however, are also treasurable for us all. Likewise to those problems we have successfully tackled, the main reasons accounting for our failure are valuable and priceless, too.

Original Plan & Failure

My original plan had three sections which covered database designing, server implementations and the linkage to the main program. I loosely set two weeks as a separation node of each section to avoid the collision of final examinations and this project. I booted my battle on March 23rd, and actually I succeed in the first two sections. In late April, I have accomplished a versatile but not powerful server implemented by a library Boost::asio, and a mediocre database implemented by a library “Muduo”. By then I did not realize how severe a situation I was in, which was that I could not encapsulate a proper gaming protocol that satisfied this game. The lack of knowledge on data serialization made me hard to implement a proper “MessageType” Structure, and the separation modeling on human players and bot players also served as my harassment. Plunging into huge volumes of debugging received no achievement I aborted my LAN system. I concluded my project as an epic tale, because I had no regrets on efforts I paid and lessons I learnt from both success and failure.

Problems encountered & Solutions

* 1. Database Design

The general idea of the database designing was to implement a register-and-login system. Only when a player has registered and was validated by my checking system could he login successfully and play games. To simplify the validation progress, at first I decided to restrict the only two register fields “Student’s Name” as accounts and “Student’s ID” as passwords. If a student registered, it read a file containing all mappings of students’ names and the responding IDs and verify it. However, my request for the file was rejected. After that I realized that it was little use to restrict students’ names, I could only check their student IDs and compare them with certain formats. For instance, undergraduate students, master students and doctors are assigned 1,2,3 as their first digits. Using this certain formats, I used regular expressions on <regex> library and made an expression of student ID like [1|2|3]1[4|5|6|7|8]0[1|2|3]\d{4}. This section is sort of easy.

* 1. Multithread

The idea of using multithread was when your server’s listen socket was set in a SYN\_RECV blocked state, once a client did not response SYN+(ACK +1) to your server, then your server socket would be blocked in this state and all work would terminate. I created a nested-socket for my server, one was for a overall global listening thread which sent heart-beat packages and invoked the clients sending socket, and the other one was for the read/write IO within clients’ read/write socket. If I wanted to break a connection between one client because of exploiting, I could kill one thread including the read/write IO with him instead of killing outer clients. If the outer thread got lame or dropped, we could refer to the error logger for further reasoning. It was a pity that I was not familiar with synchronized locks, but It happened that all read/write thread from tested clients were balanced and took up almost the same amount of CPU resources! I did not know whether it was a coincidence, but according to several tests on a tested client I had good outcomes which were enough for my satisfaction.

* 1. Asyn IO & Coroutine

During the process I found a book entitled《Linux多线程服务端编程》by accident, and I was greatly inspired by this book. Large scales of throughput among web services could not be handled with continuous piling of threads, and we needed to use one nonblocked asynchronous thread to drive one file descriptor for one event. In order not to lose control of different asynchronous threads, coroutine was needed to provide an await function for threads to submit their resources and guaranteed message pipes’ safety on fibers when ported to different platforms. Because coroutines were ill-performing when the protocol was CPU-bound, I modified and used Muduo’s Library Channel:: ReadEventCallback to compensate this situation.

Reasons of failure

* + 1. Opinions Bifurcation on core business

The main reason for my failure was that I had misunderstood the core business, which meant that I had been devoting my efforts and hard work on a similar but totally different branch from our groupmates. I was too well-planned and progressed too smooth and probably because of that, it took me some time on database designing, only to find it useless when I realized that every gaming round would be flushed as all players, planes would be reset. The register-and-login system was somehow useful, but posed little use when no validation would be taken on players. And it took me too much of time on server developing, racking my brains to understand each token on Boost::asio with such a long, dull, harassing function names. However, gaming servers had a shared core business that synchronize states, which were serialization on game structures to message streams, channelization through a encapsulated structure MessageType, deserialization to update states. I misunderstood our purposes, which accounted for my failure.

* + 1. Shortage of experience on network programming

As a layman to network programming, my learning was very smooth. However, I was lacking in the overview of socket programming and PC games development. I neglected all those differences between “High performance Server Development” and “Online Gaming System Implementations”. The former focused on portability, responding speed and stableness on high concurrency within large scales of IOs, but the latter was what our business acquired indeed, focusing on nothing but a clear data package flowing between clients and servers. The trickiest part was not Asyn or Coroutine, but the design of gaming protocol and how to serialize the structure of a human player’s plane and a bot’s plane. This task was nothing difficult but complex and heavy-loaded, as I had run out of my time to learn something new and make any difference. It was a common sense for experienced developers to build MessageType and gaming protocol, but for a layman it dawned on me just too late.

Further Exploration

The summer vacation will be a wonderful time to reboot my network tasks. At present I am planning a long-lasting, through, systematic learning on network programming, and will come up with a solution as a delayed triumph.

**Reference**

Guo Zijian. (2009). Chinese chess online game system.

“random.h” and “random.cpp” from Stanford C++ Libraries. Retrieved from https://stanford.edu/~stepp/cppdoc/

GitHub

Cocos 2d-x documents. Retrieved from https://docs.cocos2d-x.org/cocos2d-x/en/