Arbejdsblade for vejledermøde d. 20.02.13

Gruppe d402f13

19. februar 2013

1 Introduction

Board games have been played throughout almost all of history, within different cultures, societies, and countries. Backgammon is known to be at least 5,000 years old, and is still played today. Board games can consist of a board, some player pieces or tokens, a deck of cards, and/or dice. The given player or players follow a set of rules in an attempt to achieve a goal. Strategy and luck are usually involved in some form so playing the game more than once is engaging and dynamic. Most board games only involve one winner at the end of the game.

Different categories exist and board games can be placed into different genres, such as: strategy, alignment, chess variants, paper-and-pencil, territory, race, trivia, wargames, word games, and dozens of others. Obviously some games can overlap genres. Chess is an example of this. It is obviously a chess variant and is very strategy-heavy.

Programming languages exist to create programs that express algorithms to control the behaviour of machines. Most board games, as we will demonstrate, have specific rules and exact winning conditions to follow, which can be described with algorithms in a programming language to a very detailed degree.

1.1 Why board games?

So what do board games have to do with designing, defining, and implementing a programming language? If you're able to describe how board games work on a very generic and general level, it would theoretically be possible to abstract away from that to describe all games in general; something that could be very beneficial for game designers in many different ways.

If a language purely concentrates on allowing the programmer (i.e. game designer) to express how his game works, it would be quicker and easier to pick up and develop shorter and more precise programs rather than having to reimplement everything from scratch in an already existing high-level language. Furthermore, data structures and special statements specifically designed to help define board games would greatly increase the readability and writability of such a program.

A language designed with board games in mind would also allow the game designer to, relatively quickly, explore new ideas for a board game with a simple implementation. He/she could then efficiently modify the code according to a new rule or idea, and the implementation would stay exactly the same. If the language also took multiple platforms into account, it would open up the possibility to run the same game across multiple devices.

This could enable AI creation for the language that understands the rules, so you can test an early implementation of your game without the need of other human players.

An example could be four-person chess. If you already had an implementation of chess in the programming language set up with a board and separate rules for each piece,

then it could be as simple as changing the piece location and player count (and maybe editing the rules for one or two of the pieces so it's a little more fair) followed by running the program again.

1.2 Problem Statement

To help create a concise description of where the project is headed, we create a problem statement that we wish to see fulfilled. This problem statement is written as a question, which will be answered as part of the conclusion. It is stated as follows:

> How can we define, design, and implement a new programming language that aids programmers in defining rules for board games in general?

2 Analysis

2.1 Board game analysis

One may wonder what a board game really is. Could it just be any game containing some kind of a board? If so, would Trivial Pursuit be a board game and what about the game Twister, where you have to place your hands and/or feet on a spot marked with a particular color on a sheet - or board, as you could call it. Most people have a mental model of a board game that does not include games like Twister. Here is one definition of a board game

> "A board game is a game played across a board by two or more players. The board may have markings and designated spaces, and the board game may have tokens, stones, dice, cards, or other pieces that are used in specific ways throughout the game." [**?**]

The definition above is very broad and will to some extend allow a game like Twister to be categorized as a board game. All kinds of widgets like cards and dice can be contained in a board game, but one board game designer may also be able to invent a new and yet unseen widget, which he wants to include in his board game. A programming language that makes it possible to describe any board will cover a very broad category of games. You could argue that it would actually cover all games that can be made, since even a first person shooter could be played across a board. With such a broad definition, a programming language that aims to make the programming of board game easier, will be a programming language that aims to make almost everything easier. If a programming language aimed to make the programming of only a very specific kind of board games easier, there might be many things that can be optimized compared to existing general purpose programming languages.

2.2 Chess, Kalah and Naughts & Crosses

In the following sections a detailed analysis of the three games: Chess, Kalah, and Naughts & Crosses will be performed. The reason for picking these three games in particular, is first of all the fact that none of the games includes dice, cards or other related objects, accept for it's pieces and a board. The second reason why we pick these three games is because they are among those we have biggest personal interest in. Therefore we want to dig deeper into the details of the components of these games (e.g. the pieces, the board, the squares etc.) to gain a better understanding of which features are needed in LUDUS. The respective history of the games and other related information will not be included in the analysis, since this has no relevans for gaining understandig of how our programming language could be designed.

Figur 2.1: *The board game chess with the pieces in start position.*

Chess

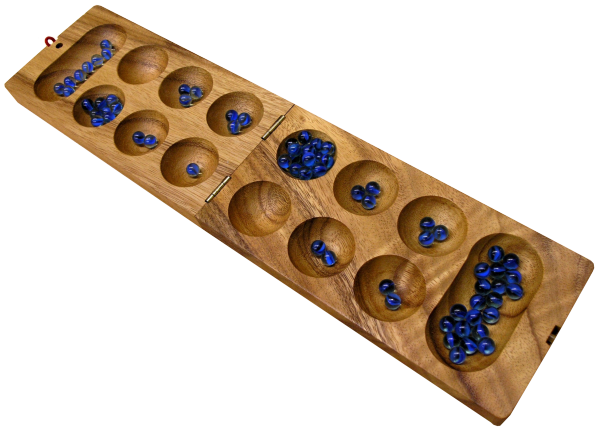
Chess is a board game of two oppenent players. It's a turn-based game which means one player makes a move, then the other player makes a move, then the first player makes a move and so on. Chess is played on a board of 8×8 squares. The squares are typically black and white, but can be any two colors (see figure 2.1). The squares can only contain one piece at a time, unlike games like Mancala and Backgammon. Each player has a total of 16 pieces: 8 pawns, 2 knights, 2 bishops, 2 rooks, a queen and a king. Each type of piece has unique ways to move. For instance a pawn can move only one square vertically forward or one square diagonal when capturing an enemy piece. A rook can move unlimited squares either forward or backward (vertical movement), or to the right or to the left (horisontal movement). This separates it's pieces from a lot of other board games where all pieces have the same abilities e.g. Naughts and Crosses, Mancala, Ludo, Backgammon etc.

Cut to the bone the game goes as follow: When a game starts the pieces are in their respective starting positions as seen in figure 2.1. The player with the white pieces always makes the first move, and after that the players shifts in turn in which clever moves are beeing taken and pieces are beeing captured until one player has checkmated the other - and the game is over. The checkmate situation is obtained when the king piece is in a position to be captured and cannot escape from capture. [**?**].

Special situation and moves. In chess there are numerus special situations and moves which doesn't follow the normal pattern of chess. Earlier we mentioned that a pawn can only move only one square vertically forward or one square diagonal when capturing an enemy piece. But this is not always true. If the pawn is in it's respective starting position it can move either one **or** two squares vertically forward. After moving from it's starting position it can only move one square forward the rest of the game. Another special move is the move called "castling". This move allows a player to move two pieces in one turn (the king and one of the rooks). But to do the move several conditions needs to be met. First: the move has to be the very first move of the king and the rook, second: there can't be any pieces standing between the king and the rook and third: there can't be any opposing pieces that could capture the king in his original square, the squares he moves through or the square he end up in [**?**].

Possible requirements

- pieces with different movement abilities.
- A square board with a number of squares in it.



Figur 2.2: *The board game Kalah*

- A winning condition - when the king has been checkmated.
- A starting state - how the pieces are placed on the board before the game's very first move.
- Defining special situations like the "castling"

Kalah

> "The object of the game is to capture more seeds than one's opponent. At the beginning of the game, three seeds are placed in each house. Each player controls the six houses and their seeds on his/her side of the board. His/her score is the number of seeds in the store to his/her right. Players take turns sowing their seeds. On a turn, the player removes all seeds from one of the houses under his/her control. Moving counter-clockwise, the player drops one seed in each house in turn, including the player's own store but not his/her opponent's. If the last sown seed lands in the player's store, the player gets an additional move. There is no limit on the number of moves a player can make in his/her turn. If the last sown seed lands in an empty house owned by the player, and the opposite house contains seeds, both the last seed and the opposite seeds are captured and placed into the player's store. When one player no longer has any seeds in any of his/her houses, the game ends. The other player moves all remaining seeds to his/her store, and the player with the most seeds in his/her store wins. It is possible for the game to end in a draw, with 18 seeds each."

When comparing Kalah and the game elements involved to other board games, some interesting requirements for a generic board game programming language able to describe this particulary game arises. The requirements described here consider seeds as pieces and the houses as squares, which makes it easier to compare the requirements of Kalah to the requirements of other board games later.

- Squares can contain an arbitrary number of pieces

- Making a move can be considered as simply choosing a square
- A turn may contain more than one move
  - If the last piece of your move is put on a specific square, you can make another move.
- A square can be related to a player.
  - A winner is found based on who has most pieces in a special square.
  - On your turn you can only choose a square beloning to you.
- Squares can be related to other squares.
  - You place pieces on squares counter-clowise.
  - If you place a last piece on an empty square, the nearest square controlled by the opponent is emptied.
- The game ends when some condition is met
  - All pieces are on one specific square.
  - If you place a last piece on an empty square, the nearest square controlled by the opponent is emptied.
- When the game ends, a winner is found
  - The player with most pieces wins
- Only one type of pieces
- The number of pieces on a square determines how long a move you can make

Naughts & Crosses

Naughts & Crosses is like Kalaha and Chess a game of two opposing players. The game is played on a board with 3×3 squares. Each square has the same properties with no acceptions or special situations to ruin this.

possible requirements

- Turn-handling.
- A start condition

2.3 Requirements

2.4 Compiler versus interpreter

Along with the design of the programing language LUDUS, we also want to make it possible to play a game once it has been written in LUDUS. There are a number of ways in which we can make people capable of playing the games written in LUDUS. Some of the options can be seen here as well as advantages of each:

- A LUDUS compiler that targets a specific system arcitechture and creates a playable game written in machine code
  - Pros
    * Gives us full control over what machine code that will be produced.
    * Makes us able to design for a specific platform and advantage from the features provided by the platform. E.g. styling, use of right click on mouse.

  - Cons
    * For every machine arcitecthure we want the LUDUS programming language to be able to compile to, we must make a compiler that targets that specific machine arcitecthure.
- A LUDUS compiler that compiles to an intermediate language, for instance C.
  - Pros
    * Lets LUDUS target many different platforms if the compiler compiles to a language that does so, C e.g.
  - Cons
    * We cannot fully control what kind of optimisations is done by the intermediate language compiler.
    * We can only target those platforms that is targeted by the intermediate language compilers.
- An interpreter that provides features for simulating all games written in LUDUS.
  - Pros
    * We can at all times modify our interpreter, for instance if we want to include new skins for the board, allow networked multiplayer, e.g, without people need to recompile their games.
    * The interpreter can be written in a language like C that targets many different platforms.
  - Cons
    * Slower execution of games written in LUDUS, since they must be interpreted at run time.
    * An interpreter is needed to play games written in LUDUS. A finished game can however be combined with the interpreter as a seemingly single executeable.

2.5 Code examples