Laboratory of Artificial Intelligence and Robotics

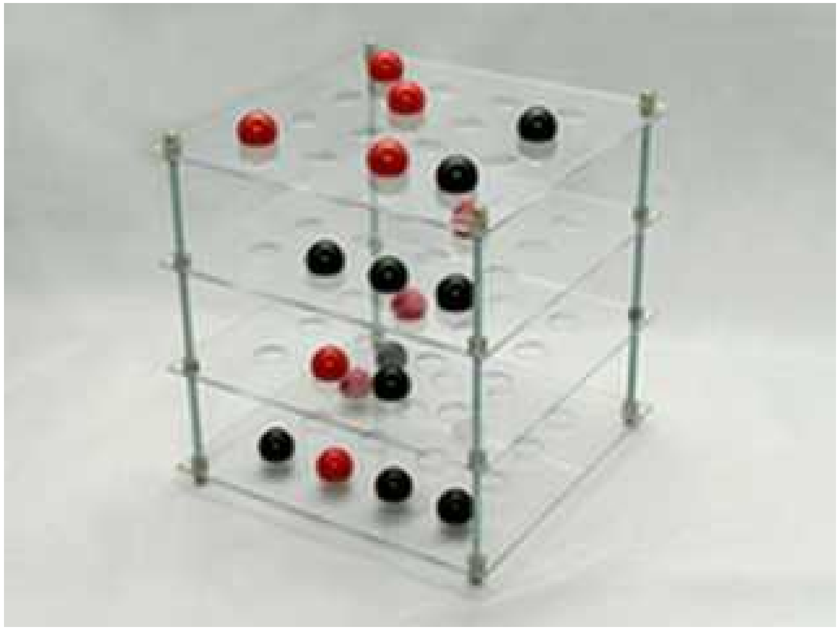
by Lucas De Marchi

Outline

- About the game
- Project goals
- Complexity
- Implementation
- How to add your own Player algorithm
- Minmax
- Future work
- Hands on!

About the game

- Tic-tac-toe 3D
- Actually, a little more: NxNxN tic-tac-toe
- The very same basic rule applies: the goal is to align N pieces in whatever direction, where N is the so called dimension of the game
- A real 4x4x4 board (University of Sao Paulo - Brazil):



Project's goals

- Implement it as a computer game
- Give the possibility to play:
    - Human vs Human
    - Computer vs Human
    - Computer vs Computer
- Create a Minmax-based algorithm to play the game
- Ease the creation of further algorithms
- Create a 3D interface to better visualize the output

Complexity

- Let n be the dimension of the game
  - The first piece can be placed in n^3 different places
  - The second piece can be placed in $n^3 - 1$ places
  - These two iterations created $n^3(n^3-1)$ states
  - Going on with this reasoning (without accounting for symmetry) the number of possible states is:

$$n^3 \cdot (n^3-1) \cdot (n^3-2) \cdot \ \ldots \ \cdot (1) = (n^3)!$$

Complexity

- Let's see the number of final states for n even. After the last possible the board will be full and since the player take turns, turn there will be $n^3/2$ pieces of each one. It's a k-combination problem.
- Number of final states (NFS):

$$NS = \binom{n^3}{\frac{n^3}{2}} = \frac{n^3\,!}{\frac{n^3}{2}! \cdot \frac{n^3}{2}!}$$

- For n = 4, this leads to roughly 1.83E+18 final states

Complexity – the goal test

- Inside a cube with side n there are 13 directions in which it's possible to align N pieces
- A “goal test” function which has the information of where the last piece was put, must just check these 13 possible directions starting from this position.
  - Example: a possible direction is [1, 0, 0]. If the last player put a piece on [2, 0, 1], we can sum and subtract the direction until reaching the cube's border. If the number of pieces counted and the dimension are equal, the last player has won.
- So, in the worst case we expend 13n operations to decide if the game is finished and who is the winner. Therefore the complexity is O(n).

Complexity – another goal test

- If we don't have information of the last position, we have to test all directions applied to all points and keep track if a certain direction already “passed through” a certain point.
- The complexity is always the same, since we have to scan the entire board every time its configuration is changed. Simply counting all the possible alignments of n pieces we get:

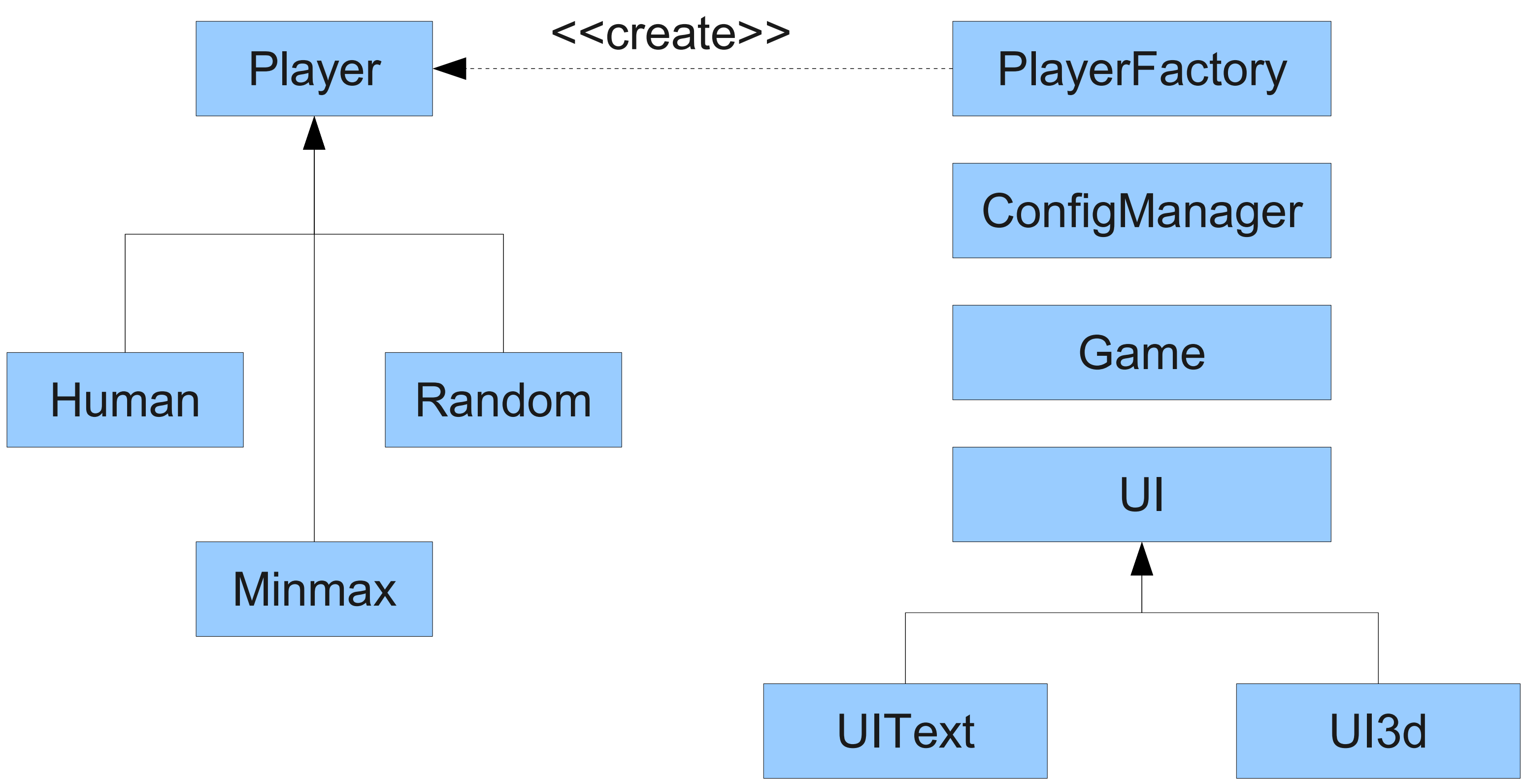
$$3n^2 + 10n$$

operations, thus having an $O(n^2)$ algorithm.

Implementation

- Powered by:

Implementation – Design



Implementation – Design (more detailed)

How to add your own Player

- You have to do only 3 steps:
  i. Implement the abstract class Player
  ii. Compile it into a shared library
  iii. Put it in the folder you configured Trissa to look for players. You can even send it to another person on another computer to prove your algorithm is better
- Implementing Player is basically a matter of implementing the “play” method, called by Trissa when its turn is arrived.
- If you use Trissa's building system, these 3 steps can be reduced to the first one

How to add your own Player

```
class MyPlayer : public trissa::Player {
public:
static char* name;
MyPlayer(int dimension, trissa::PlayerType player_type)
: trissa::Player(dimension, player_type){
//Put your constructor’s implementation here
}

~MyPlayer(){
//Put your destructor’s implementation here
}

virtual trissa::Move* play(trissa::Cube const& board, trissa::Move const& opponentMove)
    {
//Your implementation of deciding in which position to put a piece
}

virtual trissa::Move* firstPlay(){
//Your implementation of deciding in which position to put a piece when your player
//is the first one to play (in general hard-coded).
}

virtual const char * getName() const {
     return name;
}
char MyPlayer::name = (char*)“My Player Name”;
REGISTER_PLAYER(MyPlayer)
```

http://wiki.github.com/lucasdemarchi/trissa/howto-player-algorithms

Minmax

- “Minmax Player” is the better player implemented until now. It uses the well known Minimax algorithm with α-β pruning.
- As noted in the complexity study, the number of states is huge. It's not practical to predict all the states to play.
- The level L in the tree indicates how deep the search will be and is configured at compilation time

Minmax

- When level L is reach, a state evaluation function is performed to decide how good is that state
- The following heuristic is used in this implementation:
  - If there are n pieces aligned
    - Return INF if pieces are mine or -INF otherwise
  - Otherwise, return:

$$ret = \sum_{i=0}^{n-1} (\alpha_i - \beta_i) \cdot i^2$$

α_i: number of lines in which I have i aligned pieces

β_i: number of lines in which my opponent has i aligned pieces

Future work - Players

- Other possible algorithms or modifications to exist ones:
  - MTD(f)
  - Machine learning (WIP)
  - Fine-tune the heuristic of Minmax
  - Minmax with interactive-deepening (this allows to put a time constraint in which a Player has to return a position
  - Minmax with state caching: don't recalculate parts of the tree already scanned, this allows to have a greater depth and thus predict more states

Future work - Game

- The present game is at v0.98 (waiting some time to make sure it's stable to release v1.0)
- Planned for v1.1:
  - Porting to Windows / MacOS X
  - Audio integration
- Planned for v1.2:
  - “Network Player” which will make possible to play through Internet
  - Some others no-so-smart/no-so-dumb Players (for example a “Linear Player”
- Other algorithms?
  - MTD(f) ?
  - Machine-learning ? (WIP)

Hands on!

Source code:

- http://github.com/lucasdemarchi/trissa

Future site:

- http://www.politreco.com/trissa

LET'S PLAY!