WaveRider *2019 Jan 24*

Implement tictactoe.

```
namespace algo
{
	enum pos_state { O, X, empty }; // O for man, X for computer

	inline auto rc2n(unsigned short row, unsigned short col)
	{
		return row * 3 + col;
	}

	inline std::pair<unsigned short, unsigned short> n2rc(unsigned short n)
	{
		unsigned short row = n/3;
		unsigned short col = n%3;
		return std::make_pair(row, col);
	}

	inline unsigned short check_key(char c)
	{
		if (c == 'q')	return 0;
		else if (c == 'w') return 1;
		else if (c == 'e') return 2;
		else if (c == 'a') return 3;
		else if (c == 's') return 4;
		else if (c == 'd') return 5;
		else if (c == 'z') return 6;
		else if (c == 'x') return 7;
		else if (c == 'c') return 8;
		else		return 9; // invalid key
	}

	struct grid_state
	{
		grid_state()
		{
			for(unsigned short n=0; n!=9; ++n) value[n] = empty;
		}

		inline bool click(unsigned short n, bool is_man)
		{
			if (n >= 9) return false;
			if (value[n] != empty) return false;
			value[n] = (is_man ? O : X);
			return true;
		}

		inline bool self_check() const
		{
			unsigned short num_O = 0;
			unsigned short num_X = 0;
			for(int n=0; n!=9; ++n)
			{
				if (value[n] == O) ++num_O;
				if (value[n] == X) ++num_X;
			}
			if (abs(num_O - num_X) > 1) return false;
			return true;
		}

		inline auto empty_pos() const
		{
			std::vector<unsigned short> temp;
			for(unsigned short n=0; n!=9; ++n)
			{
				if (value[n] == empty) temp.push_back(n);
			}
			return temp;
		}

		inline bool is_all_filled() const
		{
			for(unsigned short n=0; n!=9; ++n)
			{
				if (value[n] == empty) return false;
			}
			return true;
		}

		inline signed short end_score() const
		{
			signed short output = 0;
			if (value[0]==value[3] && value[0]==value[6] && value[0]!=empty) output = (value[0]==X?+1:-1);
			if (value[1]==value[4] && value[1]==value[7] && value[1]!=empty) output = (value[1]==X?+1:-1);
			if (value[2]==value[5] && value[2]==value[8] && value[2]!=empty) output = (value[2]==X?+1:-1);
```

```
            if (value[0]==value[1] && value[0]==value[2] && value[0]!=empty) output = (value[0]==X?+1:-1);
            if (value[3]==value[4] && value[3]==value[5] && value[3]!=empty) output = (value[3]==X?+1:-1);
            if (value[6]==value[7] && value[6]==value[8] && value[6]!=empty) output = (value[6]==X?+1:-1);
            if (value[0]==value[4] && value[0]==value[8] && value[0]!=empty) output = (value[0]==X?+1:-1);
            if (value[2]==value[4] && value[2]==value[6] && value[2]!=empty) output = (value[2]==X?+1:-1);
            return output;
      }

      inline bool is_done() const
      {
            return (is_all_filled() || end_score() != 0);
      }

      inline void print() const
      {
            std::cout << "\n";
            for(int row=0; row!=3; ++row)
            {
                  std::cout << "|";
                  for (int col=0; col!=3; ++col)
                  {
                        unsigned short n = rc2n(row, col);
                        if (value[n] == O)       std::cout << "O";
                        else if (value[n] == X)  std::cout << "X";
                        else                     std::cout << " ";
                  }
                  std::cout << "|\n";
            }
            if (end_score() > 0) std::cout << "computer won\n";
            if (end_score() < 0) std::cout << "you won\n";
      }

      pos_state value[9];
};

// ************************ //
// *** Recursive max-min *** //
// ************************ //
namespace recursive_maxmin
{
      auto normalize(const std::vector<double>& next_scores)
      {
            std::vector<double> output = next_scores;

            double sum = 0;
            for(const auto& x : output) sum += x;
            if (fabs(sum) > 1e-5)
            {
                  std::for_each(output.begin(), output.end(), [&](auto& x){ x /= sum; });
            }
            return output;
      }

      std::vector<double> future_score(const grid_state& this_state, bool is_man, std::vector<unsigned short>& next_moves)
      {
            std::vector<double> next_scores;
            for(const auto& next_move : next_moves)
            {
                  // *** this-action and next-state *** //
                  grid_state next_state = this_state;
                  next_state.click(next_move, is_man);

                  if (next_state.is_done())
                  {
                        double temp = next_state.end_score();
                        next_scores.push_back(temp);
                  }
                  else
                  {
                        auto next_next_scores = future_score(next_state, !is_man, next_state.empty_pos());

                        // *** next-action and next-next-state *** //
                        if (is_man)
                        {
                              next_scores.push_back(*std::max_element(next_next_scores.begin(),
                                                                      next_next_scores.end()));
                        }
                        else
                        {
                              next_scores.push_back(*std::min_element(next_next_scores.begin(),
                                                                      next_next_scores.end()));
                        }
                  }
            }
      //    return normalize(next_scores); // no need to normalise
            return next_scores;
      }
```

```
		inline unsigned short best_move(const grid_state& state) // assume this is NOT end-game
		{
			auto next_moves  = state.empty_pos();
			auto next_scores = future_score(state, false, next_moves);
			auto  max_score  = next_scores[0];
			int   max_index  = 0;

			for(int n=1; n!=next_moves.size(); ++n)
			{
				if (max_score < next_scores[n])
				{	max_score = next_scores[n];
					max_index = n;
				}
			}
			return next_moves[max_index];
		}

		inline unsigned short rand_move(const grid_state& state) // assume this is NOT end-game
		{
			auto next_moves = state.empty_pos();
			return next_moves[rand() % next_moves.size()];
		}
	}

	class game
	{
	public:
		inline game(bool is_man_first) : is_man_next(is_man_first)
		{
		}

		inline void start()
		{
			while(!state.is_done())
			{
				if (is_man_next) man_one_step();
				else alg_one_step();
				is_man_next = !is_man_next;
			}

			std::cout << "\n\n[ENDGAME]";
			state.print();
		}

	private:
		inline void man_one_step()
		{
			state.print();
			std::cout << "\nnext : ";
			unsigned short n = check_key(getche());

			while(true)
			{
				if (state.click(n, true)) return;
				std::cout << "\ninvalid, try again : ";
				n = check_key(getche());
			}
		}

		inline void alg_one_step()
		{
			auto n = recursive_maxmin::best_move(state);
		//	auto n = recursive_maxmin::rand_move(state);

			if (state.click(n, false)) return;
			std::cout << "\nERROR in engine.";
			throw(std::exception());
		}

	private:
		bool is_man_next;
		grid_state state;
	};
}

void test_alg()
{
	algo::game ttt(true);
	ttt.start();
}
```