Source Code

game.h

```
#ifndef GAME_H
#define GAME_H

#include <iostream>
#include <pthread.h>

#include "board.h"
#include "player.h"

// COSC 4F00 - Software Development (2016)
// Instructor: Vlad Wojcik
// Brock University
//
// Assignment #: 2
// Due: November 18, 2016
//
// Matt Laidman
// ml12ef, 5199807
//
//
// Three Dimensional Tic-Tac-Toe
// An Exercise in AI and Concurrency in C++

// game.h

// Global Variables

Board* board;
Player* p1;
Player* p2;

// Function Prototypes

void playGame();

#endif
```

game.cpp

```
#include "game.h"

// main function
// Displays the menus, starts and drives the game.
int main(int argc, char* argv[]) {

    char c, gc;
    int ic, igc;
    bool pa;

    std::cout<<"3D Tic Tac Toe!\n\n";
    do {
        // Main Menu
        std::cout<<"Menu:\n\t1)\tPlay\n\t2)\tDisplay Rules\n\n\t0)\tQuit\n\n";
        std::cout<<"Enter a selection ([0..2]): ";
        std::cin>>c;
        ic = ((int)c)-48;
        if (ic == 0) {
            return 0;
        }
        // Display Rules
        if (ic == 2) {
            std::cout<<"\nRules:\n";
            std::cout<<"There are eight pegs arranged as follows:\n\n";
            board->printBoard();
            std::cout<<"\nEach player is given twelve pieces, the player with ";
            std::cout<<"the red pieces goes first.\n";
            std::cout<<"Players will alternate placing all twelve of their ";
            std::cout<<"pieces on to the pegs, one at\n\ta time.\n";
            std::cout<<"A peg can hold at most 3 pieces.\n";
            std::cout<<"At the end of the twenty-four turns, the player with ";
            std::cout<<"the most pieces forming a\n\tline of three is the ";
            std::cout<<"winner.\n";
            std::cout<<"A line of three occupy either one vertical level, or ";
            std::cout<<"all three vertical levels,\n\tand can be vertical, ";
            std::cout<<"horizontal, or diagonal in direction.\n";
            std::cout<<std::endl;
        }
    } while(ic != 1);
    do {
        board = new Board();
        // Game mode selection menu
        do {
            std::cout<<"\nSelect a game mode:\n";
            std::cout<<"\t1)\tHuman Player vs Human Player\n";
            std::cout<<"\t2)\tHuman Player vs Computer Player\n";
            // AI vs AI is mostly for fun..
            std::cout<<"\t3)\tComputer Player vs Computer Player\n\n";
            std::cout<<"Choose a game mode ([1..3]): ";
            std::cin>>gc;
            igc = ((int)gc)-48;
```

```
        } while((igc < 1) || (igc > 3));
        switch (igc) {
            case 1:
                p1 = new HumanPlayer(Red);
                p2 = new HumanPlayer(White);
                break;
            case 2:
                // if human vs ai get human to choose colour
                do {
                    std::cout<<"\nChoose a colour:\n\t1)\tRed\n\t2)\tWhite\n\n";
                    std::cout<<"Choice ([1..2]): ";
                    std::cin>>c;
                    ic = ((int)c)-48;
                } while((ic < 1) || (ic > 2));
                if (ic == 1) {
                    p1 = new HumanPlayer(Red);
                    p2 = new ComputerPlayer(White);
                } else {
                    p1 = new ComputerPlayer(Red);
                    p2 = new HumanPlayer(White);
                }
                break;
            case 3:
                p1 = new ComputerPlayer(Red);
                p2 = new ComputerPlayer(White);
                break;
            default:
                return 1;
        }
        std::cout<<std::endl;
        playGame();
        delete board;
        delete p1;
        delete p2;
        // play again?
        do {
            std::cout<<"\nPlay Again:\n\t1)\tYes\n\t2)\tNo\n\n";
            std::cout<<"Choice ([1..2]): ";
            std::cin>>c;
            ic = ((int)c)-48;
        } while((ic < 1) || (ic > 2));
        pa = (ic == 1 ? true : false);
    } while(pa);

    return 0;
}
```

```
// playGame procedure
// Loops until the game is over, getting player's move, printing the board, and
// keeping the score.
void playGame() {
    int peg;
    Player* current = p1;

    pthread_t t;
    void* aiMove; // Really an int ... pthreads ...

    board->printBoard();
    std::cout<<"              R: "<<p1->getScore()<<"           W: "<<
            p2->getScore()<<"\n\n";
    do {
        std::cout<<(current->getColour() == Red ? "Red" :
            "White")<<"'s turn!\n";

        // get move, if computer player join with thread
        if (current->isHuman() || board->getMoveNum() == 0) {
            peg = current->getMove(*board);
        } else {
            pthread_join(t, &aiMove);
            peg = *((int*)aiMove);
        }

        std::cout<<std::endl;
        board->addPiece(peg, current->getColour()); // add the mvoe

        // if other player is computer, immediatly start thinking about next move
        if (!(current == p1 ? p2 : p1)->isHuman() && board->getMoveNum() < 24) {
            pthread_create(&t, NULL, Player::getMoveWrap, new
GetMoveStruct((current == p1 ? p2 : p1), *board));
        }

        // update scores
        current->setScore(board->evalBoard(current->getColour()));
        current = (current == p1 ? p2 : p1);
        board->printBoard(); // print board and score
        std::cout<<"              R: "<<p1->getScore()<<"           W: "<<
            p2->getScore()<<"\n\n";
    } while (board->getMoveNum() < 24); // loop until no moves left
    std::cout<<"Game Over!\n"; // declare winner
    std::cout<<(p1->getScore() > p2->getScore() ? "Red is the winner!" :
        (p1->getScore() == p2->getScore() ? "The game is a tie!" :
        "White is the winner!"))<<std::endl;

    return;
}
```

board.h

```
#ifndef BOARD_H
#define BOARD_H

#include <iostream>

// board.h

// A board object for the 8-Peg, Three Dimensional Tic-Tac-Toe game!

// Type Definitions

// Colour enum
enum Colour {
    None,
    Red,
    White
};

// Board class
class Board {
    private:
        Colour pegs[8][3];
        int moveNum;
    public:
        Board();
        void addPiece(int, Colour);
        void removePiece(int);
        int getNumPieces(int);
        int getMoveNum();
        bool validMove(int);
        void printBoard();
        int evalBoard(Colour);
};

// Board constructor
// creates an empty board for a new game
Board::Board() {
    moveNum = 0;
    for (int peg = 0 ; peg < 8 ; peg++) {
        for (int height = 0 ; height < 3 ; height++) {
            pegs[peg][height] = None;
        }
    }
}
```

```
// Class Method Implementations

// Board Class

// addPiece method
// adds a piece of the given colour to the given peg
void Board::addPiece(int peg, Colour colour) {
    for (int height = 0 ; height < 3 ; height++) {
        if (pegs[peg][height] == None) {
            pegs[peg][height] = colour;
            moveNum++;
            break;
        }
    }
    return;
}

// removePiece method
// removes the topmost piece fromt the given peg
void Board::removePiece(int peg) {
    for (int height = 2 ; height >= 0 ; height--) {
        if (pegs[peg][height] != None) {
            pegs[peg][height] = None;
            moveNum--;
            break;
        }
    }
    return;
}

// getNumPieces method
// returns the number of pieces on the peg
int Board::getNumPieces(int peg) {
    int numPieces = 0;
    for (int height = 0 ; height < 3 ; height++) {
        if (pegs[peg][height] == None) {
            break;
        }
        numPieces++;
    }
    return numPieces;
}

// getMoveNum method
int Board::getMoveNum() {
    return moveNum;
}

// validMove method
bool Board::validMove(int peg) {
    return ((peg >= 0) && (peg <= 7) && (getNumPieces(peg) < 3));
}
```

```
// evalBoard method
// Evaluates the board with respect to the given colour
// Checks all possible scoring positions
int Board::evalBoard(Colour colour) {
    int score = 0;
    for (int peg = 0 ; peg < 8 ; peg++)
        if (pegs[peg][0] == colour && pegs[peg][1] == colour && pegs[peg][2] ==
colour) score++;
    for (int height = 0 ; height < 3 ; height++)
        if (pegs[0][height] == colour && pegs[1][height] == colour && pegs[2]
[height] == colour) score++;
    for (int height = 0 ; height < 3 ; height++)
        if (pegs[5][height] == colour && pegs[6][height] == colour && pegs[7]
[height] == colour) score++;
    for (int height = 0 ; height < 3 ; height++)
        if (pegs[7][height] == colour && pegs[4][height] == colour && pegs[1]
[height] == colour) score++;
    for (int height = 0 ; height < 3 ; height++)
        if (pegs[6][height] == colour && pegs[3][height] == colour && pegs[0]
[height] == colour) score++;
    for (int height = 0 ; height < 3 ; height++)
        if (pegs[5][height] == colour && pegs[3][height] == colour && pegs[1]
[height] == colour) score++;
    for (int height = 0 ; height < 3 ; height++)
        if (pegs[6][height] == colour && pegs[4][height] == colour && pegs[2]
[height] == colour) score++;
    if (pegs[0][0] == colour && pegs[1][1] == colour && pegs[2][2] == colour)
score++;
    if (pegs[0][2] == colour && pegs[1][1] == colour && pegs[2][0] == colour)
score++;
    if (pegs[5][0] == colour && pegs[6][1] == colour && pegs[7][2] == colour)
score++;
    if (pegs[5][2] == colour && pegs[6][1] == colour && pegs[7][0] == colour)
score++;
    if (pegs[0][0] == colour && pegs[3][1] == colour && pegs[6][2] == colour)
score++;
    if (pegs[0][2] == colour && pegs[3][1] == colour && pegs[6][0] == colour)
score++;
    if (pegs[1][0] == colour && pegs[4][1] == colour && pegs[7][2] == colour)
score++;
    if (pegs[1][2] == colour && pegs[4][1] == colour && pegs[7][0] == colour)
score++;
    if (pegs[1][0] == colour && pegs[3][1] == colour && pegs[5][2] == colour)
score++;
    if (pegs[1][2] == colour && pegs[3][1] == colour && pegs[5][0] == colour)
score++;
    if (pegs[2][0] == colour && pegs[4][1] == colour && pegs[6][2] == colour)
score++;
    if (pegs[2][2] == colour && pegs[4][1] == colour && pegs[6][0] == colour)
score++;
    return score;
}
```

```
// printBoard method
// Prints the Board, line by line..
void Board::printBoard() {

    std::cout<<"\t     "<<(pegs[5][2] == None ? "|" : (pegs[5][2] == Red ? "R" :
"W"));
    std::cout<<"         "<<(pegs[6][2] == None ? "|" : (pegs[6][2] == Red ? "R" :
"W"));
    std::cout<<"         "<<(pegs[7][2] == None ? "|" : (pegs[7][2] == Red ? "R" :
"W"));
    std::cout<<"\n";
    std::cout<<"\t ----"<<(pegs[5][1] == None ? "|" : (pegs[5][1] == Red ? "R" :
"W"));
    std::cout<<"---------"<<(pegs[6][1] == None ? "|" : (pegs[6][1] == Red ? "R" :
"W"));
    std::cout<<"---------"<<(pegs[7][1] == None ? "|" : (pegs[7][1] == Red ? "R" :
"W"));
    std::cout<<"----\n";
    std::cout<<"\t|    "<<(pegs[5][0] == None ? "|" : (pegs[5][0] == Red ? "R" :
"W"));
    std::cout<<"         "<<(pegs[6][0] == None ? "|" : (pegs[6][0] == Red ? "R" :
"W"));
    std::cout<<"         "<<(pegs[7][0] == None ? "|" : (pegs[7][0] == Red ? "R" :
"W"));
    std::cout<<"    |\n";
    std::cout<<"\t|   (F)   "<<(pegs[3][2] == None ? "|" : (pegs[3][2] == Red ?
"R" : "W"));
    std::cout<<"   (G)   "<<(pegs[4][2] == None ? "|" : (pegs[4][2] == Red ? "R" :
"W"));
    std::cout<<"   (H)   |\n";
    std::cout<<"\t|         "<<(pegs[3][1] == None ? "|" : (pegs[3][1] == Red ?
"R" : "W"));
    std::cout<<"         "<<(pegs[4][1] == None ? "|" : (pegs[4][1] == Red ? "R" :
"W"));
    std::cout<<"         |\n";
    std::cout<<"\t|         "<<(pegs[3][0] == None ? "|" : (pegs[3][0] == Red ?
"R" : "W"));
    std::cout<<"         "<<(pegs[4][0] == None ? "|" : (pegs[4][0] == Red ? "R" :
"W"));
    std::cout<<"         |\n";
    std::cout<<"\t|    "<<(pegs[0][2] == None ? "|" : (pegs[0][2] == Red ? "R" :
"W"));
    std::cout<<"   (D)   "<<(pegs[1][2] == None ? "|" : (pegs[1][2] == Red ? "R" :
"W"));
    std::cout<<"   (E)   "<<(pegs[2][2] == None ? "|" : (pegs[2][2] == Red ? "R" :
"W"));
    std::cout<<"    |\n";
    std::cout<<"\t|    "<<(pegs[0][1] == None ? "|" : (pegs[0][1] == Red ? "R" :
"W"));
    std::cout<<"         "<<(pegs[1][1] == None ? "|" : (pegs[1][1] == Red ? "R" :
"W"));
    std::cout<<"         "<<(pegs[2][1] == None ? "|" : (pegs[2][1] == Red ? "R" :
"W"));
```

```
    std::cout<<"    |\n";
    std::cout<<"\t|    "<<(pegs[0][0] == None ? "|" : (pegs[0][0] == Red ? "R" :
"W"));
    std::cout<<"         "<<(pegs[1][0] == None ? "|" : (pegs[1][0] == Red ? "R" :
"W"));
    std::cout<<"         "<<(pegs[2][0] == None ? "|" : (pegs[2][0] == Red ? "R" :
"W"));
    std::cout<<"    |\n";
    std::cout<<"\t|   (A)       (B)       (C)   |\n";
    std::cout<<"\t|                             |\n";
    std::cout<<"\t -----------------------------"<<std::endl;
    return;
}

#endif
```

player.h

```
#ifndef PLAYER_H
#define PLAYER_H

#include <cstdlib>
#include <ctime>
#include <iostream>
#include <unistd.h>

#include "board.h"

// player.h

// Type Definitions

// Player class
class Player {
    protected:
        Colour colour;
        bool human;
        int score;
    public:
        Player(Colour, bool);
        virtual ~Player() {};
        Colour getColour();
        bool isHuman();
        int getScore();
        void setScore(int);
        virtual int getMove(Board) = 0;

        static void* getMoveWrap(void* arg);
        int __aiMove; // secret
};

// Constructor
// Creates a Player of given colour
Player::Player(Colour colour, bool human) : colour(colour), human(human), score(0)
{}

// Helper struct to be passed to getMoveWrap...
// Can only pass one parameter to pthread.
struct GetMoveStruct {
    Player* p;
    Board b;
    GetMoveStruct(Player* p, Board b) : p(p), b(b) {}
};
```

```
// HumanPlayer class
class HumanPlayer : public Player {
    public:
        HumanPlayer(Colour);
        int getMove(Board);
};

// Constructor
// Calls the Player constructor
HumanPlayer::HumanPlayer(Colour colour) : Player(colour, true) {}

// ComputerPlayer class
class ComputerPlayer : public Player {
    private:
        int maxDepth;
        int bestPeg;
        int negamax(Board, int, int, int, Colour);
    public:
        ComputerPlayer(Colour);
        int getMove(Board);
};

// Constructor
// Calls the Player constructor
ComputerPlayer::ComputerPlayer(Colour colour) : Player(colour, false),
maxDepth(12) {
    srand(time(NULL)); // seed first move random number with current time
}

// Class Method Implementations

// Player class

// getColour method
Colour Player::getColour() {
    return colour;
}

// isHuman method
bool Player::isHuman() {
    return human;
}

// getScore method
int Player::getScore() {
    return score;
}

// setScore method
void Player::setScore(int score) {
    this->score = score;
    return;
}
```

```
// messy pthreads...
// This is literally just a horrifying wrapper so getmove can be given
// as pthread start routine...
void* Player::getMoveWrap(void* arg) {
    GetMoveStruct* gms = (GetMoveStruct*)arg;
    Player* player = gms->p;
    player->__aiMove = player->getMove(gms->b);
    return (void*)(&(player->__aiMove));
}

// HumanPlayer class

// getMove method
// Creates a move from user input
int HumanPlayer::getMove(Board board) {
    char peg;
    int ipeg;
    do {
        std::cout<<"Peg ([A..F], case-sensitive): ";
        std::cin>>peg;
        ipeg = ((int)peg)-65;
    } while((ipeg < 0) || (ipeg > 7));
    return ipeg;
}

// ComputerPlayer class

// getMove method
int ComputerPlayer::getMove(Board board) {
    if (board.getMoveNum() <= 1) {
        sleep(1); // some "realness" to the first move...
        return (rand() % 8);
    }
    negamax(board, maxDepth, -1000, 1000, colour);
    return bestPeg;
}
```

```
// NegaMax with Alpha-Beta Pruning
// Calculates the move to be played.

// function negamax(board, depth, alpha, beta, colour)
//     if depth = 0 or no moves left
//         return colour*board
//     bestValue := −1000
//     foreach possible move
//         value := −negamax(child, depth−1, −beta, −alpha, -Colour)
//         bestValue := max(bestValue, value)
//         alpha := max(alpha, value)
//         if alpha >= beta
//             break
//     return bestValue
int ComputerPlayer::negamax(Board board, int depth, int alpha, int beta, Colour
colour) {
    if (depth == 0 || board.getMoveNum() == 24) { // if max depth or final move
        if (colour == this->colour) { // return board evaluation
            return board.evalBoard(colour);
        } else {
            return (-1)*board.evalBoard(colour);
        }
    }
    int bestValue = -1000;
    for (int peg = 0 ; peg < 8 ; peg++) { // for each possible move
        if (board.getNumPieces(peg) != 3) { // check legal move
            board.addPiece(peg, colour);
            int value = (-1)*negamax(board, depth-1, (-1)*beta, (-1)*alpha, colour
== Red ? White : Red);
            board.removePiece(peg);
            if (value > bestValue) { // if better move
                bestValue = value; // update best move
                bestPeg = peg;
            }
            if (value > alpha) {
                alpha = value;
            }
            if (alpha >= beta) { // prune
                break;
            }
        }
    }
    return bestValue; // return the best evaluation
}



#endif
```