Board.c:

#define \_CRT\_SECURE\_NO\_WARNINGS

#include "Board.h"

//Fills the board cells with their strigns

void BasicBoardFill(int i, cell \*data)

{

char temp[SIZE\_OF\_INT];

\_itoa(i, temp, 10);

data->destination\_cell = i-1;

strcpy(data->value, " ");

if(i<10)

{

strcat(data->value, "0");

strcat(data->value, temp);

strcat(data->value, " ");

}

else if(i>9 && i<100)

{

strcat(data->value,temp);

strcat(data->value, " ");

}

else if(i=100)

{

strcat(data->value,temp);

strcat(data->value, " ");

}

}

//Update the cell to be snake head

void UpdateSnakeHead(cell \*curr, int dest)

{

curr->value[POS\_IN\_STRING\_OF\_S]='v';

curr->destination\_cell=dest-1;

}

//Update the cell to be snake tail

void UpdateSnaketail(cell \*curr)

{

curr->value[POS\_IN\_STRING\_OF\_S]='\_';

}

//Update the cell to be ladder head

void UpdateLadderHead(cell \*curr)

{

curr->value[POS\_IN\_STRING\_OF\_L]='=';

}

//Update the cell to be ladder tail

void UpdateLadderTail(cell \*curr, int dest)

{

curr->value[POS\_IN\_STRING\_OF\_L]='^';

curr->destination\_cell=dest-1;

}

//Prints a line seperator

void PrintLineSeperator()

{

int i;

for(i=0;i<LINE\_SIZE;i++)

printf("-");

printf("\n");

}

//Prints a cell

void PrintCell(cell cur\_cell)

{

int j;

int numOfPlayers;

char players[MAX\_NUM\_OF\_PLAYERS + 1];

memset(players, '\0', MAX\_NUM\_OF\_PLAYERS + 1);

numOfPlayers = 0;

for (j = 0; j < MAX\_NUM\_OF\_PLAYERS; j++)

{

if (cur\_cell.players\_in\_cell[j] == TRUE)

{

players[numOfPlayers] = GAME\_PIECES[j];

numOfPlayers++;

}

}

switch(numOfPlayers)

{

case 0:

printf("%s",cur\_cell.value);

break;

case 1:

printf(" %s ", players);

break;

case 2:

printf(" %s ", players);

break;

case 3:

printf(" %s ", players);

break;

case 4:

printf(" %s ", players);

break;

}

printf("|");

}

//Prints a board line

void PrintBoardLine(cell board[], int line)

{

int i;

int start;

int factor;

int end;

if (line % 2 == 0)

{

//asc

start = line\*NUM\_OF\_CELLS\_IN\_LINE;

end = start + NUM\_OF\_CELLS\_IN\_LINE;

factor = 1;

}

else

{

//dsc

start = (line + 1)\*NUM\_OF\_CELLS\_IN\_LINE - 1;

end = start - NUM\_OF\_CELLS\_IN\_LINE;

factor = -1;

}

printf("|");

for (i=start; i!=end; i+=factor)

{

PrintCell(board[i]);

}

printf("\n");

PrintLineSeperator();

}

//Print the whole board

void PrintBoard(game\_board \*board)

{

int i;

PrintLineSeperator();

for(i=9; i>=0; i--)

{

PrintBoardLine(board->cells, i);

}

}

//Build the board

void BuildBoard(game\_board \*board)

{

int i;

int up\_ladder[]={14,31,38,84,59,67,81,91};

int bottom\_ladder[]={4,9,20,28,40,51,63,71};

int snake\_head[]={17,54,64,62,93,95,87,99};

int snake\_tail[] = {7,34,60,19,73,75,24,78};

for(i=0;i<BOARD\_SIZE;i++)

BasicBoardFill(i+1, &(board->cells[i]));

for(i=0; i<NUM\_OF\_LADDERS\_OR\_SNAKES; i++)

{

UpdateLadderHead(&(board->cells[up\_ladder[i]-1]));

UpdateLadderTail(&(board->cells[bottom\_ladder[i]-1]),up\_ladder[i]);

UpdateSnaketail(&(board->cells[snake\_tail[i]-1]));

UpdateSnakeHead(&(board->cells[snake\_head[i]-1]),snake\_head[i]);

}

for (i=0; i<MAX\_NUM\_OF\_PLAYERS; i++)

{

board->players\_location[i] = -1;

}

}

//Updates the board on a game\_piece dice\_result

BOOL UpdateBoard(game\_board \*board, char game\_piece, int dice\_result)

{

int location;

int player\_index;

player\_index = (int)(strchr(GAME\_PIECES, game\_piece) - GAME\_PIECES);

location = board->players\_location[player\_index];

if (location > 0)

board->cells[location].players\_in\_cell[player\_index] = FALSE;

location += dice\_result;

if (location >= (BOARD\_SIZE - 1))

{

return TRUE;

}

//Get destination cell

location = board->cells[location].destination\_cell;

//Update player location

board->players\_location[player\_index] = location;

board->cells[location].players\_in\_cell[player\_index] = TRUE;

return FALSE;

}

Board.h:

#ifndef BOARD\_HEADER

#define BOARD\_HEADER

#include <stdio.h>

#include <stdlib.h>

#include <windows.h>

#include <string.h>

#include "GameConsts.h"

#define CELL\_SIZE 7

#define BOARD\_SIZE 100

#define SIZE\_OF\_INT 33

#define NUM\_OF\_LADDERS\_OR\_SNAKES 8

#define POS\_IN\_STRING\_OF\_S 4

#define POS\_IN\_STRING\_OF\_L 1

#define LINE\_SIZE 71

#define NUM\_OF\_CELLS\_IN\_LINE 10

#define GAME\_PIECES "@#%\*"

typedef struct cell

{

char value[CELL\_SIZE];

int destination\_cell;

BOOL players\_in\_cell[MAX\_NUM\_OF\_PLAYERS];

}cell;

typedef struct game\_board

{

cell cells[BOARD\_SIZE];

int players\_location[MAX\_NUM\_OF\_PLAYERS];

}game\_board;

void BasicBoardFill(int i, cell \*data);

void BuildBoard(game\_board \*board);

void UpdateSnakeHead(cell \*curr, int dest);

void UpdateSnaketail(cell \*curr);

void UpdateLadderHead(cell \*curr);

void UpdateLadderTail(cell \*curr, int dest);

void PrintLineSeperator();

void PrintCell(cell cur\_cell);

void PrintBoardLine(cell board[], int line);

void PrintBoard(game\_board \*board);

BOOL UpdateBoard(game\_board \*board, char game\_piece, int dice\_result);

#endif

ClientCommunication:

#include "ClientCommunication.h"

//Runs the client communication thread. This thread reads msgs from the socket.

DWORD WINAPI RunClientCommunication(LPVOID lpParam)

{

data\_communication \*communication = (data\_communication \*) lpParam;

while (1)

{

memset(communication->message, '\0', MAX\_COMMAND\_LENGTH);

if (receive\_from\_socket(communication->socket, communication->message) == TRUE)

{

ReleaseSemaphoreSimple(communication->IncomingMessageFromServerSemaphore);

WaitForSingleObject(communication->EngineDoneWithServerMessageSemaphore, INFINITE);

}

else

{

SetLastError(WSAGetLastError());

write\_log\_and\_print("Error while trying to receive data to socket. Error code: 0x%x\n", GetLastError());

ReleaseSemaphoreSimple(communication->IncomingMessageFromServerSemaphore);

communication->communication\_error = TRUE;

return GetLastError();

}

}

return GetLastError();

}

ClientCommunication.h:

#ifndef CLIENT\_COMMUNICATION\_HEADER

#define CLIENT\_COMMUNICATION\_HEADER

#include "SocketWrapper.h"

#include <Windows.h>

#include "Mutex.h"

#include "Semaphore.h"

typedef struct data\_communication

{

HANDLE IncomingMessageFromServerSemaphore;

HANDLE EngineDoneWithServerMessageSemaphore;

SOCKET socket;

char message[MAX\_COMMAND\_LENGTH];

char \*username;

char game\_piece;

int port;

BOOL communication\_error;

} data\_communication;

DWORD WINAPI RunClientCommunication(LPVOID lpParam);

#endif

Engine.c:

#include "Engine.h"

/\*oOoOoOoOoOoOoOoOoOoOoOoOoOoOoOoOoOoOoOoOoOoOoOoOoOoOoOoOoOoOoOoOoOoOoOoO\*/

void RunClient(int port, char \*username)

{

HANDLE semaphores[NUMBER\_OF\_THREADS];

HANDLE threads[NUMBER\_OF\_THREADS];

DWORD lock\_result;

game\_board board;

data\_ui ui;

data\_communication communication;

int i;

//Init port and username

communication.port = port;

communication.username = username;

//Set communication flag

communication.communication\_error = FALSE;

//Set all handles to NULL

communication.EngineDoneWithServerMessageSemaphore = NULL;

communication.IncomingMessageFromServerSemaphore = NULL;

//Set all handles to NULL

ui.EngineDoneWithUserMessageSemaphore = NULL;

ui.UserEnteredTextSemaphore = NULL;

ui.PlayersTurnEvent = NULL;

//Init all threads to NULL

for (i = 0; i < NUMBER\_OF\_THREADS; i++)

{

threads[i] = NULL;

semaphores[i] = NULL;

}

//Connect to the server

if (ConnectToServer(&communication) == FALSE)

ExitGame(&communication, &ui, threads);

threads[0] = RunUiThread(&ui);

if (threads[0] == NULL)

ExitGame(&communication, &ui, threads);

threads[1] = RunClientCommunicationThread(&communication);

if (threads[1] == NULL)

ExitGame(&communication, &ui, threads);

semaphores[0] = ui.UserEnteredTextSemaphore;

semaphores[1] = communication.IncomingMessageFromServerSemaphore;

//init random

srand(time(NULL));

//Builds the board

BuildBoard(&board);

while (1)

{

lock\_result = (WaitForMultipleObjects(NUMBER\_OF\_THREADS, semaphores, FALSE, INFINITE));

switch (lock\_result)

{

case WAIT\_OBJECT\_0:

//If thread 0 finished, receive user msg

if (ReceivedUserMessage(&communication, &ui, &board) == TRUE)

ExitGame(&communication, &ui, threads);

break;

//If thread 1 finished, receive server msg

case WAIT\_OBJECT\_0 + 1:

//Error occured while reading from the server

if (communication.communication\_error == TRUE)

ExitGame(&communication, &ui, threads);

else if (HandleServerMessage(&communication, &ui, &board) == TRUE)

ExitGame(&communication, &ui, threads);

break;

}

}

ExitGame(&communication, &ui, threads);

}

// Connects to the server using socket. Return TRUE if succeed

BOOL ConnectToServer(data\_communication \*communication)

{

char username\_message[MAX\_SIZE\_OF\_USERNAME\_MESSAGE];

if (connect\_socket(communication->port, &communication->socket) == TRUE)

{

write\_log\_and\_print("Connected to server on port %d\n", communication->port);

memset(username\_message, '\0', MAX\_SIZE\_OF\_USERNAME\_MESSAGE);

strcat(username\_message, "username=");

strcat(username\_message, communication->username);

strcat(username\_message, "\n");

return SendMessageToServer(communication->socket, username\_message);

}

else

{

write\_log\_and\_print("Failed connecting to server on port %d\n", communication->port);

return FALSE;

}

}

//Creates the semaphore that related to the client communication thread.

//Starts the client communication thread. return TRUE if succeed

HANDLE RunClientCommunicationThread(data\_communication \*communication)

{

HANDLE clientCommunicationHandle = NULL;

communication->IncomingMessageFromServerSemaphore =

CreateSemaphoreSimple("IncomingMessageFromServerSemaphore");

if (communication->IncomingMessageFromServerSemaphore == NULL)

{

write\_log\_and\_print("Failed to create semaphore - Error code: 0x%x\n", GetLastError());

return NULL;

}

communication->EngineDoneWithServerMessageSemaphore =

CreateSemaphoreSimple("EngineDoneWithServerMessageSemaphore");

if (communication->EngineDoneWithServerMessageSemaphore == NULL)

{

write\_log\_and\_print("Failed to create semaphore - Error code: 0x%x\n", GetLastError());

return NULL;

}

clientCommunicationHandle = CreateThread(NULL, 0, RunClientCommunication, communication, 0, NULL);

if(clientCommunicationHandle == NULL)

{

write\_log\_and\_print("Failed to create thread - Error code: 0x%x\n", GetLastError());

}

return clientCommunicationHandle;

}

//Creates the semaphore that related to the ui thread. Runs the HANDLE of the thread

HANDLE RunUiThread(data\_ui \*ui)

{

HANDLE uiHandle = NULL;

ui->UserEnteredTextSemaphore =

CreateSemaphoreSimple("UserEnteredTextSemaphore");

if (ui->UserEnteredTextSemaphore == NULL)

{

write\_log\_and\_print("Failed to create semaphore - Error code: 0x%x\n", GetLastError());

return NULL;

}

ui->EngineDoneWithUserMessageSemaphore =

CreateSemaphoreSimple("EngineDoneWithUserMessageSemaphore");

if (ui->EngineDoneWithUserMessageSemaphore == NULL)

{

write\_log\_and\_print("Failed to create semaphore - Error code: 0x%x\n", GetLastError());

return NULL;

}

uiHandle = CreateThread(NULL, 0, RunUiManager, ui, 0, NULL);

if(uiHandle == NULL)

{

write\_log\_and\_print("Failed to create thread - Error code: 0x%x\n", GetLastError());

}

return uiHandle;

}

//Handles a message from the user. Returns TRUE if the game ended

BOOL ReceivedUserMessage(data\_communication \*communication, data\_ui \*ui, game\_board \*board)

{

DWORD return\_value;

char \*token = NULL;

int num\_of\_args;

char command\_copy[MAX\_COMMAND\_LENGTH];

strcpy(command\_copy, ui->command);

command\_copy[strlen(command\_copy) - 1] = '\0';

num\_of\_args = NumOfArgInCommand(command\_copy);//returns one if there are no spaces - one word

token = strtok(command\_copy, " ");

if(num\_of\_args > 1) //atleast two words

{

if(strcmp(token, COMMAND\_MESSAGE) == 0)

{

//If failed to send message to the server, finish the game

if (HandleMessageCommand(ui->command, num\_of\_args, communication->socket) == FALSE)

return TRUE;

}

else if(strcmp(token, COMMAND\_BROADCAST) == 0)

{

//If failed to send message to the server, finish the game

if (HandleBroadcastCommand(ui->command, num\_of\_args, communication->socket) == FALSE)

return TRUE;

}

else if(strcmp(token, COMMAND\_PLAY) == 0 || strcmp(token, COMMAND\_PLAYERS) == 0)

write\_log\_and\_print("Illegal argument for command %s. Command format is %s\n",

token, token);

else

write\_log\_and\_print("Command %s is not recognized. Possible commands are: players, message, broadcast and play.\n",

ui->command);

}

else if(strcmp(command\_copy, COMMAND\_PLAY) == 0)

{

if (HandlePlayCommand(communication, ui, board) == TRUE)

return TRUE; //The game is ended

}

else if (strcmp(command\_copy, COMMAND\_PLAYERS) == 0)

{

if (SendMessageToServer(communication->socket, ui->command) == FALSE)

return TRUE; //Error occured when sending message to the server

}

else

write\_log\_and\_print("Command %s is not recognized. Possible commands are: players, message, broadcast and play.\n",

token);

//Release the semaphore that engine done with user msgs

ReleaseSemaphoreSimple(ui->EngineDoneWithUserMessageSemaphore);

return FALSE;

}

//Handles a message command from the user

BOOL HandleMessageCommand(char \*command, int num\_of\_args, SOCKET socket)

{

char \*token = NULL;

if(num\_of\_args < COMMAND\_MESSAGE\_MIN\_ARGS)

{

write\_log\_and\_print("Illegal argument for command %s. Command format is %s <user> <message>\n",token, token);

return TRUE;

}

token = strtok(NULL, " ");

if(token == NULL)

{

write\_log\_and\_print("Illegal username\n");

return TRUE;

}

token = strtok(NULL, " ");

if(CheckIfMessageValid(token) == FALSE)

{

write\_log\_and\_print("Illegal message\n");

return TRUE;

}

return SendMessageToServer(socket, command);

}

//Handles a braodcast message from the user

BOOL HandleBroadcastCommand(char \*command, int num\_of\_args, SOCKET socket)

{

char \*token = NULL;

if(num\_of\_args < COMMAND\_BROADCAST\_MIN\_ARGS)

{

write\_log\_and\_print("Illegal argument for command %s. Command format is %s <message>\n", token, token);

return TRUE;

}

token = strtok(NULL, "\n");

if(CheckIfMessageValid(token) == FALSE)

{

write\_log\_and\_print("Illegal message");

return TRUE;

}

return SendMessageToServer(socket, command);

}

//Handles play command from the user. Plays only if its his turn. returns TRUE if the game ended

BOOL HandlePlayCommand(data\_communication \*communication, data\_ui \*ui, game\_board \*board)

{

int dice\_result;

char message[MAX\_COMMAND\_LENGTH];

DWORD lock\_result;

BOOL is\_game\_ended = FALSE;

lock\_result = WaitForSingleObject(ui->PlayersTurnEvent, 0);

switch(lock\_result)

{

case WAIT\_OBJECT\_0:

break;

default:

//Its not the player turn, not playing...

printf("It is not your turn...\n");

return FALSE;

}

memset(message, '\0', MAX\_COMMAND\_LENGTH);

//Rolls the dice to random number between 1 - 6

dice\_result = (((double)rand() / (RAND\_MAX + 1)) \* (MAX\_DICE\_VALUE - MIN\_DICE\_VALUE + 1)) + MIN\_DICE\_VALUE;

//Updating the board

is\_game\_ended = UpdateBoard(board, communication->game\_piece, dice\_result);

PrintBoard(board);

sprintf(message, "Player %c (%s) drew a %d.\n",

communication->game\_piece, communication->username, dice\_result);

printf("%s", message);

//Updates the server about the play

if (SendMessageToServer(communication->socket, message) == FALSE)

return TRUE; //Error occured when sending message to the server

if (is\_game\_ended == TRUE)

{

memset(message, '\0', MAX\_COMMAND\_LENGTH);

sprintf(message, "Player %s won the game. Congratulations.\n",

communication->username);

write\_log\_and\_print("%s", message);

if (SendMessageToServer(communication->socket, message) == FALSE)

return TRUE; //Error occured when sending message to the server

}

ResetEvent(ui->PlayersTurnEvent);

return is\_game\_ended;

}

//Handles the msgs received from the server. returns TRUE if the game ended

BOOL HandleServerMessage(data\_communication \*communication, data\_ui \*ui, game\_board \*board)

{

write\_log\_format("Received from server: %s", communication->message);

printf("%s", communication->message);

if (strstr(communication->message, "Private message from") == NULL &&

strstr(communication->message, "Broadcast from") == NULL)

{

if (strcmp(communication->message, "Your turn to play.\n") == 0)

SetEvent(ui->PlayersTurnEvent);

else if (strstr(communication->message, "your game piece is") != NULL)

{

communication->game\_piece =

communication->message[strlen(communication->message) - GAME\_PIECE\_POSITION\_FROM\_END];

}

else if (strstr(communication->message, "drew a") != NULL)

{

HandleOponentTurn(board, communication->message);

}

else if (strstr(communication->message, "won the game") != NULL)

return TRUE;

else if (strstr(communication->message, CONNETION\_REFUSED\_MSG) != NULL)

return TRUE; //Connection refused. Exiting...

else if (strstr(communication->message, "Cannot accept connection. Username already exists") != NULL)

return TRUE; //Username doens't exist

}

//Finish handling the message from the server

ReleaseSemaphoreSimple(communication->EngineDoneWithServerMessageSemaphore);

return FALSE;

}

void HandleOponentTurn(game\_board \*board, char \*message)

{

char \*token = NULL;

char game\_piece;

int dice\_result;

game\_piece = message[GAME\_PIECE\_POSITION\_IN\_DREW\_COMMAND]; //game piece

dice\_result = atoi(&(message[strlen(message) - DICE\_RESULT\_POSITION]));

//Updates the board on the user turn

UpdateBoard(board, game\_piece, dice\_result);

PrintBoard(board);

}

//Sends a message to the server and prints to the log.

BOOL SendMessageToServer(SOCKET socket, char \*message)

{

BOOL result;

write\_log\_format("Sent to server: %s", message);

result = write\_to\_socket(socket, message);

if (result == FALSE)

{

SetLastError(WSAGetLastError());

write\_log\_and\_print("Error while trying to write data to socket. Error code: 0x%x\n", GetLastError());

}

return result;

}

//Check is the message is valid.

//Should contain only digits, chars or ' ', ',', '.'.

//Should be less than 80 chars

BOOL CheckIfMessageValid(char \*message)

{

int length,i;

if (message == NULL)

return FALSE;

length = strlen(message); //returns not including \0

if(length > MAX\_USER\_MESSAGE\_LENGTH)

return FALSE;

else

{

for(i=0; i < length; i++)

{

if(message[i] != ' ' &&

message[i] != '.' &&

message[i] != ',' &&

isdigit(message[i])==0 &&

isalpha(message[i])==0)

return FALSE;

}

}

return TRUE;

}

//Returns the num of args in the command. Splitting by space.

int NumOfArgInCommand(char \*command)

{

char \*token=NULL;

int counter=0;

char command\_copy[MAX\_COMMAND\_LENGTH];

/\* get the first token \*/

strcpy(command\_copy, command);

token = strtok(command\_copy, " ");

/\* walk through other tokens \*/

while( token != NULL )

{

counter++;

token = strtok(NULL, " ");

}

return counter;

}

//Closes all the HANDLE and socket and exits the program

void ExitGame(data\_communication \*communication, data\_ui \*ui, HANDLE \*threads)

{

LPDWORD exit\_code;

int i;

close\_log();

//Close sockets

for (i = 0; i < NUMBER\_OF\_THREADS; i++)

{

if (threads[i] == NULL)

continue;

TerminateThread(threads[i], 0);

CloseHandle(threads[i]);

}

//Close semaphores

ForceCloseHandle(communication->EngineDoneWithServerMessageSemaphore);

ForceCloseHandle(communication->IncomingMessageFromServerSemaphore);

ForceCloseHandle(ui->EngineDoneWithUserMessageSemaphore);

ForceCloseHandle(ui->UserEnteredTextSemaphore);

//Close events

ForceCloseHandle(ui->PlayersTurnEvent);

//Close sockets

if (communication->socket != NULL)

close\_socket(communication->socket);

//Exit

exit(GetLastError());

}

//Close all the handles

void ForceCloseHandle(HANDLE handle)

{

if (handle != NULL)

CloseHandle(handle);

}

Engine.h:

#ifndef ENGINE\_HEADER

#define ENGINE\_HEADER

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

#include "SocketWrapper.h"

#include <windows.h>

#include "UiManager.h"

#include "ClientCommunication.h"

#include "Log.h"

#include "Mutex.h"

#include "Semaphore.h"

#include "GameConsts.h"

#include "Board.h"

#define MIN\_DICE\_VALUE 1

#define MAX\_DICE\_VALUE 6

#define NUMBER\_OF\_THREADS 2

#define GAME\_PIECE\_POSITION\_FROM\_END 2

#define DICE\_RESULT\_POSITION 3

#define GAME\_PIECE\_POSITION\_IN\_DREW\_COMMAND 7

HANDLE RunClientCommunicationThread(data\_communication \*communication);

HANDLE RunUiThread(data\_ui \*ui);

BOOL ReceivedUserMessage(data\_communication \*communication, data\_ui \*ui, game\_board \*board);

BOOL HandleServerMessage(data\_communication \*communication, data\_ui \*ui, game\_board \*board);

BOOL ConnectToServer(data\_communication \*communication);

BOOL SendMessageToServer(SOCKET socket, char \*message);

int NumOfArgInCommand(char \*command);

BOOL CheckIfMessageValid(char \*message);

BOOL HandleMessageCommand(char \*command, int num\_of\_args, SOCKET socket);

BOOL HandleBroadcastCommand(char \*command, int num\_of\_args, SOCKET socket);

BOOL HandlePlayCommand(data\_communication \*communication, data\_ui \*ui, game\_board \*board);

void RunClient(int port, char \*username);

void ExitGame(data\_communication \*communication, data\_ui \*ui, HANDLE \*threads);

void ForceCloseHandle(HANDLE handle);

void HandleOponentTurn(game\_board \*board, char \*message);

#endif

Events.c:

#include "Events.h"

//this function initialize the events//

HANDLE InitEvent(char \*event\_name)

{

DWORD last\_error = 0;

HANDLE evnt = NULL;

char unique\_event\_name[MAX\_EVENT\_LENGTH];

memset(unique\_event\_name, '\0', MAX\_EVENT\_LENGTH);

sprintf(unique\_event\_name, "%d\_%s", GetCurrentProcessId(), event\_name);

last\_error = GetLastError();

evnt = CreateEvent(NULL, TRUE, FALSE, (LPCWSTR)unique\_event\_name);

if(evnt == NULL)

{

write\_log\_and\_print("!!! ERROR in CreateEvent function. Error code: 0x%x !!!\n", GetLastError());

return FALSE;

}

SetLastError(last\_error);

return evnt;

}

Events.h:

#ifndef EVENTS\_HEADER

#define EVENTS\_HEADER

#include <Windows.h>

#include "Log.h"

#include <stdlib.h>

#include <stdio.h>

#define MAX\_EVENT\_LENGTH 100

HANDLE InitEvent(char \*event\_name);

#endif

GameConsts.h:

#ifndef GAME\_CONSTS\_HEADER

#define GAME\_CONSTS\_HEADER

#define MAX\_USER\_NAME\_LENGTH 31 // including \0

#define MAX\_NUM\_OF\_PLAYERS 4

#define MAX\_USER\_NAME\_MESSAGE\_LENGTH (10+MAX\_USER\_NAME\_LENGTH) //username=<username>\n

#define MAX\_WELCOME\_MESSAGE\_LENGTH (MAX\_USER\_NAME\_LENGTH+20) // <username> you game piece is <game piece>\n

#define MAX\_PLAYER\_JOINED\_MESSAGE\_LENGTH (35+MAX\_USER\_NAME\_LENGTH) // New player joined the game: <username> <game piece>\n

#define MAX\_PLAYERS\_LIST\_MESSAGE\_LENGTH (MAX\_NUM\_OF\_PLAYERS\*(MAX\_USER\_NAME\_LENGTH + 4) + 3) // <user1>-<game\_piece>, <user2>-<game\_piece>, <user3>-<game\_piece>, <user4>-<game\_piece>.\n

#define MAX\_USER\_NOT\_EXIST\_MESSAGE\_LENGTH (MAX\_USER\_NAME\_LENGTH+35) // User <username> doesn't exist in the game.\n

#define MAX\_PLAYER\_TURN\_MESSAGE\_LENGTH (MAX\_USER\_NAME\_LENGTH+30) // It is now <username>'s turn to play.\n

#define MAX\_MESSAGE\_LENGTH 80

#define MAX\_PRIVATE\_MESSAGE\_LENGTH (MAX\_USER\_NAME\_LENGTH+MAX\_MESSAGE\_LENGTH+30) // Private message from <username>: <message>

#define MAX\_BROADCAST\_MESSAGE\_LENGTH (MAX\_USER\_NAME\_LENGTH+MAX\_MESSAGE\_LENGTH+20) // Broadcast from <username>: <message>

#define MAX\_COMMAND\_LENGTH 200 // longer than the max command(players message response) just to be on the safe side..

#define MAX\_SIZE\_OF\_USERNAME\_MESSAGE 41

#define MAX\_USER\_MESSAGE\_LENGTH 80

#define COMMAND\_MESSAGE "message"

#define COMMAND\_PLAY "play"

#define COMMAND\_PLAYERS "players"

#define COMMAND\_BROADCAST "broadcast"

#define CONNETION\_REFUSED\_MSG "Connection to server refused. Exiting."

#define COMMAND\_MESSAGE\_MIN\_ARGS 3

#define COMMAND\_BROADCAST\_MIN\_ARGS 2

#endif

Log.c

#include <stdio.h>

#include <windows.h>

#include <stdarg.h>

#include "Mutex.h"

#include "MutexConstants.h"

#include "Log.h"

static FILE\* log\_file;

/\*oOoOoOoOoOoOoOoOoOoOoOoOoOoOoOoOoOoOoOoOoOoOoOoOoOoOoOoOoOoOoOoOoOoOoOoO\*/

//the file containe all the functions that responsable for the log file//

/\*oOoOoOoOoOoOoOoOoOoOoOoOoOoOoOoOoOoOoOoOoOoOoOoOoOoOoOoOoOoOoOoOoOoOoOoO\*/

void open\_log(char\* log\_path)

{

DWORD last\_error = 0;

if(log\_file != NULL)

return;

last\_error = GetLastError();

log\_file = fopen(log\_path, "w+");

if (log\_file == NULL)

{

printf("Unable to open log file\n");

exit(GetLastError());

}

SetLastError(last\_error);

}

void close\_log()

{

if(log\_file == NULL)

return;

fclose(log\_file);

log\_file = NULL;

close\_mutex("LogFile");

}

void write\_log(const char\* message)

{

if(log\_file == NULL)

return;

lock\_mutex(LOG\_MUTEX);

fputs(message, log\_file);

unlock\_mutex(LOG\_MUTEX);

}

void write\_log\_format(\_In\_z\_ \_Printf\_format\_string\_ const char \* \_Format, ...)

{

va\_list args;

if(log\_file == NULL)

return;

va\_start(args, \_Format);

lock\_mutex(LOG\_MUTEX);

vfprintf(log\_file, \_Format, args);

unlock\_mutex(LOG\_MUTEX);

va\_end(args);

}

void write\_log\_and\_print(\_In\_z\_ \_Printf\_format\_string\_ const char \* \_Format, ...)

{

va\_list args;

if(log\_file == NULL)

return;

va\_start(args, \_Format);

lock\_mutex(LOG\_MUTEX);

vfprintf(log\_file, \_Format, args);

vprintf(\_Format, args);

unlock\_mutex(LOG\_MUTEX);

va\_end(args);

}

Log.h:

#ifndef LOG\_HEADER

#define LOG\_HEADER

void open\_log(char\* log\_path);

void close\_log();

void write\_log\_format(\_In\_z\_ \_Printf\_format\_string\_ const char \* \_Format, ...);

void write\_log\_and\_print(\_In\_z\_ \_Printf\_format\_string\_ const char \* \_Format, ...);

void write\_log(const char\* message);

#endif

Main.c:

/\*Limor Mendelzburg 308081389

Mor Ben Ami 203607536

Exercise 4\*/

#define \_WINSOCKAPI\_ // stops windows.h including winsock.h

#include <windows.h>

#include <stdio.h>

#include "Engine.h"

#include "ServerGameManagement.h"

#include "Log.h"

#include "SocketWrapper.h"

#define SERVER\_MODE "server"

#define CLIENT\_MODE "client"

/\*oOoOoOoOoOoOoOoOoOoOoOoOoOoOoOoOoOoOoOoOoOoOoOoOoOoOoOoOoOoOoOoOoOoOoOoO\*/

int main(int argc, char\* argv[])

{

char\* mode = NULL;

char\* log\_path = NULL;

int port = 0;

char\* username = NULL;

mode = argv[1];

log\_path = argv[2];

port = atoi(argv[3]);

open\_log(log\_path);

if(init\_WSA() == FALSE)

{

printf("Failed to init WSA, Error\_Code: 0x%x\n", GetLastError());

}

if(strcmp(mode,SERVER\_MODE) == 0)

{

start\_server(port);

}

else if(strcmp(mode, CLIENT\_MODE) == 0)

{

username = argv[4];

RunClient(port, username);

}

clean\_WSA();

close\_log();

exit(GetLastError());

}//main1

Mutex.c:

#include "Mutex.h"

/\*oOoOoOoOoOoOoOoOoOoOoOoOoOoOoOoOoOoOoOoOoOoOoOoOoOoOoOoOoOoOoOoOoOoOoOoO\*/

//the file containe all the functions that responsable for all the mutexs//

/\*oOoOoOoOoOoOoOoOoOoOoOoOoOoOoOoOoOoOoOoOoOoOoOoOoOoOoOoOoOoOoOoOoOoOoOoO\*/

BOOL lock\_mutex(char\* mutex\_name)

{

HANDLE mutex = NULL;

DWORD lock\_result = 0;

char unique\_mutex\_name[MAX\_MUTEX\_LENGTH];

DWORD last\_error = 0;

last\_error = GetLastError();

memset(unique\_mutex\_name, '\0', MAX\_MUTEX\_LENGTH);

sprintf(unique\_mutex\_name, "%d\_%s", GetCurrentProcessId(), mutex\_name);

mutex = CreateMutex(NULL, FALSE, (LPCWSTR)unique\_mutex\_name);

if(mutex == NULL)

{

printf("failed to get mutex: %s\n",unique\_mutex\_name);

return FALSE;

}

SetLastError(last\_error);

lock\_result = WaitForSingleObject(mutex, INFINITE);

switch(lock\_result)

{

case WAIT\_OBJECT\_0:

return TRUE;

break;

}

printf("unable to gain ownership on mutex: %s\n",unique\_mutex\_name);

return FALSE;

}

BOOL unlock\_mutex(char\* mutex\_name)

{

HANDLE mutex = NULL;

DWORD lock\_result = 0;

char unique\_mutex\_name[MAX\_MUTEX\_LENGTH];

DWORD last\_error = 0;

memset(unique\_mutex\_name, '\0', MAX\_MUTEX\_LENGTH);

sprintf(unique\_mutex\_name, "%d\_%s", GetCurrentProcessId(), mutex\_name);

last\_error = GetLastError();

mutex = CreateMutex(NULL, FALSE, (LPCWSTR)unique\_mutex\_name);

if(mutex == NULL)

{

printf("failed to get mutex: %s, Error code: 0x%x\n",unique\_mutex\_name, GetLastError());

return FALSE;

}

SetLastError(last\_error);

return ReleaseMutex(mutex);

}

void close\_mutex(char\* mutex\_name)

{

HANDLE mutex = NULL;

DWORD lock\_result = 0;

char unique\_mutex\_name[MAX\_MUTEX\_LENGTH];

DWORD last\_error = 0;

memset(unique\_mutex\_name, '\0', MAX\_MUTEX\_LENGTH);

sprintf(unique\_mutex\_name, "%d\_%s", GetCurrentProcessId(), mutex\_name);

last\_error = GetLastError();

mutex = CreateMutex(NULL, FALSE, (LPCWSTR)unique\_mutex\_name);

if(mutex == NULL)

return;

SetLastError(last\_error);

CloseHandle(mutex);

}

Mutex.h:

#ifndef MUTEX\_HEADER

#define MUTEX\_HEADER

#include <stdio.h>

#include <stdlib.h>

#include <windows.h>

#include "Events.h"

#define MAX\_MUTEX\_LENGTH 200

BOOL lock\_mutex(char\* mutex\_name);

BOOL unlock\_mutex(char\* mutex\_name);

void close\_mutex(char\* mutex\_name);

HANDLE open\_mutex(char\* mutex\_name);

#endif

MutexConstants.h:

#define MUTEX\_NAME\_INCOMING\_MESSAGE "IncomingMessageFromServerMutex"

#define MUTEX\_NAME\_ENGINE\_DONE\_SRV\_MSG "EngineDoneWithServerMessageMutex"

#define BROADCAST\_MUTEX "BroadcastMutex"

#define LOG\_MUTEX "LogMutex"

#define HANDLE\_INCOMING\_MESSAGE\_MUTEX "HandleIncomingMessageMutex"

Semaphore.c:

#include "Semaphore.h"

HANDLE CreateSemaphoreSimple(char\* name)

{

HANDLE created\_semaphore;

DWORD last\_error = 0;

char unique\_semaphore\_name[MAX\_SEMAPHORE\_LENGTH];

memset(unique\_semaphore\_name, '\0', MAX\_SEMAPHORE\_LENGTH);

sprintf(unique\_semaphore\_name, "%d\_%s", GetCurrentProcessId(), name);

last\_error = GetLastError();

created\_semaphore = CreateSemaphore(NULL, 0, 1, (LPCWSTR)unique\_semaphore\_name);

if(created\_semaphore == NULL)

return NULL;

SetLastError(last\_error);

return created\_semaphore;

}

void ReleaseSemaphoreSimple(HANDLE semaphore)

{

ReleaseSemaphore(semaphore, 1, NULL);

}

Semaphore.h:

#ifndef SEMAPHORE\_HEADER

#define SEMAPHORE\_HEADER

#include <windows.h>

#include <stdio.h>

#define MAX\_SEMAPHORE\_LENGTH 100

HANDLE CreateSemaphoreSimple(char\* name);

void ReleaseSemaphoreSimple(HANDLE semaphore);

#endif

ServerCommunication.c:

#include "ServerCommunication.h"

DWORD WINAPI ServerCommunicationThreadStart(LPVOID param)

{

communication\_data \*data = (communication\_data\*)param;

SetLastError(0);

StartServerCommunication(data);

return GetLastError();

}

void StartServerCommunication(communication\_data\* data)

{

while(1)

{

if(ShouldFinishThread(data->all\_threads\_must\_end\_event) == TRUE)

return;

memset(data->message, '\0', MAX\_COMMAND\_LENGTH);

if(receive\_from\_socket(data->socket, data->message) == FALSE)

{

if(GetLastError() == 0)

{

//connection closed..

return;

}

write\_log\_and\_print("Error while receiving message from user: %s, ErrorCode: 0x%x\n", data->username, GetLastError());

SetEvent(data->all\_threads\_must\_end\_event);

return;

}

if(HandleIncomingMessage(data) == FALSE)

{

SetEvent(data->all\_threads\_must\_end\_event);

return;

}

}

}

BOOL HandleIncomingMessage(communication\_data\* data)

{

BOOL handle\_result = TRUE;

lock\_mutex(HANDLE\_INCOMING\_MESSAGE\_MUTEX);

lock\_mutex(BROADCAST\_MUTEX);

if(strstr(data->message, "message") == data->message)

handle\_result = HandleSendMessage(data);

else if(strstr(data->message, "broadcast") == data->message)

handle\_result = HandleBroadcastMessage(data);

else if(strstr(data->message, "players") == data->message)

handle\_result = HandlePlayersMessage(data);

else if(strstr(data->message, "Player") == data->message)

{

if(strstr(data->message, "drew") != NULL)

handle\_result = HandlePlayerTurnMessage(data);

else if(strstr(data->message, "won") != NULL)

handle\_result = HandlePlayeWonMessage(data);

// else: unknown message.. ignore

}

// else: unknown message.. ignore

unlock\_mutex(BROADCAST\_MUTEX);

unlock\_mutex(HANDLE\_INCOMING\_MESSAGE\_MUTEX);

return handle\_result;

}

BOOL HandleSendMessage(communication\_data\* data)

{

//message <user> <message>

char \*command\_name = NULL;

char \*username = NULL;

char \*message = NULL;

int i = 0;

char private\_message[MAX\_PRIVATE\_MESSAGE\_LENGTH];

char user\_not\_exist\_message[MAX\_USER\_NOT\_EXIST\_MESSAGE\_LENGTH];

command\_name = strtok(data->message, " ");

username = strtok(NULL, " ");

message = strtok(NULL, "\n");

for(i = 0; i < MAX\_NUM\_OF\_PLAYERS; i++)

{

if(strcmp(username, data->all\_users[i]) == 0)

{

memset(private\_message, '\0', MAX\_PRIVATE\_MESSAGE\_LENGTH);

sprintf(private\_message, "Private message from %s: %s\n", data->username, message);

write\_log\_format("Private message sent from %s to %s: %s\n", data->username, username, message);

if(write\_to\_socket(data->all\_users\_sockets[i], private\_message) == FALSE)

return FALSE;

return TRUE;

}

}

//user not exist..

memset(user\_not\_exist\_message, '\0', MAX\_USER\_NOT\_EXIST\_MESSAGE\_LENGTH);

sprintf(user\_not\_exist\_message, "User %s doesn't exist in the game.\n", username);

write\_log(user\_not\_exist\_message);

if(write\_to\_socket(data->socket, user\_not\_exist\_message) == FALSE)

return FALSE;

return TRUE;

}

BOOL HandleBroadcastMessage(communication\_data\* data)

{

//broadcast <message>

char \*command\_name = NULL;

char \*message = NULL;

int i = 0;

char broadcast\_message[MAX\_BROADCAST\_MESSAGE\_LENGTH];

command\_name = strtok(data->message, " ");

message = strtok(NULL, "\n");

memset(broadcast\_message, '\0', MAX\_BROADCAST\_MESSAGE\_LENGTH);

sprintf(broadcast\_message, "Broadcast from %s: %s\n", data->username, message);

write\_log\_format("Broadcast message from user %s: %s\n", data->username, message);

for(i = 0; i < MAX\_NUM\_OF\_PLAYERS; i++)

{

if(data->all\_users\_sockets[i] != INVALID\_SOCKET && data->all\_users\_sockets[i] != data->socket)

{

if(write\_to\_socket(data->all\_users\_sockets[i], broadcast\_message) == FALSE)

return FALSE;

}

}

return TRUE;

}

BOOL HandlePlayerTurnMessage(communication\_data\* data)

{

//Player <game piece> (<username>) drew a <dice result>.

int i = 0;

HANDLE turn\_finished\_event = NULL;

write\_log(data->message);

for(i = 0; i < MAX\_NUM\_OF\_PLAYERS; i++)

{

if(data->all\_users\_sockets[i] != INVALID\_SOCKET && data->all\_users\_sockets[i] != data->socket)

{

if(write\_to\_socket(data->all\_users\_sockets[i], data->message) == FALSE)

{

write\_log\_and\_print("Failed to send player draw result, Error\_code: 0x%x\n", GetLastError());

return FALSE;

}

}

}

turn\_finished\_event = InitEvent("TurnFinished");

if(turn\_finished\_event == NULL)

{

write\_log\_and\_print("Failed to create TurnFinished Event, Error\_code: 0x%x\n", GetLastError());

return FALSE;

}

SetEvent(turn\_finished\_event);

return TRUE;

}

BOOL HandlePlayeWonMessage(communication\_data\* data)

{

int i = 0;

HANDLE won\_event = NULL;

write\_log(data->message);

for(i = 0; i < MAX\_NUM\_OF\_PLAYERS; i++)

{

if(data->all\_users\_sockets[i] != INVALID\_SOCKET && data->all\_users\_sockets[i] != data->socket)

{

if(write\_to\_socket(data->all\_users\_sockets[i], data->message) == FALSE)

{

write\_log\_and\_print("Failed to send player won message, Error\_code: 0x%x\n", GetLastError());

return FALSE;

}

}

}

won\_event = InitEvent("PlayerWon");

if(won\_event == NULL)

{

write\_log\_and\_print("Failed to create PlayerWon Event, Error\_code: 0x%x\n",GetLastError());

return FALSE;

}

SetEvent(won\_event);

return TRUE;

}

BOOL HandlePlayersMessage(communication\_data\* data)

{

char players\_message[MAX\_PLAYERS\_LIST\_MESSAGE\_LENGTH];

int i = 0;

int j = 0;

BOOL send\_result;

memset(players\_message, '\0', MAX\_PLAYERS\_LIST\_MESSAGE\_LENGTH);

for(i = 0; i < MAX\_NUM\_OF\_PLAYERS; i++)

{

if(data->all\_users\_sockets[i] != INVALID\_SOCKET)

{

if(i > 0)

j += sprintf(players\_message + j, ", ");

j += sprintf(players\_message + j, "%s-%c", data->all\_users[i], data->all\_symbols[i]);

}

}

strcat(players\_message, ".\n");

send\_result = write\_to\_socket(data->socket, players\_message);

return send\_result;

}

BOOL ShouldFinishThread(HANDLE all\_threads\_must\_end\_event)

{

DWORD allThreadsMustEndSignal = WaitForSingleObject(all\_threads\_must\_end\_event,0);

switch(allThreadsMustEndSignal)

{

case WAIT\_OBJECT\_0:

return TRUE;

break;

case WAIT\_TIMEOUT:

return FALSE;

break;

default:

return TRUE;

break;

}

}

ServerCommunication.h:

#define \_WINSOCKAPI\_ // stops windows.h including winsock.h

#include <Windows.h>

#include <stdio.h>

#include "Log.h"

#include "Mutex.h"

#include "MutexConstants.h"

#include "GameConsts.h"

#include "SocketWrapper.h"

typedef struct communication\_data

{

SOCKET socket;

char message[MAX\_COMMAND\_LENGTH];

char username[MAX\_USER\_NAME\_LENGTH];

char symbol;

char (\*all\_users)[MAX\_USER\_NAME\_LENGTH];

char \*all\_symbols;

SOCKET\* all\_users\_sockets;

HANDLE all\_threads\_must\_end\_event;

} communication\_data;

DWORD WINAPI ServerCommunicationThreadStart(LPVOID param);

void StartServerCommunication(communication\_data\* data);

BOOL HandleIncomingMessage(communication\_data\* data);

BOOL HandleSendMessage(communication\_data\* data);

BOOL HandlePlayerTurnMessage(communication\_data\* data);

BOOL HandlePlayerWonMessage(communication\_data\* data);

BOOL HandlePlayersMessage(communication\_data\* data);

BOOL ShouldFinishThread(HANDLE all\_threads\_must\_end\_event);

ServerGameManagment.c:

#include "ServerGameManagement.h"

void start\_server(int port)

{

char users[MAX\_NUM\_OF\_PLAYERS][MAX\_USER\_NAME\_LENGTH];

SOCKET user\_sockets[MAX\_NUM\_OF\_PLAYERS] = {INVALID\_SOCKET};

char symbols[MAX\_NUM\_OF\_PLAYERS] = {'@','#','%','\*'};

communication\_data players\_communication\_data[MAX\_NUM\_OF\_PLAYERS];

HANDLE players\_communication\_thread[MAX\_NUM\_OF\_PLAYERS] = {NULL};

int num\_of\_threads = 0;

HANDLE all\_threads\_must\_end\_event;

all\_threads\_must\_end\_event = InitEvent("AllThreadsMustEnd");

if(all\_threads\_must\_end\_event == NULL)

{

write\_log\_and\_print("Failed to create event!. Error\_code: 0x%x\n", GetLastError());

return;

}

if(WaitForPlayers(port, user\_sockets, users, symbols, players\_communication\_data, players\_communication\_thread, all\_threads\_must\_end\_event) == FALSE)

{

CloseConnections(user\_sockets);

return;

}

WriteToLogOrderOfPlayers(user\_sockets, users, symbols);

if(BroadcastPlayers(user\_sockets, users, symbols) == FALSE)

{

write\_log\_and\_print("Failed to send players list to all players, Error\_code: 0x%x\n", GetLastError());

CloseConnections(user\_sockets);

return;

}

if(PlayGame(users, user\_sockets, symbols, all\_threads\_must\_end\_event) == FALSE)

{

write\_log\_and\_print("Failed to play game, Error\_code: 0x%x\n", GetLastError());

CloseConnections(user\_sockets);

return;

}

while(players\_communication\_thread[num\_of\_threads] != NULL)

num\_of\_threads++;

WaitForMultipleObjects(num\_of\_threads, players\_communication\_thread, TRUE, INFINITE);

CloseConnections(user\_sockets);

}

BOOL WaitForPlayers(int port, SOCKET user\_sockets[MAX\_NUM\_OF\_PLAYERS],

char users[MAX\_NUM\_OF\_PLAYERS][MAX\_USER\_NAME\_LENGTH], char symbols[MAX\_NUM\_OF\_PLAYERS],

communication\_data players\_communications[MAX\_NUM\_OF\_PLAYERS], HANDLE players\_communication\_thread[MAX\_NUM\_OF\_PLAYERS],

HANDLE all\_threads\_must\_end\_event)

{

SOCKET listen\_sock;

int i = 0;

int connected\_users\_count = 0;

time\_t start\_time;

time\_t stop\_time;

char player\_symbol[2];

for(i = 0; i < MAX\_NUM\_OF\_PLAYERS; i++)

{

user\_sockets[i] = INVALID\_SOCKET;

memset(users[i],'\0', MAX\_USER\_NAME\_LENGTH);

}

if(sock\_listen(port, MAX\_NUM\_OF\_PLAYERS, &listen\_sock) == FALSE)

{

write\_log\_and\_print("Failed on listening port:%d, Error\_code: 0x%x\n",port, GetLastError());

return FALSE;

}

time(&start\_time);

while(connected\_users\_count < MAX\_NUM\_OF\_PLAYERS)

{

//wait for connection to occure

while(user\_sockets[connected\_users\_count] == INVALID\_SOCKET)

{

time(&stop\_time);

if(difftime(stop\_time,start\_time) > MAX\_TIME\_TO\_WAIT\_FOR\_PLAYERS)

{

if(i == 0)

{

write\_log\_and\_print("No players connected, exiting...\n");

close\_socket(listen\_sock);

return FALSE;

}

else

{

close\_socket(listen\_sock);

return TRUE;

}

}

Sleep(100);

if(accept\_connection(listen\_sock, &user\_sockets[connected\_users\_count]) == FALSE)

{

close\_socket(listen\_sock);

return FALSE;

}

}

lock\_mutex(BROADCAST\_MUTEX);

if(ReceiveUsername(user\_sockets[connected\_users\_count], users[connected\_users\_count]) == FALSE)

{

close\_socket(listen\_sock);

unlock\_mutex(BROADCAST\_MUTEX);

return FALSE;

}

if(IsUsernameExists(users[connected\_users\_count], users, connected\_users\_count) == TRUE)

{

write\_log\_format("Cannot accept connection. Username(%s) already exists\n", users[connected\_users\_count]);

write\_to\_socket(user\_sockets[connected\_users\_count], "Cannot accept connection. Username already exists\n");

close\_socket(user\_sockets[connected\_users\_count]);

user\_sockets[connected\_users\_count] = INVALID\_SOCKET;

memset(users[connected\_users\_count],'\0', MAX\_USER\_NAME\_LENGTH);

unlock\_mutex(BROADCAST\_MUTEX);

continue;

}

if(SendWelcomeMessage(user\_sockets[connected\_users\_count], users[connected\_users\_count], symbols[connected\_users\_count]) == FALSE)

{

close\_socket(listen\_sock);

unlock\_mutex(BROADCAST\_MUTEX);

return FALSE;

}

if(BroadcastNewPlayerJoined(user\_sockets, connected\_users\_count, users[connected\_users\_count], symbols[connected\_users\_count]) == FALSE)

{

close\_socket(listen\_sock);

unlock\_mutex(BROADCAST\_MUTEX);

return FALSE;

}

players\_communications[connected\_users\_count].socket = user\_sockets[connected\_users\_count];

strcpy(players\_communications[connected\_users\_count].username, users[connected\_users\_count]);

players\_communications[connected\_users\_count].symbol = symbols[connected\_users\_count];

memset(players\_communications[connected\_users\_count].message, '\0', MAX\_COMMAND\_LENGTH);

players\_communications[connected\_users\_count].all\_users\_sockets = user\_sockets;

players\_communications[connected\_users\_count].all\_users = users;

players\_communications[connected\_users\_count].all\_symbols = symbols;

players\_communications[connected\_users\_count].all\_threads\_must\_end\_event = all\_threads\_must\_end\_event;

players\_communication\_thread[connected\_users\_count] = CreateThread(NULL, 0, ServerCommunicationThreadStart, &(players\_communications[connected\_users\_count]), 0, NULL);

if(players\_communication\_thread[connected\_users\_count] == NULL)

{

write\_log\_and\_print("Failed to create ServerCommunicationThread for user: %s, ErrorCode: 0x%x\n", users[connected\_users\_count], GetLastError());

close\_socket(listen\_sock);

unlock\_mutex(BROADCAST\_MUTEX);

return FALSE;

}

connected\_users\_count++;

unlock\_mutex(BROADCAST\_MUTEX);

}

close\_socket(listen\_sock);

return TRUE;

}

BOOL PlayGame(char users[MAX\_NUM\_OF\_PLAYERS][MAX\_USER\_NAME\_LENGTH], SOCKET user\_sockets[MAX\_NUM\_OF\_PLAYERS], char symbols[MAX\_NUM\_OF\_PLAYERS], HANDLE all\_threads\_must\_end\_event)

{

int current\_player = 0;

int i = 0;

int num\_of\_players = 0;

char player\_turn\_message[MAX\_PLAYER\_TURN\_MESSAGE\_LENGTH];

HANDLE wait\_handles[3];

DWORD wait\_result = 0;

for(i = 0; i < MAX\_NUM\_OF\_PLAYERS; i++)

{

if(user\_sockets[i] != INVALID\_SOCKET)

num\_of\_players++;

}

wait\_handles[0] = InitEvent("TurnFinished");

if(wait\_handles[0] == NULL)

{

write\_log\_and\_print("Failed creating TurnFinished Event, Error\_code: 0x%x\n", GetLastError());

return FALSE;

}

wait\_handles[1] = InitEvent("PlayerWon");

if(wait\_handles[1] == NULL)

{

write\_log\_and\_print("Failed creating PlayerWon Event, Error\_code: 0x%x\n", GetLastError());

return FALSE;

}

wait\_handles[2] = all\_threads\_must\_end\_event;

while(1)

{

if(ShouldFinishExecution(all\_threads\_must\_end\_event) == TRUE)

return FALSE;

lock\_mutex(BROADCAST\_MUTEX);

write\_log("Your turn to play.\n");

if(write\_to\_socket(user\_sockets[current\_player], "Your turn to play.\n") == FALSE)

{

write\_log\_and\_print("Failed to write to socket, Error\_code: 0x%x\n", GetLastError());

unlock\_mutex(BROADCAST\_MUTEX);

return FALSE;

}

memset(player\_turn\_message, '\0', MAX\_PLAYER\_TURN\_MESSAGE\_LENGTH);

sprintf(player\_turn\_message, "It is now %s's turn to play.\n", users[current\_player]);

write\_log(player\_turn\_message);

for(i = 0; i < num\_of\_players; i++)

{

if(i != current\_player)

{

if(write\_to\_socket(user\_sockets[i], player\_turn\_message) == FALSE)

{

unlock\_mutex(BROADCAST\_MUTEX);

return FALSE;

}

}

}

unlock\_mutex(BROADCAST\_MUTEX);

wait\_result = WaitForMultipleObjects(3, wait\_handles, FALSE, INFINITE);

switch(wait\_result)

{

case WAIT\_OBJECT\_0: // Turn Finished

ResetEvent(wait\_handles[0]);

break;

case WAIT\_OBJECT\_0 + 1: // Player Won

return TRUE;

break;

case WAIT\_OBJECT\_0 + 2: // all threads must end

return FALSE;

break;

default:

write\_log\_and\_print("Unexpected wait result, result: %d, Error\_code: 0x%x\n", wait\_result, GetLastError());

return FALSE;

break;

}

current\_player++;

current\_player = current\_player % num\_of\_players;

}

}

void WriteToLogOrderOfPlayers(SOCKET user\_sockets[MAX\_NUM\_OF\_PLAYERS], char users[MAX\_NUM\_OF\_PLAYERS][MAX\_USER\_NAME\_LENGTH], char symbols[MAX\_NUM\_OF\_PLAYERS])

{

int i = 0;

write\_log("The order of players' in the game is ");

for(i = 0; i<MAX\_NUM\_OF\_PLAYERS; i++)

{

if(user\_sockets[i] != INVALID\_SOCKET)

{

if(i > 0)

write\_log(",");

write\_log(users[i]);

}

}

write\_log(".\n");

}

BOOL BroadcastPlayers(SOCKET user\_sockets[MAX\_NUM\_OF\_PLAYERS], char users[MAX\_NUM\_OF\_PLAYERS][MAX\_USER\_NAME\_LENGTH], char symbols[MAX\_NUM\_OF\_PLAYERS])

{

int i = 0;

lock\_mutex(BROADCAST\_MUTEX);

for(i = 0; i < MAX\_NUM\_OF\_PLAYERS; i++)

{

if(user\_sockets[i] != INVALID\_SOCKET)

{

if(SendPlayersToUser(user\_sockets[i], user\_sockets, users, symbols) == FALSE)

{

unlock\_mutex(BROADCAST\_MUTEX);

return FALSE;

}

}

}

unlock\_mutex(BROADCAST\_MUTEX);

write\_log("Players' game pieces' selection broadcasted to all users.\n");

return TRUE;

}

BOOL SendPlayersToUser(SOCKET user\_sock, SOCKET user\_sockets[MAX\_NUM\_OF\_PLAYERS], char users[MAX\_NUM\_OF\_PLAYERS][MAX\_USER\_NAME\_LENGTH], char symbols[MAX\_NUM\_OF\_PLAYERS])

{

char players\_message[MAX\_PLAYERS\_LIST\_MESSAGE\_LENGTH];

int i = 0;

int j = 0;

memset(players\_message, '\0', MAX\_PLAYERS\_LIST\_MESSAGE\_LENGTH);

for(i = 0; i < MAX\_NUM\_OF\_PLAYERS; i++)

{

if(user\_sockets[i] != INVALID\_SOCKET)

{

if(i > 0)

j += sprintf(players\_message + j, ", ");

j += sprintf(players\_message + j, "%s-%c", users[i], symbols[i]);

}

}

strcat(players\_message, ".\n");

return write\_to\_socket(user\_sock, players\_message);

}

void CloseConnections(SOCKET user\_sockets[MAX\_NUM\_OF\_PLAYERS])

{

int i =0;

lock\_mutex(BROADCAST\_MUTEX);

for(i = 0; i<MAX\_NUM\_OF\_PLAYERS; i++)

{

if(user\_sockets[i] != INVALID\_SOCKET)

close\_socket(user\_sockets[i]);

}

unlock\_mutex(BROADCAST\_MUTEX);

}

BOOL ReceiveUsername(SOCKET user\_sock, char username[MAX\_USER\_NAME\_LENGTH])

{

char username\_message[MAX\_USER\_NAME\_MESSAGE\_LENGTH];

char\* username\_prefix;

char\* username\_suffix;

memset(username\_message, '\0', MAX\_USER\_NAME\_MESSAGE\_LENGTH);

if(receive\_from\_socket(user\_sock, username\_message) == FALSE)

{

write\_log\_and\_print("Failed to receive username message, ErrorCode: 0x%x\n", GetLastError());

return FALSE;

}

username\_prefix = strtok(username\_message, "=");

username\_suffix = strtok(NULL, "\n");

if(strcmp(username\_prefix, "username") != 0)

{

printf("Unexpected username message received! received message: %s", username\_message);

write\_log("Unexpected username message received! received message: %s", username\_message);

return FALSE;

}

strcpy(username, username\_suffix);

return TRUE;

}

BOOL SendWelcomeMessage(SOCKET user\_sock, char username[MAX\_USER\_NAME\_LENGTH], char user\_symbol)

{

char server\_welcome\_message[MAX\_WELCOME\_MESSAGE\_LENGTH];

memset(server\_welcome\_message, '\0', MAX\_WELCOME\_MESSAGE\_LENGTH);

sprintf(server\_welcome\_message, "%s your game piece is %c\n", username, user\_symbol);

write\_log(server\_welcome\_message);

if(write\_to\_socket(user\_sock, server\_welcome\_message) == FALSE)

{

write\_log\_and\_print("Failed to send game piece to user, Error\_code: 0x%x\n",GetLastError());

return FALSE;

}

return TRUE;

}

BOOL BroadcastNewPlayerJoined(SOCKET user\_sockets[MAX\_NUM\_OF\_PLAYERS], int index\_of\_new\_player, char new\_player\_username[MAX\_USER\_NAME\_LENGTH], char new\_player\_symbol)

{

char player\_joined\_message[MAX\_PLAYER\_JOINED\_MESSAGE\_LENGTH];

char new\_player\_symbol\_str[2];

int i = 0;

memset(player\_joined\_message, '\0', MAX\_PLAYER\_JOINED\_MESSAGE\_LENGTH);

sprintf(player\_joined\_message, "New player joined the game: %s %c\n", new\_player\_username, new\_player\_symbol);

write\_log(player\_joined\_message);

for(i = 0; i < index\_of\_new\_player; i++)

{

if(write\_to\_socket(user\_sockets[i], player\_joined\_message) == FALSE)

{

write\_log\_and\_print("Failed to send player joined message! ErrorCode: 0x%x\n", GetLastError());

return FALSE;

}

}

return TRUE;

}

BOOL IsUsernameExists(char username[MAX\_USER\_NAME\_LENGTH], char users[MAX\_NUM\_OF\_PLAYERS][MAX\_USER\_NAME\_LENGTH], int user\_index)

{

int i = 0;

for(i = 0; i < user\_index; i++)

{

if(strcmp(username, users[i]) == 0)

return TRUE;

}

return FALSE;

}

BOOL ShouldFinishExecution(HANDLE all\_threads\_must\_end\_event)

{

DWORD allThreadsMustEndSignal = WaitForSingleObject(all\_threads\_must\_end\_event,0);

switch(allThreadsMustEndSignal)

{

case WAIT\_OBJECT\_0:

return TRUE;

break;

case WAIT\_TIMEOUT:

return FALSE;

break;

default:

return TRUE;

break;

}

}

ServerGameManagment.h:

#define \_WINSOCKAPI\_ // stops windows.h including winsock.h

#include <stdio.h>

#include <time.h>

#include "Log.h"

#include "Semaphore.h"

#include "Mutex.h"

#include "MutexConstants.h"

#include "SocketWrapper.h"

#include "ServerCommunication.h"

#include "GameConsts.h"

#define MAX\_TIME\_TO\_WAIT\_FOR\_PLAYERS 60

void start\_server(int port);

BOOL WaitForPlayers(int port, SOCKET user\_sockets[MAX\_NUM\_OF\_PLAYERS],

char users[MAX\_NUM\_OF\_PLAYERS][MAX\_USER\_NAME\_LENGTH], char symbols[MAX\_NUM\_OF\_PLAYERS],

communication\_data players\_communications[MAX\_NUM\_OF\_PLAYERS], HANDLE players\_communication\_thread[MAX\_NUM\_OF\_PLAYERS],

HANDLE all\_threads\_must\_end\_event);

BOOL PlayGame(char users[MAX\_NUM\_OF\_PLAYERS][MAX\_USER\_NAME\_LENGTH], SOCKET user\_sockets[MAX\_NUM\_OF\_PLAYERS], char symbols[MAX\_NUM\_OF\_PLAYERS], HANDLE all\_threads\_must\_end\_event);

void CloseConnections(SOCKET user\_sockets[MAX\_NUM\_OF\_PLAYERS]);

void WriteToLogOrderOfPlayers(SOCKET user\_sockets[MAX\_NUM\_OF\_PLAYERS], char users[MAX\_NUM\_OF\_PLAYERS][MAX\_USER\_NAME\_LENGTH], char symbols[MAX\_NUM\_OF\_PLAYERS]);

BOOL BroadcastPlayers(SOCKET user\_sockets[MAX\_NUM\_OF\_PLAYERS], char users[MAX\_NUM\_OF\_PLAYERS][MAX\_USER\_NAME\_LENGTH], char symbols[MAX\_NUM\_OF\_PLAYERS]);

BOOL SendPlayersToUser(SOCKET user\_sock, SOCKET user\_sockets[MAX\_NUM\_OF\_PLAYERS], char users[MAX\_NUM\_OF\_PLAYERS][MAX\_USER\_NAME\_LENGTH], char symbols[MAX\_NUM\_OF\_PLAYERS]);

BOOL ReceiveUsername(SOCKET user\_sock, char username[MAX\_USER\_NAME\_LENGTH]);

BOOL SendWelcomeMessage(SOCKET user\_sock, char username[MAX\_USER\_NAME\_LENGTH], char user\_symbol);

BOOL BroadcastNewPlayerJoined(SOCKET user\_sockets[MAX\_NUM\_OF\_PLAYERS], int index\_of\_new\_player, char new\_player\_username[MAX\_USER\_NAME\_LENGTH], char new\_player\_symbol);

BOOL IsUsernameExists(char username[MAX\_USER\_NAME\_LENGTH], char users[MAX\_NUM\_OF\_PLAYERS][MAX\_USER\_NAME\_LENGTH], int user\_index);

void CloseThreads(HANDLE players\_communication\_thread[MAX\_NUM\_OF\_PLAYERS]);

BOOL ShouldFinishExecution();

SocketWrapper.c:

#include "SocketWrapper.h"

BOOL init\_WSA()

{

WSADATA wsaData;

int iResult = WSAStartup(MAKEWORD(2, 2), &wsaData);

if (iResult != NO\_ERROR)

{

SetLastError(iResult);

return FALSE;

}

return TRUE;

}

void clean\_WSA()

{

WSACleanup();

}

BOOL sock\_listen(int port, int max\_connections, SOCKET\* listen\_sock)

{

SOCKADDR\_IN listen\_addr;

u\_long nonblockingMode = 1;

int iResult = 0;

\*listen\_sock = socket(AF\_INET, SOCK\_STREAM, IPPROTO\_TCP);

if(\*listen\_sock == INVALID\_SOCKET)

{

SetLastError(WSAGetLastError());

return FALSE;

}

listen\_addr.sin\_family = AF\_INET;

listen\_addr.sin\_port = htons(port);

listen\_addr.sin\_addr.s\_addr = inet\_addr(SRV\_ADDRESS);

if(bind(\*listen\_sock, (SOCKADDR\*)&listen\_addr, sizeof(listen\_addr)) == SOCKET\_ERROR)

{

SetLastError(WSAGetLastError());

return FALSE;

}

if(listen(\*listen\_sock, SOMAXCONN) == SOCKET\_ERROR)

{

SetLastError(WSAGetLastError());

return FALSE;

}

iResult = ioctlsocket(\*listen\_sock, FIONBIO, &nonblockingMode);

if (iResult != NO\_ERROR)

return FALSE;

return TRUE;

}

void close\_socket(SOCKET sock)

{

char buffer[2];

int result = 0;

shutdown(sock, SD\_SEND);

do

{

result = recv(sock, buffer, 1,0);

} while( result > 0);

closesocket(sock);

}

BOOL accept\_connection(SOCKET listen\_socket, SOCKET\* accepted\_socket)

{

struct sockaddr\_in connect\_socket\_addr;

int addr\_len;

\*accepted\_socket = INVALID\_SOCKET;

addr\_len = sizeof(connect\_socket\_addr);

\*accepted\_socket = accept(listen\_socket, (struct sockaddr\*)&connect\_socket\_addr, &addr\_len);

if(\*accepted\_socket == INVALID\_SOCKET)

{

if(WSAGetLastError() == WSAEWOULDBLOCK)

{

//no waiting socket.. return true but not socket was accepted!

WSASetLastError(0);

return TRUE;

}

SetLastError(WSAGetLastError());

return FALSE;

}

return TRUE;

}

//Receive a message from the socket until '\n' recieved

BOOL receive\_from\_socket(SOCKET socket, char\* buffer)

{

char\* cur\_place\_ptr = buffer;

int bytes\_just\_transferred;

while (1)

{

/\* send does not guarantee that the entire message is sent \*/

bytes\_just\_transferred = recv(socket, cur\_place\_ptr, 1, 0);

while(WSAGetLastError() == WSAEWOULDBLOCK) //when the socket is nonblocking: no data available yet

{

WSASetLastError(0);

Sleep(100);

bytes\_just\_transferred = recv(socket, cur\_place\_ptr, 1, 0);

}

if (bytes\_just\_transferred == SOCKET\_ERROR)

{

SetLastError(WSAGetLastError());

return FALSE;

}

else if (bytes\_just\_transferred == 0)

return FALSE; // recv() returns zero if connection was gracefully disconnected.

cur\_place\_ptr += bytes\_just\_transferred; // <ISP> pointer arithmetic

if (cur\_place\_ptr[-1] == '\n') {

//Received all message

return TRUE;

}

}

return TRUE;

}

//Writes the message\_to\_send to the socket

BOOL write\_to\_socket(SOCKET socket, char \*message\_to\_send)

{

const char\* cur\_place\_ptr = message\_to\_send;

int bytes\_transferred;

int remaining\_bytes\_to\_send = strlen(message\_to\_send);

while ( remaining\_bytes\_to\_send > 0 )

{

/\* send does not guarantee that the entire message is sent \*/

bytes\_transferred = send(socket, cur\_place\_ptr, remaining\_bytes\_to\_send, 0);

if ( bytes\_transferred == SOCKET\_ERROR )

{

SetLastError(WSAGetLastError());

return FALSE;

}

remaining\_bytes\_to\_send -= bytes\_transferred;

cur\_place\_ptr += bytes\_transferred; // <ISP> pointer arithmetic

}

return TRUE;

}

//Connects to local socket to specific port. return TRUE if succeed

BOOL connect\_socket(int port, SOCKET \*socket\_client)

{

SOCKADDR\_IN clientService;

\*socket\_client = socket(AF\_INET, SOCK\_STREAM, IPPROTO\_TCP);

if(\*socket\_client == INVALID\_SOCKET)

{

SetLastError(WSAGetLastError());

return FALSE;

}

clientService.sin\_family = AF\_INET;

clientService.sin\_port = htons(port);

clientService.sin\_addr.s\_addr = inet\_addr(SRV\_ADDRESS);

if (connect(\*socket\_client, (SOCKADDR\*) &clientService, sizeof(clientService) ) == SOCKET\_ERROR)

{

SetLastError(WSAGetLastError());

WSACleanup();

return FALSE;

}

return TRUE;

}

SocketWrapper.h:

#ifndef SOCKET\_WRAPPER\_HEADER

#define SOCKER\_WRAPPER\_HEADER

#define \_WINSOCKAPI\_ // stops windows.h including winsock.h

#include <winsock2.h>

#include <windows.h>

#include <conio.h>

#include "GameConsts.h"

#pragma comment(lib, "ws2\_32.lib")

#define SRV\_ADDRESS "127.0.0.1"

BOOL init\_WSA();

void clean\_WSA();

BOOL accept\_connection(SOCKET listen\_sock, SOCKET\* accepted\_sock);

BOOL sock\_listen(int port, int max\_connections, SOCKET\* listen\_sock);

void close\_socket(SOCKET sock);

BOOL receive\_from\_socket(SOCKET sock, char\* received\_message);

BOOL write\_to\_socket(SOCKET sock, char\* message\_to\_send);

BOOL connect\_socket(int port, SOCKET\* socket);

#endif

UiManager.c:

#include "UiManager.h"

#include "Mutex.h"

#include "Semaphore.h"

#include "Events.h"

void ReadFromClient(char \*command);

//Runs the ui thread

DWORD WINAPI RunUiManager(LPVOID lpParam)

{

data\_ui \*data = (data\_ui \*)lpParam;

data->PlayersTurnEvent = InitEvent("PlayersTurnEvent");

while (1)

{

//Reads an input for the client

ReadFromClient(data->command);

//Notifies the engine that a message received

ReleaseSemaphoreSimple(data->UserEnteredTextSemaphore);

//Waits for the engine to handle the message

WaitForSingleObject(data->EngineDoneWithUserMessageSemaphore, INFINITE);

}

return GetLastError();

}

//Reads a message from the client until '\n'

void ReadFromClient(char \*command)

{

int index = 0;

char current\_char;

//Resets the buffer

memset(command, '\0', MAX\_COMMAND\_LENGTH);

while ((current\_char = getchar()) != '\n')

{

command[index++] = current\_char;

}

command[index] = '\n';

}

UiManager.h:

#ifndef UI\_MANEGER\_HEADER

#define UI\_MANEGER\_HEADER

#include <Windows.h>

#include "GameConsts.h"

DWORD WINAPI RunUiManager(LPVOID lpParam);

void ReadFromClient(char \*command);

typedef struct data\_ui

{

HANDLE UserEnteredTextSemaphore;

HANDLE EngineDoneWithUserMessageSemaphore;

HANDLE PlayersTurnEvent;

char command[MAX\_COMMAND\_LENGTH];

} data\_ui;

#endif