Rzymkowice
27.01.2022

Semester: 3

Group: MS_Section (4)

Section: MS_Section

Computer Programming Laboratory

Go Game

Author: Jakub Kuzak

E-mail: jakukuz814@polsl.pl

Tutor: Michał Staniszewski

1. *Task topic*

My task was to write the game "Go Game" with the appropriate logic of the game and the form of playing in the following modes:
- player vs player
- player vs computer
The game should include showing who won a given game as well as some kind of saving / reading of the game.

2. *Project analysis*

At the very beginning, to create the game, I needed an external game library, I decided that Allegro 5 would be appropriate.

Then, in order for the program to work properly, I needed to create a class board that would be responsible for creating and maintaining the board on which the results will be shown, the possibility of surrendering and the game itself.

Using the Allegro I had to implement a mouse service, because this is my only way to interact with the game. The mouse handles events such as closing the game by clicking on the X in the upper right corner of the program, as well as all interactive places where it can click, for example:
- each button has a specific range where it can be clicked to make it work
- the board that creates pieces for the go game.

Using allegro 5 I had to somehow draw the whole board and then create a method that would create stones (pawns of the game) in the right place.

To facilitate the work, I have compiled a list of legal neighbors around the pawn position. I also had to take care and include in the code dealing with suicide, because a pawn that has no breaths cannot be created.

To handle the font of the file i used this form that load the Arial from the Windows, if the machine running the program does not have access to Arial, the game uses the built-in adapter in allegro for proper operation.

```
ALLEGRO_FONT* font = al_load_font("c:/windows/fonts/Arial.ttf", 22,
NULL);
   if (font == NULL) {
         font = al_create_builtin_font();
   }
```

3. *External specification*

The user, after launching the game, can choose from 3 options, but not during the first launch, because the option to load the last game has not been used yet, so after pressing the "Load Game" button, nothing will happen. The user has a choice, therefore, to play against another player (usually, the other player is needed) or the option of playing with a computer that will not take into account your decisions, but will only create pawns with the logic of the game, during such a game you can devise strategies and familiarize yourself with the mechanics games.

After starting the game in the computer mode, you will be shown on the interface, where you can see the board, the field that counts the points of the black and white players and who currently has the move, because you can get confused with the larger boards. The "pass" option in the computer mode will end the game and show who won the game, that is the person who scored more points by capturing the opponent's pieces. The difference is in the two-player mode, because the first press of the pass button causes the player to take the turn and at this point the opponent is faced with the option of surrendering (the pass button has changed into "end game") or continuing the game, which will end until two players surrender.

Regardless of the selected game mode, the player who clicks the X in the top right corner will close the game, and thus save the current game state, which can be resumed the next time the game is started using the "Load Game" button in the menu.

The game can also be run from go-game.exe without any needed file.

4. *Internal specification*

In the Stone.h i created a class Stone that handles the type of the stone and is the base class with virtual int GetType().

In the Position.h i created a class Position that has a constructor that gives us position x and y.

In the Black.h i created a class Black that inverits publicly from class Stone and returns stone_type_black.
In the White.h i created a class White that inverits publicly from class Stone and returns stone_type_white.

In the Go Game.h that has all the methods used in Go Game.cpp:

void draw_board(Board* board, int& current, ALLEGRO_FONT* font);

void draw_surrender(ALLEGRO_FONT* font);

void draw_point(Board* board, ALLEGRO_FONT* font);

void draw_current_player(int& current, ALLEGRO_FONT* font);

void horizontal_lines(float half_cell);

void vertical_lines(float half_cell);

void mouse_click(ALLEGRO_EVENT& event, int& current, int& retflag, Board* board);

void drawing_stone(float half_cell);

void drawing_board(float half_cell);

void draw_stones(float half_cell, Board* board);

void draw_menu(ALLEGRO_FONT* font);

void handle_menu(ALLEGRO_FONT* font);

void handle_menu(int ix, int iy, int& current, Board* board);

void save_game(int& current, Board* board);

void flip_side(int& current);

void handle_end_game(Board* board, int& current);

In the Board.h i used methods and classes like:

class IllegalMove_Exception

class Board that handles all the staff necessary to properly utilize the board.

void init_board()

Stone* GetAtLocation(int y, int x)

void clear_at_location(int y, int x)

void SetAtLocation(int y, int x,Stone* stone)

void handle_enemy(int y, int x, Stone* stone)

bool handle_suicide(int y, int x, Stone* stone)

list <Position*> legal_neighbours(Position* pos)

list <Position*>* group(Position* start)

void reccur_group(Position* start,list<Position*>* group)

bool have_breath(list<Position*>* group)

int get_black_score()

int get_white_score()

void board_reset()

5. *Source code*

- Go Game.cpp

```
#include <iostream>
#include <fstream>
#include <allegro5/allegro.h>
#include <allegro5/allegro_ttf.h>
#include <allegro5/allegro_image.h>
#include <allegro5/allegro_primitives.h>
#include <allegro5/allegro_native_dialog.h>
#include "Go Game.h"
#include <random>
#include <string>

#define ALLEGRO_STATICLINK

using namespace std;

void make_AI_move(int& current, Board* board);


#define BLACK (1)
#define WHITE (2)
#define CELL_SIZE (40)

int Stone::stone_type_empty = 0;
```

```
int White::stone_type_white = 2;
int Black::stone_type_black = 1;

int AI = 1;

random_device rd; // obtain a random number from hardware
mt19937 gen(rd()); // seed the generator
uniform_int_distribution<> distr(0, 18); // define the range

enum Game_State{menu,ai,multi};
int STATE = menu;
int SURRENDER = 0;

ALLEGRO_DISPLAY* display;

int main()
{

    Board* board = new Board;
    bool running = true;
    float x = 0, y = 0;
    int current = BLACK;

    al_init();
    al_init_font_addon();
    al_init_ttf_addon();
    al_init_image_addon();
    al_install_mouse();
    al_init_primitives_addon();


    //anti-aliasing
    al_set_new_display_option(ALLEGRO_SAMPLE_BUFFERS, 1, ALLEGRO_SUGGEST);
    al_set_new_display_option(ALLEGRO_SAMPLES, 8, ALLEGRO_SUGGEST);

    ALLEGRO_EVENT_QUEUE* queue;
    display = al_create_display(CELL_SIZE * 23, CELL_SIZE * 19);
    ALLEGRO_TIMER* timer = al_create_timer(1.0 / 60);


    ALLEGRO_FONT* font = al_load_font("c:/windows/fonts/Arial.ttf", 22, NULL);
    if (font == NULL) {
            font = al_create_builtin_font();
    }


    queue = al_create_event_queue();
    al_register_event_source(queue, al_get_display_event_source(display));
    al_register_event_source(queue, al_get_mouse_event_source());
    al_register_event_source(queue, al_get_timer_event_source(timer));
    al_start_timer(timer);



    while (running) {

            ALLEGRO_EVENT event;
            al_wait_for_event(queue, &event);
```

```
		if (event.type == ALLEGRO_EVENT_TIMER) {


			if (STATE == menu) {
				draw_menu(font);
			}
			else {
				draw_board(board,current,font);
			}


		}


		if (event.type == ALLEGRO_EVENT_DISPLAY_CLOSE) {
			if (STATE != menu) {
				save_game(current, board);
			}
			running = false;

		}



		if (event.type == ALLEGRO_EVENT_MOUSE_BUTTON_UP) {
			int retflag;
			mouse_click(event, current, retflag, board);


			if (retflag == 3) continue;

		}


		if (STATE == ai && AI == current) {

			make_AI_move(current, board);

		}


	}

	al_destroy_display(display);
	al_uninstall_mouse();
	al_destroy_font(font);
	delete board;

	return 0;
}

int make_move(int y, int x, int& current, Board* board) {
	try {
		if (current == BLACK) {
			board->SetAtLocation(y, x, new Black);
		}
		else {
```

```
                board->SetAtLocation(y, x, new White);
        }

        flip_side(current);
    }
    catch (IllegalMove_Exception* e) {
        // illegal move
        return 1;
    }
    return 0;
}

void flip_side(int& current)
{
    if (current == BLACK) {
        current = WHITE;
    }
    else {
        current = BLACK;
    }
}

void save_game(int& current, Board* board) {

    ofstream saved_file;
    saved_file.open("saved_file.txt");
    saved_file << STATE << " "<< current << " ";
    for (int i = 0;i < 19;i++) {
        for (int j = 0;j < 19;j++) {
            if (board->GetAtLocation(j, i)->GetType() == BLACK) {
                saved_file << BLACK << " ";
            }
            else if (board->GetAtLocation(j, i)->GetType() == WHITE) {
                saved_file << WHITE << " ";
            }
            else {
                saved_file << 0 << " ";
            }

        }
    }
    saved_file.close();

}

void load_game(int& current, Board* board) {

    int tempState;
    int tempCurrent;
    ifstream load_file;
    load_file.open("saved_file.txt");
    if (load_file.is_open() != true) {
        return;
    }
    load_file >> tempState;
    load_file >> tempCurrent;
    current = tempCurrent;
    STATE = tempState;
    for (int i = 0;i < 19;i++) {
```

```
		for (int j = 0;j < 19;j++) {
			int num;
			load_file >> num;

			if (num == Black::stone_type_black) {
				board->SetAtLocation(j, i, new Black);
			}
			if (num == White::stone_type_white) {
				board->SetAtLocation(j, i, new White);
			}
		}
	}
	load_file.close();

}

void make_AI_move(int& current, Board* board) {
	while (true) {
		int x = distr(gen);
		int y = distr(gen);

		if (make_move(y, x, current, board) == 0) {
			return;
		}
	}
}

void mouse_click(ALLEGRO_EVENT& event, int& current, int& retflag, Board* board)
{
	if (STATE == menu) {
		int ix = event.mouse.x;
		int iy = event.mouse.y;
		handle_menu(ix, iy, current, board);

	}
	else {
		if (event.mouse.x >= (CELL_SIZE * 18.9) && event.mouse.x <= (CELL_SIZE * 22.7)) {
			if (event.mouse.y >= (CELL_SIZE * 5) && event.mouse.y <= (CELL_SIZE * 7)) {
				if (SURRENDER > 0 || STATE == ai) {
					handle_end_game(board,current);
					return;
				}



				flip_side(current);
				SURRENDER++;
				return;
			}
		}
		retflag = 1;
		SURRENDER = 0;
		if (STATE == ai && current == AI) {
			return;
		}

		int ix = event.mouse.x / CELL_SIZE;
		int iy = event.mouse.y / CELL_SIZE;
		if (board->GetAtLocation(iy, ix)->GetType() != 0) {
```

```
                retflag = 3;
                return;
            }
            make_move(iy, ix, current, board);
    }

}

void handle_end_game(Board* board, int& current) {

    int button;
    if (board->get_black_score() > board->get_white_score()) {

            button = al_show_native_message_box(display, "GAME ENDED", " ", "Black Win", NULL,
ALLEGRO_MESSAGEBOX_OK_CANCEL);
    }
    else if (board->get_black_score() < board->get_white_score()) {

            button = al_show_native_message_box(display, "GAME ENDED", " ", "White Win", NULL,
ALLEGRO_MESSAGEBOX_OK_CANCEL);
    }
    else {
            button = al_show_native_message_box(display, "GAME ENDED", " ", "Tie", NULL,
ALLEGRO_MESSAGEBOX_OK_CANCEL);
    }

    STATE = menu;
    board->board_reset();
    SURRENDER = 0;
    current = BLACK;
}

void handle_menu(int ix, int iy, int& current, Board* board) {
    if (ix >= (CELL_SIZE * 8) && ix <= (CELL_SIZE * 15)) {
            if (iy >= (CELL_SIZE * 4) && iy <= (CELL_SIZE * 6)) {
                    STATE = multi;
            }

            if (iy >= CELL_SIZE * 8 && iy <= CELL_SIZE * 10) {
                    STATE = ai;
            }

            if (iy >= CELL_SIZE * 12 && iy <= CELL_SIZE * 14) {

                    load_game(current, board);
            }
    }
}

void draw_menu(ALLEGRO_FONT* font) {

    al_flip_display();
    al_clear_to_color(al_map_rgb(255, 222, 0));

    if (font == NULL) {
            return;
    }
    handle_menu(font);
```

```
}

void handle_menu(ALLEGRO_FONT* font)
{
   //multiplayer mode
   al_draw_filled_rectangle(CELL_SIZE * 8, CELL_SIZE * 4, CELL_SIZE * 15, CELL_SIZE * 6,
al_map_rgb(137, 135, 0));
   al_draw_text(font, al_map_rgb(0, 0, 0), CELL_SIZE * 11.5, CELL_SIZE * 4.68,
ALLEGRO_ALIGN_CENTRE, "Multiplayer mode.");

   //player vs AI mode
   al_draw_filled_rectangle(CELL_SIZE * 8, CELL_SIZE * 8, CELL_SIZE * 15, CELL_SIZE * 10,
al_map_rgb(137, 135, 0));
   al_draw_text(font, al_map_rgb(0, 0, 0), CELL_SIZE * 11.5, CELL_SIZE * 8.68,
ALLEGRO_ALIGN_CENTRE, "Player vs AI.");

   //load the game
   al_draw_filled_rectangle(CELL_SIZE * 8, CELL_SIZE * 12, CELL_SIZE * 15, CELL_SIZE * 14,
al_map_rgb(137, 135, 0));
   al_draw_text(font, al_map_rgb(0, 0, 0), CELL_SIZE * 11.5, CELL_SIZE * 12.68,
ALLEGRO_ALIGN_CENTRE, "Load the game.");
}

void draw_board(Board* board, int& current, ALLEGRO_FONT* font)
{
   al_flip_display();
   al_clear_to_color(al_map_rgb(255, 222, 0));


   float half_cell = 1.0 * CELL_SIZE / 2;
   bool start = true;

   horizontal_lines(half_cell);

   vertical_lines(half_cell);

   //current color moves area
   draw_current_player(current,font);


   draw_point(board, font);



   //surrender arrea
   draw_surrender(font);



   for (int i = 0;i < 3;i++) {
            for (int j = 0;j < 3;j++) {
                     al_draw_circle(half_cell + (3 + 6 * j) * CELL_SIZE, half_cell + (3 + 6 * i) * CELL_SIZE, 1,
al_map_rgb(0, 0, 0), 5);
            }

   }
   draw_stones(half_cell, board);
}
```

```
void draw_surrender(ALLEGRO_FONT* font)
{
   al_draw_filled_rectangle(CELL_SIZE * 18.9, CELL_SIZE * 5, CELL_SIZE * 22.7, CELL_SIZE * 7,
al_map_rgb(137, 135, 0));
   if (SURRENDER == 1) {
            al_draw_text(font, al_map_rgb(0, 0, 0), CELL_SIZE * 20.8, CELL_SIZE * 5.7,
ALLEGRO_ALIGN_CENTRE, "End Game.");
   }
   else {
            al_draw_text(font, al_map_rgb(0, 0, 0), CELL_SIZE * 20.8, CELL_SIZE * 5.7,
ALLEGRO_ALIGN_CENTRE, "Pass.");
   }
}

void draw_point(Board* board, ALLEGRO_FONT* font)
{
   string white = (to_string(board->get_white_score()));
   string black = (to_string(board->get_black_score()));
   const char* black_string = black.c_str();
   const char* white_string = white.c_str();


   //points of white player area
   al_draw_filled_rectangle(CELL_SIZE * 20, CELL_SIZE * 14, CELL_SIZE * 22, CELL_SIZE * 16,
al_map_rgb(137, 135, 0));
   al_draw_text(font, al_map_rgb(0, 0, 0), CELL_SIZE * 21, CELL_SIZE * 13, ALLEGRO_ALIGN_CENTRE,
"White points.");
   al_draw_text(font, al_map_rgb(0, 0, 0), CELL_SIZE * 21, CELL_SIZE * 14.8,
ALLEGRO_ALIGN_CENTRE, white_string);
   //points of black player area
   al_draw_filled_rectangle(CELL_SIZE * 20, CELL_SIZE * 9, CELL_SIZE * 22, CELL_SIZE * 11,
al_map_rgb(137, 135, 0));
   al_draw_text(font, al_map_rgb(0, 0, 0), CELL_SIZE * 21, CELL_SIZE * 8, ALLEGRO_ALIGN_CENTRE,
"Black points.");
   al_draw_text(font, al_map_rgb(0, 0, 0), CELL_SIZE * 21, CELL_SIZE * 9.8,
ALLEGRO_ALIGN_CENTRE, black_string);
}

void draw_current_player(int& current, ALLEGRO_FONT* font)
{
   al_draw_text(font, al_map_rgb(0, 0, 0), CELL_SIZE * 21, CELL_SIZE * 3, ALLEGRO_ALIGN_CENTRE,
"Current player.");

   al_draw_filled_rectangle(CELL_SIZE * 20, CELL_SIZE, CELL_SIZE * 22, CELL_SIZE * 3,
al_map_rgb(137, 135, 0));
   if (current == BLACK) {

            al_draw_filled_circle(CELL_SIZE * 21, CELL_SIZE * 2, 15, al_map_rgb(0, 0, 0));
   }
   if (current == WHITE) {

            al_draw_filled_circle(CELL_SIZE * 21, CELL_SIZE * 2, 15, al_map_rgb(255, 255, 255));
   }
}

void horizontal_lines(float half_cell)
{
   for (int i = 0;i < 19;i++) {
```

```
        al_draw_line(half_cell, half_cell + i * CELL_SIZE, CELL_SIZE * 19 - half_cell, half_cell + i *
CELL_SIZE, al_map_rgb(0, 0, 0), 2);

    }
}

void vertical_lines(float half_cell)
{
    for (int i = 0;i <19;i++) {
        al_draw_line(half_cell + i * CELL_SIZE, half_cell, half_cell + i * CELL_SIZE, CELL_SIZE * 19 -
half_cell, al_map_rgb(0, 0, 0), 2);

    }
}

void draw_stones(float half_cell, Board* board)
{
    for (int y = 0; y < 19; y++) {
        for (int x = 0; x < 19; x++) {

            if (board->GetAtLocation(y, x)->GetType() == BLACK) {

                al_draw_filled_circle(half_cell + CELL_SIZE * x, half_cell + y * CELL_SIZE, 15,
al_map_rgb(0, 0, 0));
            }
            if (board->GetAtLocation(y, x)->GetType() == WHITE) {

                al_draw_filled_circle(half_cell + CELL_SIZE * x, half_cell + y * CELL_SIZE, 15,
al_map_rgb(255, 255, 255));
            }
        }

    }
}
```

- Go Game.h

```
#pragma once
#include "Stone.h"
#include "Black.h"
#include "White.h"
#include "Board.h"

void draw_board(Board* board, int& current, ALLEGRO_FONT* font);

void draw_surrender(ALLEGRO_FONT* font);

void draw_point(Board* board, ALLEGRO_FONT* font);

void draw_current_player(int& current, ALLEGRO_FONT* font);

void horizontal_lines(float half_cell);

void vertical_lines(float half_cell);

void mouse_click(ALLEGRO_EVENT& event, int& current, int& retflag, Board* board);

void drawing_stone(float half_cell);
```

```
void drawing_board(float half_cell);

void draw_stones(float half_cell, Board* board);

void draw_menu(ALLEGRO_FONT* font);

void handle_menu(ALLEGRO_FONT* font);

void handle_menu(int ix, int iy, int& current, Board* board);

void save_game(int& current, Board* board);

void flip_side(int& current);

void handle_end_game(Board* board, int& current);
```

- White.h

```
#pragma once
#include "Stone.h"
class White : public Stone{

public:

        White() {

        }

        virtual int GetType() {
        return stone_type_white;
        }

};
```

- Black.h

```
#pragma once
#include "Stone.h"
class Black : public Stone{

public:

        Black() {

        }

        virtual int GetType() {
        return stone_type_black;
        }
```

```
};
```

- Stone.h

```
#pragma once

class Stone{

public:
   static int stone_type_white;
   static int stone_type_black;
   static int stone_type_empty;

   virtual int GetType() {
            return stone_type_empty;
   }


};
```

- Position.h

```
#pragma once
class Position
{
public:
   int x, y;
   Position(int a, int b) {
            y = a;
            x = b;
   }
};
```

- Board.h

```
#pragma once
#include "Stone.h"
#include <list>
#include <iterator>
#include "Position.h"
#include <exception>

using namespace std;

class IllegalMove_Exception : public exception {
public:
```

```
};

class Board
{
private:
   Stone* board[19][19];
   int black_score = 0;
   int white_score = 0;
public:

   Board() {
         init_board();
   }
   void init_board()
   {
         Stone* empty = new Stone;
         for (int i = 0;i < 19;i++) {
               for (int j = 0;j < 19;j++) {
                     board[i][j] = empty;
               }
         }
   }
   Stone* GetAtLocation(int y, int x) {
         return board[y][x];
   }

   void clear_at_location(int y, int x) {
         board[y][x] = new Stone;

   }

   void SetAtLocation(int y, int x,Stone* stone) {

         if (y < 0 || y>18 || x < 0 || x>18) {
               throw new IllegalMove_Exception;
         }
         //part of the "AI"
         if (GetAtLocation(y,x)->GetType() != 0) {
               throw new IllegalMove_Exception;
         }

         board[y][x] = stone;

         list < Position* > list_neighbours = legal_neighbours(new Position(y, x));
         for (Position* pos : list_neighbours) {
               handle_enemy(pos->y, pos->x, stone);

         }
         if(handle_suicide(y,x,stone)==true){
               clear_at_location(y, x);
               throw new IllegalMove_Exception;
         }
   }

   void handle_enemy(int y, int x, Stone* stone)
   {
         Stone* stn = GetAtLocation(y, x);
         if (stn->GetType() == Stone::stone_type_empty) {
               return;
```

```
	}
	if (stn->GetType() != stone->GetType()) {
		list<Position*>* enemy_group = group(new Position(y, x));
		if (have_breath(enemy_group) == false) {
			for (Position* pos : *enemy_group) {
				clear_at_location(pos->y, pos->x);
				if (stn->GetType() == Stone::stone_type_black) {
					white_score++;
				}else{
					black_score++;

				}
			}
		}
	}
}

bool handle_suicide(int y, int x, Stone* stone) {
	list<Position*>* suicide_group = group(new Position(y, x));
	if (have_breath(suicide_group) == false) {
		return true;
	}
	return false;

}

list <Position*> legal_neighbours(Position* pos) {
	// list of leegel neighbours around the position
	list <Position*> returned;
	if (pos->y != 0) {
		returned.push_back(new Position((pos->y - 1), pos->x));
	}

	if (pos->y != 18) {
		returned.push_back(new Position((pos->y + 1), pos->x));
	}

	if (pos->x != 0) {
		returned.push_back(new Position((pos->y), pos->x - 1));
	}

	if (pos->x != 18) {
		returned.push_back(new Position((pos->y), pos->x + 1));
	}
	return returned;
}

list <Position*>* group(Position* start) {

	list <Position*>* returned = new list <Position*>;
	reccur_group(start,returned);
	return returned;

}
void reccur_group(Position* start,list<Position*>* group) {

	bool is_in_group = false;

	for (Position* pos : *group) {
```

```
                if (start->x == pos->x && start->y == pos->y) {
                        is_in_group = true;
                        break;
                }
        }
        if (is_in_group == false) {
                Stone* s = GetAtLocation(start->y, start->x);
                group->push_back(start);
                list < Position* > list_neighbours = legal_neighbours(new Position(start->y, start->x));
                for (Position* pos : list_neighbours) {
                        Stone* s_at_pos = GetAtLocation(pos->y, pos->x);
                        if (s_at_pos->GetType() == s->GetType()) {
                                reccur_group(pos, group);
                        }

                }
        }

}

bool have_breath(list<Position*>* group) {
        for (Position* pos : *group) {
                list < Position* > list_neighbours = legal_neighbours(new Position(pos->y, pos->x));
                for (Position* neighbour : list_neighbours) {
                        // cheking is the neighbour is empty
                        Stone* stone_at_neighbour = GetAtLocation(neighbour->y, neighbour->x);
                        if (stone_at_neighbour->GetType() == Stone::stone_type_empty) {
                                return true;
                        }
                }
        }
        return false;
}

int get_black_score() {


        return black_score;
}

int get_white_score() {


        return white_score;
}

void board_reset() {
        black_score = 0;
        white_score = 0;

        init_board();

}
};
```

6. *Testing*

7. *Conclusions*

This part is optional. There may be written any comments about the program and other remarks about your experience during development or future todo's or practical usage of the program.

Remember that the final mark is influenced not only by the software that you submit, but also by hereby report and the way you prove your competence during laboratory.