Documentație

1. Titlu Proiect: Chess game

2. Descriere:

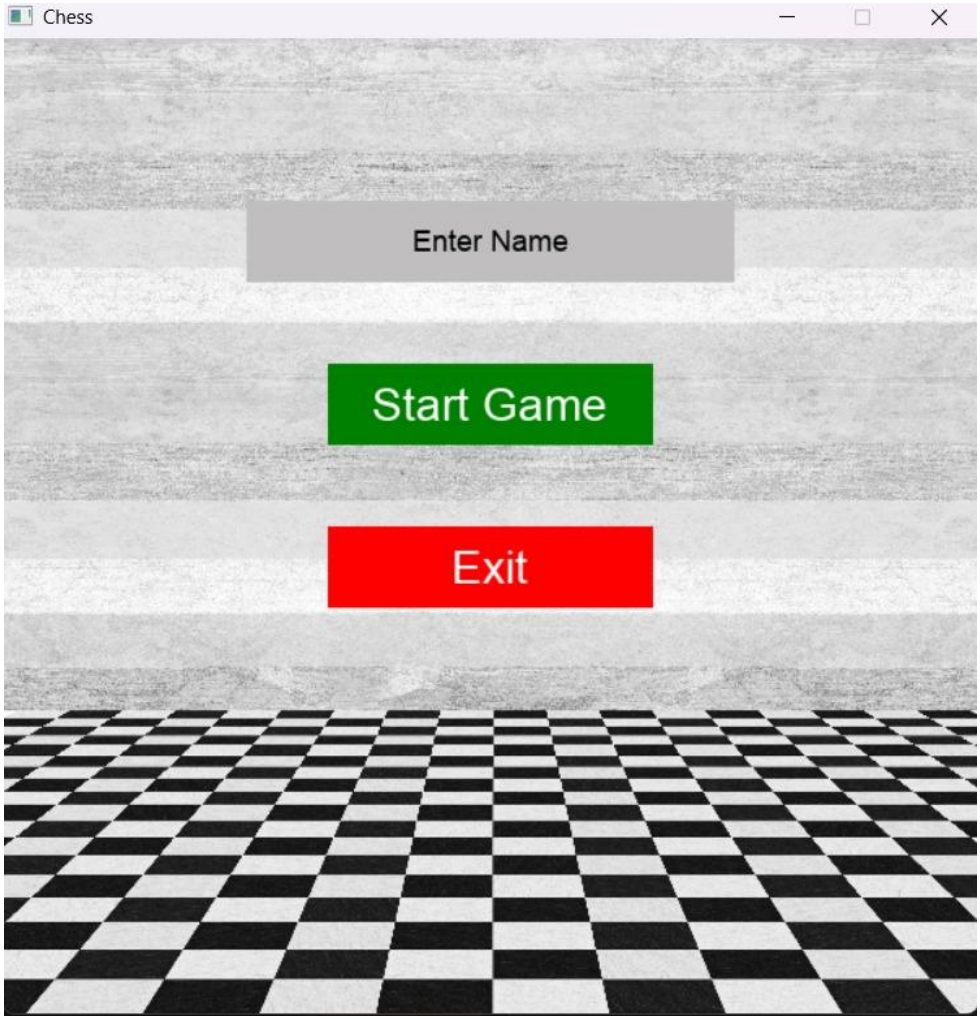
Acest proiect reprezintă o aplicație completă de șah cu interfață grafică, dezvoltată în limbajul C, ce permite jocul între doi jucători conectați în rețea (server și client). Interfața este realizată cu ajutorul bibliotecilor SDL2, SDL2_image și SDL2_ttf, iar comunicarea între jucători se face prin socket-uri TCP folosind Winsock2 (Windows).

Jocul oferă o experiență interactivă, cu meniu grafic, introducerea numelui jucătorului, validarea mutărilor conform regulilor oficiale de șah, afișarea grafică a tablei și pieselor, precum și salvarea rezultatelor fiecărei partide într-un fișier text.

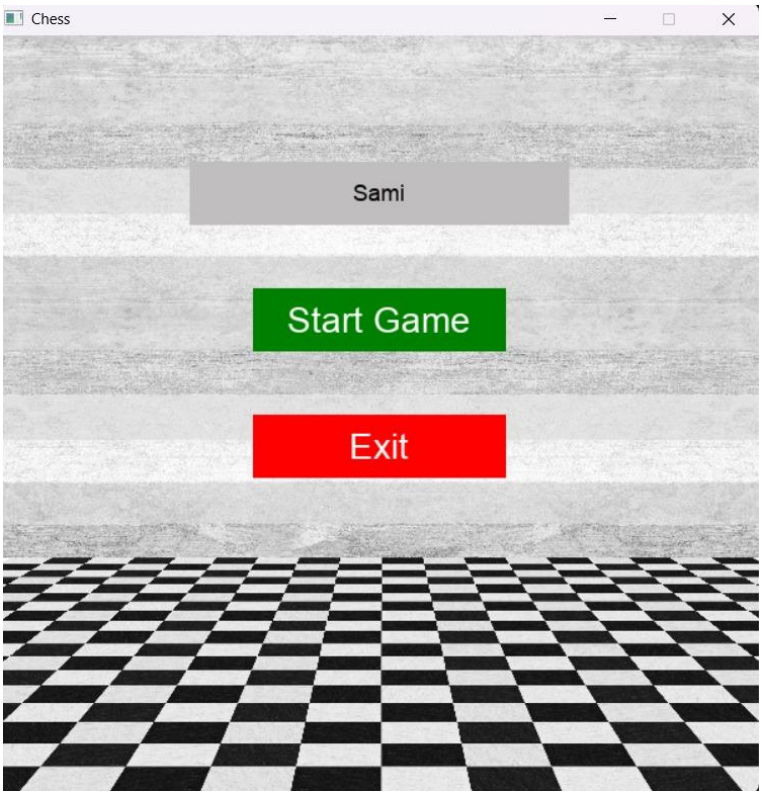
Proiectul este destinat atât exersării programării orientate pe evenimente și grafice, cât și înțelegerii principiilor de bază ale comunicației în rețea și sincronizării între procese. Este ideal pentru uz educațional, laborator sau proiecte personale.

3. Functionalitati:

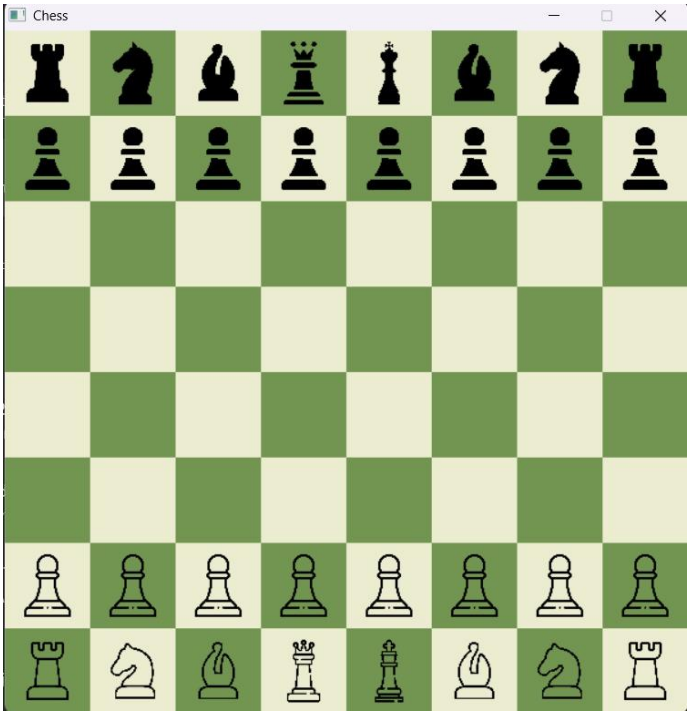
1. **Meniu grafic interactiv**: menu() – Afișează un ecran de start cu fundal, butoane "Play" și "Exit" și o casetă pentru introducerea numelui jucătorului.



2. Gestionează evenimentele de mouse și tastatură pentru selectarea butoanelor și introducerea textului.



3. La apăsarea "Play", numele introdus este trimis către server (dacă ești client) sau salvat local (dacă ești server).



4. Inițializarea jocului:

init_game(GameState* state, Button* startBtn, Button* exitBtn, Button* nameBtn)

Creează fereastra și renderer-ul SDL, inițializează tabla de șah cu piesele în poziția de start, resetează variabilele de stare și configurează butoanele pentru meniu.

```
167
168  // Inițializare joc
169  void init_game(GameState* state, Button* startBtn, Button* exitBtn, Button* nameBtn) {
170      state->window = SDL_CreateWindow("Chess", SDL_WINDOWPOS_CENTERED,
171                                       SDL_WINDOWPOS_CENTERED, WINDOW_WIDTH,
172                                       WINDOW_HEIGHT, SDL_WINDOW_SHOWN);
173      state->renderer = SDL_CreateRenderer(state->window, -1, SDL_RENDERER_ACCELERATED);
174
175      // Inițializare tablă
176      for (int i = 0; i < BOARD_SIZE; i++) {
177          for (int j = 0; j < BOARD_SIZE; j++) {
178              state->board[i][j] = (Piece){EMPTY, WHITE};
179          }
180      }
181
182      // Configurare piese inițiale
183      state->board[0][0] = (Piece){ROOK, BLACK};
184      state->board[0][1] = (Piece){KNIGHT, BLACK};
185      state->board[0][2] = (Piece){BISHOP, BLACK};
186      state->board[0][3] = (Piece){QUEEN, BLACK};
187      state->board[0][4] = (Piece){KING, BLACK};
188      state->board[0][5] = (Piece){BISHOP, BLACK};
189      state->board[0][6] = (Piece){KNIGHT, BLACK};
190      state->board[0][7] = (Piece){ROOK, BLACK};
191
192      for (int i = 0; i < BOARD_SIZE; i++) {
193          state->board[1][i] = (Piece){PAWN, BLACK};
194          state->board[6][i] = (Piece){PAWN, WHITE};
195      }
```

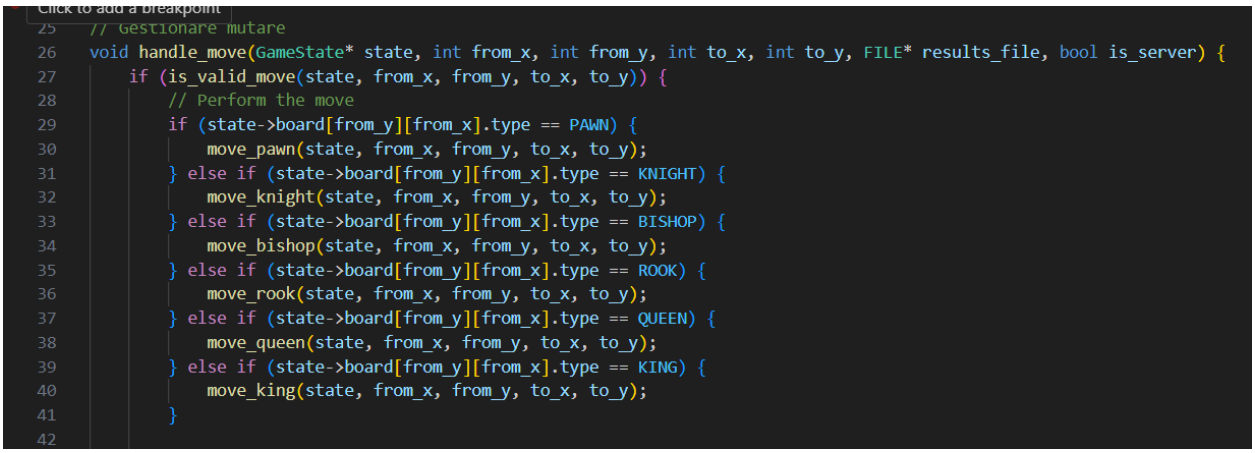
```
169  void init_game(GameState* state, Button* startBtn, Button* exitBtn, Button* nameBtn) {
208      state->selected_x = -1;
209      state->selected_y = -1;
210
211      // Inițializare jucători
212      state->player.is_ready_player_1 = false;
213      state->player.is_ready_player_2 = false;
214      strcpy(state->player.name_player_1, "Guest");
215      strcpy(state->player.name_player_2, "Guest");
216
217      // Inițializare butoane
218      startBtn->rect = (SDL_Rect){200, 200, 200, 50};
219      startBtn->color_normal = (SDL_Color){0, 128, 0, 255}; // verde
220      startBtn->color_hover = (SDL_Color){0, 180, 0, 255}; // verde deschis
221      startBtn->text = "Start Game";
222      startBtn->hovered = false;
223
224      exitBtn->rect = (SDL_Rect){200, 300, 200, 50};
225      exitBtn->color_normal = (SDL_Color){255, 0, 0, 255}; // roșu
226      exitBtn->color_hover = (SDL_Color){255, 100, 100, 255}; // roșu deschis
227      exitBtn->text = "Exit";
228      exitBtn->hovered = false;
229
230      nameBtn->rect = (SDL_Rect){150, 100, 300, 50};
231      nameBtn->color_normal = (SDL_Color){192, 190, 190, 255}; // gri
232      nameBtn->color_hover =  (SDL_Color){216, 214, 214, 255}; //
233      nameBtn->text = "Enter Name";
234      nameBtn->hovered = false;
  }
```

```
typedef enum {
    WHITE, BLACK
} PieceColor;

typedef struct{
    char name_player_1[MAX_NAME_LENGTH];
    char name_player_2[MAX_NAME_LENGTH];
    bool is_ready_player_1;
    bool is_ready_player_2;
}Player;

typedef struct {
    PieceType type;
    PieceColor color;
} Piece;
typedef struct {
    SDL_Window* window;
    SDL_Renderer* renderer;
    Piece board[BOARD_SIZE][BOARD_SIZE];
    Player player;
    int is_white_turn;
    int selected_x;
    int selected_y;
} GameState;
typedef enum{
```

5. Mutări și validare: User-ul selectează piesa pe care dorește să o mute, urmând ca selecția facută să fie interpretată.



```
Click to add a breakpoint
// Gestionare mutare
void handle_move(GameState* state, int from_x, int from_y, int to_x, int to_y, FILE* results_file, bool is_server) {
    if (is_valid_move(state, from_x, from_y, to_x, to_y)) {
        // Perform the move
        if (state->board[from_y][from_x].type == PAWN) {
            move_pawn(state, from_x, from_y, to_x, to_y);
        } else if (state->board[from_y][from_x].type == KNIGHT) {
            move_knight(state, from_x, from_y, to_x, to_y);
        } else if (state->board[from_y][from_x].type == BISHOP) {
            move_bishop(state, from_x, from_y, to_x, to_y);
        } else if (state->board[from_y][from_x].type == ROOK) {
            move_rook(state, from_x, from_y, to_x, to_y);
        } else if (state->board[from_y][from_x].type == QUEEN) {
            move_queen(state, from_x, from_y, to_x, to_y);
        } else if (state->board[from_y][from_x].type == KING) {
            move_king(state, from_x, from_y, to_x, to_y);
        }

```

Fiecare tip de piesă de pe tabla de șah este tratată în particular, fiind implementate biblioteci. Structura bibliotecilor(corespunzatoare tipului de piesă) este asemanatoare: o funcție principală void move_(tip_piesa)() unde are loc mutarea piesei și o funcție care verifică dacă aceasta este validă, int is_valid_(tip_piesa)_move(), care returnează -1 dacă mutarea intenționată nu este posibilă.

a. Pion

```
void move_pawn(GameState* state, int from_x, int from_y, int to_x, int to_y){
    if(is_valid_pawn(state, from_x, from_y, to_x, to_y) == -1){
        return;
    }
    if(is_valid_pawn(state, from_x, from_y, to_x, to_y) > 0){
        Piece temp_piece = state->board[to_y][to_x];
        state->board[to_y][to_x] = state->board[from_y][from_x];
        state->board[from_y][from_x] = (Piece){EMPTY, WHITE};
        if(is_check(state)){
            // Mutare invalidă, revenire la starea
            state->board[from_y][from_x] = state->b
            state->board[to_y][to_x] = temp_piece;
            printf("Invalid move! King is in check!\n");
            return;
        }
        // Verificare promovare
        int promotion_row = state->board[to_y][to_x].color == WHITE ? 0 : 7;
        if (to_y == promotion_row) {
            promotion(state, to_x, to_y);
        }


        state->is_white_turn = !state->is_white_turn;
        return;
    }
}
```

(char [33])"Invalid move! King is in check!\n"

Generate Copilot summary

```
int is_valid_pawn(GameState* state,int from_x, int from_y, int to_x, int to_y){
    int direction = state->board[from_y][from_x].color == WHITE ? -1 : 1;
    // Verificare mutare verticală (în față)
    if (to_x == from_x && to_y == from_y + direction && state->board[to_y][to_x].type == EMPTY) {
        return 1; // Mutare normală
    }
    // Verificare mutare diagonală (capturare)
    if (to_y == from_y + direction) {
        if(to_x == from_x - 1){ // merg in stanga
            if(state->board[to_y][to_x].type != EMPTY && state->board[to_y][to_x].color != state->board[from_y][from_x].color){
                return 2; // Capturare validă
            }
        }
        else if(to_x == from_x + 1){ // merg in dreapta
            if(state->board[to_y][to_x].type != EMPTY && state->board[to_y][to_x].color != state->board[from_y][from_x].color){
                return 3; // Capturare validă
            }
        }

    }

    if(from_y == 6 && state->board[from_y][from_x].color == WHITE){ // alb
        if(to_y == from_y - 2 && to_x == from_x && is_empty(state, to_x, to_y) && is_empty(state, to_x, to_y + 1)){
            return 4; // mutare dubla
        }
    }
    else if(from_y == 1 && state->board[from_y][from_x].color == BLACK){ // negru
```

În funcție de culoarea piesei, direcția pionului poate fi y_current + direction = 1 – negru, -1 -alb. Această funcție verifică pozițiile pe care pionul selectat poate ajunge și, totodată, dacă aceasta este și cea selectată de jucător.

b. Tură

```
int is_valid_rook_move(GameState* state, int from_x, int from_y, int to_x, int to_y) {
    if(state->board[from_y][from_x].type != ROOK && state->board[from_y][from_x].type != QUEEN) {
        return -1; // invalid piece type
    }

    //  verifying if the move is either horizontal or vertical
    if (from_x != to_x && from_y != to_y) {
        return -1; // Mutare invalidă
    }

    // path verification
    if (from_x == to_x) { // vertical move
        int step = (to_y > from_y) ? 1 : -1;
        for (int i = from_y + step; i != to_y; i += step) {
            if (state->board[i][from_x].type != EMPTY) {
                return -1; // path is not clear
            }
        }
    } else if (from_y == to_y) { // horizontal move
        int step = (to_x > from_x) ? 1 : -1;
        for (int i = from_x + step; i != to_x; i += step) {
            if (state->board[from_y][i].type != EMPTY) {
                return -1; // path is not clear
            }
        }
    }

    // verifying if the destination square is occupied by a piece of the same color
    if (state->board[to_y][to_x].type != EMPTY &&
```

```
    }

    // verifying if the destination square is occupied by a piece of the same color
    if (state->board[to_y][to_x].type != EMPTY &&
        state->board[to_y][to_x].color == state->board[from_y][from_x].color) {
        return -1; // invalid capture
    }

    return 1; // valid move
}
```

- int is_valid_rook_move(...):verifică ca “drumul” parcurs să fie liber și se asigură că în celula finală nu se afla o piesa de aceeasi culoare.

c. Cal

```c
int is_valid_knight_move(GameState* state,int from_x, int from_y, int to_x, int to_y) {
    int dx = abs(to_x - from_x);
    int dy = abs(to_y - from_y);
    return ( ( (dx == 2 && dy == 1) || (dx == 1 && dy == 2) ) && (state->board[to_y][to_x].type == EMPTY ||
            state->board[to_y][to_x].color != state->board[from_y][from_x].color)); // sa fie valida mutarea si sa nu fie ocupata de o piesa de acee
}

void move_knight(GameState* state, int from_x, int from_y, int to_x, int to_y){
    if(is_valid_knight_move(state,from_x, from_y, to_x, to_y) == 1){
        Piece temp_piece = state->board[to_y][to_x];
        // Mutare piesă
        state->board[to_y][to_x] = state->board[from_y][from_x];
        state->board[from_y][from_x] = (Piece){EMPTY, WHITE};
        if(is_check(state)){
            // Mutare invalidă, revenire la starea anterioară
            state->board[from_y][from_x] = state->board[to_y][to_x];
            state->board[to_y][to_x] = temp_piece;
            printf("Invalid move! King is in check!\n");
            return;
        }
        state->is_white_turn = !state->is_white_turn;
    }
    else{
        printf("Invalid move for knight!\n");
    }

}
```

- int is_valid_knight_move(...): generează toate pozițiile unde calul poate ajunge și verifică dacă se găsește cea dorită

d. Nebun

```c
int is_valid_bishop_move(GameState* state, int from_x, int from_y, int to_x, int to_y) {
    int dx = abs(to_x - from_x);
    int dy = abs(to_y - from_y);
    if(dx != dy)
        return -1; // Not a diagonal move
    if(state->board[to_y][to_x].type != EMPTY && state->board[to_y][to_x].color == state->board[from_y][from_x].color) {
        return -1; // Cannot capture own piece
    }
    // Check for obstacles in the path
    int step_x = to_x > from_x ? 1 : -1;
    int step_y = to_y > from_y ? 1 : -1;
    for(int i = 1; i < dx; ++i){
        int x = from_x + i * step_x;
        int y = from_y + i * step_y;
        if(state->board[y][x].type != EMPTY) {
            return -1; // Obstacle in the way
        }
    }

    return 1; // Valid move
}
```

- int is_valid_bishop_move(...): verifică “drumul” – să fie liber.

e. Regina

```
int is_valid_queen_move(GameState* state, int from_x, int from_y, int to_x, int to_y){
    if(is_valid_rook_move(state, from_x, from_y, to_x, to_y) == -1 && is_valid_bishop_move(state, from_x, from_y, to_x, to_y) == -1){
        return -1; // Invalid move
    }
    return 1; // valid move
}
void move_queen(GameState* state, int from_x, int from_y, int to_x, int to_y){
    if(is_valid_queen_move(state, from_x, from_y, to_x, to_y) == 1){
        Piece temp_piece = state->board[to_y][to_x];
        state->board[to_y][to_x] = state->board[from_y][from_x];
        state->board[from_y][from_x] = (Piece){EMPTY, WHITE};
        if(is_check(state)){
            state->board[from_y][from_x] = state->board[to_y][to_x];
            state->board[to_y][to_x] = temp_piece;
            printf("Invalid move! King is in check!\n");
            return;
        }
        state->is_white_turn = !state->is_white_turn;
    }
    else{
        printf("Invalid move for queen!\n");
    }
}
```

- int is_valid_queen_move(...):folosește funcțiile implementate pentru tură și nebun.

f. Regele

```
int is_valid_king_move(GameState* state, int from_x, int from_y, int to_x, int to_y) {
    int dy[] = {-1, -1, -1, 0, 1, 1, 1, 0},
        dx[] = {-1, 0, 1, 1, 1, 0, -1, -1};
    for(int i = 0; i < 8; i++){
        if(to_x == from_x + dx[i] && to_y == from_y + dy[i] && (state->board[to_y][to_x].type == EMPTY  || state->board[to_y][to_x].color != state->
            return 1; // Valid move
        }
    }
    return -1; // Invalid move
}
```

- Se generează pozițiile unde acesta poate să fie mutat, iar în cazul în care se găsește poziția dorita, returnează 1 – mutare validă.

6. Promovarea pionului

```
        }
        // Verificare promovare
        int promotion_row = state->board[to_y][to_x].color == WHITE ? 0 : 7;
        if (to_y == promotion_row) {
            promotion(state, to_x, to_y);
        }
```

```
void promotion(GameState* state, int x, int y){
    // Promovare la regină
    Piece promoted_piece = {QUEEN, state->board[y][x].color};
    state->board[y][x] = promoted_piece;
    printf("Pawn promoted to Queen at (%d, %d)\n", x, y);
}
```

Promovează pionul ajuns pe ultima linie la regină (implicit).

Pentru o experiență completă, se poate implementa o funcție care permite user-ului să aleagă ce piesă să promoveze.

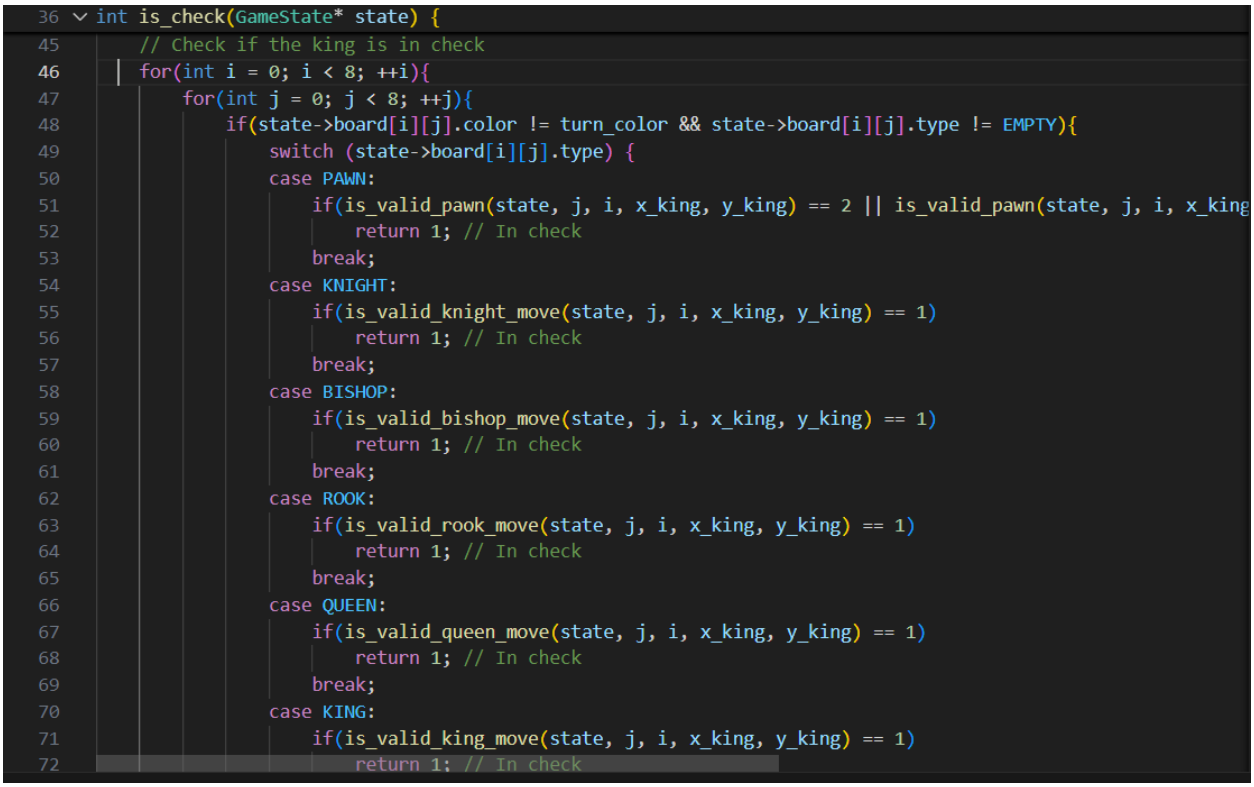
7. Detectare șah/șah mat

```
int get_coordinates_king(GameState* state, int* x_king, int* y_king){
    PieceColor turn_color = BLACK;
    if(state->is_white_turn)
        turn_color = WHITE;
    for(int i = 0; i < 8; ++i){
        for(int j = 0; j < 8; ++j){
            if(state->board[i][j].type == KING && state->board[i][j].color == turn_color){
                (*x_king) = j;
                (*y_king) = i;
                return 1; // Found the king
            }
        }
    }

    return -1; // King not found

}
```

- int get_coordinates_king(...): caută pe toată tabla de șah poziția regelui jucătorului care trebuie să mute, pentru a verifica ulterior dacă acesta se află în șah



```
36  int is_check(GameState* state) {
45      // Check if the king is in check
46      for(int i = 0; i < 8; ++i){
47          for(int j = 0; j < 8; ++j){
48              if(state->board[i][j].color != turn_color && state->board[i][j].type != EMPTY){
49                  switch (state->board[i][j].type) {
50                  case PAWN:
51                      if(is_valid_pawn(state, j, i, x_king, y_king) == 2 || is_valid_pawn(state, j, i, x_king
52                          return 1; // In check
53                      break;
54                  case KNIGHT:
55                      if(is_valid_knight_move(state, j, i, x_king, y_king) == 1)
56                          return 1; // In check
57                      break;
58                  case BISHOP:
59                      if(is_valid_bishop_move(state, j, i, x_king, y_king) == 1)
60                          return 1; // In check
61                      break;
62                  case ROOK:
63                      if(is_valid_rook_move(state, j, i, x_king, y_king) == 1)
64                          return 1; // In check
65                      break;
66                  case QUEEN:
67                      if(is_valid_queen_move(state, j, i, x_king, y_king) == 1)
68                          return 1; // In check
69                      break;
70                  case KING:
71                      if(is_valid_king_move(state, j, i, x_king, y_king) == 1)
72                          return 1; // In check
```

- parcurge toată tabla de șah și în funcție de piesa gasită(a adeversarului) verifică dacă poate ataca regele. În caz afirmativ se returnează 1 => regele este în șah. Această funcție este folosită și pentru a verifica dacă o piesă este blocată(pinned).

```
if(is_check(state)){
    // Mutare invalidă, revenire la starea anterioară
    state->board[from_y][from_x] = state->board[to_y][to_x];
    state->board[to_y][to_x] = temp_piece;
    printf("Invalid move! King is in check!\n");
    return;
}
```

```
82  int is_checkmate(GameState* state) {
83      // Check if the king is in checkmate
84      PieceColor turn_color = BLACK;
85      if(state->is_white_turn)
86          turn_color = WHITE;
87
88      for(int i = 0; i < 8; ++i){
89          for(int j = 0; j < 8; ++j){
90              if(state->board[i][j].color == turn_color && state->board[i][j].type != EMPTY){
91                  for(int y = 0; y < 8; ++y){
92                      for(int x = 0; x < 8; ++x){
93                          if(is_valid_move(state, j, i, x, y)){
94                              Piece temp_piece = state->board[y][x];
95                              // Move the piece
96                              // state->board[y][x] = state->board[i][j];
97                              // state->board[i][j] = (Piece){EMPTY, WHITE};
98                              switch (state->board[i][j].type) {
99                                  case PAWN:
100                                     if(is_valid_pawn(state, j, i, x, y) > 0){
101                                         state->board[y][x] = state->board[i][j];
102                                         state->board[i][j] = (Piece){EMPTY, WHITE};
103                                         if(is_check(state) == 0){
104                                             // Undo the move if it doesn't put the king in check
105                                             state->board[i][j] = state->board[y][x];
```

- int is_checkmate(...): în această funcție se fac toate mutările posibile, iar dacă toate indică faptul că nu pot fi mutate => este șah mat, fiind câștigător adversarul. Spre exemplu dacă este albul la mutat, iar în urma generării tuturor mutărilor posibile nu poate găsi minim una validă, atunci înseamnă că negrul a câștigat.

8. Multiplayer(socket programming):

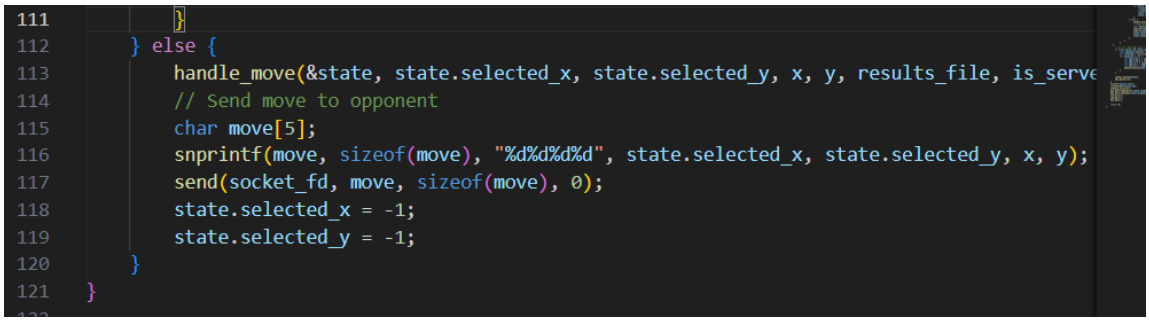
setup_server(...), setup_client(...):

Inițializează socket-ul pentru server/client și stabilește conexiunea TCP.

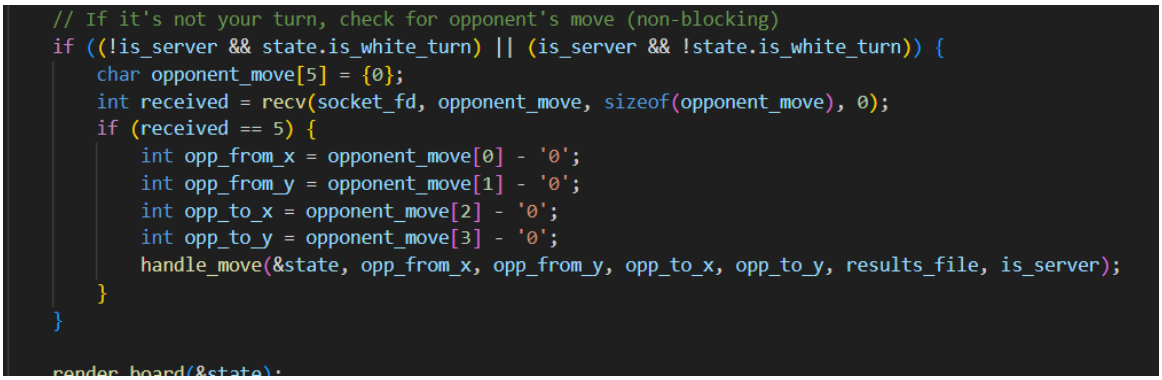
```
55  SOCKET setup_client(const char* ip, int port) {
56      init_winsock();
57      typedef UINT_PTR SOCKET
58      SOCKET client_fd = socket(AF_INET, SOCK_STREAM, 0);
59      if (client_fd == INVALID_SOCKET) {
60          printf("Socket creation failed. Error Code : %d\n", WSAGetLastError());
61          cleanup_winsock();
62          exit(EXIT_FAILURE);
63      }
64
65      struct sockaddr_in server_addr = {0};
66      server_addr.sin_family = AF_INET;
67      server_addr.sin_port = htons(port);
68      server_addr.sin_addr.s_addr = inet_addr(ip);
69      if (server_addr.sin_addr.s_addr == INADDR_NONE) {
70          printf("Invalid address/ Address not supported\n");
71          closesocket(client_fd);
72          cleanup_winsock();
73          exit(EXIT_FAILURE);
74      }
75      if (connect(client_fd, (struct sockaddr*)&server_addr, sizeof(server_addr)) == SOCKET_ERROR) {
76          printf("Connection failed. Error Code : %d\n", WSAGetLastError());
77          closesocket(client_fd);
78          cleanup_winsock();
79          exit(EXIT_FAILURE);
80      }
```

```
22  SOCKET setup_server(int port) {
23      init_winsock();
24
25      SOCKET server_fd = socket(AF_INET, SOCK_STREAM, 0);
26      if (server_fd == INVALID_SOCKET) {
27          printf("Socket creation failed. Error Code : %d\n", WSAGetLastError());
28          cleanup_winsock();
29          exit(EXIT_FAILURE);
30      }
31
32      struct sockaddr_in server_addr = {0};
33      server_addr.sin_family = AF_INET;
34      server_addr.sin_addr.s_addr = INADDR_ANY;
35      server_addr.sin_port = htons(port);
36
37      if (bind(server_fd, (struct sockaddr*)&server_addr, sizeof(server_addr)) == SOCKET_ERROR) {
38          printf("Bind failed. Error Code : %d\n", WSAGetLastError());
39          closesocket(server_fd);
40          cleanup_winsock();
41          exit(EXIT_FAILURE);
42      }
43
44      if (listen(server_fd, 1) == SOCKET_ERROR) {
45          printf("Listen failed. Error Code : %d\n", WSAGetLastError());
46          closesocket(server_fd);
47          cleanup_winsock();
48          exit(EXIT_FAILURE);
```

- send() și recv() trimit și primesc mutări și nume de jucători între client și server.



```
111             }
112         } else {
113             handle_move(&state, state.selected_x, state.selected_y, x, y, results_file, is_serve
114             // Send move to opponent
115             char move[5];
116             snprintf(move, sizeof(move), "%d%d%d%d", state.selected_x, state.selected_y, x, y);
117             send(socket_fd, move, sizeof(move), 0);
118             state.selected_x = -1;
119             state.selected_y = -1;
120         }
121     }
```



```
// If it's not your turn, check for opponent's move (non-blocking)
if ((!is_server && state.is_white_turn) || (is_server && !state.is_white_turn)) {
    char opponent_move[5] = {0};
    int received = recv(socket_fd, opponent_move, sizeof(opponent_move), 0);
    if (received == 5) {
        int opp_from_x = opponent_move[0] - '0';
        int opp_from_y = opponent_move[1] - '0';
        int opp_to_x = opponent_move[2] - '0';
        int opp_to_y = opponent_move[3] - '0';
        handle_move(&state, opp_from_x, opp_from_y, opp_to_x, opp_to_y, results_file, is_server);
    }
}

render_board(&state);
```

9. Salvarea rezultatelor
    - Clientul îi trimite serverului numele introdus(clientul este întotdeauna jucătorul cu piesele negre). Când este șah mat, rezultatul jocului este salvat în fișierul “results.txt”.

```c
// Check for check or checkmate
if (is_check(state)) {
    if (is_checkmate(state)) {
        printf("Checkmate!\n");
        printf(state->is_white_turn ? "Black wins!\n" : "White wins!\n");
        if(is_server){
            fprintf(results_file, "Player1: %s vs Player2: %s - Winner: %s\n",
            state->player.name_player_1,
            state->player.name_player_2,
            state->is_white_turn ? state->player.name_player_2 : state->player.name_player_1);
            fflush(results_file); // Ensure it's written immediately
        }
    } else {
        printf("Check!\n");
    }
```

results.txt

```
1   Player1: alb vs Player2: Sami - Winner: Sami
2
```

4. Instalare:

1. Instalează dependențele:

- SDL2, SDL2_image, SDL2_ttf (prin MSYS2: `pacman -S mingw-w64-i686-SDL2 mingw-w64-i686-SDL2_image mingw-w64-i686-SDL2_ttf`)

2. Compilează proiectul:

- Folosește Makefile-ul:

mingw32-make build

3. Pornește serverul:

-Deschide un terminal în folderul unde se află jocul:

./sdl_game.exe server 12345

4. Pornește clientul:

- În alt terminal sau pe alt calculator:

./sdl_game.exe client 127.0.0.1 12345

- Înlocuiește `127.0.0.1` cu IP-ul serverului dacă nu rulezi local.

5. Utilizare:

- Meniu:
  a. Introdu numele în caseta de text.
  b. Click pe "Play" pentru a începe jocul sau "Exit" pentru a ieși.
- Joc:
  a. Selectează piesa cu mouse-ul, apoi dă click pe pătratul destinație.
  b. Mutările sunt transmise automat adversarului.
- Rezultate:
  a. La finalul fiecărei partide, rezultatul este salvat în `results.txt`.

6. Structura Proiectului:

```
Proiect_chess/
│
├── assets/
│   ├── fonts/
│   │   └── arial.ttf
│   └── img/
│       ├── background/
│       │   └── menu_background.jpg
│       └── pieces/
│           ├── white/
│           │   └── pawn.png, knight.png, ...
│           └── black/
│               └── pawn.png, knight.png, ...
│
├── src/
│   ├── main.c
│   ├── board.c
│   ├── board.h
│   ├── moves.c
│   ├── moves.h
```

```
│   ├── socket_utils.c
│   ├── socket_utils.h
│   ├── pawn.c, knight.c, bishop.c, rook.c, queen.c, king.c
│
├── results.txt
├── Makefile
└── README.md
```

7. Extensii posibile:

a. Implementare AI pentru single-player.
b. Adăugare sunete și animații.
c. Salvare și încărcare automată a partidelor.
d. Suport pentru chat între jucători.
e. Diferite moduri de joc(blitz, rapid etc).
f. Implementarea jocului pe timp.
g. Posibilitatea ca user-ul să aleagă piesa dorită atunci când pionul ajunge în baza adversarului.

8. Dependencies

SDL2, SDL2_image, SDL2_ttf

Winsock2 (Windows, included with MinGW)

9. Link Github:

https://github.com/samyro14/chess_project

Succes la șah!