

CPS2008 Assignment

Super Battle Tetris

Tony Valentine

Contents

Project Overview	1
Server Backend	2
Server Architecture	2
Connection Handling	2
Message Handling	2
Chat Messages	3
Nickname	3
Leaderboard(s)	3
Players	4
Playerstats	4
Battle	5
Quickplay	6
Chill	7
Gamestats	7
Go	7
Ignore	7
Score Updates	8
Game End	8
Client Connectivity	9
Client-Server	9
Connecting to the server	9
Sending Messages to the Server	9
Receiving Messages from the Server	9
Chat/Command Encoding	9
Client-Client	10
Client Front-end	12
Handling Messages	12
Chat Messages	13
Game Messages	13
Games	14
Boomer	14
Rising Tide	14
Fast Track	14
Compiling and Running	16

Project Overview

Super battle tetris required the implementation of multiple distinct components, each of which is implemented in its own sub-directory.

- **src/client**: The client user interface implemented in **ncurses**
- **src/tetris**: Implementation of the Tetris games using **ncurses**
- **src/p2p**: Library used for peer-to-peer connectivity during tetris games
- **src/client-server**: Library used to connect to the server, aswell as send/recieve messages.
- **src/server**: The server

The video presentation can be found at: <https://drive.google.com/file/d/16sZE9PuygCVKbzsyVyfeevZNzGqS-cjw/view?usp=sharing>.

Server Backend

Super Battle Tetris required the implementation of an IRC style server, where users interact with the server through a text based interface. The main functionality required was the implementation of the chat relay, and for the server to execute custom commands denoted by the `!` symbol at the beginning of a line.

Server Architecture

To start the server the `run_server` command is used, where the only argument given is the port number to bind to.

`run_server` creates a new TCP socket before attempting to bind the socket to the requested port number. If this is successful then the server calls `listen` and will attempt to `accept` new connections until shutdown.

When the `accept` call succeeds the server logs the clients IP address, port number and socket file descriptor before creating a new thread to run the `handle_connection` function. An important note here is the use of `setsockopt` to apply timeouts when sending to the new socket, this prevents slow clients from disrupting the execution of the server as `send` is typically a blocking call.

New Client with IP: 127.0.0.1 On port: 61095 with socket number 4

Figure 1: Client Connection Message

Connection Handling

As mentioned previously connections are handled in a new thread running the `handle_connection` function.

Initially the function generates a new player name and creates a `player_data` class to store all the information needed for a client, this data is stored in a map (`player_list`) accessible to all threads. The thread will then enter an infinite loop trying to process data from the client. To reduce server overhead the socket file descriptor is monitored using the Linux `poll` system call, this allows the `recv` call to only be used when there is data to be received.

Upon successful return of the `poll` call, `recv` is used to get the data from the socket and the total number of bytes read are recorded. If zero bytes are read from the socket this is indicative of the client on the other end closing the socket, the thread will then enter a cleanup phase to remove all references to the client it's serving before terminating. Alternatively if the server reads the required amount of data then the message will be decoded from network byte ordering (`ntohl`) before calling `handle_message`.

Client with IP: 127.0.0.1:61095 connected to socket 4 disconnected

Figure 2: Client Disconnection Message

Message Handling

To reduce load on the server a strict message protocol was adopted, this protocol can be found in the `tprotocol.hh` header file.

```
struct tmessage {
    tmessage_t message_type; /**< Enumeration denoting the type of the message*/
    int32_t arg1, arg2, arg3, arg4, arg5,
        arg6; /**< Optional argument sfor each message type*/
    char buffer[MESSAGE_LENGTH]; /**< Char buffer for chat messages or information to decode */
} typedef tmessage;
```

`tmessage_t` is especially important to the implementation as it allows the server to switch over handlers for the different message types decreasing the overall processing time.

Chat Messages

Chat messages are handled by copying the contents of the message `buffer` and prepending the player name to the buffer. To send messages to all the clients a `relay` function was implemented. The function works by creating and encoding a valid `tmessage` with the requested string before sending the message to all the keys from `player_list` (map from sockets to player data).

```
1 case CHAT: {
2     string str;
3     auto iter = player_list.find(sock);
4     if (iter != player_list.end()) {
5         str = iter->second.name;
6     }
7     str.append("> ");
8     str.append(string(msg->buffer));
9     relay(str);
10    break;
11 }
```

Nickname

Nickname changing is implemented through changing the `name` field in the `player_data` class corresponding to the current user. If the requested player name is already being used by another player, or the nickname is too long then an error message is sent to the client as a chat message.

To solve the situation where two clients request the same nickname at the same time, the `player_list` has a corresponding mutex `player_list_mutex` which is used to lock the map during manipulation.

```
1 case NICKNAME: {
2
3     string str;
4     str = "Updated Nickname to: " + string(msg->buffer);
5     // If an error is encountered str will be updated to the error message
6
7     player_list_mutex.lock();
8     // Critical Section setting the player nickname
9     player_list_mutex.unlock();
10
11    send_chat(sock, str);
12    break;
13 }
```

Leaderboard(s)

The win, loss and scores for games are stored in the `player_data` entry for each client. To reduce the amount of time in the critical section the `player_list` is locked and a local copy is created. Since none of the entities in the map are dynamically allocated the default copy constructor in c++ will implement the behaviour required.

```
player_list_mutex.lock();
auto local_plist(player_list);
player_list_mutex.unlock();
```

Another added benefit is that the leader-board shown will be at the time of the message request, not including any new games ending during processing.

Generating the leaderboard required sorting the `player_data` in descending order according to the specified gamemode, the implementation of which used the `<algorithm>` library in combination with a custom comparison function. The top 3 entries for the gamemode are formatted into a string and placed in a `vector` which is returned at the end of the function. All of this functionality is wrapped inside the `get_leaderboard` function to increase re-usability.

```
1 auto sort_order = [game](player_data &p1, player_data &p2) -> bool {
2     switch (game) {
3         case RISING_TIDE:
4             return get<0>(p1.rising_games) > get<0>(p2.rising_games);
5         case BOOMER:
6             return get<0>(p1.boomer_games) > get<0>(p2.boomer_games);
7         case FAST_TRACK:
8             return get<0>(p1.fasttrack_games) > get<0>(p2.fasttrack_games);
9         case CHILLER:
10            return p1.chill > p2.chill;
11        default:
12            throw logic_error("Invalid Gamemode");
13    }
14 };
15
16 sort(plist.begin(), plist.end(), sort_order);
```

To ensure that the multiple messages which construct the leaderboard arrive uninterrupted the socket file descriptor for the client is locked using the `lockf` system call. Following this all messages can be sent to the client before unlocking the file descriptor. This locking and unlocking behaviour is implemented in all the sending commands from the server to ensure correctness.

```
1 if (lockf(sock_fd, F_ULOCK, 0) < 0) {
2     perror("lockf");
3 }
4 if (send(sock_fd, (char *)&msg, sizeof(tmessage), 0) < 0) {
5     perror("send");
6 };
7 if (lockf(sock_fd, F_ULOCK, 0) < 0) {
8     perror("lockf");
9 }
```

Leaderboards The implementation for the `!leaderboards` command involved calling the `get_leaderboard` function for each gamemode and sending the messages to the client.

Players

Listing the current players required traversing the `player_list` and extracting the `name` field for each active user's `player_data`. These names are then appended to a string before sending the message to the user.

Playerstats

The `playerstats` command is required to list the current win/loss statistics for all the active players. As with the `leaderboards` command a local copy of the map is created. The local copy is iterated over where the stats are obtained from the `player_data`, the information is formatted and placed into a vector before sending all the information to the client. A small delay is added in between the stats for each player to allow the client a chance to read the incoming stats.

```

1 for (auto &[k, v] : local_plist) { // Iterating over local player list
2     string str = "Player: " + string(v.name);
3     scores.push_back(str);
4     formatted_out("Mode", "Score", "Wins", "Losses"); // lambda for string formatting
↪ and pushing to scores
5     scores_out("Rising", v.rising, v.rising_games); // lambda for string formatting and
↪ pushing to scores
6     scores_out("Boomer", v.boomer, v.boomer_games);
7     scores_out("FastTrack", v.fasttrack, v.fasttrack_games);
8     scores.push_back(
9         formatted_out("Chiller", to_string(v.chill), "N/A", "N/A"));
10    send_multiple(sock, scores);
11    scores.clear();
12    this_thread::sleep_for(
13        chrono::milliseconds(250)); // As to allow the user to read a bit
14 }

```

Battle

Currently super battle tetris supports 4 gamemodes, of which 3 are supported in the battle command. Upon receiving a battle request the server must first decode the player names into the corresponding socket file descriptors.

If all or some of the players are found then a **game** object can be created, otherwise an error message is sent to the user. The **game** object is used to store all the information required to host a game, including the gamemode, gamemode arguments, and playerlist. The playerlist is a mapping from the socket file descriptor to a boolean, where the boolean flags if a player has accepted or declined a match.

The new game is then added to the **game_list** map where the key is generated from an **atomic_uint32_t**, this guarantees that each pending game has a unique number with an upper limit of no more than 4×10^9 games being pending at one time. Invited players are then messaged, and the game id is added to the **games** vector inside of their **player_data**. Game invitations and active games are stored in the player's data to allow the server to cleanup active and pending games upon client termination.

Finally a new thread is created running the **handle_game** function which used to handle game creation and initiation logic.

Game Launching The **handle_game** function first goes into a 30 second timeout to give players a chance to accept the game request. After 30 seconds the game is removed from the **game_list** to prevent late accept or decline messages. Following this the function collects the list of accepting players, if less than 2 players accept then an error message is sent to the client.

```

1 void handle_game(int game_id) {
2     // Start a 30 second timeout
3     this_thread::sleep_for(chrono::seconds(LOBBY_TIME));
4     // Timeout ended
5     game_list_mutex.lock(); // locking the game_list so no one can edit
6     auto match = game_list.at(game_id);
7     game_list.erase(game_id);
8     game_list_mutex.unlock();

```

Typically multiplayer games use a client-server architecture, this allows all clients to send and receive updates on a single port number. Unfortunately one of the requirements of Super Battle Tetris was for game updates to be implemented in a peer-to-peer fashion, as well as allowing multiple clients to exist on the same computer.

This means that the server needs to assign unique port numbers to each of the clients so that data can be transmitted and received properly.

Therefore the `handle_game` function needs to generate an unique port number for each client when sending the list of participating ip addresses. Apart from this quirk the rest of the function involves creating the message with the game information to send to the participating clients.

```

1 random_device rd;           // Used to generate random numbers
2 tmessage msg;               // message to send to the user
3 msg.message_type = INIT_GAME;
4 msg.arg2 = match.gamemode;   // Gamemode
5 msg.arg3 = game_id;
6 msg.arg4 = match.arg1;       // Gamemode Arguments [Time or Baselines]
7 msg.arg5 = match.arg2;       // Gamemode Arguments [Winlines]
8 msg.arg6 = rd();             // Game Seed

```

The participating IPs and port numbers are stored in the `buffer` member variable of `tmessage` where the final ip and port belong to the respective client.

Quickplay

The quickplay implementation is almost identical to that of Battle. The primary difference is that game arguments and players are generated by the server rather than being included in the command arguments.

```

1 std::random_device rd;
2 std::mt19937 gen(rd());
3 std::uniform_int_distribution<> game_distribution(0, 2); // Valid Gamemodes
4 game new_game;
5 int gamemode = game_distribution(gen);
6 new_game.gamemode = gamemode;
7 switch (gamemode) {
8     case BOOMER: {
9         std::uniform_int_distribution<> time_distribution(30,300); // 30 seconds to 5
10         ↪ minutes
11         new_game.arg1 = time_distribution(gen);
12     } break;
13     case FAST_TRACK: {
14         std::uniform_int_distribution<> line_distribution(1,10); // 1-10 lines
15         new_game.arg1 = line_distribution(gen); // Baselines
16         new_game.arg2 = line_distribution(gen); // Winlines
17     } break;

```

Parameter randomisation was implemented by switching over the current gamemode, and generating random numbers in a range deemed reasonable for the respective parameter. In the code snippet above the time duration for a boomer game is being set in the range of 30 seconds to 5 minutes.

Given an iterator number n the `advance` function can be used to get the $n + k$ iterator, and since c++ maps implement iterators this can be used to generate random players. The implementation locks the `player_list` and then generates a random offset from the start of the list, if the player is not the game creator or an already selected player then they are added to the invite list. After this point the `game` object is configured and the implementation is identical to that of Battle.

```

1 vector<int> offsets;
2 player_list_mutex.lock();
3 std::uniform_int_distribution<> invite_distribution(0, player_list.size() - 1); // Used
4 ↪ to generate offsets from the starting iterator

```



```

4 while (new_game.players.size() < msg->arg1) {
5     auto iter = player_list.begin();
6     advance(iter, invite_distribution(gen)); // Get a random player
7     // Preventing the game creator being added, and preventing a player from
8     // having multiple invites
9     if (iter->first != sock && new_game.players.count(iter->first) == 0) {
10         new_game.players.insert({iter->first, false});
11     }
12 }
13 player_list_mutex.unlock();

```

Chill

The `!chill` command is used by clients to initiate a single player game. The server creates a `game_data` struct to store the game information, and places it in the `chill_games` map. Players are sent an `INIT_GAME` message with the chill gamemode, game id, and random seed.

Gamestats

The `!gamestats` command is intended to output the individual player scores and lines cleared for all active games. Since the list of active games is stored in the `ongoing_games` map, then games can be iterated over where the scores will be formatted and sent to the user.

Go

The `!go` command is intended to allow users to accept a specific game invitation. To implement this functionality the game number is parsed from the command, where the server can then check the `game_list` for this game id. If the game is found and the client was invited to the game, then the `players` map inside the game is updated to indicate the clients choice. Otherwise the client is sent an error message informing them that either the game does not exist or they were not invited.

```

1 string str;
2 game_list_mutex.lock();
3 auto g = game_list.find(msg->arg1);
4 if (g != game_list.end()) {
5     auto entry =
6         g->second.players.find(sock); // Checking if we are part of the game
7     if (entry != g->second.players.end()) {
8         entry->second = true; // Setting acceptance boolean to true
9         str = "You sucessfully accepted to join the game";
10    } else {
11        str = "You were not invited to the game in question";
12    }
13 } else {
14     str = "The game you requested to join does not exist!";
15 }
16 game_list_mutex.unlock();
17 send_chat(sock, str); // Sending outside of critical section

```

Ignore

The `!ignore` command allows users to decline invitations to games. The processing of the command is identical to that of the `!go` command, with the boolean invitation acceptance being set to false rather than true.

Score Updates

Score updates are commands issued to the server by clients during battle games. The messages consists of the game id, current score and current lines cleared. The server can then update the game in the `ongoing_games` map using the game id.

```
1 case SCORE_UPDATE: {
2     int game_number = msg->arg1;
3     int score = msg->arg2;
4     int lines = msg->arg3;
5     if (ongoing_games.find(game_number) != ongoing_games.end()) {
6         auto game = ongoing_games.find(game_number);
7         game->second.update_player(sock, score, lines);
8     } else if (chill_games.find(game_number) != chill_games.end()) {
9         auto game = chill_games.find(game_number);
10        game->second.score = score;
11        game->second.lines = lines;
12    }
13    break;
14 }
```

Game End

The game end command is issued to the server by clients to update the server with the final scores of a game. The message consists of the game id, final score and lines. The server then updates the game with the players information and checks if all players have ended the match. When updating the player information the time is recorded for later use.

Once all players have ended the match a winner needs to be decided.

- Boomer: Select the player with the highest score
- Rising Tide: Select the player with the longest game duration
- Fast Track: If both players completed the required number of lines then select the player with the shortest game duration. Otherwise select the player with the larger number of lines with duration as a fallback in the case of a tie.

This functionality is wrapped inside a lambda function so that the player can be selected using `std::max_element` from the `<algorithm>` library. All players will then have their individual stats updated for win/loss numbers as well as high-scores.

Client Connectivity

Client-Server

Client server connectivity is implemented in the `src/client-server` directory. The library exposes functions for connecting to the server as well as sending or receiving messages.

Connecting to the server

Connecting to the server is handled by the `establish_connection` function. The function attempts to create a TCP socket with the server, whose ip and port number are passed as arguments to the function.

By using the `getaddrinfo` command the client can even use DNS addresses for the server ip address or even “localhost” when on the same machine.

Sending Messages to the Server

The `send_message` function is used to send `tmessage` struct to the server. Before sending the structs must first be passed through `encode_message` which is used to convert the structure into network byte ordering.

Receiving Messages from the Server

The `receive_message` function is used to receive `tmessage` structs from the server. The structs will be automatically decoded from network byte ordering into host ordering by calling the `decode_message` function.

Chat/Command Encoding

The `parse_message` function is used to convert strings (entered by the user) into `tmessage` structs. Message parsing was implemented client side to reduce the load in the server. If an unknown or malformed command string is passed to the function an exception will be thrown, this exception can then be caught and the message can be displayed to the user.

Client-Client

Peer to peer connectivity is implemented in the `src/p2p` directory.

The main component of the library is the `communicate_state` function. The function is designed to take a `gamestate` struct which contains the information used during a game. This struct is passed by reference to the function by the tetris games to ensure that the information can be read properly. The function is designed to establish the necessary sockets to send or receive state from peers during the game. Once the sockets have been established then `receive_state` and `broadcast_state` are called.

`receive_state` is used to retrieve game updates from peers and update the `gamestate` with the new information. Peer information is only updated by the `receive_state` function with local stats only being updated by the tetris games as such mutual exclusion was not required in this case.

```
1 update_msg *msg = (update_msg *)buffer;
2 decode_state(msg); // Network to host
3 if (state.players.find(msg->player_no) != state.players.end()) {
4     auto player = state.players.find(msg->player_no);
5     player->second.score = msg->score;
6     player->second.lines = msg->lines;
7 } else {
8     // First update, add player to list
9     state.players.insert({msg->player_no, (playstate){.score = 0, .lines = 0}});
10 }
11 }
```

`broadcast_state` is used to send the current `gamestate` information to all peers every 25 milliseconds, and to the server every one second. The information is sent in the form of an `update_msg` struct to peers and a `tmessage` to the server. To increase the frequency of updates a `TICKRATE` macro is defined in the `p2p.hh` header file which alters the number of updates to peers per second.

```
1 while (1) {
2     if (*termination_flag) {
3         break;
4     }
5     counter = (counter + 1) % TICKRATE;
6     struct update_msg msg;
7     msg.score = state.local.score;
8     msg.lines = state.local.lines;
9     msg.player_no = state.player_no;
10    encode_state(msg);
11    for (auto &[sock, other] : peers) {
12        // send data to peer
13    }
14    if (counter == 0) {
15        tmessage server_msg;
16        server_msg.message_type = (tmessage_t)htonl((int32_t)SCORE_UPDATE);
17        server_msg.arg1 = htonl(state.game_no);
18        server_msg.arg2 = htonl(state.local.score);
19        server_msg.arg3 = htonl(state.local.lines);
20        send(sock_fd, (char *)&server_msg, sizeof(server_msg), 0);
21    }
22
23    this_thread::sleep_for(TICKDURATION); // Tick duration = 1s / tickspeed
24 }
```

Terminating threads safely in c++ is slightly more cumbersome than with native POSIX threads. To work around this problem a `termination_flag` is used to inform threads when they need to terminate, the flag is set by tetris games when the player either wins or loses the match. This ensures that all threads are cleaned up properly and the cleanup code is self contained within the function.

An alternative implementation involved using `native_handle`, however this proved to be rather messier and inelegant.

Client Front-end

The client front-end consists of two distinct components;

- The chat client used to send and display messages
- The tetris games

On startup the client uses the `establish_connection` command to open a TCP socket with the server. The arguments to the command are the servers hostname, as well as the port, both of which should be passed by the command line.

The client will then enter a loop where the socket file descriptor and standard input are polled using the `poll` system call. If standard input is ready then the `send_message` function is used to take the input, wrap it in a `tmessage` struct and send it to the server. If data is ready to be received from the server then `recieve_message` is used to get the message from the server and handle it. The possible messages from the server are either chat messages, or start game messages.

```
1  struct pollfd pfds[2];
2  pfds[0].fd = 0; // STDIN
3  pfds[0].events = POLLIN;
4
5  pfds[1].fd = sockfd;
6  pfds[1].events = POLLIN;
7
8  int xMax, yMax;
9  getmaxyx(stdscr, yMax, xMax);
10
11 for (;;) {
12     move(yMax - 4, 1);
13     int poll_count = poll(pfds, 2, -1); // Wait till we have input
14
15     if (poll_count == -1) {
16         exit(1);
17     } else {
18         for (int i = 0; i < 2; i++) {
19             if (pfds[i].revents & POLLIN) {
20                 if (pfds[i].fd == 0) {
21                     // Standard Input is ready
22                     send_message(sockfd);
23                 } else if (pfds[i].fd == sockfd) {
24                     // Data to receive from the client
25                     recieve_message(sockfd);
26                 }
27             }
28         }
29     }
30 }
```

Handling Messages

As mentioned previously `recieve_message` is used to handle incoming messages from the client. The received messages are handled according to their `message_type` parameter.

Chat Messages

Chat messages are handled by extracting the `buffer` field from the received message, and appending it to a limited buffer. The limited buffer is used to store the n most recent messages to emulate text scrolling on the display.

```
1 case CHAT: {
2     chat_messages.push_back(string(msg->buffer));
3
4     wclear(chat_window);
5     box(chat_window, 0, 0);
6
7     if (chat_messages.size() > (yMax - 12)) { // yMax is the maximum buffer size based on
        ↳ the screen
8         chat_messages.erase(chat_messages.begin(), chat_messages.begin() +
        ↳ (chat_messages.size() - (yMax - 12))); // Remove the old messages
9     }
10    // Display the messages
11    for (int i = 0; i < chat_messages.size(); i++) {
12        mvwprintw(chat_window, i + 1, 1, chat_messages[i].c_str());
13        wrefresh(chat_window);
14    }
15    break;
16 }
```

Game Messages

Tetris games are triggered by receiving `INIT_GAME` messages from the server, this applies to both single and multiplayer games. The message structure contains all the information to start the game;

1. Player number
2. Gamemode
3. Game id
4. Gamemode Argument 1
5. Gamemode Argument 2
6. Game seed

The arguments are then passed to the constructors for the tetris games, before allowing the game to run and getting the final score.

```
1 switch (msg->arg2) {
2     case BOOMER: {
3         BoomerGame game(msg->arg6, ips, msg->arg1, msg->arg3, sockfd, msg->arg4);
4         game.run();
5         score = game.get_final_score();
6     } break;
```

The score is needed to send a final `GAME_END` message to the server before returning to the chat client.

Games

The tetris game implementation code is located in the `src/tetris` directory. The base `TetrisGame` class is used to implement a basic tetris game with the code for piece generation, rotation, collision, scoring, line clearing etc.

This gamemode also acts as the implementation for the singleplayer CHILL gamemode.

Boomer

The Boomer gamemode was intended to end the game after a specified duration. To implement this functionality the current time elapsed is checked after each gametick, meaning that the game will terminate with a maximum delay of 25 milliseconds.

```
1 auto start = chrono::high_resolution_clock::now();
2 auto current = chrono::high_resolution_clock::now();
3 auto time_elapsed =
4     chrono::duration_cast<chrono::milliseconds>(current - start);
5 while (do_gametick(new_piece, piece_flag, counter) == 0 &&
6     time_elapsed < game_duration) {
7     current = chrono::high_resolution_clock::now();
8     time_elapsed = chrono::duration_cast<chrono::milliseconds>(current - start);
9 }
```

Rising Tide

Rising tide requires that all lines completed by other players are transferred to your own board.

This functionality was implemented by keeping track of the total number of lines completed at each game tick. If on the new gametick more lines have been completed then the lines are added to the players board. The `insert_lines` function was also implemented to handle the logic of inserting lines and managing the pieces.

```
1 while (do_gametick(new_piece, piece_flag, counter) == 0) {
2     new_lines = 0;
3     if (state.players.size()) { // Only checking lines if there are lines to check
4         old_lines = total_lines; // Noting the previous number of lines completed
5         total_lines = 0;
6         for (auto &[k, v] : state.players) {
7             total_lines += v.lines; // Calculating the total number of lines completed
8         }
9         new_lines = total_lines - old_lines; // Calculating the new lines completed
10        if (new_lines > 0) {
11            if (insert_lines() != 0) { // Inserting the lines
12                /*
13                 * Cleanup current piece
14                 */
15            }
16        }
17    }
18 }
```

Fast Track

In the fast track gamemode the playing field is initially populated with a number of rows and the game ends when the player either completes the stipulated number of lines or loses.

Filling the board was a trivial implementation and involved setting all the columns in the specified rows to an unclearable number.


```
1 for (int i = playing_field.size() - 1; i >= playing_field.size() - init_lines; i--) {  
2     playing_field[i] = vector<char>(playing_field[i].size() + 1, STATIC_ROW);  
3 }
```

Ensuring that the game ends after the stipulated number of lines involved checking the current number of completed lines against the needed amount.

```
1 while (do_gametick(new_piece, piece_flag, counter) == 0 &&  
2     lines_cleared < win_lines) {  
3 }
```

Compiling and Running

The source code for the implementation can be found in the `src` directory of the folder.

- `src/client`: The client user interface
- `src/tetris`: Implementation of the Tetris games
- `src/p2p`: Library used for peer-to-peer connectivity during tetris games
- `src/client-server`: Library used to connect to the server, aswell as send/recieve messages.
- `src/server`: The server

To compile the implementations a `compile.sh` script is included in the root directory. The script will compile the server into `src/server/server`, and the client into `src/client/client`. Before compiling the `client` program users are given the choice between compiling with `client-server` and `p2p` as shared libraries. This choice was made as the assignment specified their implementation as shared libraries. If you experience any issues compiling with shared libraries please select `N` when prompted.