System Requirements Plan

The system shall:

- Allow players to use keyboard input to steer their Tron motorcycle.
- Leave a snake-like trail behind the players as they move
- Make players be able to crash into the trails they create
- Players can also crash into the walls (i.e. the screen edge)
- Allow new players to connect to the server
- Allow current players to disconnect from the server

The system should:

- Have some sort of chat feature so the players can talk to each other while playing
- Tell all current players when someone has joined or disconnected
- Operate in rounds, once only one player is left the round ends and that player gains a point
- Keep a log of the number of rounds won so far (possible limit for the match) for each player

Shared Data:

WindowSize

This struct contains constant values for the game's screen dimensions and the grid that the game takes place in. The client uses this to set their window size and for colouring in the grid with the player's colours. The server uses the grid size when updating the state of the grid to send to each client.

NetMsg

- Enum attached to the start of all packets for telling server and client what to do with the packet data they receive

PlayerMove

- Enum sent from client to server to indicate the direction that the player is moving

Controls

- Struct to hold the key values used for each client's input checks

PlayerData

Each client (client-side and server-side) uses this struct. It stores player-specific data such as: move directions, client IDs, player controls, whether the player is alive and positions on the grid.

StartData

A struct that contains a set of vectors. Each vector contains four values (since the implementation is supposed to be 4 players max.) that each player uses. They are indexed so that each player can use the data at the index equal to their client id, e.g. player one uses the data in index[0] in each vector, player two uses index[1] and so on.

Classes

UserClient (Client-side):

UserClient
- my_data : PlayerData - mutex : std::mutex - grid : std::vector<sf::Int32>
+ connect(TcpClient&) : void + input(TcpClient&) void + client() : void + getGrid() : vector<sf::Int32> + setControls(Controls&) : void

This class is where the client-side networking happens. The client method connects to the server, then sets up a thread for receiving and sending network messages. It then calls the input method, which is an infinite loop that takes keyboard input based on the controls stored in my_data and sends them to the server to indicate changes in movement. The grid stores IDs to represent spaces that players have been in, blank spaces are represented with -1. Main.cpp gets this grid when updating the graphics in its window loop.

Client (Server-side):

Client
- id : sf::Uint8 - timestamp : chrono::steady_clock::time_point - latency : chrono::microseconds - socket : unique_ptr<sf::TcpSocket> - data : PlayerData
+ setStartDirection(PlayerMove&) : void + setDirection(PlayerMove&) : void + setLatency(chrono::microseconds) : void + setSpawn(sf::Vector2i&) : void + killMe() : void + getSocket() : sf::TcpSocket& + getIndexPosition() : sf::Vector2i& + getPingTime() : sf::Vector2i& + getLatency() : chrono::microseconds + getClientID() : sf::Uint8 + isAlive() : bool + ping() : void + pong() : void + tick() : void + respawn() : void

This class stores and manipulates player data for the server to use in its game loop. The tick method is used to update the client's position based on current movement direction. Ping and pong are used for latency checks with the server if a player doesn't input anything after a certain amount of time. The respawn method simply resets the player's position and sets them to be alive again.

Server:

Server
- player_number : sf::Uint8 - next_id : sf::Uint8 - disconnected_player_ids : priority_queue<sf::Uint8> - start_data : StartData - grid : vector<sf::Int32> - mutex : sf::Mutex
+ bindServerPort(sf::TcpListener&) : bool + connect(sf::TcpListener&, sf::SocketSelector&, TcpClients&) : void + disconnect(TcpClients&, Client&, sf::SocketSelector&) : void + receiveMsg(sf::TcpListener&, sf::SocketSelector, TcpClients&) : void + runMe() : void + ping(TcpClients&) : void + runGame(TcpClients&) : void + sendPositions(TcpClients&) : void + updateGrid(TcpClients&) : void + refreshGrid(TcpClients&) : void + killPlayer(TcpClients&) : void

The server class processes the game's logic and sends the current grid to each connected client. The runMe method sets up threads for updating and sending the grid, and running the game. The grid is updated in the updateGrid method and sent with sendPositions.

When a new player attempts to connect the listener calls the connect method; this adds the client to the current client list and creates a packet that stores the new client's player data and sends it to them. Disconnect is called when someone leaves the game; this method removes the client from the list of current clients and adds their id to the disconnected_player_ids queue. A priority queue was chosen for this because the next id should be the lowest available number, i.e. if client id 4 disconnects, followed by client id 2, the next id to be given should be 2.

Post Mortem

The Good:

Implementing the base functionality for the server and client was relatively quick since a lot of the code was similar to or exactly the same as the code used in tutorials. Setting up the first connection was also quick and easy. Once collisions were setup for one and two players, scaling the game for three and four players was simple. The implementation aimed to allow four players in the game at once from the start, making the setup for each player easy to do (see StartData struct).

The Not So Good:

It was a slow start with getting the light trails in because for a while the player movement was purely vector based. There was also some confusion with what functions the client and server were supposed to perform game-wise, for example to start with the client would process the game loop then send their entire game state to the server. This was later changed so that the server handled all of the game logic. Because the game is designed for a maximum of four players adding any more will create several issues.

Known Issues

A current bug is that sometimes the loop that draws the grid in the client will break. This is possibly an error to do with the scope of the mutexes being used for the main and in the client class – each has their own mutex, meaning locking one does not affect the other which sometimes leads to the data racing

Since the window and grid size were the same throughout the development process the game is only known to work with the default settings. The game did not work properly when testing other sizes.

The game will start to lag when the fourth player joins (it also lags a bit with three players).