LLP Assignment 2: Tron-like

# Planning and design

* Server should handle as much as possible
* Clients should be lightweight
* Data sent over the network should be minimal
* Any code that may block execution should be moved out to its own task/thread
* Threads must terminate properly when window closes
* Up to 8 players/clients
* Clients can drop in/drop out without interrupting the game

# Class overview

Server

* TronServer32
  + Globals
    - std::atomic\_bool running
      * Serves as a flag so threads know to close when no longer running
  + Functions
    - bindServerPort
      * Binds listener to a given port
    - listen
      * Constantly checks for incoming connections and messages, calls ‘ping’ every 250 ms
    - connect
      * Accepts incoming connections, creates client objects and adds them to the client vector
    - recieveMsg
      * Deals with messages picked up by ‘listen’
        + PLAYERDIRECTION will update a player’s direction, i.e. an input from the client
        + PONG will be passed on to the ‘Client’ object’s ‘pong’ function
        + INVALID or DISCONNECT will remove that player from the vector, despawning them
    - ping
      * Sends a ping message to all clients
    - runServer
      * Sets up variables and sockets, calls ‘bindServerPort’, creates players vector and GameGrid, launches ‘runGame’ and ‘broadcastGridData’ asynchronously and returns ‘listen’
    - runGame
      * Main bulk of game logic. Runs in its own thread. Updates player positions, handles collisions and calls to ‘broadcastPlayerData’ every 33 ms (i.e 30 ticks per second).
    - broadcastPlayerData
      * Takes one player’s data as an argument, packs it up and sends it out to all clients.
    - broadcastGridData
      * Loops infinitely while program runs. Packs up game grid into an sf::Packet and sends to all clients as often as possible.
    - spawnPlayer
      * Finds a random position for the player to start in, that has at least a 3x3 empty square around it to give players a chance to get their bearings as the player character starts moving.
    - despawnPlayer
      * Removes player from grid, including their light trail.
* Client
  + Private
    - std::unique<sf::TcpSocket> socket
    - static sf::Uint8 next\_id
    - std::chrono::microseconds latency
    - std::chrono::steady\_clock::time\_point timestamp
    - sf::Uint8 failed\_pings
    - bool connected
    - PlayerData data
  + Functions
    - Constructors
      * Copy constructors needed since sf sockets aren’t copyable
    - operator==
      * Comparison is just based on ID, since it is unique
    - Getters
      * Socket, ID, xPos, yPos, direction, playerdata, latency, timestamp, failed pings, connected
    - Setters
      * Latency, connected, ID, direction, pos
    - ping
      * sends a ping message to the actual client over the network
    - pingFailed
      * failed pings++
    - pingSuccess
      * failed pings = 0
    - pong
      * called when pong message received, updates timestamp and latency

Client

* Main
  + Globals
    - sf::Uint8 clientID
    - sf::RenderWindow window
    - std::vector<std::unique\_ptr<Player>> players
    - std::unique\_ptr<GameGrid> m\_grid
    - std::mutex m\_mutex
    - std::atomic\_bool running
  + Functions
    - input
      * Runs in its own thread. Handles all keyboard-related variables. Chooses a control scheme (one of four) based on client ID. Waits for input and then immediately sends input message to server.
    - disconnectFromServer
      * Send a disconnect packet to the server so it knows to remove the player on that end.
    - draw
      * Draw the grid, and the players if they have a special sprite. Otherwise the grid is enough. Creates just one sf rectangle and draws it in multiple places with multiple colours to conserve memory.
    - client
      * Connects to server, launches ‘recieveMessage’ and returns ‘input’.
    - recieveMessage
      * Listens for incoming messages, launches ‘processMsg’ if any are received to as to not block and listen for further messages
    - processMsg
      * Deals with different types of message from the server.
        + GRID will either create or update the grid, depending on if the client has a copy already.
        + PLAYERDATA will either create or update a player, depending on if the client has a copy already.
        + PING will return a PONG
        + CONNECT will initialise this client’s player, as well as the player id
        + DISCONNECT will remove a given player from the vector
* Player : sf::CircleShape
  + Private
    - PlayerData data
  + Functions
    - Constructor/Destructor
    - Getters
      * ID, xPos, yPos, direction
    - Setters
      * ID, pos, direction

Common

* Stream operators
  + sf::Packet << PlayerData
  + sf::Packet >> PlayerData
* GameGrid
  + Private
    - sf::Uint16 m\_width
    - sf::Uint16 m\_height
    - std::vector<sf::Uint8> m\_grid
  + Functions
    - Constructor/Destructor
    - Getters
      * Grid, value at grid coordinate, value at grid index, width, height
    - Setters
      * Value at grid coordinates, value at grid index
* PlayerData
  + enum PlayerDirection : sf:Uint8
    - Up, down, left, right
  + struct PlayerData
    - sf:Uint8 ID
    - sf:Uint8 xPos
    - sf:Uint8 yPos
    - sf:Uint8 direction
* MessageTypes
  + enum NetMsg : sf::Uint8
    - INVALID, DISCONNECT, CONNECT, PING, PONG, GRID, PLAYERDATA, PLAYERDIRECTION, BLANK

# Post-mortem

The program is functional. A multiplayer game of Tron can be played with drop-in/drop-out gameplay. It is quite basic, as there is no scoring system, lobby or in-game indication of controls. However it is technically stable, and the clients are quite lightweight.

Too much data is sent over the network. The entire grid is sent to every player as often as possible, leading to high network traffic. A better solution would be to only send grid points that have updated since the last broadcast. Additionally, UDP could be used for these quick updates while TCP would continue to be used on a regular interval to ensure players’ copies of the grid are accurate. I attempted some of this in the experimental ‘vectorgrid’ branch, however there were issues with dropped/skipped updates.

Players also have no indication of which player they are, and there is no guarantee they will be a different colour to their opponents, potentially leading to confusion. They also spawn in at almost random points, meaning it can be hard for players to recover if they spawn near a wall. Ideally the players would have fixed spawn positions, though that may interfere with other players’ light trails.