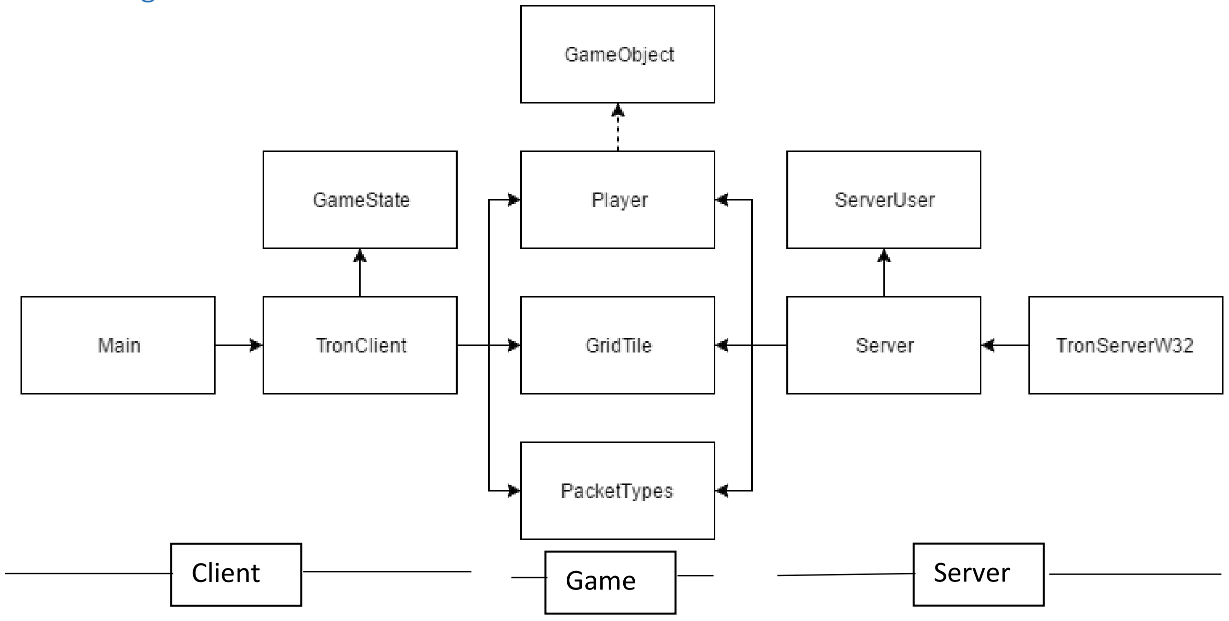
Elliott Smith - 14012566

Tron Game Documentation

Class diagram – Solid line = Owns. Dotted Line = Inherits from



Discussion

The main focus of this project was to ensure that the networking was robust and encapsulated into the correct areas. As displayed by the class diagram, the client, server and game projects had their own specific classes; the client and server made use of game classes for shared data. As the game is run on the server, it needs access to the objects being manipulated so that it can do the calculations and relay that to each client for them to apply it to their own version of the game to display for the user.

Main / TronServerW32 – These classes are the starting points of their respective projects and run an instantiation of their relative main classes (TronClient and Server).

GameState – The GameState is used for client’s to determine what to display and what input to take. For example, in the lobby state it displays all the relevant sprites and only allows players to move up and down on the two buttons and to select the one they are currently on.

TronClient – This is one of the two main classes, the other being the Server. The TronClient is responsible for all the heavy lifting in the client project. This is where packets are received from and sent to the server, sprites are displayed, input is taken and game states are managed.

GameObject – This is the base class for sprite objects. It contains all the base pieces of data such as a sprite, texture and position for classes that inherit from it to take.

Player – The Player class contains information for the user’s player character. The class is mostly used for manipulation of the sprite that represents the player such as movement and rotation. This class also contains information such as whether the player is still alive and other game logic.

GridTile – GridTiles are what the game map is made of. The TronClient contains a vector of GridTiles. These are assigned a texture and are only drawn if they are active. Players move between these and are set to dead if they hit an active one. Players moving over these set them to active to give the appearance of a trail.

PacketTypes – This enum class contains the headers for packets. Every packet that is sent is given a header so that when it is received they receiving party knows what it contains.

ServerUser – This is used as part of the interface between the server and client and is responsible for things such as ping and latency checks.

Server – This is the main server project class. This is where the server side game is run. This is also where packets are received and sent and generally where most of the networking is done. The server sends out all of its own player details to each client to update their versions.

Evaluation

Positive

This project has, overall, been a success. With the goal of implementing a networked Tron-like game the milestones have been completed as the game seems to represent the original multiplayer snake type game fairly well. Iterative development was key to the success of this project; the project as well as the game itself went through many different iterations as design decisions were made and changed. The evolution of the project to the resulting server-run game was extensive and the waterfall development cycle was used in each iteration with specific functionality being implemented and tested before moving on to the next. The fairly stable current build of the project can be attributed to this rigorous iterative testing.

Firstly, the lobby system works as intended with no known bugs. Having players connect and disconnect with the correct IDs being sent provides and very clear and smooth system with connections being clearly shown. This is mostly due to the correct information being sent using the packets between the clients and the server.

The way the packets are handled in this system also works without any known bugs. Using packet headers means that the correct information is sent as well as received so that accurate networking can be done.

Also coming across as a minor success would be the theme implementation. When deciding on a potential theme, Nyan Cat seemed like a perfect fit as it already involved trails being left by the chief character; the possibility of the music being too irritating was brought up however.

The class structure, as it is, also appears to be in a good state. Whilst some things could have been split up into their own class (mainly the Server and TronClient's slightly bloated functionality), the class structure that is there is clear and easy to use.

Finally, the FSM being used in the client project also works as well as can be expected. Whilst only basic, being a simple enum class, it is effective. The correct sprites are drawn and the correct logic is called as well as the transitions between each state also being smooth and clear.

Negative

With its positives also come its negatives. The centre of all setbacks for this project was the networking. During the early stages of the project, the networking seemed to be far too complex and out of scope for the goals and milestones that had been set but these worries were slowly quelled throughout the development.

Arguably the biggest issue with the implementation of the game is how the server-side game is run. To prevent things like cheating and desynchronization on the clients, the design choice of running the main game on the server was made. This has prevented cheating and desynchronization but at the cost of performance. After each game update tick on the server it sends this new data to each client for them to update their versions but the tick rate of this update message is far too slow. Attempts were made to thread this on the server but were unsuccessful. This results in clients receiving movement updates at a rate which leaves players moving on the screen too slowly. The distance in which they move each tick can be increased on the server but this leaves the client players jumping great distances each tick which is far from ideal.

As mentioned previously, a lot of the main code is held in the main classes for each project. The TronClient and Server classes handle far more than they actually need to and so a lot of it could have been separated into its own specific class.

One minor problem that has been encountered is that the updating of menu sprites occasionally does not complete successfully. This results in players remaining dead or spawning with the wrong rotation.

Finally, the sophistication of the code does not match what had been originally envisioned at the start of the project. Whilst it runs as intended, additional functionality could have been added to aid stability. Firstly, the only threading that is done is for the networking of the clients; other threads for functionality such as input or screen drawing would have been nice to have been implemented. With these new threads the addition of mutexes and atomic variables was also planned to protect data that would have been accessed across multiple threads but this fell out of scope when more threads were not implemented.

Potential Improvements

Many potential improvements such as additional classes and threads have been already mentioned but there yet remain more that could potentially improve the project and the game.

As mentioned, the key improvement that is needed is how the server game is run. Sending updates to players more quickly would result in a game with much better flow and a much better ‘feel’ for the gameplay.

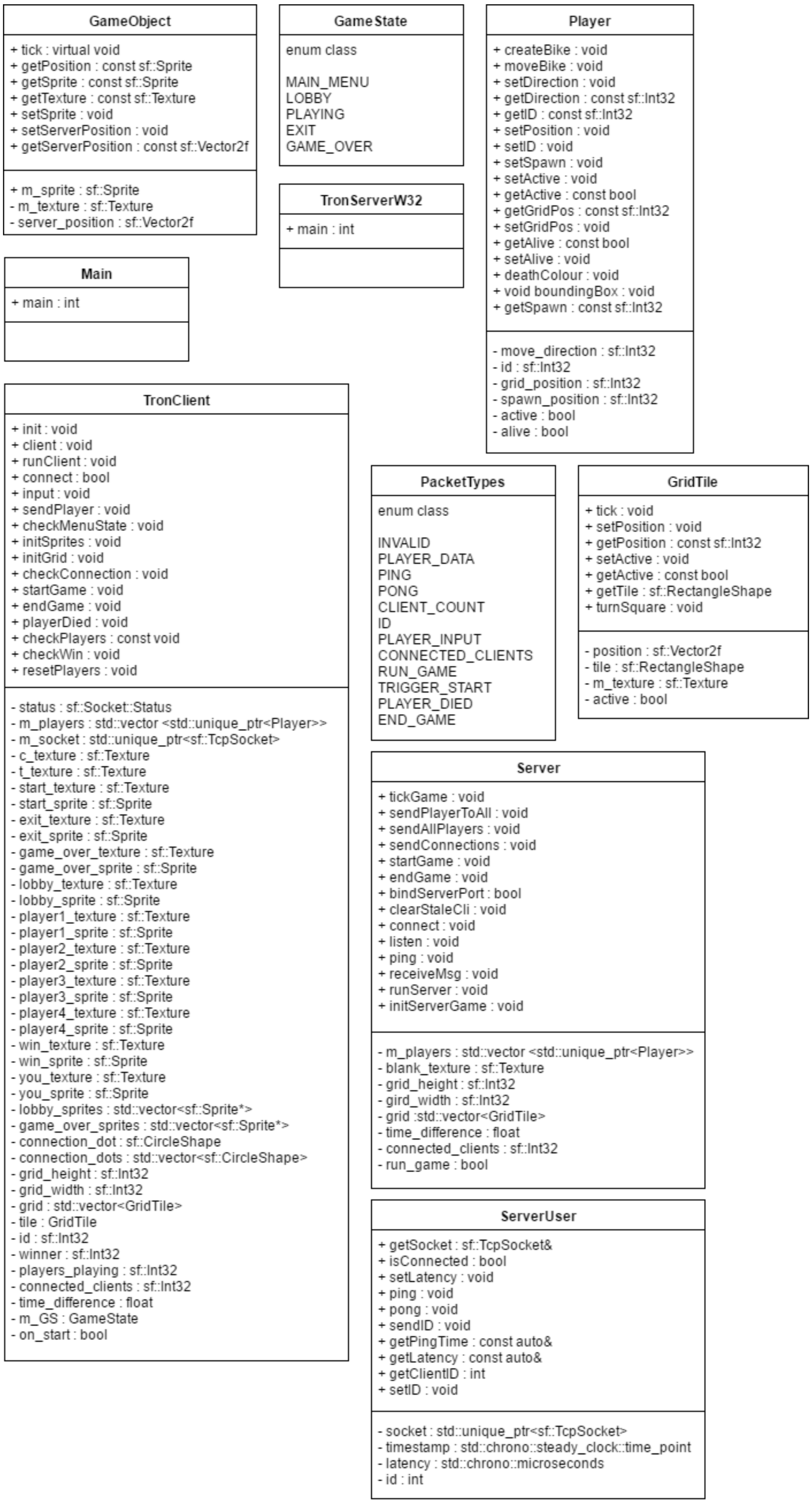
In the original design for this project there was a major stretch goal to add to the gameplay. Whilst it doesn’t exactly fit with the new theme, some kind of projectile that players could collect in some way to fire at trails to destroy part of them was original envisioned. This, of course, fell out of scope as the project went through development but would have potentially been a welcome addition to the gameplay to add variety.

In addition, the audio of the game is currently slightly lacklustre. Additional game sounds for game aspects such as menu navigation and player death could potentially add to the user experience.

Finally, the ability to play across multiple machines would be a very welcome addition. This is possible as other projects with the same brief have achieved this but, again, this functionality fell out of scope. The game works fine when run on the same machine but without differing controls for each player each client must be selected to move the player which does not help with multiplayer gameplay.

Elliott Smith - 14012566

UML Diagram


GameObject
+ tick : virtual void
+ getPosition : const sf::Sprite
+ getSprite : const sf::Sprite
+ getTexture : const sf::Texture
+ setSprite : void
+ setServerPosition : void
+ getServerPosition : const sf::Vector2f
+ m_sprite : sf::Sprite
- m_texture : sf::Texture
- server_position : sf::Vector2f
GameState
enum class
MAIN_MENU
LOBBY
PLAYING
EXIT
GAME_OVER
TronServerW32
+ main : int
Player
+ createBike : void
+ moveBike : void
+ setDirection : void
+ getDirection : const sf::Int32
+ getID : const sf::Int32
+ setPosition : void
+ setID : void
+ setSpawn : void
+ setActive : void
+ getActive : const bool
+ getGridPos : const sf::Int32
+ setGridPos : void
+ getAlive : const bool
+ setAlive : void
+ deathColour : void
+ void boundingBox : void
+ getSpawn : const sf::Int32
- move_direction : sf::Int32
- id : sf::Int32
- grid_position : sf::Int32
- spawn_position : sf::Int32
- active : bool
- alive : bool
Main
+ main : int
TronClient
+ init : void
+ client : void
+ runClient : void
+ connect : bool
+ input : void
+ sendPlayer : void
+ checkMenuState : void
+ initSprites : void
+ initGrid : void
+ checkConnection : void
+ startGame : void
+ endGame : void
+ playerDied : void
+ checkPlayers : const void
+ checkWin : void
+ resetPlayers : void
- status : sf::Socket::Status
- m_players : std::vector <std::unique_ptr<Player>>
- m_socket : std::unique_ptr<sf::TcpSocket>
- c_texture : sf::Texture
- t_texture : sf::Texture
- start_texture : sf::Texture
- start_sprite : sf::Sprite
- exit_texture : sf::Texture
- exit_sprite : sf::Sprite
- game_over_texture : sf::Texture
- game_over_sprite : sf::Sprite
- lobby_texture : sf::Texture
- lobby_sprite : sf::Sprite
- player1_texture : sf::Texture
- player1_sprite : sf::Sprite
- player2_texture : sf::Texture
- player2_sprite : sf::Sprite
- player3_texture : sf::Texture
- player3_sprite : sf::Sprite
- player4_texture : sf::Texture
- player4_sprite : sf::Sprite
- win_texture : sf::Texture
- win_sprite : sf::Sprite
- you_texture : sf::Texture
- you_sprite : sf::Sprite
- lobby_sprites : std::vector<sf::Sprite*>
- game_over_sprites : std::vector<sf::Sprite*>
- connection_dot : sf::CircleShape
- connection_dots : std::vector<sf::CircleShape>
- grid_height : sf::Int32
- grid_width : sf::Int32
- grid : std::vector<GridTile>
- tile : GridTile
- id : sf::Int32
- winner : sf::Int32
- players_playing : sf::Int32
- connected_clients : sf::Int32
- time_difference : float
- m_GS : GameState
- on_start : bool
PacketTypes
enum class
INVALID
PLAYER_DATA
PING
PONG
CLIENT_COUNT
ID
PLAYER_INPUT
CONNECTED_CLIENTS
RUN_GAME
TRIGGER_START
PLAYER_DIED
END_GAME
GridTile
+ tick : void
+ setPosition : void
+ getPosition : const sf::Int32
+ setActive : void
+ getActive : const bool
+ getTile : sf::RectangleShape
+ turnSquare : void
- position : sf::Vector2f
- tile : sf::RectangleShape
- m_texture : sf::Texture
- active : bool
Server
+ tickGame : void
+ sendPlayerToAll : void
+ sendAllPlayers : void
+ sendConnections : void
+ startGame : void
+ endGame : void
+ bindServerPort : bool
+ clearStaleCli : void
+ connect : void
+ listen : void
+ ping : void
+ receiveMsg : void
+ runServer : void
+ initServerGame : void
- m_players : std::vector <std::unique_ptr<Player>>
- blank_texture : sf::Texture
- grid_height : sf::Int32
- gird_width : sf::Int32
- grid :std::vector<GridTile>
- time_difference : float
- connected_clients : sf::Int32
- run_game : bool
ServerUser
+ getSocket : sf::TcpSocket&
+ isConnected : bool
+ setLatency : void
+ ping : void
+ pong : void
+ sendID : void
+ getPingTime : const auto&
+ getLatency : const auto&
+ getClientID : int
+ setID : void
- socket : std::unique_ptr<sf::TcpSocket>
- timestamp : std::chrono::steady_clock::time_point
- latency : std::chrono::microseconds
- id : int