Joe da Silva - 15000015

LOW-LEVEL PROGRAMMING ASSIGNMENT 2 - DOCUMENTATION

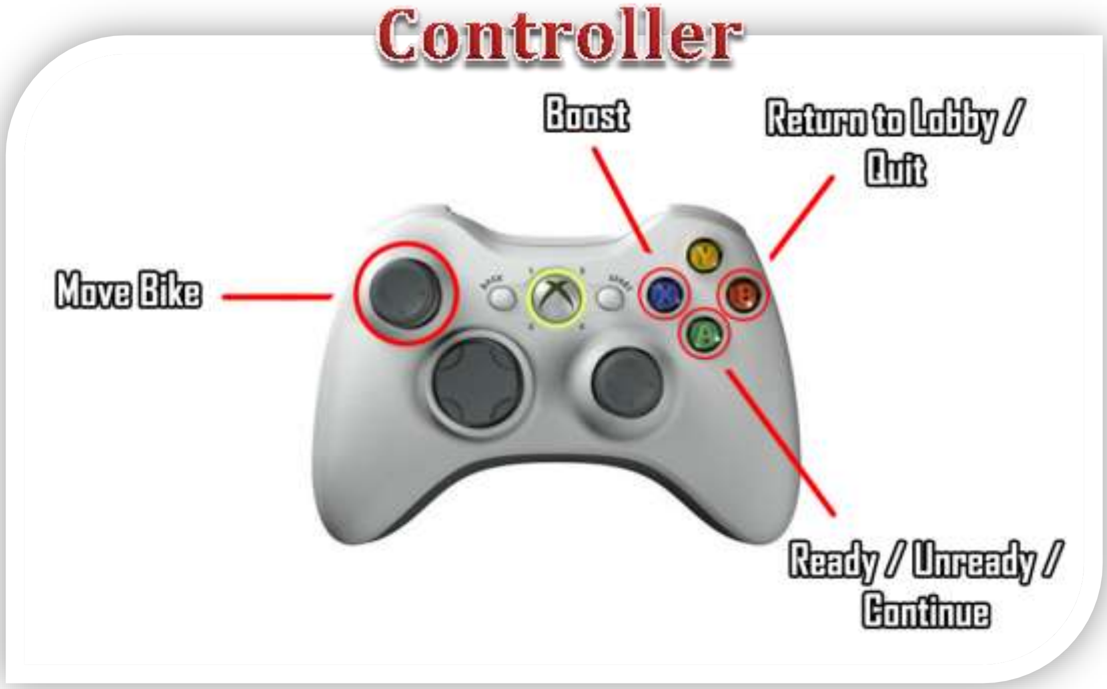
DEVELOPER NOTES

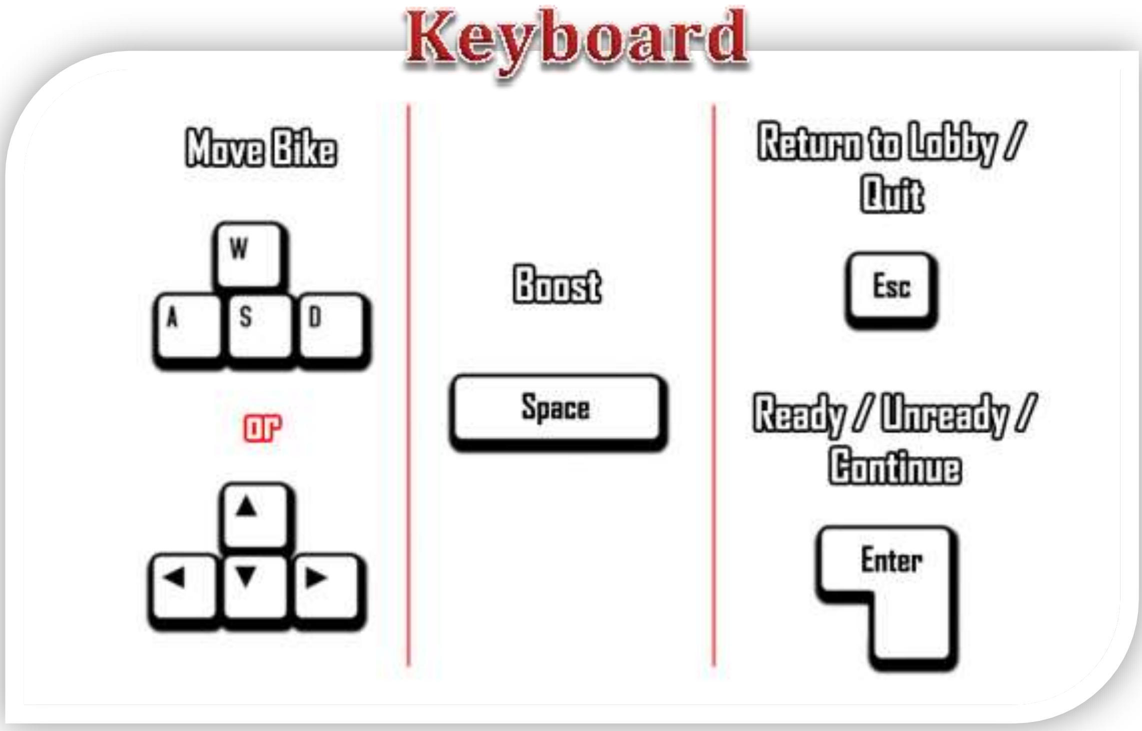
- The game supports a maximum of 4 players.
- The game requires a minimum of 2 players to start (though all connected players must be ready).
- The game has Xbox controller and Keyboard support (See documentation for controls).
- The game has audio, enjoy it!
- Take note of the colour the server assigns you in the lobby as it is the colour of your bike.
- Your boost charges replenish over time. Be aggressive!
- After a round, all players must go back to the lobby before a new one can be started.
- Players who connect while a round is in progress must wait until it is over to play.

SERVER SETTINGS

The game uses a server_settings.txt file in Resources to determine the destination server. The first line of the file is the server's IP; the second line is the server's TCP port.

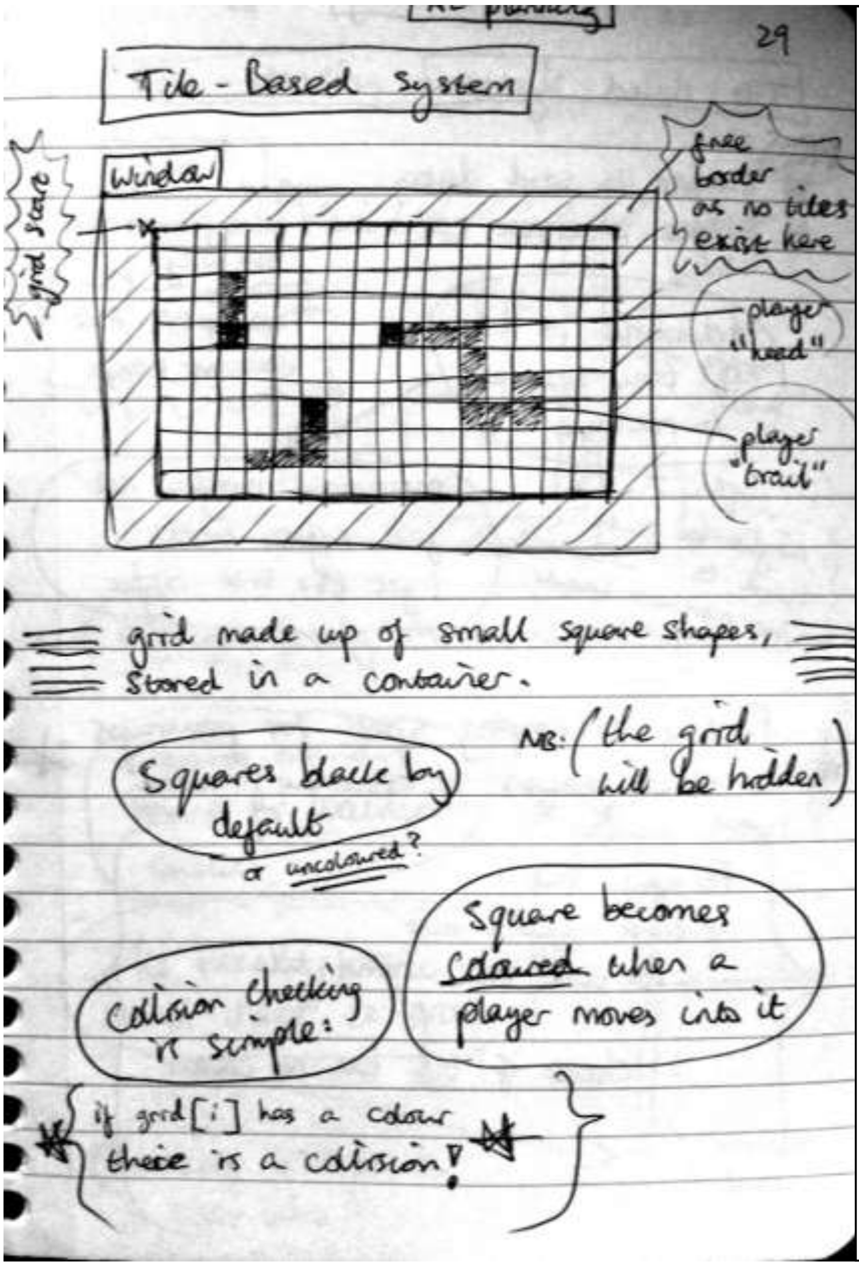
CONTROLS


Controller
Boost
Return to Lobby / Quit
Move Bike
Ready / Unready / Continue


Keyboard
Move Bike
W
A
S
D
or
▲
◀
▼
▶
Boost
Space
Return to Lobby / Quit
Esc
Ready / Unready / Continue
Enter

PLANNING

NOTEBOOK SCANS



grid made up of small square shapes, stored in a container.

NB: (the grid will be hidden)

Squares black by default or uncoloured?

Square becomes coloured when a player moves into it

Collision checking is simple:

if grid[i] has a colour there is a collision!

One of the first things I considered when planning the structure of the game was what sort of underlying system I was going to use for the gameplay.

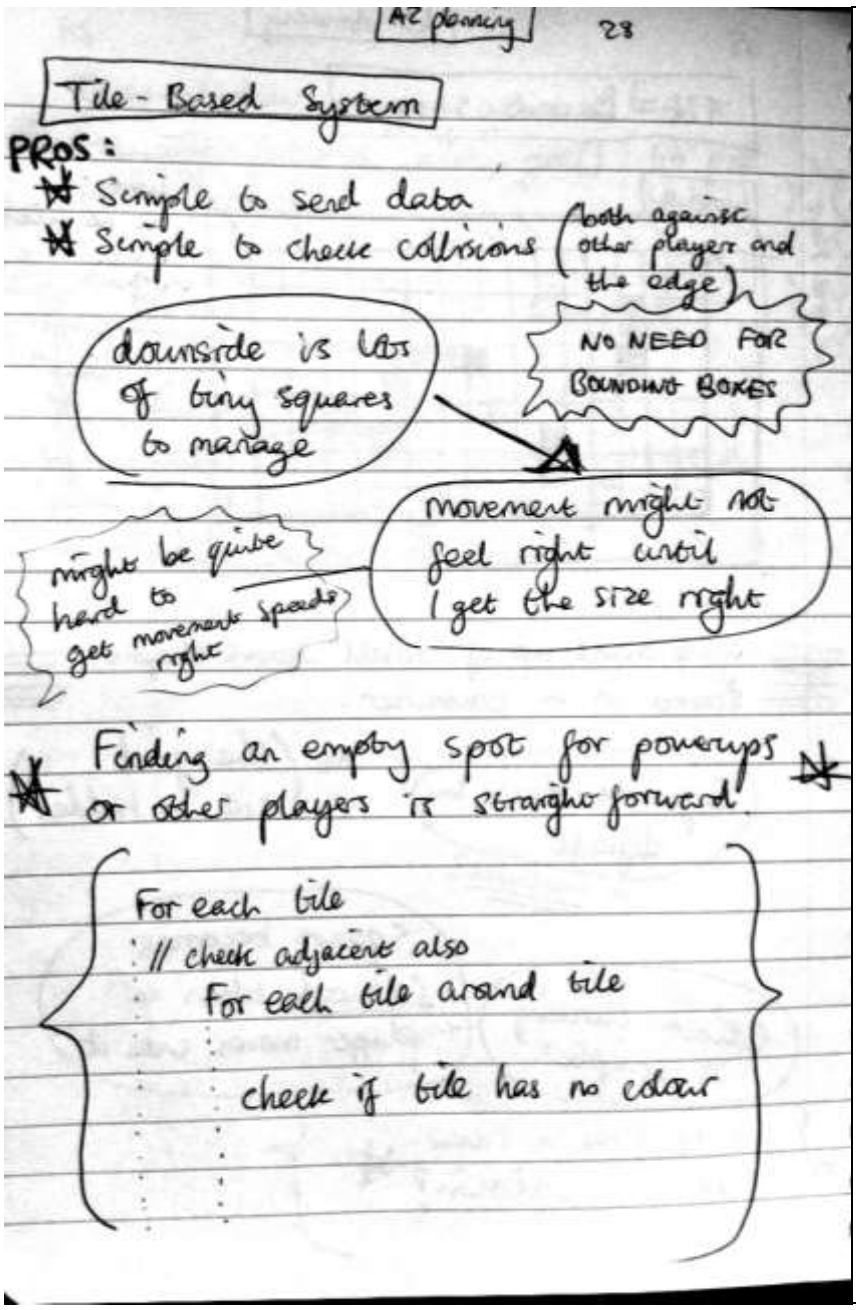
I identified two distinct types of infrastructure: a Tile-Based system, and a Free-Standing system.

The former system sounded a lot easier on paper. Having the movement on a grid meant a lot of important events would be easy to handle, such as player collision (with other players or with the edge of the playing area).

The idea I had with this system was straight forward. Each time I moved a bike I would check if the destination space was valid. A space would only be valid if it was within the playing area and it was currently unoccupied, i.e. the cell had a value of 0 or similar.

Then, if the space is valid, I simply colour that square in or mark it with a value, and repeat the process.

Joe da Silva - 15000015



A2 planning 28

Tile-Based System

PROS:

- Simple to send data
- Simple to check collisions (both against other players and the edge)

downside is lots of tiny squares to manage

NO NEED FOR BOUNDING BOXES

movement might not feel right until I get the size right

might be quite hard to get movement speeds right

- Finding an empty spot for powerups or other players is straighforward

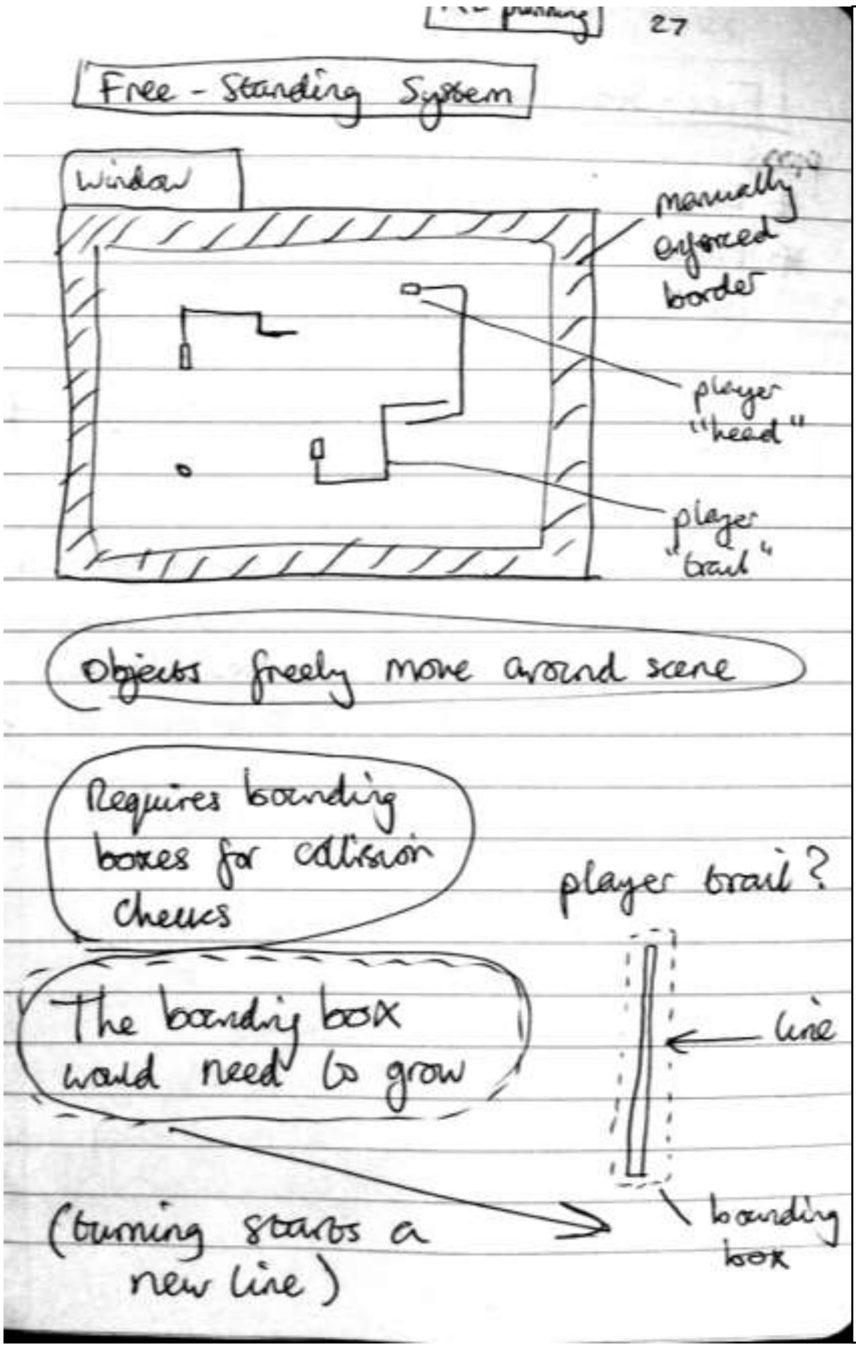
```
For each tile
    // check adjacent also
    For each tile around tile
        check if tile has no colour
```

I considered some pros and cons of this system.

On the one hand, I thought it would be fairly straightforward to send data, and handle collisions. On the other, I thought it would be tricky to get the game feeling "right", due to all movement being handled on the grid, and thus it may feel sluggish or jerky if the speeds weren't right.

I also considered the ease of determining spawn points for power ups or players using this system. Since I would be running without bounding boxes, I would need to check surrounding cells.

As highlighted in the scan, a downside with this system is that I now have a grid to manage, and there's a lot that comes with setting that up and making sure it's used correctly. E.g. calculating indexes or displaying the tiles on screen.


27
Free - Standing System
Window
manually enforced border
player "head"
player "trail"
Objects freely move around scene
Requires bounding boxes for collision checks
player trail?
The bounding box would need to grow
line
(turning starts a new line)
bounding box

While the Tile-Based system sounded fairly straight forward to implement, I did consider that it came with a fairly big overhead.

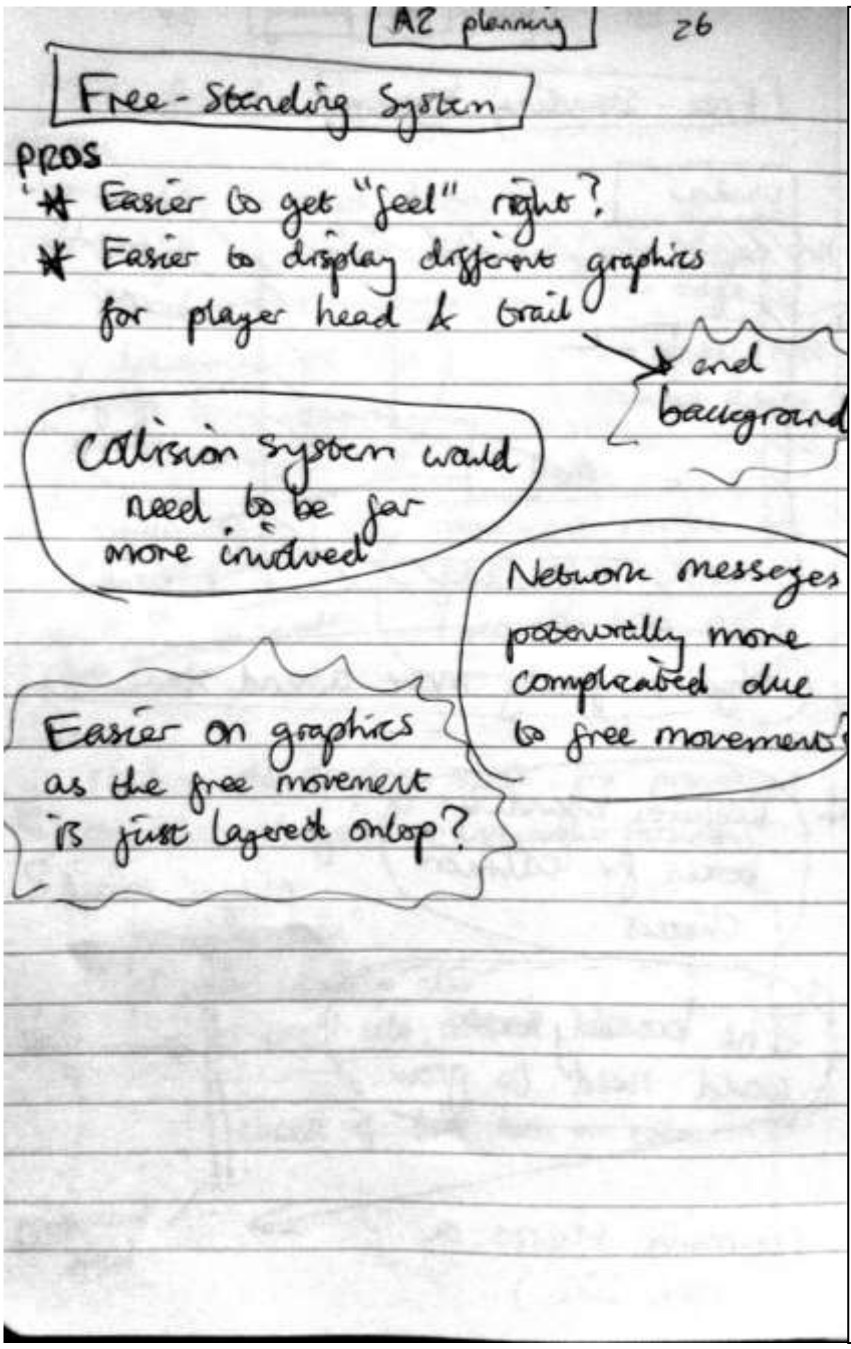
The alternative was the Free-Standing system, which would have objects be able to move about freely without being bound to a grid.

To implement this properly, I figured that bounding boxes or some form of more involved collision checking would be required.

I had thought of a system that would create a line behind the player, which would grow as long as the player headed in the same direction.

Then whenever the player changed direction, the current line would end and a new one would start.

This would be in place of constantly dropping small squares or other objects behind the player. It would just be one continuous shape with the same bounding box.


A2 planning 26

Free-Standing System

PROS

* Easier to get "feel" right?
* Easier to display different graphics for player head & trail

and background

Collision system would need to be far more involved

Network messages potentially more complicated due to free movement

Easier on graphics as the free movement is just layered ontop?

With this approach, I thought that it would be easier to get the game feeling right. And I think that was mainly due to the movement not being confined to a grid, thus it would be easier to tamper with movement speeds without also having to consider if that needed the grid size to change.

I also thought that achieving satisfactory graphics would be easier with this approach, as the player would most likely be a sprite, rather than a coloured square in a sea of squares.

The main concern with this system was that the collision system would be much more involved than the Tile-Based approach, and that a lot of time would need to go into perfecting that.

In regards to sending data over the network, I actually weighed this one as being more difficult to synchronise correctly than the Tile-based approach, even though there would be potentially far less data to broadcast.

I think back then I drastically underestimated how big my grid would actually need to be to have a reasonably sized playing area.

SUMMARY

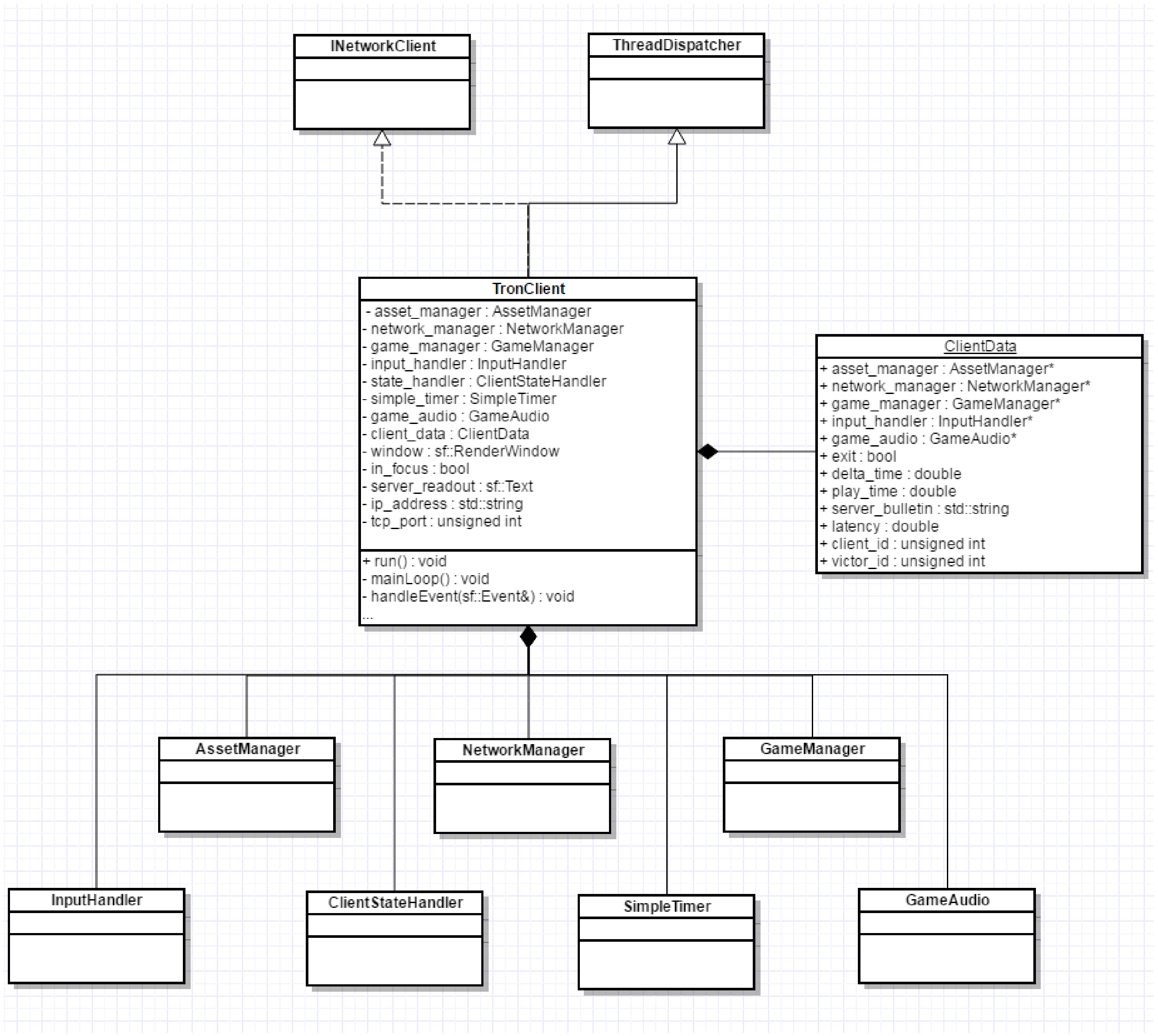
Ultimately, I decided to go for the Tile-Based approach, as I felt that the simplicity of its implementation would allow more time to focus on networking, which was integral to this assignment.

I had also played a number of tron-like games whilst deciding how to implement the game, and it seemed like in many of them there was a sense of grid-like movement, at least from a visual standpoint. Thus, I decided it was a feasible approach and went with it.

One final thing I noticed from playing these games was that some incorporated a boost mechanic to compliment the claustrophobic and risk-taking atmosphere. In games with this system, a player usually had a limited number of boosts they could use to gain a brief burst of speed. There's clear risk-reward to this mechanic, in that you can try to outmaneuver your opponents and catch them off guard, but you also risk being overly aggressive and force your own demise. Games that I played without this mechanic didn't feel as intense or competitive, so I decided it would be a good mechanic to implement in my own game.

CLASS HIERARCHIES

TRONCLIENT


INetworkClient
ThreadDispatcher
TronClient
- asset_manager : AssetManager
- network_manager : NetworkManager
- game_manager : GameManager
- input_handler : InputHandler
- state_handler : ClientStateHandler
- simple_timer : SimpleTimer
- game_audio : GameAudio
- client_data : ClientData
- window : sf::RenderWindow
- in_focus : bool
- server_readout : sf::Text
- ip_address : std::string
- tcp_port : unsigned int
+ run() : void
- mainLoop() : void
- handleEvent(sf::Event&) : void
...
ClientData
+ asset_manager : AssetManager*
+ network_manager : NetworkManager*
+ game_manager : GameManager*
+ input_handler : InputHandler*
+ game_audio : GameAudio*
+ exit : bool
+ delta_time : double
+ play_time : double
+ server_bulletin : std::string
+ latency : double
+ client_id : unsigned int
+ victor_id : unsigned int
AssetManager
NetworkManager
GameManager
InputHandler
ClientStateHandler
SimpleTimer
GameAudio

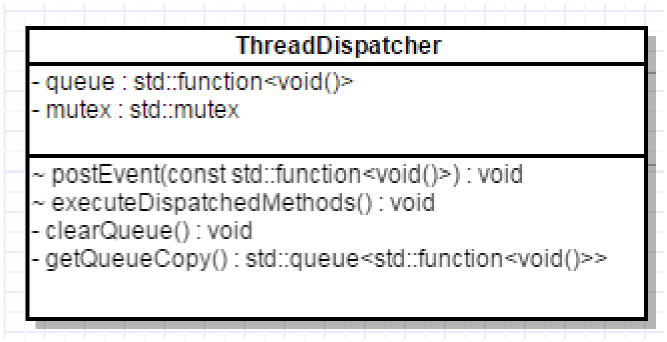
TronClient sits at the heart of the game on the client's side of things, as it owns the majority of core systems used throughout the game's lifetime.

Once the TronClient has been started, it enters a main loop where it begins ticking core systems such as the ClientStateHandler, which acts mainly as a flow control for certain aspects of the program.

During runtime, the TronClient is mainly occupied with processing events that have been posted to it from the NetworkManager. These events are usually passed to the GameManager, as that is where the client-side simulation is kept.

A ClientData struct is used to pass common important data between different areas of the program, including the GameManager and ClientStateHandler.

THREADDISPATCHER


ThreadDispatcher
- queue : std::function<void()>
- mutex : std::mutex
~ postEvent(const std::function<void()>) : void
~ executeDispatchedMethods() : void
- clearQueue() : void
- getQueueCopy() : std::queue<std::function<void()>>

ThreadDispatcher is a thread-safe class that is designed to be inherited from by classes that wish to have events posted to them from other threads.

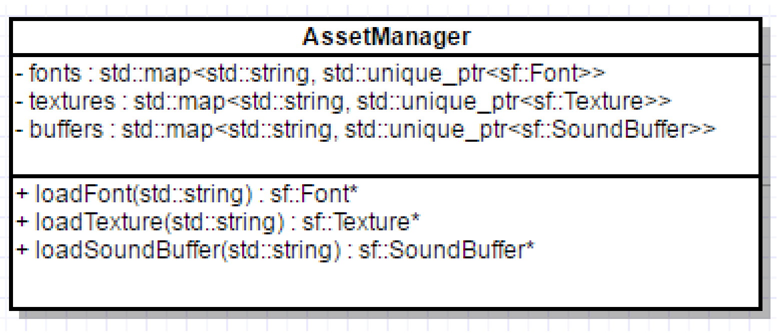
A class that inherits from ThreadDispatcher should include the 'executeDispatchedMethods' in its main loop to ensure prompt execution of posted events.

When a derived class attempts to execute the methods in the function queue, it first tries to make a copy of the queue, so that messages can still be posted to the queue if iterating through the queue and executing the methods happens to take a significant amount of time.

A mutex lock guard in both the postEvent function and getQueueCopy function ensures that threads aren't able to post events to a queue that is currently in the process of being copied.

Within the program, four classes derive from ThreadDispatcher: **TronClient**, **NetworkManager**, **TronServer**, and **SimulationThread**. In each of these cases, the purpose is to form a thread-safe line of communication between client-side or server-side systems.

ASSETMANAGER

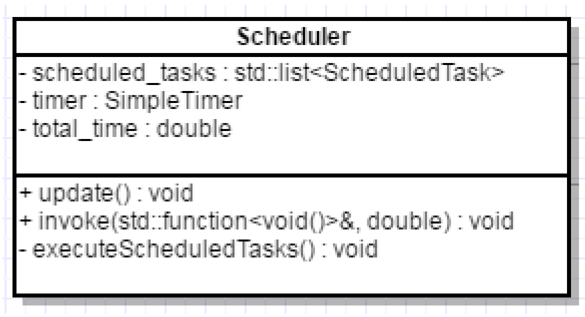

AssetManager
- fonts : std::map<std::string, std::unique_ptr<sf::Font>>
- textures : std::map<std::string, std::unique_ptr<sf::Texture>>
- buffers : std::map<std::string, std::unique_ptr<sf::SoundBuffer>>
+ loadFont(std::string) : sf::Font*
+ loadTexture(std::string) : sf::Texture*
+ loadSoundBuffer(std::string) : sf::SoundBuffer*

AssetManager is a fairly straightforward class that acts as a repository for all assets that are to be used inside the game.

AssetManager keeps a separate map for each type of asset, and a function to request each asset type. When an asset is first requested, it is constructed and placed in the map. Thereafter, a pointer to the asset is returned. This avoids unecessary reloading of data, and centralises the ownership of the assets inside the AssetManager.

Ideally the AssetManager would be templated, or made more generic. Currently there is a fair amount of code duplication just to populate the different maps.

SCHEDULER



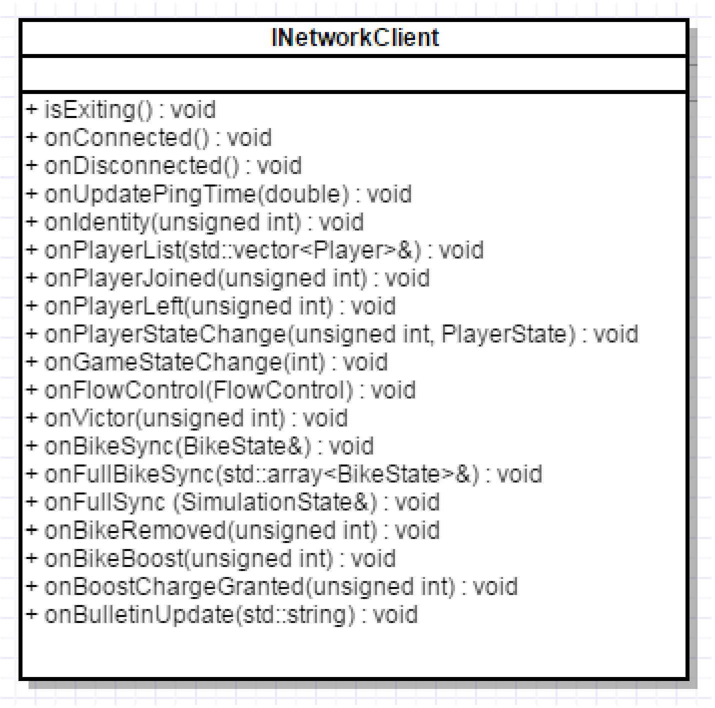
```
Scheduler
- scheduled_tasks : std::list<ScheduledTask>
- timer : SimpleTimer
- total_time : double

+ update() : void
+ invoke(std::function<void()>&, double) : void
- executeScheduledTasks() : void
```

Scheduler is a simple class inspired by the Invoke functionality found in Unity.

As long as the scheduler's update function is called from the owning class' main loop, functions can be scheduled with the Scheduler and they will automatically be invoked after the specified time has passed.

This class has been used to great effect across the program, for the purposes of scheduling repeating events such as synchronisation methods and ping requests.

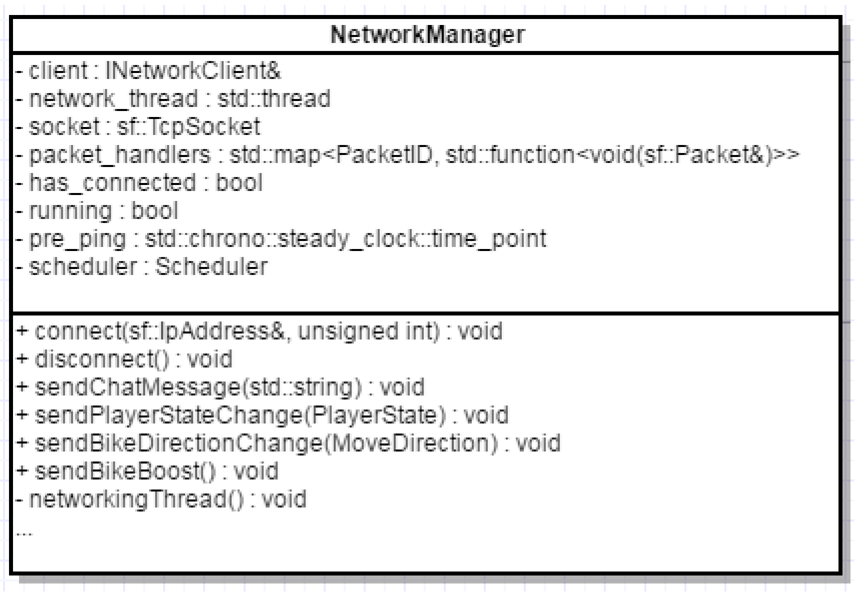
INETWORKCLIENT



```
INetworkClient

+ isExiting() : void
+ onConnected() : void
+ onDisconnected() : void
+ onUpdatePingTime(double) : void
+ onIdentity(unsigned int) : void
+ onPlayerList(std::vector<Player>&) : void
+ onPlayerJoined(unsigned int) : void
+ onPlayerLeft(unsigned int) : void
+ onPlayerStateChange(unsigned int, PlayerState) : void
+ onGameStateChange(int) : void
+ onFlowControl(FlowControl) : void
+ onVictor(unsigned int) : void
+ onBikeSync(BikeState&) : void
+ onFullBikeSync(std::array<BikeState>&) : void
+ onFullSync (SimulationState&) : void
+ onBikeRemoved(unsigned int) : void
+ onBikeBoost(unsigned int) : void
+ onBoostChargeGranted(unsigned int) : void
+ onBulletinUpdate(std::string) : void
```

INetworkClient is an abstract class designed to be derived by TronClient and used by the NetworkManager. This ensures that only the network-related aspects of the TronClient are accessible by the NetworkManager.

NETWORKMANAGER


NetworkManager
- client : INetworkClient&
- network_thread : std::thread
- socket : sf::TcpSocket
- packet_handlers : std::map<PacketID, std::function<void(sf::Packet&)>>
- has_connected : bool
- running : bool
- pre_ping : std::chrono::steady_clock::time_point
- scheduler : Scheduler
+ connect(sf::IpAddress&, unsigned int) : void
+ disconnect() : void
+ sendChatMessage(std::string) : void
+ sendPlayerStateChange(PlayerState) : void
+ sendBikeDirectionChange(MoveDirection) : void
+ sendBikeBoost() : void
- networkingThread() : void
...

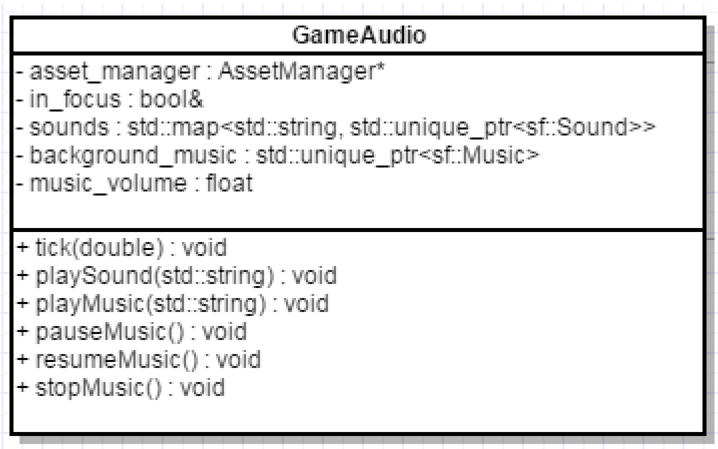
NetworkManager is the client-side hub for all network traffic to and from the server.

Like TronClient, NetworkManager derives from ThreadDispatcher so that events can be posted to it safely across the thread boundary.

NetworkManager uses the INetworkClient interface for informing the TronClient of network messages that come in from the server.

Likewise, when the TronClient's members need to send a message across the server, such as when a bike needs to change direction based on user input, they can access the NetworkManager through the ClientData struct and call the appropriate method. The event is then posted to the NetworkManager's queue and is executed the next time it has a chance to execute it.

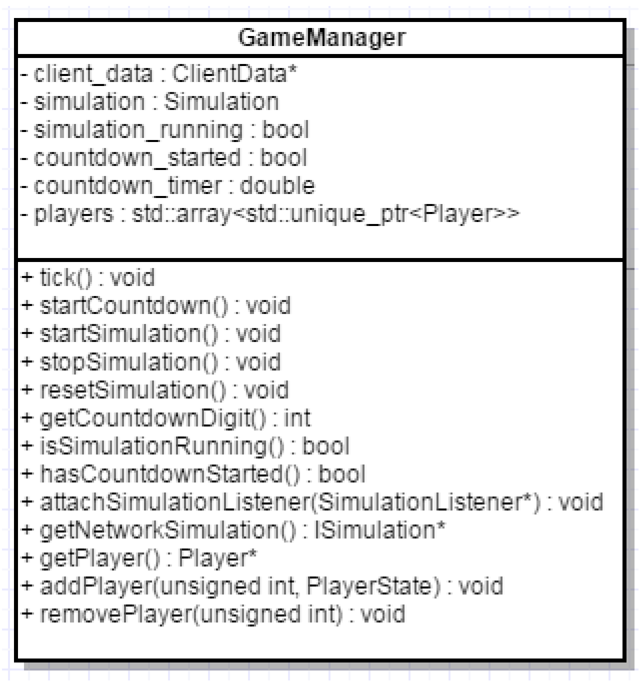
GAMEAUDIO


GameAudio
- asset_manager : AssetManager*
- in_focus : bool&
- sounds : std::map<std::string, std::unique_ptr<sf::Sound>>
- background_music : std::unique_ptr<sf::Music>
- music_volume : float
+ tick(double) : void
+ playSound(std::string) : void
+ playMusic(std::string) : void
+ pauseMusic() : void
+ resumeMusic() : void
+ stopMusic() : void

GameAudio is used for playing sounds and music on the client. GameAudio uses the AssetManager to populate its list of available sounds, and it streams music straight from a file in accordance with SFML guidance.

GameAudio stores the in_focus bool from TronClient so it can mute audio when the window is not in focus, which helps to avoid sound pollution (especially when testing with multiple clients!).

GAMEMANAGER


GameManager
- client_data : ClientData*
- simulation : Simulation
- simulation_running : bool
- countdown_started : bool
- countdown_timer : double
- players : std::array<std::unique_ptr<Player>>
+ tick() : void
+ startCountdown() : void
+ startSimulation() : void
+ stopSimulation() : void
+ resetSimulation() : void
+ getCountdownDigit() : int
+ isSimulationRunning() : bool
+ hasCountdownStarted() : bool
+ attachSimulationListener(SimulationListener*) : void
+ getNetworkSimulation() : ISimulation*
+ getPlayer() : Player*
+ addPlayer(unsigned int, PlayerState) : void
+ removePlayer(unsigned int) : void

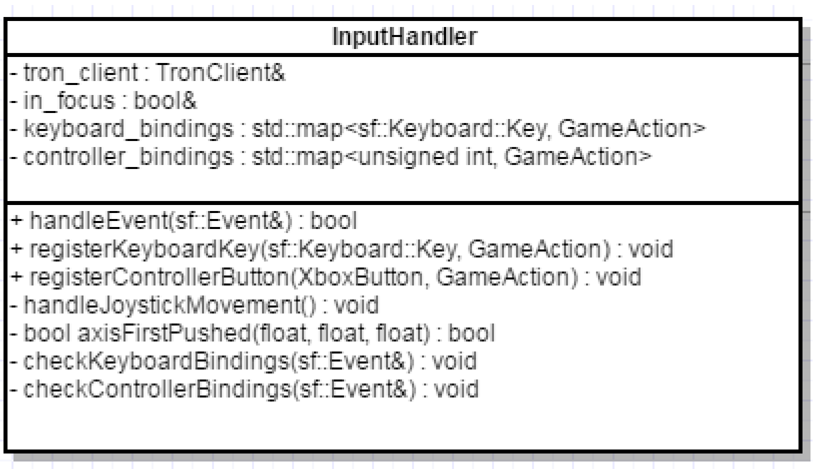
GameManager is the managerial class which oversees the flow of the client's simulation. Additionally, GameManager is charged with maintaining the list of Players which are connected to the server. The client themselves resides within this list, and the list exists for the purposes of displaying player information in the lobby.

The Player class stores just the player's id and their current state.

Because the client's simulation can't start or stop without the server's say, GameManager exposes a number of methods to control the flow of the simulation from network messages, so that it can synchronise its simulation with the server's.

The countdown information is used by the client's game state for displaying the full countdown before the simulation actually starts. The server doesn't actually simulate the whole countdown, it instead simply uses a Scheduler invoke call to schedule the simulation start after a certain amount of time.

INPUTHANDLER


InputHandler
- tron_client : TronClient&
- in_focus : bool&
- keyboard_bindings : std::map<sf::Keyboard::Key, GameAction>
- controller_bindings : std::map<unsigned int, GameAction>
+ handleEvent(sf::Event&) : bool
+ registerKeyboardKey(sf::Keyboard::Key, GameAction) : void
+ registerControllerButton(XboxButton, GameAction) : void
- handleJoystickMovement() : void
- bool axisFirstPushed(float, float, float) : bool
- checkKeyboardBindings(sf::Event&) : void
- checkControllerBindings(sf::Event&) : void

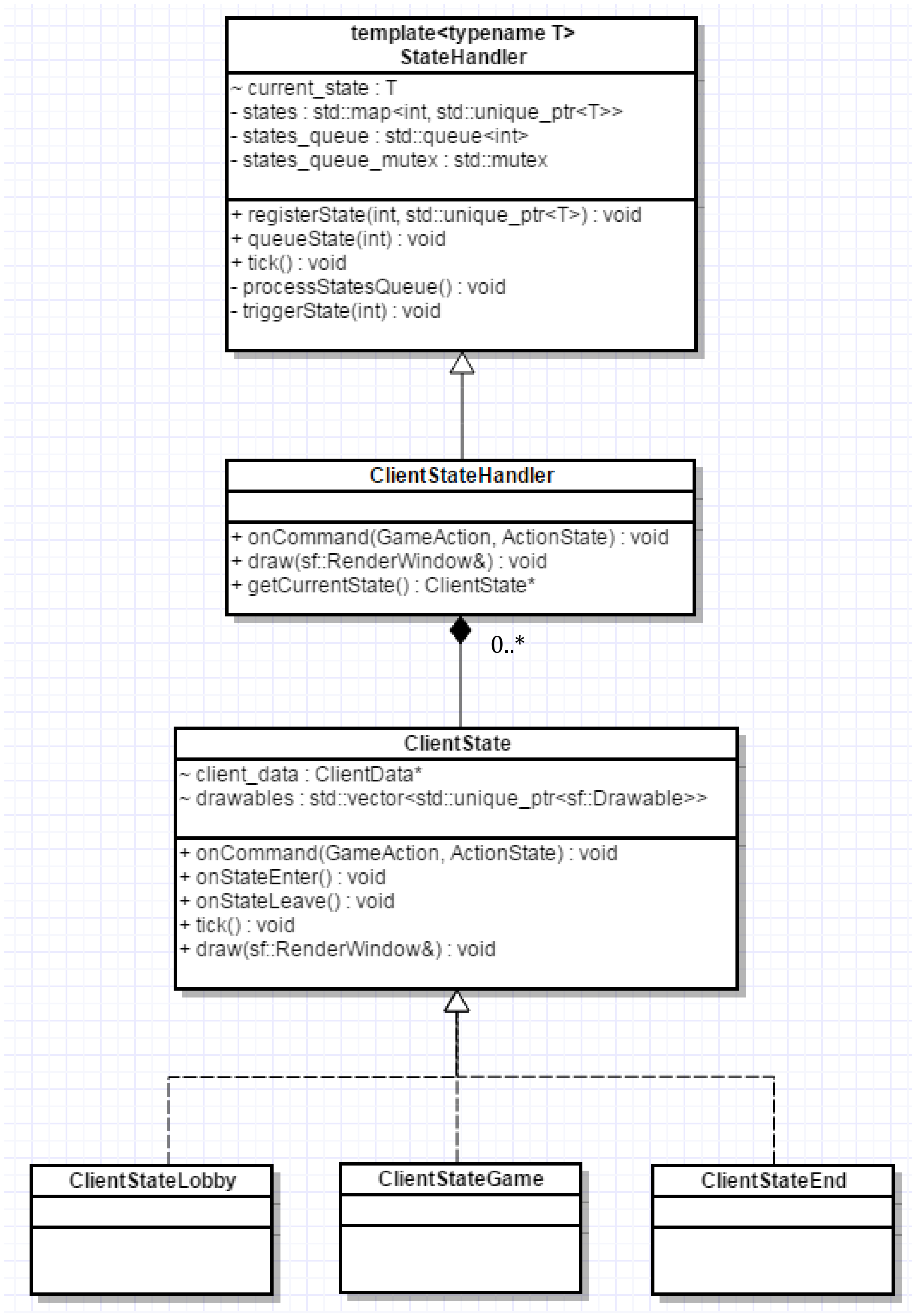
InputHandler's focus is to translate any SFML events it is passed into GameActions. It does this by checking its list of bindings, which will have been registered earlier in the program.

The evaluated GameAction is then passed back to the TronClient via the onCommand function, which is forwarded on to the states to perform the appropriate behaviour.

The bindings themselves are stored in maps separated by Keyboard and Controller. In the current implementation, InputHandler will only store unique key bindings, the same key cannot be registered twice.

Controller support is enabled, but the InputHandler only listens to inputs from the first connected controller.

CLIENTSTATEHANDLER

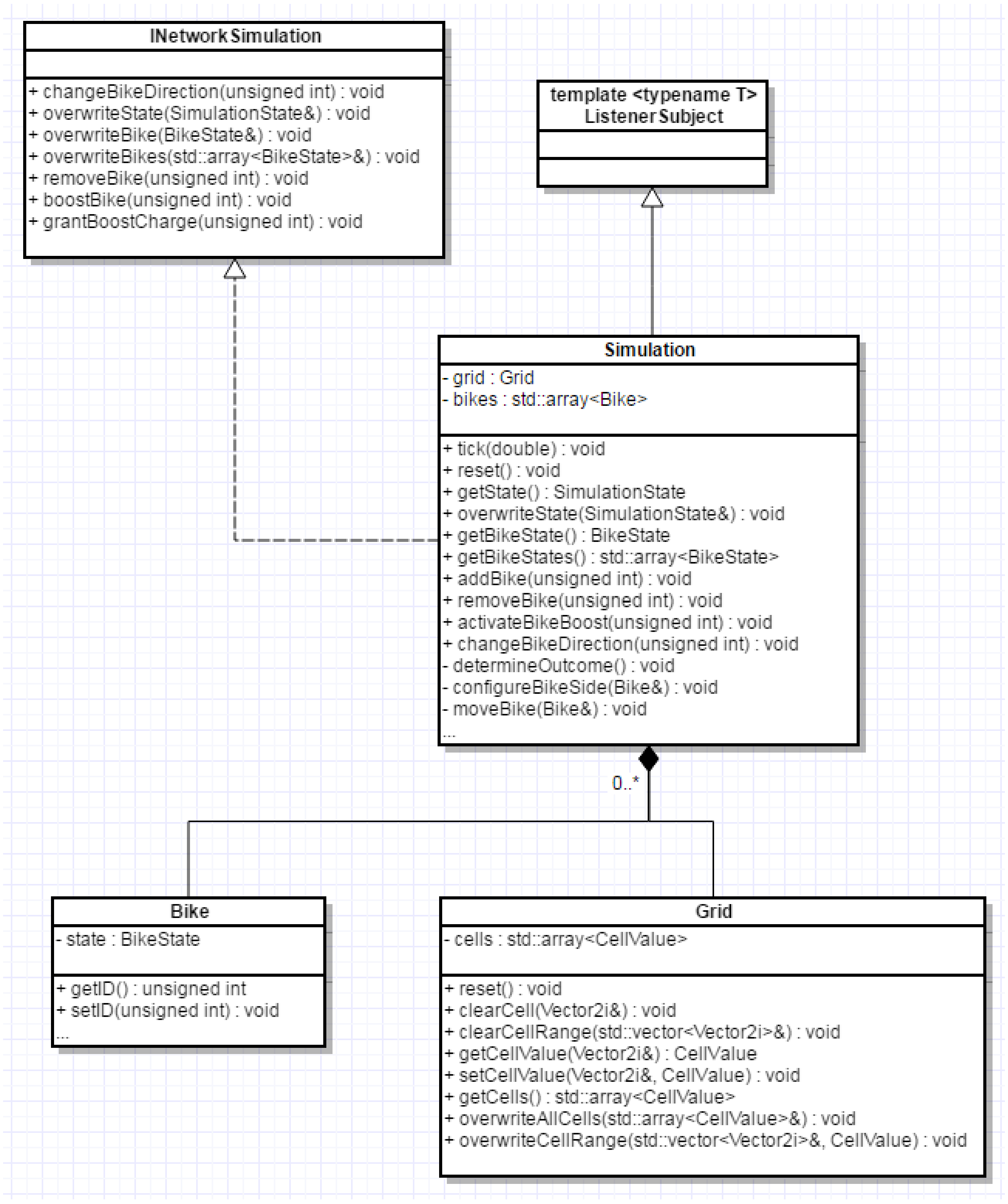

template<typename T>
StateHandler
~ current_state : T
- states : std::map<int, std::unique_ptr<T>>
- states_queue : std::queue<int>
- states_queue_mutex : std::mutex
+ registerState(int, std::unique_ptr<T>) : void
+ queueState(int) : void
+ tick() : void
- processStatesQueue() : void
- triggerState(int) : void
ClientStateHandler
+ onCommand(GameAction, ActionState) : void
+ draw(sf::RenderWindow&) : void
+ getCurrentState() : ClientState*
0..*
ClientState
~ client_data : ClientData*
~ drawables : std::vector<std::unique_ptr<sf::Drawable>>
+ onCommand(GameAction, ActionState) : void
+ onStateEnter() : void
+ onStateLeave() : void
+ tick() : void
+ draw(sf::RenderWindow&) : void
ClientStateLobby
ClientStateGame
ClientStateEnd

ClientStateHandler extends the templated StateHandler class and handles states of type ClientState, which extend the templated State class with client-specific functionality such as onCommand and draw.

The templated StateHandler was designed to be reusable, I intended to use it for both the client and server states. Unfortunately there wasn't enough time to implement a ServerStateHandler. However, use of the templated pattern has been demonstrated with the ClientStateHandler.

On the client, the states mainly serve as flow control for drawing/ticking certain elements.

SIMULATION


INetworkSimulation
+ changeBikeDirection(unsigned int) : void
+ overwriteState(SimulationState&) : void
+ overwriteBike(BikeState&) : void
+ overwriteBikes(std::array<BikeState>&) : void
+ removeBike(unsigned int) : void
+ boostBike(unsigned int) : void
+ grantBoostCharge(unsigned int) : void
template <typename T>
ListenerSubject
Simulation
- grid : Grid
- bikes : std::array<Bike>
+ tick(double) : void
+ reset() : void
+ getState() : SimulationState
+ overwriteState(SimulationState&) : void
+ getBikeState() : BikeState
+ getBikeStates() : std::array<BikeState>
+ addBike(unsigned int) : void
+ removeBike(unsigned int) : void
+ activateBikeBoost(unsigned int) : void
+ changeBikeDirection(unsigned int) : void
- determineOutcome() : void
- configureBikeSide(Bike&) : void
- moveBike(Bike&) : void
...
0..*
Bike
- state : BikeState
+ getID() : unsigned int
+ setID(unsigned int) : void
...
Grid
- cells : std::array<CellValue>
+ reset() : void
+ clearCell(Vector2i&) : void
+ clearCellRange(std::vector<Vector2i>&) : void
+ getCellValue(Vector2i&) : CellValue
+ setCellValue(Vector2i&, CellValue) : void
+ getCells() : std::array<CellValue>
+ overwriteAllCells(std::array<CellValue>&) : void
+ overwriteCellRange(std::vector<Vector2i>&, CellValue) : void

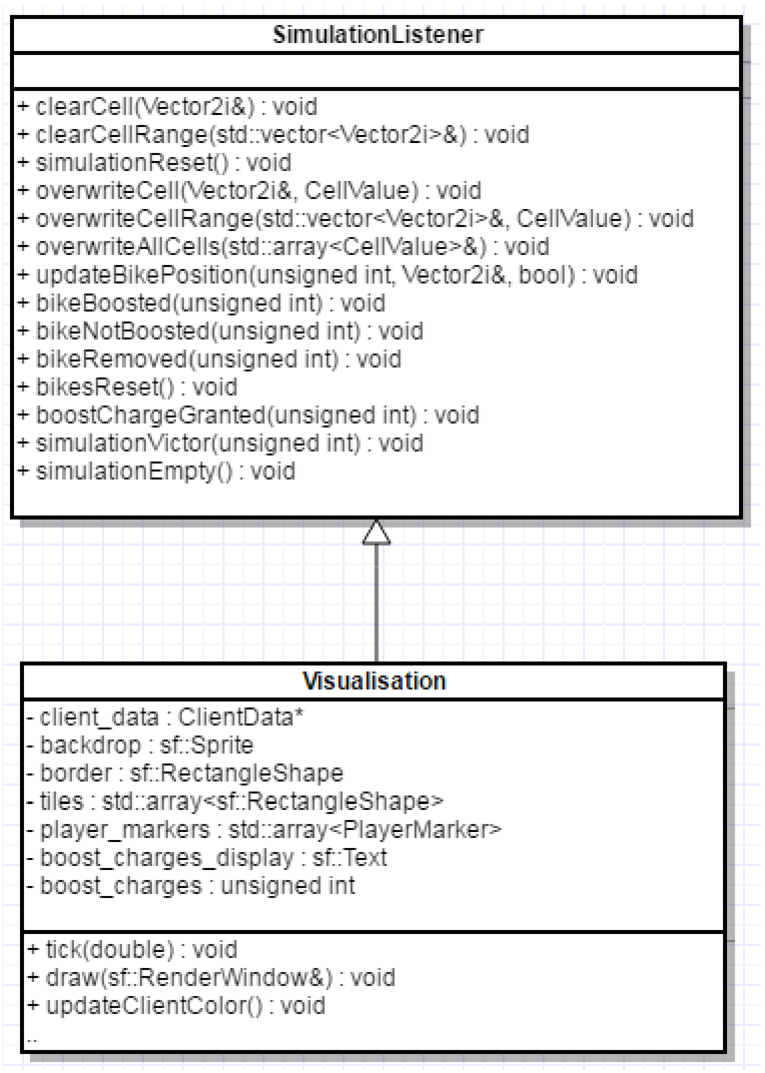
Simulation is the class where the majority of the gameplay takes place. The simulation is generic in that no graphics are displayed as part of it, it is purely a number simulation. The server runs its own version of the simulation via its SimulationThread, and the client runs the simulation on its end via GameManager.

Simulation implements abstract class INetworkSimulation, which is used by areas of the TronClient that are only concerned with using the simulation's network interface, such as for sending game events based on user input.

Simulation also inherits from the templated ListenerSubject class of type SimulationListener. Throughout the simulation the class sends out various events to any listener that may be attached. Any class that then derives from SimulationListener and is attached to the simulation will be able to react to those events.

The client's Visualisation class is a great example of this. The server's SimulationThread also acts as a SimulationListener in order to send information about the server's simulation across the network to the clients.

VISUALISATION

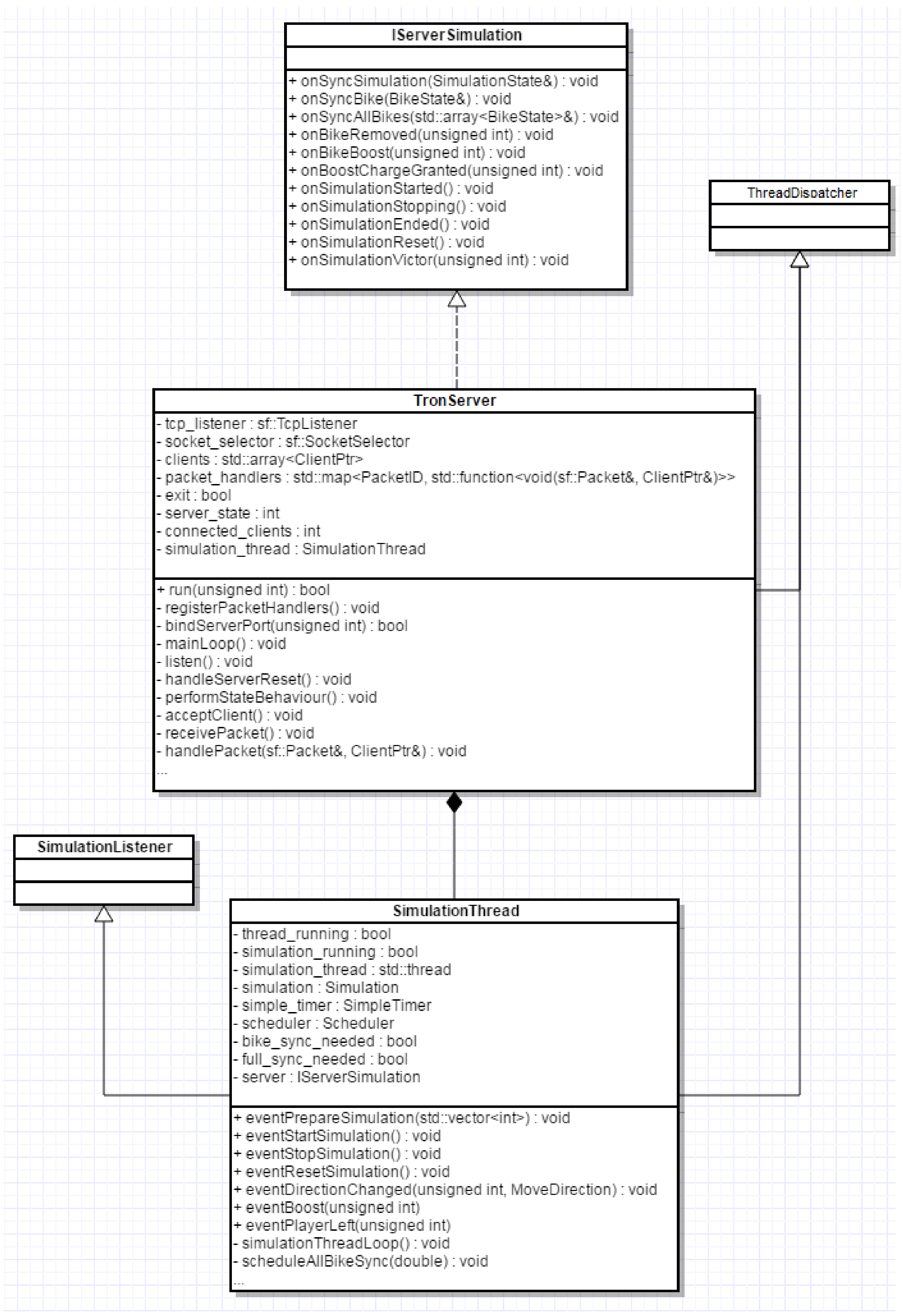


```
SimulationListener

+ clearCell(Vector2i&) : void
+ clearCellRange(std::vector<Vector2i>&) : void
+ simulationReset() : void
+ overwriteCell(Vector2i&, CellValue) : void
+ overwriteCellRange(std::vector<Vector2i>&, CellValue) : void
+ overwriteAllCells(std::array<CellValue>&) : void
+ updateBikePosition(unsigned int, Vector2i&, bool) : void
+ bikeBoosted(unsigned int) : void
+ bikeNotBoosted(unsigned int) : void
+ bikeRemoved(unsigned int) : void
+ bikesReset() : void
+ boostChargeGranted(unsigned int) : void
+ simulationVictor(unsigned int) : void
+ simulationEmpty() : void

Visualisation
- client_data : ClientData*
- backdrop : sf::Sprite
- border : sf::RectangleShape
- tiles : std::array<sf::RectangleShape>
- player_markers : std::array<PlayerMarker>
- boost_charges_display : sf::Text
- boost_charges : unsigned int

+ tick(double) : void
+ draw(sf::RenderWindow&) : void
+ updateClientColor() : void
..
```

Visualisation is the client-side graphical representation of the simulation. Visualisation listens to updates from the simulation and updates its visuals accordingly.

Visualisation is also responsible to playing certain sounds during gameplay, such as when a bike boosts or when an additional boost charge is granted.

TRONSERVER


IServerSimulation
+ onSyncSimulation(SimulationState&) : void
+ onSyncBike(BikeState&) : void
+ onSyncAllBikes(std::array<BikeState>&) : void
+ onBikeRemoved(unsigned int) : void
+ onBikeBoost(unsigned int) : void
+ onBoostChargeGranted(unsigned int) : void
+ onSimulationStarted() : void
+ onSimulationStopping() : void
+ onSimulationEnded() : void
+ onSimulationReset() : void
+ onSimulationVictor(unsigned int) : void
ThreadDispatcher
TronServer
- tcp_listener : sf::TcpListener
- socket_selector : sf::SocketSelector
- clients : std::array<ClientPtr>
- packet_handlers : std::map<PacketID, std::function<void(sf::Packet&, ClientPtr&)>>
- exit : bool
- server_state : int
- connected_clients : int
- simulation_thread : SimulationThread
+ run(unsigned int) : bool
- registerPacketHandlers() : void
- bindServerPort(unsigned int) : bool
- mainLoop() : void
- listen() : void
- handleServerReset() : void
- performStateBehaviour() : void
- acceptClient() : void
- receivePacket() : void
- handlePacket(sf::Packet&, ClientPtr&) : void
...
SimulationListener
SimulationThread
- thread_running : bool
- simulation_running : bool
- simulation_thread : std::thread
- simulation : Simulation
- simple_timer : SimpleTimer
- scheduler : Scheduler
- bike_sync_needed : bool
- full_sync_needed : bool
- server : IServerSimulation
+ eventPrepareSimulation(std::vector<int>) : void
+ eventStartSimulation() : void
+ eventStopSimulation() : void
+ eventResetSimulation() : void
+ eventDirectionChanged(unsigned int, MoveDirection) : void
+ eventBoost(unsigned int)
+ eventPlayerLeft(unsigned int)
- simulationThreadLoop() : void
- scheduleAllBikeSync(double) : void
...

TronServer is the main class for all network activity and events on the server side. Similarly to the relationship between TronClient and NetworkManager, the TronServer and SimulationThread classes communicate with one another over the thread boundary via ThreadDispatcher.

In addition to running the simulation, the SimulationThread class also derives from SimulationListener, allowing it to listen in on events from the simulation and post events to TronServer as soon as they happen.

The TronServer state system is fairly primitive compared to TronClient, but the framework is still there for handling the flow of the simulation as well as the different behaviours that can be triggered by clients in each server state.

Whenever a packet is handled by the server that concerns the simulation, it is posted to the Simulation thread where it is evaluated and a response is posted back to TronServer if needed.

POST MORTEM

I can see why networking is said to be something that should be considered very early on in a project's development. It was very challenging to determine exactly what data should be sent over the server to best synchronise the clients, even once a number of stable systems were in place to support networking.

Having the simulation generic and able to be run on both the server and client was both a boon and a drawback. Having it run on the client was useful for simulating the perpetual movement without constant need for the server, but it also confused things somewhat when certain functionality was only intended to be called on the server but it was also accessible by the client, due to them sharing the same class.

From the start of the project, I considered the user experience throughout the program. The entire lobby system and server flow was designed so that the user always had useful information about what was going on with the server at a given moment, and I think this has been achieved, albeit in a basic form. Nonetheless I think the lobby system is a definite strength of my game.

I will admit I drastically underestimated the amount of data I would need to send to fully synchronise the whole simulation with my Tile-based system. At one point I was sending over 15k variables in a single packet, to multiple clients. While this didn't seem to give a noticeable performance hit, it definitely felt inefficient. My simplified approach involved synchronising only the bikes, and I did successfully manage to have it so only unsynchronised parts of the line were sent, rather than the whole bike line each time. I ended up reverting that optimisation however, because I wanted to be able to send the whole bike line, and I couldn't figure out how to have two different ways of loading a bike state into a packet.

As far as gameplay goes, I didn't actually spend that much time balancing the game this time. I settled on a grid size of 125 x 125 fairly early on in development and never felt the need to change it. I tweaked the movement speed of bikes a couple of times but generally it stayed around the same value. I occasionally play-tested the game with a friend and we found it enjoyable, even more so once I added boosting and accompanying sounds.

The boosting mechanic was a definite improvement for this sort of game, where you often need to take risks in order to beat your opponent. The game tends to drone on or not be very exciting without boosting. Since I found it so enjoyable, I implemented replenishing boost charges just to encourage players to use them, and keep the game aggressive and exciting. I think it worked well.