Politecnico di Milano
Scuola di Ingegneria Industriale e dell'Informazione
Computer Science and Engineering
Dipartimento di Elettronica, Informazione e Bioingegneria

Fast communication training through asymmetrical multiplayer videogames

Relatore: Prof. Marco Gribaudo

Tesi di Laurea di:
Jacopo Grandi, Matr. 947077

Anno Accademico 2020/2021

Abstract

Videogames are a powerful learning tool. For instance typing games and rhythm games are genres defined by the skill they require. These games reward players for improving and try to make the experience as fun as possible. The game we developed is focused in training the communication, planning and problem solving skills of the players. The game puts the players under time pressure and requires a significant amount of information to be exchanged quickly. It also forces a half duplex communication protocol emulating a radio, such that only one player may speak at a time.

Sommario

I videogiochi sono un efficace modo per imparare. Per esempio i giochi di digitazione e i giochi di ritmo sono generi definiti dalle abilità che richiedono. Questi giochi ricompensano il giocatore quando migliora e cercano di rendere l'esperienza il più divertente possibile. Il gioco che abbiamo sviluppato è incentrato ad allenare le abilità comunicative, quelle di pianificazione e quelle di risoluzione dei problemi. I giocatori sono spinti da un conto alla rovescia a scambiarsi una grande quantità di informazioni. Il gioco simula una radio half duplex, quindi i giocatori sono forzati a parlare uno alla volta.

Contents

1 Introduction

Failure is an effective way to learn. This is the trial and error approach to problem solving: keep trying until a success or stop trying [22]. Videogames provide an environment where the players can fail more often and reward the players who keep trying and succeed. The environment has a clear goal, such as play the correct notes [2] or type the correct word [10], and a clear definition of a mistake with immediate consequences.

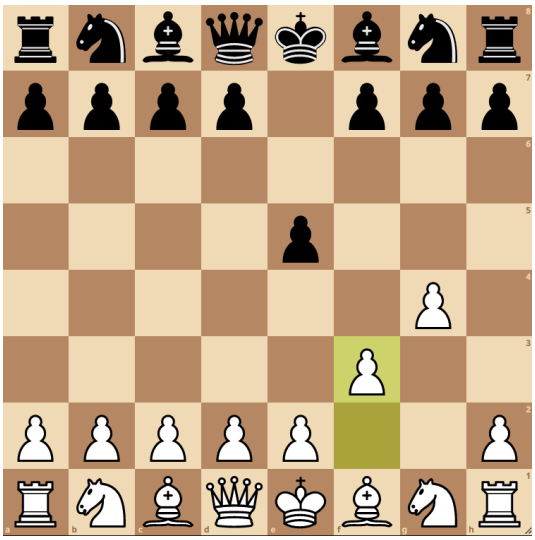


Figure 1: White played f3, a clear mistake as Black has Qh4 checkmate.

Serious games Videogames have been subject of study in educational research and are starting to be used by schools. For instance, Minecraft [13] has an educational edition with the purpose of being a platform for learning chemistry, coding and being a virtual classroom. A serious game can be defined as "Any piece of software that merges a non-entertaining purpose (serious) with a video game structure (game)" [15]. Serious games thus represent a larger set of games than educational games. Skill training games are serious games, often called exergames if they are intended to make the

player do physical exercise. The game we developed is designed to be a serious game, merging the fun of playing a videogame to the tedious learning task.

Skill training The skills we have designed the game to train are problem-solving and communication. Videogames are a good medium to train problem-solving as stated above. Communication, chat or voice chat, is only in multiplayer games. In these games it is rarely used as a core aspect of the game and is just a platform to socialize or banter. Voice chats are often used effectively in competitive team games such as Counterstrike [3] or Forged Alliance [4], but they are not required. We required the use of a voice chat by forcing the players to exchange information that is needed to beat the game. The main difficulty of the game is to get the information from the other players.

1.1 Applicable scenarios

There are situations in which an operation team have to be guided by a remote team. This situations are common in cave exploration, cave rescues, bomb defusal, warfare and even firefighting operations in thick fog. The operation team lacks vital information to act sensibly, information that the remote team can provide. Although, the remote team may need to ask for local data, for instance a description of the operation team surroundings, to remove uncertainties from its model of reality. If the exchange of information is slow and inefficient the operation team may have to take unnecessary risks and possibly lead to a failure, which in the situations described in the examples is catastrophic.

1.2 Asymmetrical multiplayer

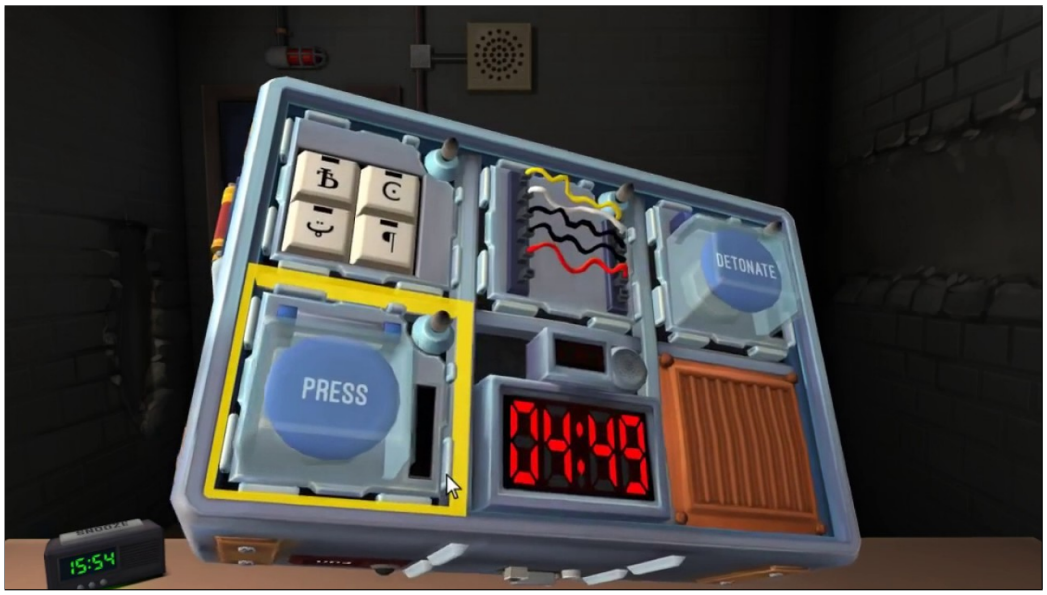
A game has asymmetrical multiplayer when the players play the game differently. There are multiple levels of asymmetry: from a slight imbalance

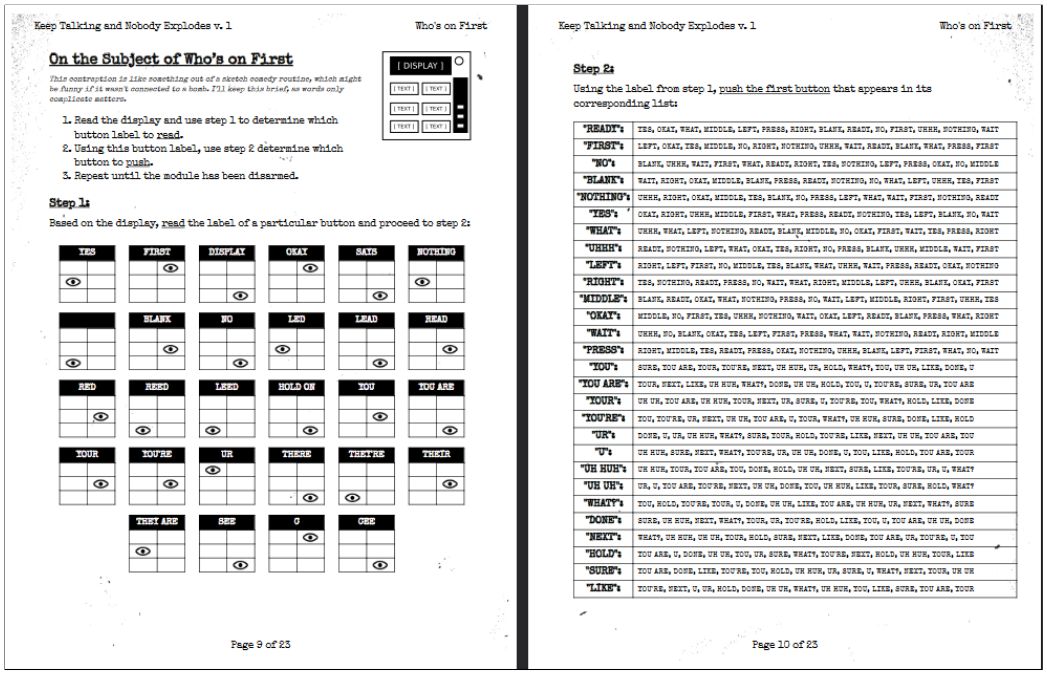
in the mechanics to a completely separate set of rules. In fps (first person shooter) games like Halo [5] the players usually start with the same action set but the map can be asymmetric. Games like Starcraft [9] have three different races to choose from that offer wildly different strategies and gameplay, which is much more asymmetric. Keep Talking and Nobody Explodes is the most asymmetrical game in the cited games: a player defuses a virtual bomb while the other just has a printed manual on how to do it. The game we designed is an asymmetrical game because it's easier to hide information between players. They are separated in different virtual spaces, they can gather information and act differently.

1.3 Similar games

We analyzed more in depth a set of games that inspired the design, are good examples of serious games or feature asymmetrical multiplayer.

Keep Talking and Nobody Explodes [6] The game consists in defusing a ticking bomb by solving puzzles attached to it. It's a two player game, one player is the defuser and the other has a bomb-defusal manual. The players have to exchange the puzzle state and solutions. This game fits the serious game definition. The primary purpose of the game is to entertain, but to be entertained the game requires certain skills by the players.





Keep Talking and Nobody Explodes v. 1 — Who's on First

On the Subject of Who's on First

This contraption is like something out of a sketch comedy routine, which might be funny if it wasn't connected to a bomb. I'll keep this brief, as words only complicate matters.

[DISPLAY]

1. Read the display and use step 1 to determine which button label to read.
2. Using this button label, use step 2 determine which button to push.
3. Repeat until the module has been disarmed.

Step 1:

Based on the display, read the label of a particular button and proceed to step 2:

YES	FIRST	DISPLAY	OKAY	SAYS	NOTHING
BLANK	NO	LED	LEAD	READ	
RED	REED	LEED	HOLD ON	YOU	YOU ARE
YOUR	YOU'RE	UR	THERE	THEY'RE	THEIR
THEY ARE	SEE	C	CEE		

Page 9 of 23

Keep Talking and Nobody Explodes v. 1 — Who's on First

Step 2:

Using the label from step 1, push the first button that appears in its corresponding list:

"READY":	YES, OKAY, WHAT, MIDDLE, LEFT, PRESS, RIGHT, BLANK, READY, NO, FIRST, UHHH, NOTHING, WAIT
"FIRST":	LEFT, OKAY, YES, MIDDLE, NO, RIGHT, NOTHING, UHHH, WAIT, READY, BLANK, WHAT, PRESS, FIRST
"NO":	BLANK, UHHH, WAIT, FIRST, WHAT, READY, RIGHT, YES, NOTHING, LEFT, PRESS, OKAY, NO, MIDDLE
"BLANK":	WAIT, RIGHT, OKAY, MIDDLE, BLANK, PRESS, READY, NOTHING, NO, WHAT, LEFT, UHHH, YES, FIRST
"NOTHING":	UHHH, RIGHT, OKAY, MIDDLE, YES, BLANK, NO, PRESS, LEFT, WHAT, WAIT, FIRST, NOTHING, READY
"YES":	OKAY, RIGHT, UHHH, MIDDLE, FIRST, WHAT, PRESS, READY, NOTHING, YES, LEFT, BLANK, NO, WAIT
"WHAT":	UHHH, WHAT, LEFT, NOTHING, READY, BLANK, MIDDLE, NO, OKAY, FIRST, WAIT, YES, PRESS, RIGHT
"UHHH":	READY, NOTHING, LEFT, WHAT, OKAY, YES, RIGHT, NO, PRESS, BLANK, UHHH, MIDDLE, WAIT, FIRST
"LEFT":	RIGHT, LEFT, FIRST, NO, MIDDLE, YES, BLANK, WHAT, UHHH, WAIT, PRESS, READY, OKAY, NOTHING
"RIGHT":	YES, NOTHING, READY, PRESS, NO, WAIT, WHAT, RIGHT, MIDDLE, LEFT, UHHH, BLANK, OKAY, FIRST
"MIDDLE":	BLANK, READY, OKAY, WHAT, NOTHING, PRESS, NO, WAIT, LEFT, MIDDLE, RIGHT, FIRST, UHHH, YES
"OKAY":	MIDDLE, NO, FIRST, YES, UHHH, NOTHING, WAIT, OKAY, LEFT, READY, BLANK, PRESS, WHAT, RIGHT
"WAIT":	UHHH, NO, BLANK, OKAY, YES, LEFT, FIRST, PRESS, WHAT, WAIT, NOTHING, READY, RIGHT, MIDDLE
"PRESS":	RIGHT, MIDDLE, YES, READY, PRESS, OKAY, NOTHING, UHHH, BLANK, LEFT, FIRST, WHAT, NO, WAIT
"YOU":	SURE, YOU ARE, YOUR, YOU'RE, NEXT, UH HUH, UR, HOLD, WHAT?, YOU, UH UH, LIKE, DONE, U
"YOU ARE":	YOUR, NEXT, LIKE, UH HUH, WHAT?, DONE, UH UH, HOLD, YOU, U, YOU'RE, SURE, UR, YOU ARE
"YOUR":	UH UH, YOU ARE, UH HUH, YOUR, NEXT, UR, SURE, U, YOU'RE, YOU, WHAT?, HOLD, LIKE, DONE
"YOU'RE":	YOU, YOU'RE, UR, NEXT, UH UH, YOU ARE, U, YOUR, WHAT?, UH HUH, SURE, DONE, LIKE, HOLD
"UR":	DONE, U, UR, UH HUH, WHAT?, SURE, YOUR, HOLD, YOU'RE, LIKE, NEXT, UH UH, YOU ARE, YOU
"U":	UH HUH, SURE, NEXT, WHAT?, YOU'RE, UR, UH UH, DONE, U, YOU, LIKE, HOLD, YOU ARE, YOUR
"UH HUH":	UH HUH, YOUR, YOU ARE, YOU, DONE, HOLD, UH UH, NEXT, SURE, LIKE, YOU'RE, UR, U, WHAT?
"UH UH":	UR, U, YOU ARE, YOU'RE, NEXT, UH UH, DONE, YOU, UH HUH, LIKE, YOUR, SURE, HOLD, WHAT?
"WHAT":	YOU, HOLD, YOU'RE, YOUR, U, DONE, UH UH, LIKE, YOU ARE, UH HUH, UR, NEXT, WHAT?, SURE
"DONE":	SURE, UH HUH, NEXT, WHAT?, YOUR, UR, YOU'RE, HOLD, LIKE, YOU, U, YOU ARE, UH UH, DONE
"NEXT":	WHAT?, UH HUH, UH UH, YOUR, HOLD, SURE, NEXT, LIKE, DONE, YOU ARE, UR, YOU'RE, U, YOU
"HOLD":	YOU ARE, U, DONE, UH UH, YOU, UR, SURE, WHAT?, YOU'RE, NEXT, HOLD, UH HUH, YOUR, LIKE
"SURE":	YOU ARE, DONE, LIKE, YOU'RE, YOU, HOLD, UH HUH, UR, SURE, U, WHAT?, NEXT, YOUR, UH UH
"LIKE":	YOU'RE, NEXT, U, UR, HOLD, DONE, UH UH, WHAT?, UH HUH, YOU, LIKE, SURE, YOU ARE, YOUR

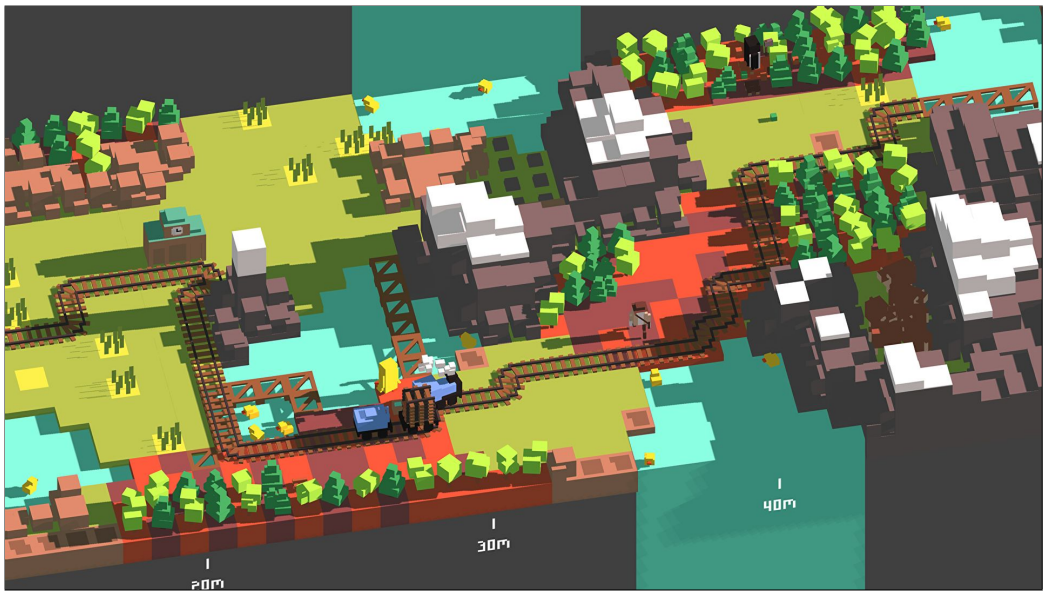
Page 10 of 23

The manual of the game is overcomplicated and explicitly designed to be convoluted. It contains the solutions to the puzzles, but they are obfuscated. Some players have simplified it and memorized it, so they can play without

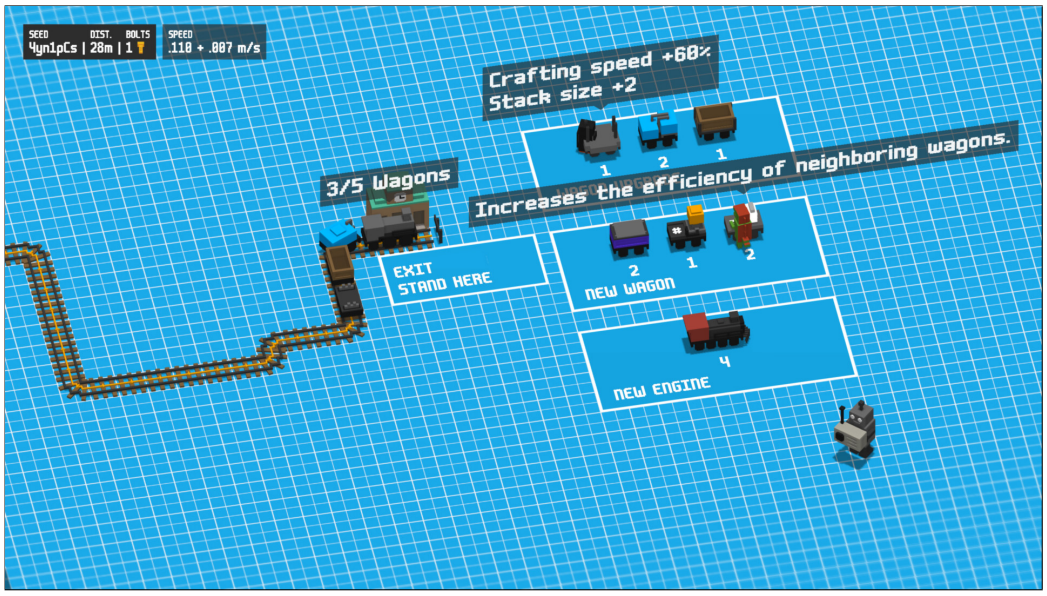
an assistant. This game is the most asymmetrical of the cited games. It's the main source of inspiration to the design of the game we developed. Both players need each other to complete the game, the manual player can't defuse the bomb and the defuser doesn't know how to solve the puzzles without having the manual memorized. This information unbalance is the basis of our game design.

Keep Talking and Nobody Explodes - Elevator of Doom [7] This is an example of how this game trained the players. A group of players of Keep Talking and Nobody Explodes completed a challenge called the "Elevator of Doom", which is a exceptionally difficult level. To finish in the 20 minutes time limit, the players spoke quickly in a condensed and clear manner to exchange the puzzles state and the solution. While the players learned how to beat the game, they also developed their communication skills.

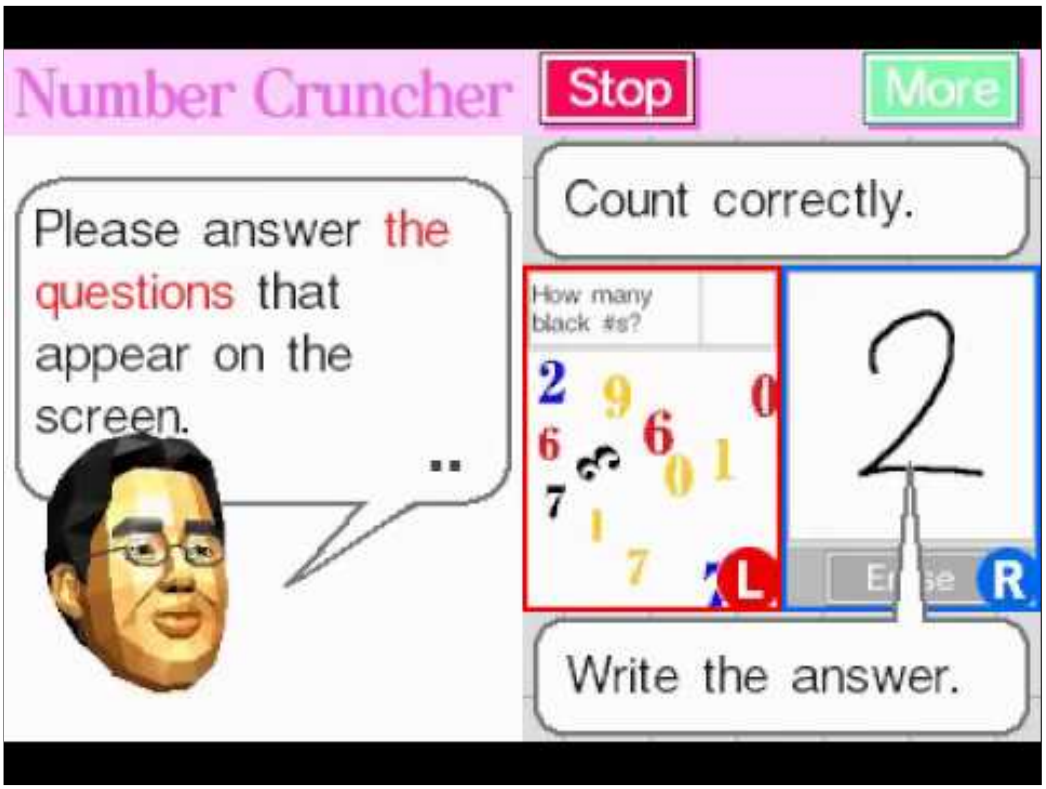
Unrailed [12] The game consists in constructing a railway in front of a moving train. If the trains runs out of track, the game is over. In order to construct rails, the players have to chop trees, mine stone and combine materials in a specialized train cart. It's a 2 to 4 player game. The players usually talk to split tasks among themselves and get time-sensitive tasks done quickly. This game is less of a serious game and is far more geared towards entertainment value: the purpose of the game isn't to learn external knowledge. However, players can use this game as a tool to improve their coordination, communication and reflexes.



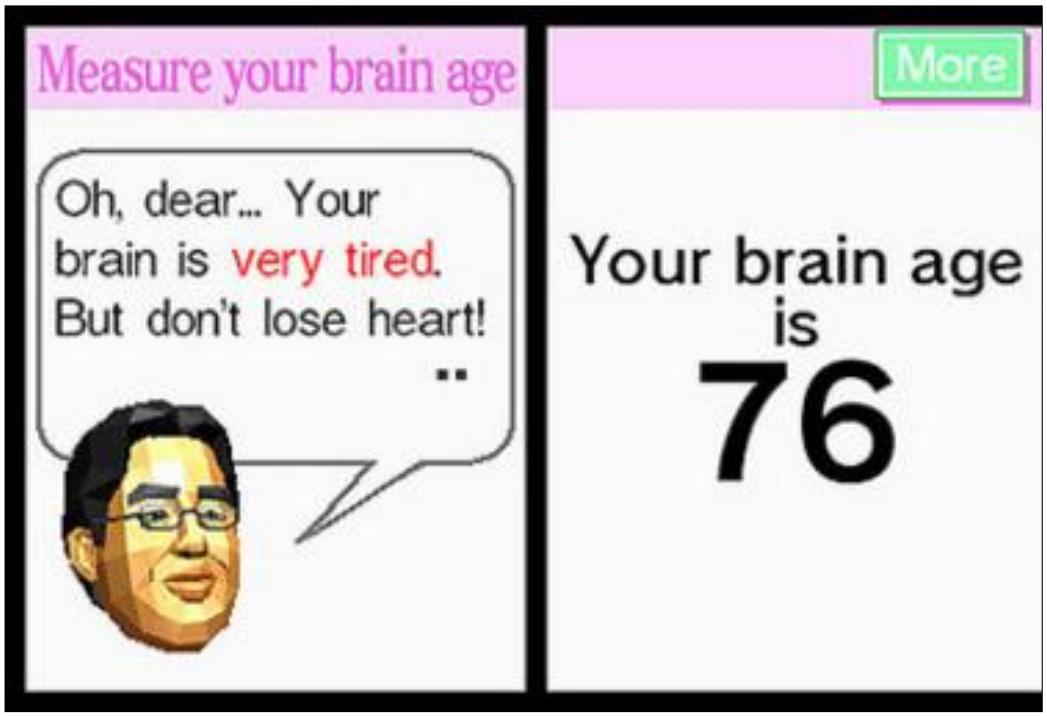
The game includes pickups, scattered in the map and difficult to get. This pickups are used to upgrade the train and wagons. The upgrade screen is cooperative too, as the players can choose the upgrades by physically grabbing them.


SEED
4yn1pCs | 28m | 1
DIST.
BOLTS
SPEED
.110 + .007 m/s
Crafting speed +60%
Stack size +2
1
2
1
Increases the efficiency of neighboring wagons.
3/5 Wagons
EXIT
STAND HERE
2
1
2
NEW WAGON
4
NEW ENGINE

More Brain Training [8] The game is a collection of puzzles that rate the player's mental skills. The game score is the mental age of the corrisponding average individual. The game is a serious game by design, it's purpose is to keep the player's mind sharp.



This game can be categorized as an exergame, term which is usually used for physical activity, for the mental training. However, the puzzles in the game are singleplayer and no speech is involved. The multiplayer aspect of the game is just the highscores, which can lead to competition which is not realtime.


Measure your brain age
Oh, dear... Your brain is very tired. But don't lose heart!
..
More
Your brain age is
76

2 Game design

The game is set in a city, with manhattan grid roads.

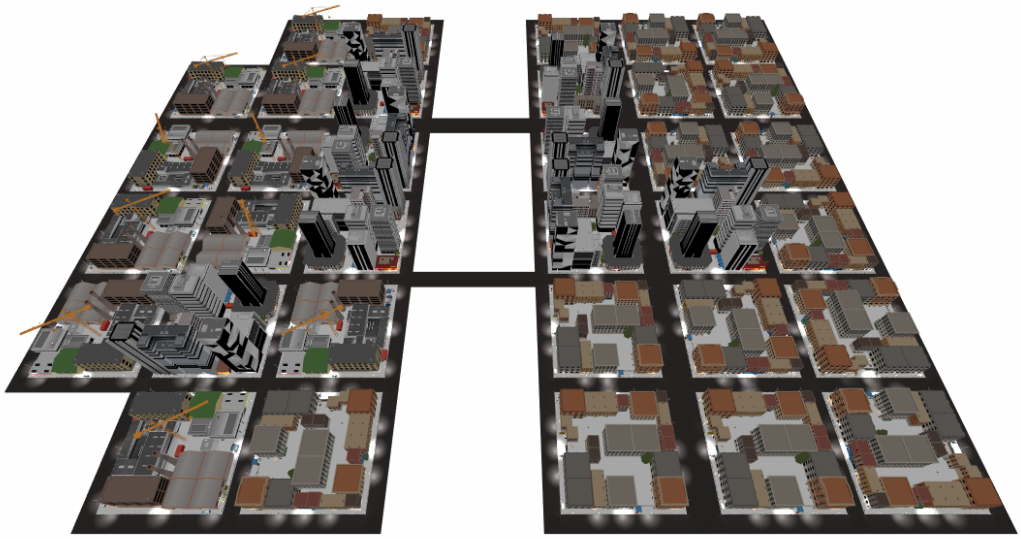


Figure 2: One of the game's maps: Town

There are up to 2 couriers, which have to fulfil a list of delivery orders. There is a master player who has a map, which shows the courier's positions and the street's traffic status. The master has the order list, the couriers do not. The orders consist of a pickup site, a dropoff site and an item to be delivered. The streets are blocked by many obstacles:

- variable level of traffic makes the street difficult to navigate
- a drawbridge over a river
- sidewalk decorations

The master has to plan the route in realtime and relay the path to each courier in the broadcast channel based on the level of traffic. The couriers

drive following the path that the remote master suggests: they only see the pickup point and the dropoff point, they don't know the way from the pickup to the dropoff. The game is lost if the players did not complete all tasks within the time limit, they win if they can complete all tasks.

2.1 Traffic simulation

The cars are agents that move through a graph representation of the streets. The cars roam through the city by turning randomly at intersections. The cars respect traffic lights and occasionally will crash so that traffic jams happen dynamically. Hitting a car will trample the player and push him until the car stops or the player is no longer in front of the car. During this trampling, the players will constantly lose lives.

2.2 Items

The items to be delivered are named and have two attributes: lives and weight. When the player has an item and crashes into a wall, the item is damaged. If the item is too damaged, the task is failed and the item reverted to the start. The players get a few invincibility frames when they lose a life. The weight of an item slows the player down by applying an acceleration penalty and a max speed penalty. The weight is displayed in the game only to the couriers.

Item table

Name	Lives	Weight
water bottle	10	0.9
atomic bomb	4	0.3
money bag	12	0.7
cat	9	1
glass panel	3	0.5
hamburger	6	0.8
box of fireworks	2	0.4
apple	7	1
pizza	10	0.6
chandelier	3	0.4
bowling ball	15	0.1
candy	8	1
pretzel	10	1
bicycle	12	0.4
saxophone	6	0.8

2.3 Master's information

The information that the master has of the players can be reduced in the settings. The master has three channels of information: geolocalization, video feed and audio from the radio. Turning off any of these channels the game becomes significantly harder, except when turning off the radio. The radio is essential for completing the game, so it cannot be switched off. Turning off the video feed and the geolocalization the master has no way of tracking where the players are, so the players have to constantly update the master with their position verbally. This exchange is made difficult by the half duplex radio, which jams if two players speak simultaneously.

2.4 Interface design and menus

The interface has been prototyped directly in Unity. The game is divided in the main menu and the game scenes. The interface is simple and intuitive, only the essential features to connect and play are present.

Figure 3: The main submenu

In the main submenu the player can choose a name and navigate to the lobby, settings and join submenu. [1]

[1]Bicycle model from Tidominer, `https://sketchfab.com/tidominer`

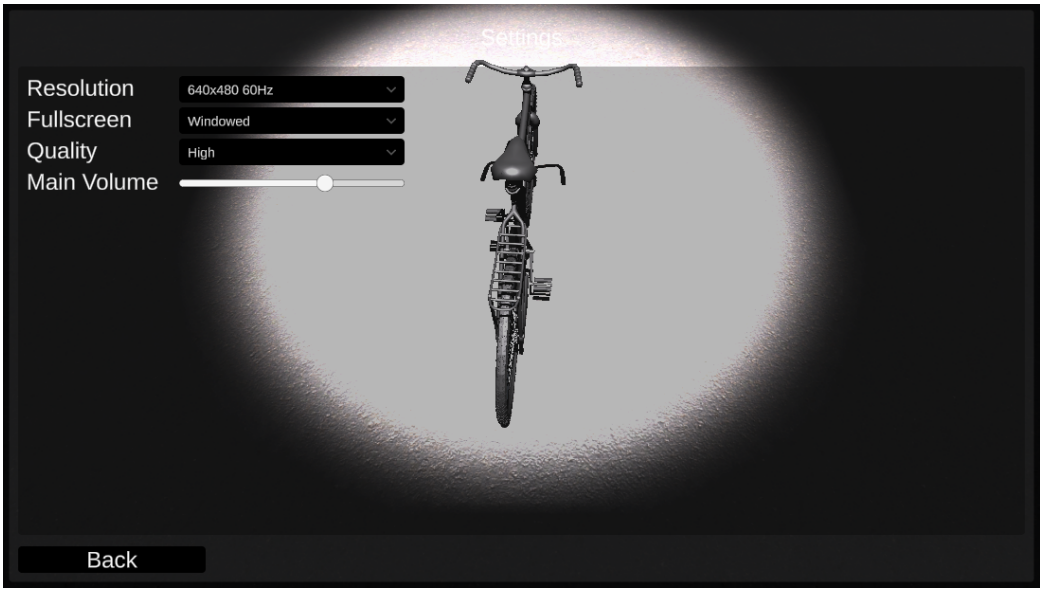


Figure 4: The settings submenu

The settings allows to control the game window, the game's graphic quality and the audio volumes. The menu is constructed dynamically and is easily expandable.

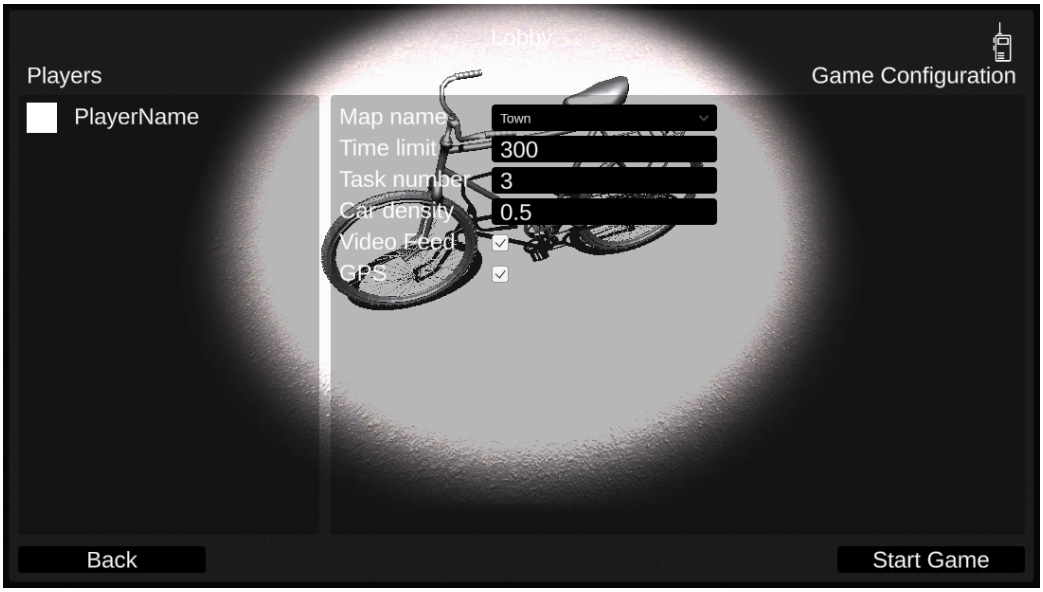


Figure 5: The lobby submenu

The lobby is where the master (which is the host) accepts connections and configures the game. The players can see the configuration and chat vocally. The other players can't change the configuration but they can ask the master to change it. The configuration is constructed line by line with prefabs, allowing it to be quickly customized. Further development may be focused on automating the visualization of the configuration class by using reflections and getting a list of the arguments. The settings are constructed similarly.

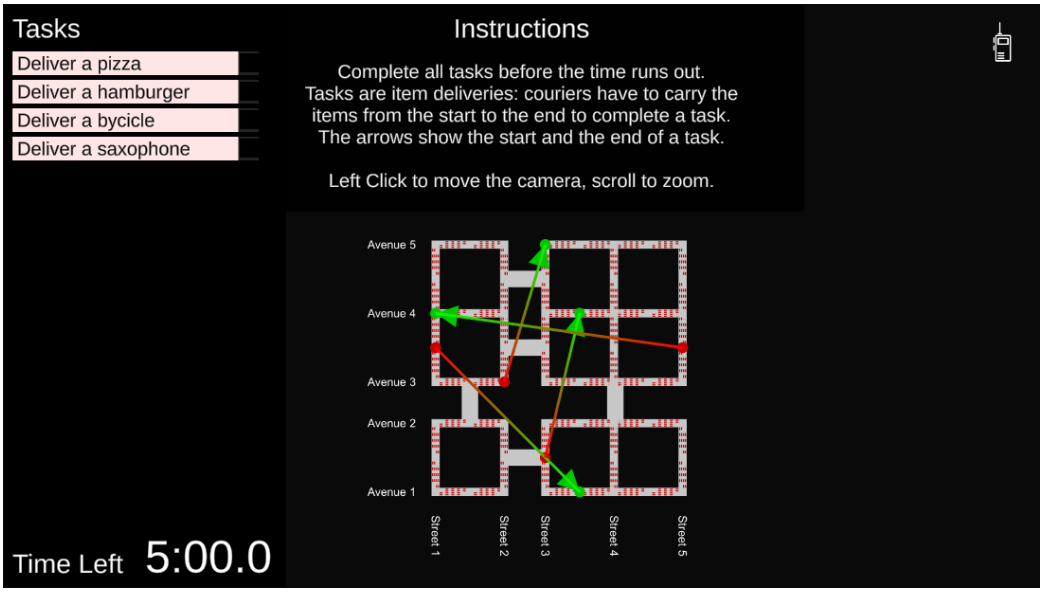


Figure 6: The master interface

The master has at his left the task list and the time left in the game. In the bottom right there are the video feeds of the players and an arrow connects the feed to the corresponding player. The arrows which connects the pickup place and the dropoff of a task can be toggled by clicking on the corresponding task in the list. The task status (none, worked on, completed) changes the color of the task in the list. The master has an instruction panel which can be toggled on and off. The instructions are very brief and are intended to be read by beginners. The master is the only one that has access to instructions.

Figure 7: The player interface

The player can see the carried item name, the item weight and the amount of lives remaining until the item is broken. When a player loses a life an animation plays flashing a white vignette and decrementing the lives counter with a floating -1 label. The lower bar is the only visualization of the weight of the items. If the master wants to know it, he has to ask the player once the players has picked the item up.

Radio visualizer In the top left of every screen (except the main submenu) there is the radio icon. This icon is animated based on the state of the radio connection, if the local player is transmitting and if the channel is jammed by too many people speaking.

3 Implementation

3.1 Level design and 3D models

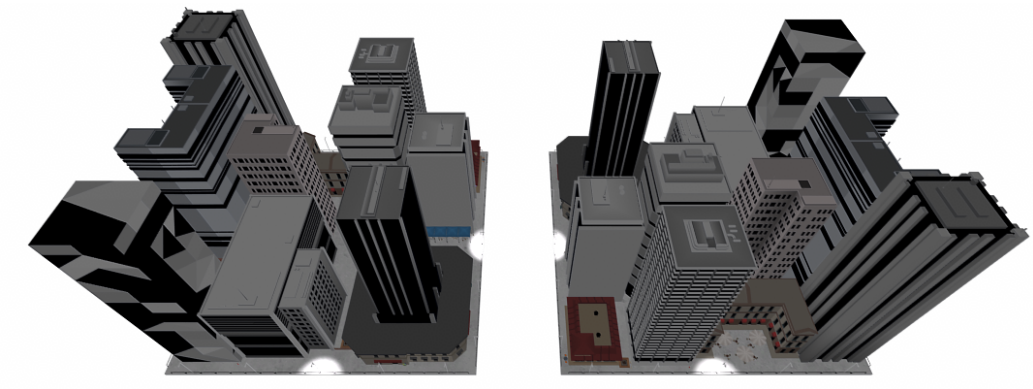


Figure 8: Downtown block

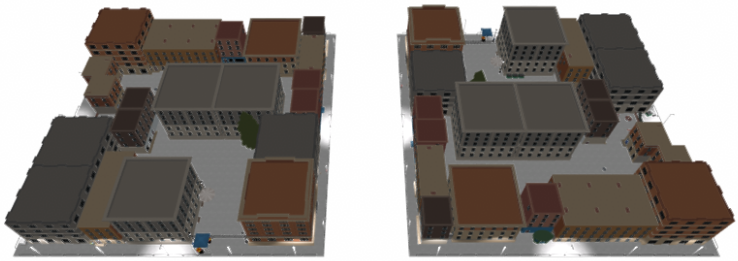


Figure 9: Residential block

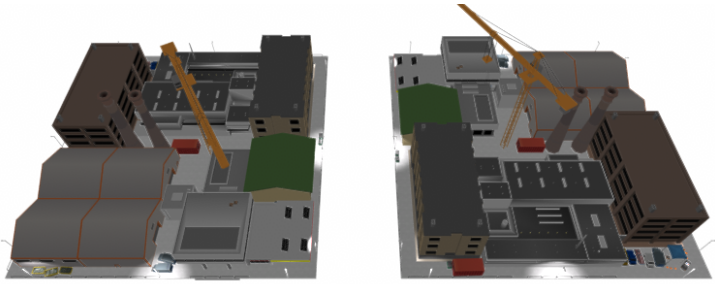


Figure 10: Industrial block

The game maps are constructed by placing blocks, which are made of multiple buildings. Blocks are all contained in a square with a 110m edge. Blocks are assembled using the Everything Library [2]. The blocks are designed by hand focusing on differentiating each side as much as possible from the others so that when they are placed in the map rotated randomly they appear to be unique. The roads are placed between blocks and their width is reflected in the number of lanes, which are then used to place cars at runtime. Signs that indicate the names of the roads are generated following the road layout. Traffic lights and cars are generated at runtime. The bridges are placed in the editor and included in the road layout as special obstacle to track their state. This workflow allows for big maps to be constructed quickly.

Currently, in the game there are three maps: City, Town and Village. The maps are stored in separate scenes, so changes can be affecting only one map. This allows to have different lighting, decorative objects and gameplay elements.

[2]Assets from Everything Library © David OReilly, `https://www.davidoreilly.com/library`

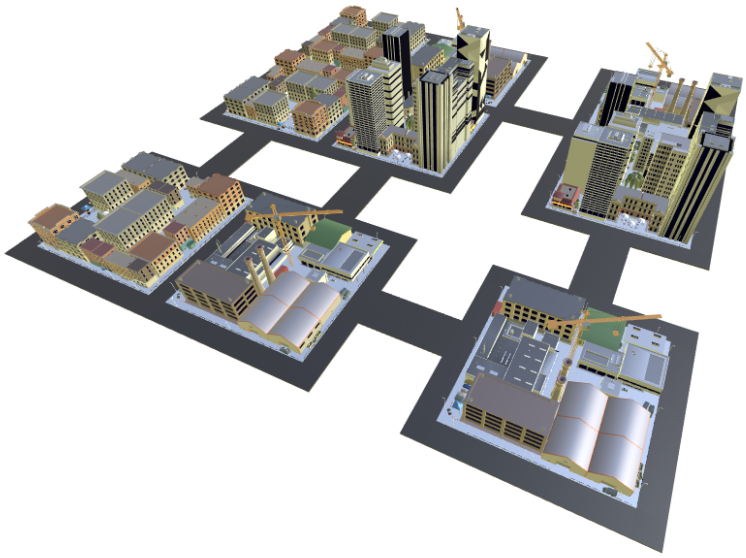


Figure 11: Game map: Village. This map is the smallest and has the most bridges, the map is sunny and with a lot of ambient light.

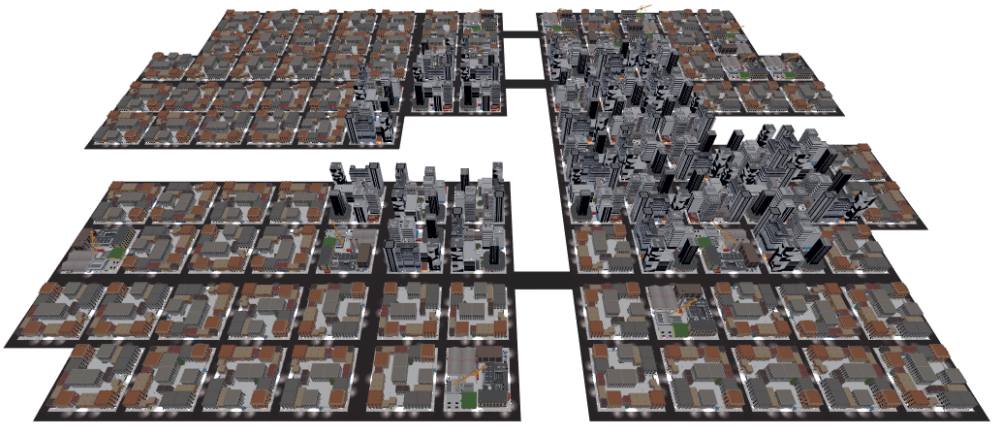


Figure 12: Game map: City. This map is the largest and the most taxing performance-wise. The map is set at night with a lot of traffic lights and bright billboards lighting baked in.

Bridges The bridges in the maps are moving drawbridges that open and close following a timer. The time it takes for a bridge to open and close is set in the editor along with the maximum angle it reaches when opened and it's size. These bridges are one implementation of a more abstract timed obstacle.

3.2 UDP Sockets with fragmentation, optional retransmission and integrity check

The connection is handled using native C# asynchronous UDP sockets. The network can send a message arbitrarily large (It will be fragmented into smaller packets). This system substitutes the deprecated Unity HLAPI (Unity Networking High Level API). Every message has a protocol number, which is used as a multiplexer allowing differently formatted messages to be sent along the same channel. The protocol is as follows:

```
public enum Protocol {
        normal, kill, ack,
        joinreq = 100, syncconf, start,
        masterstate=1000, startgame, ready,
        clientstate, over,
        videoframe,
        mastercars,
        radio=10000
}
```

If a kill is received, the network is shut down. This functionality is used internally to exit from the networking callback cycle. Ack is used to send back a receive confirmation. The joinreq, syncconf and start protocol numbers are used by the lobby controller. The masterstate, startgame, ready, clientstate and over are used during the game. Mastercars is used to syncronize the initial state of the cars from the master to the clients.

Fragmentation If the message is longer than 1024 bytes, it's fragmented in parts no longer than 1024. Each part has the protocol, the serial number and the id along with an offset and a size. The offset is what part of the original is sent and the size is the size of the whole message. At the receiving end, the datagrams received are assembled back into the original messages using the serial number, the offset and size. The partial messages are stored in a dictionary indexed by the serial number and are made available externally only when full. If the fragments arrive not in order it isn't a problem as they are assembled based on their offset.

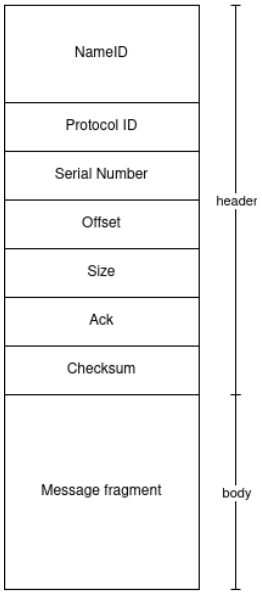


Figure 13: Network packet composition

Optional retransmission If the packet is marked as resend the system will wait for a response (ack) indicating that the packet was received. The system checks periodically if the timeouts of each packet is over. If the ack isn't received and the timeout is over the system will resend the packet with the same serial number.

Integrity check Each packet has a checksum byte to verify that the message has not been corrupted during transit. To generate the byte, the message is first assembled with the byte set to zero. The checksum is then set to 256 - the sum of all bytes. The receiver sums all bytes in the packets: if the sum is zero the message is correct. This technique is known as 1's complement sum [18] and detects any single bit error, but has a lot of collision. A collision happens when the corruption doesn't modify the checksum.

3.3 Game architecture

The game is based on the model-view-controller pattern. [17]

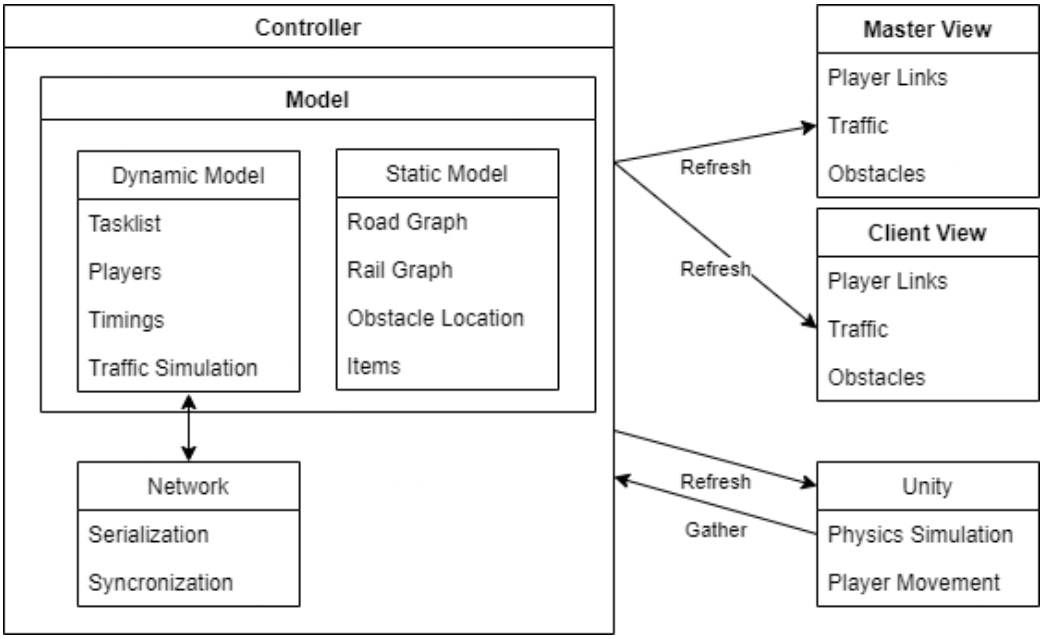


Figure 14: Game architecture diagram

Model A subset of the game rules which constitute the model are implemented decoupling the code from Unity. This subset is everything apart from the physics simulation, player movement and input management. The reason behind the separation from Unity is that the master mode has to be as light as possible. The master mode is meant to be able to run smoothly on slower devices and be ready to be ported to a native browser integration, which is much harder if it relies on some engine features. The model is divided into static and dynamic. The dynamic model is the state that changes during the game and is synchronized from the master to every client. The static model is loaded at the start and contains unchanging data.

View The view is split into master and client. In both instances, it creates a visual representation of the model and links each element to the model with it's identifier. When the model is changed, the view is updated. We implemented two different views: a master view and a client view. Both views are thightly integrated in Unity systems. As the master is meant to be easily ported, the view is the only part to be completely rewritten for the other language and libraries (particularly javascript and webgl).

Controller The controller interfaces with the model, the view, the networking and Unity. The controller behaves differently between client and master. The client gathers player data from and sends it to the master through the network. The master applies the client changes and sends back the updated dynamic model state.

3.4 Synchronization

Multiple instances of a game can be synchronized either by passing only the inputs or the whole state. This approach assumes that the server simulates the game in parallel with respect to the clients.

Inputs synchronization By passing only the inputs, every instance has to update their simulation based on all inputs gathered before the previous update. As the server doesn't have a full simulation of the game (physics and player movement are only in the clients), this approach is unfeasible.

State synchronization The master sends the whole dynamic state to the clients. The clients pass the outcome of the inputs in the form of the player position and a flag if the player happens to collide with the enviroment. This approach is unfeasible if the state is too large to be efficiently sent tens of times a second over the network, but in this game's case the dynamic state is small enough.

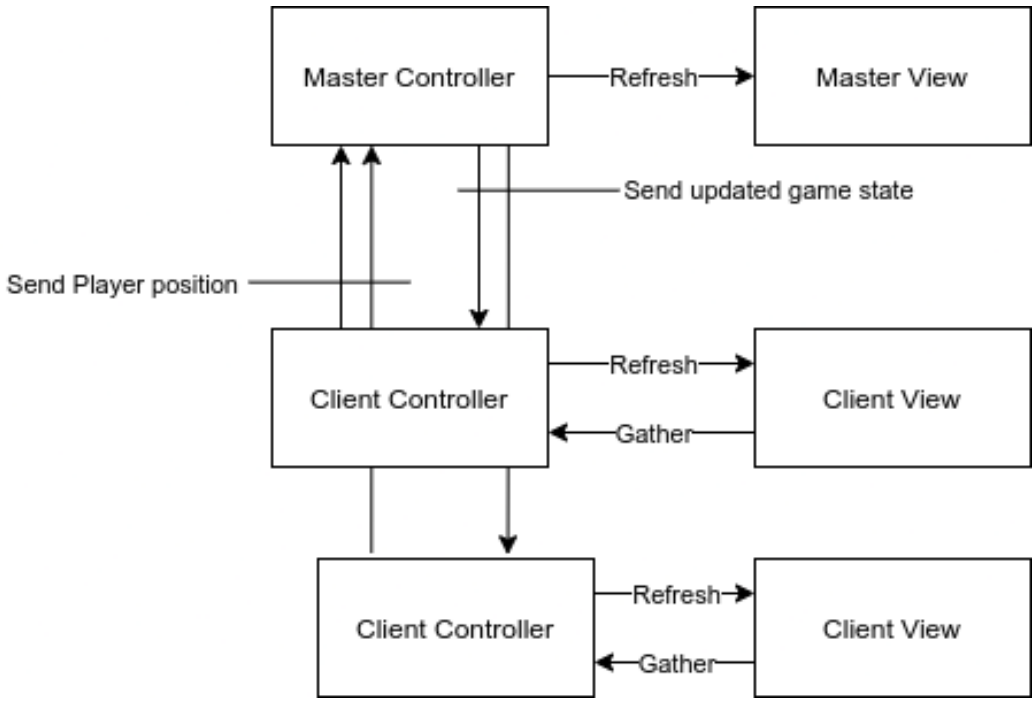


Figure 15: Network Architecture diagram

3.4.1 Dead reckoning and latency hiding

The network layer is unreliable and has variable latency. To mitigate for the delay and the unreliability there are multiple solutions: introduce a fixed delay or rollback.

Delay The fixed delay (usually less than 100 milliseconds) negates the variance in latency if the packets all arrive before the fixed delay. If one packet is late, every instance of the game must wait for the missing packet, ignore it and move forward without or rollback when it arrives.

Rollback When the server receives a late packet, it reverts the game state to the time it was sent and updates it back to the current time. While waiting

for the late packet, the server computes the next states trying to predict the missing packet effects. This extrapolation, also called dead reckoning, can be effective with a trivial implementation when using input synchronization (use the inputs from the previous packet). The game uses state synchronization, so the extrapolation is nontrivial to define.

Extrapolation The extrapolation used in the game involves only the remote players position in the client view. The position is modified by the speed of the players based on how much time has passed since the last packet and the last speed known is kept constant.

Serializer To synchronize the state we wrote a custom serializer that converts objects to bytes and vice versa. The implementation is unit tested and supports only basic types, no containers. To make an object serializable it needs to implement serialize and deserialize. The serializer works as a queue: it's constructed by appending basic types and deconstructed by popping from the bottom of the queue indicating the type of the popped object to interpret the byte array correctly. The serializer can be generalized by writing the serialize and deserialize methods automatically or by using templated methods. The serializer methods are implemented in GameState, TrafficState and LobbyConfiguration. To serialize a list or a dictionary, the size of the collection is pushed and then the elements.

3.4.2 HLAPI [11]

Unity provides the multiplayer high level API which is a system that offers messages, serialization, state synchronization and network classes. HLAPI is deprecated and highly integrated in Unity, which is not desirable as it makes it very difficult to decouple the state of the game from Unity.

3.5 Agent based traffic simulation

3.5.1 Definition and assumptions

The agents of the simulation, cars, are defined by their position relative to a graph, a velocity and other paramters (random acceleration, random seed). This graph called "rail graph" is derived from the street level graph, which is a graph that defines the layout of the roads. The rail graph gets it's name from the cars behaving more like trains or trams, as they can't deviate from the rail graph edges. The rail graph is constructed by replacing every node of the road graph with an intersection and every edge with as many edges as lanes. The road graph has a parameter for every edge indicating how many lanes the street has. The road graph is an undirected graph while the rail graph is directed. The direction of the rail graph edges are determined by calculating if they are on the left or the right side of the road graph edge using cross products. The following figure shows the conversion of 4 way intersections.

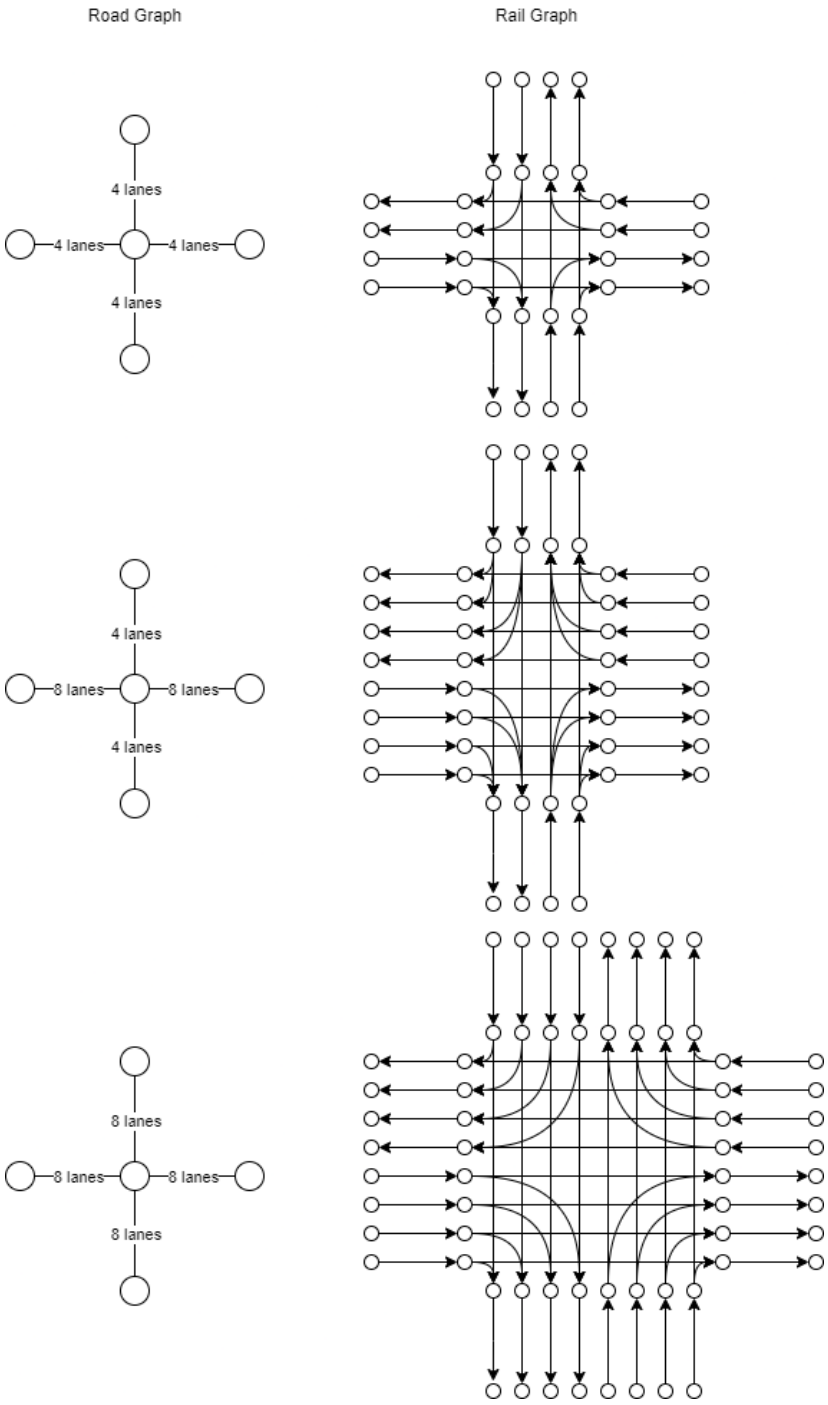


Figure 16: Rail Graph construction from the Road Graph

The rail graph is directed and the direction of an edge is the direction of the flow of traffic. At the intersection the rails are connected by bezier curves. The left turn rails are not connected to not allow cars to turn left, as it causes a jam instantly.

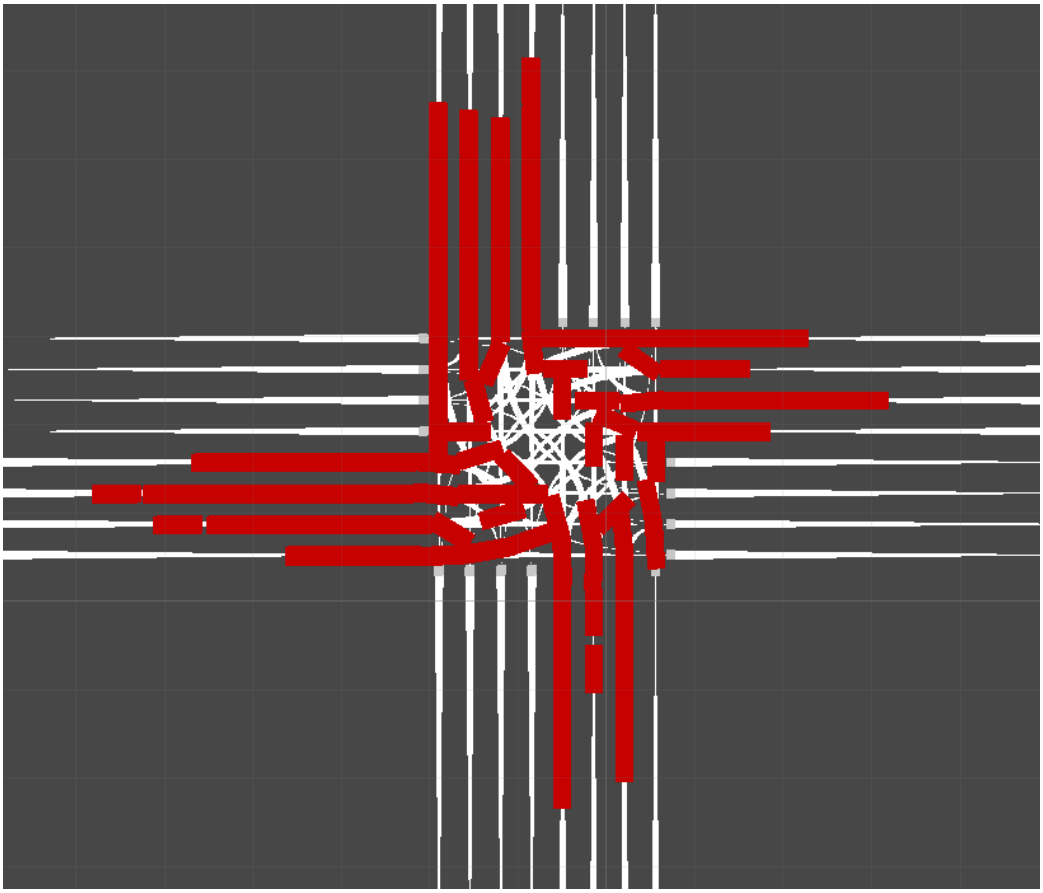


Figure 17: Left turn without traffic lights causes instant jam, this is the state after only 10 steps

To alleviate this jam, the left turning rails would have to be non intersecting and a left turn semaphore would have to be put in place. However, it is not necessary to do so as every point is still reachable without turning left.

Traffic lights Traffic lights are placed on the intersection's inward rails. This lights are coordinated at a street node level, so every traffic lights in an intersection is controlled by a single timer. Based on the position of this timer, the traffic lights states is set. The traffic lights are grouped into odd and even parity: odd parity lights are phase shifted by half a timer cycle. This grouping makes lights switch on and off based on their direction. The grouping algorithm checks if the edge on which the traffic light is placed (if the traffic light is on a node, it considers the edge which is not in the intersection) is closer to the x axis or the z axis (Unity defines the y axis as the up axis). This assumes that the roads are all aligned to the x or z axis, which is the case in the game's maps. The algorithm can be generalized to non aligned roads by modifying the alignment check on a per intersection basis.

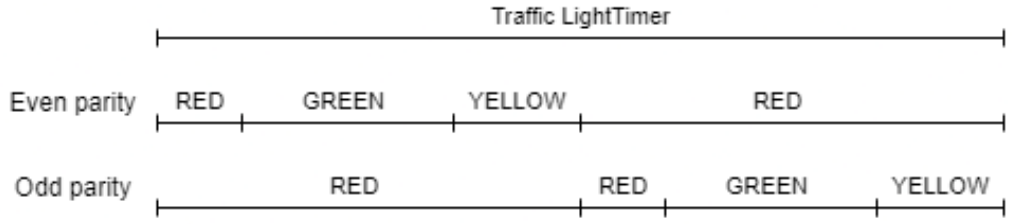


Figure 18: Traffic light timer and states

3.5.2 Implementation

Car-Train agent The car's position is defined by the starting node, the ending node (so the edge on which the car is) and the relative position $rel \in [0, 1]$ along the edge. The absolute position of a car is the position of the car within the frame of reference of the position of the nodes of the rail graph.

To get the absolute position a linear interpolation based on the relative position from the start to the end node is performed. If the edge on which the car is on is a bezier, the position is just the evaluation of the bezier equation

at the car's relative position (see Appendix A for a summary of quadratic bezier curves).
To get the absolute direction the normalized difference of the end and start node is sufficient for the linear case. In the bezier case, the derivative of the bezier equation is the tangent vector, which normalized gives the absolute direction.

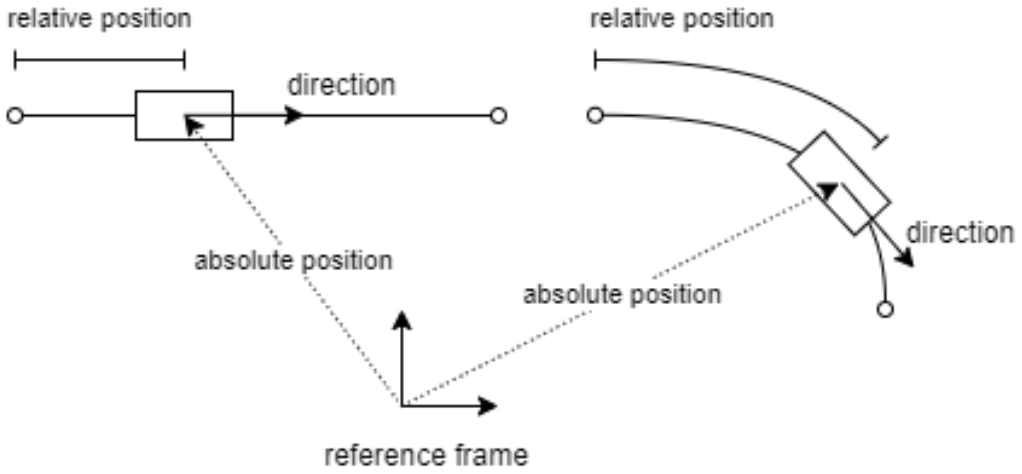


Figure 19: From the relative position of a car to the absolute position and direction. Left: linear, Right: quadratic bezier

Integration The traffic state is a collection of cars and traffic lights. The state is integrated discretely with a time differential dt. Every car calculates how much distance it has to move by using it's velocity. This distance is then consumed by navigating the graph. The car's relative position is incremented by the relative distance, which is the distance divided by the edge length. This is easily calculated for linear edges, but bezier edges in general have a complex elliptical integral to solve to get their lenght [1]. Therefore, the relative distance is calculated using an approximation of the bezier lenght by sampling a few points (in the game its 10 samples, which are also used to display the curves in the editor for debugging).
When a car reaches an intersection, if it has a red semaphore it stops. Other-

wise, it chooses a rail based on it's initial random seed using a deterministic formula, which doesn't compromise the determinstim of the simulation. The formula is

```
next = car.seed % forwardStar(car.node);
car.seed++;
```

The determinism of the whole simulation is important as the simulation will be univocally identified by an initial condition and a time. This eases synchronization as the only state which is passed is the first one and to keep every simulation in sync only the time needs to be known.

3.5.3 Optimization

Collision detection: lookahead dictionary Before the cars are moved, a lookup dictionary is constructed. This lookahead dictionary stores for every car the position it would have had if it moved some integration steps. The number of lookahead steps is a sensitive parameter with respect to number of incidents and runtime speed. Every time a car is moved, it checks if it would intersect any other future cars using the lookahead dictionary. If it would intersect, the car is stopped.
The intersection between two cars is an overlap check on two circles per car that approximate the bounds of a car. Using a box to represent the car bounding box is significantly slower, because it would be a convex polygon check [16].

Lookup optimization and caching Every collection used in the simulation can be accessed randomly in constant time and the absolute position and direction of a car are recalculated only when the relative position is modified. The rail graph has precalculated structures such as an index for accessing edges directly and a forward star collection for every node. The allocations have been reduced by preallocating the lookup dictionary of all the next positions. To calculate the next positions a support object is created that contains the state of the car in order to mutate the state without mutating the original object.

Intersection optimization: Grid indexing The intersection checking is the most costly operation for the simulation. In order to check faster, the cars further than a radius are excluded from the check. This is a coarse filter, but it requires a quadratic number of checks w.r.t. the number of cars. Therefore, an index based on the car's position is implemented. This index divides the space in a grid and for every square it has a collection listing all cars contained in the square. This index can be queried to return a neighbor

of a car, and it does so by returning all cars in the original car square and all adjacent squares.

Parallel integration The integration and collision detection algorithm can operate on a copy of the traffic state (which is a dictionary of cars) and apply the changes at the end of computation thanks to the lookahead dictionary. This makes it possible to parallelize the creation of the lookahead dictionary, the position integration and the collision detection. The threads have access to all the indexed data and traffic state and place their result into thread-safe containers (C# ConcurrentDictionary). The threads work on a subset of cars and can't contaminate each other subset as they are changed at the end of computation. The parallelization is done using Parallel.For, however the for does not loop through all the cars. The cars are split into 8 subsets: these subset are iterated by the Parallel.For. During performance analysis the Parallel.For performed way better when splitting the cars: a possible reason is that the thread pool .Net Parallel is tuned for shorter containers.

Stopped cars linking The cars which are traveling and detect a still car on their path are stopped and linked to the other car. This link is broken when the car detects that the linked car is moving. When a car is linked to another, it's not needed in the lookahead dictionary and can skip being moved.
This linking procedure helps speeding up the intersection check, which is the slowest part of the whole simulation. Although, in the initial state the cars are placed in such a way that no car is linked, which forces the simulation to run slower until some cars are stopped in order for the linking procedure to take effect.

3.5.4 Traffic synchronization

The traffic simulation is synchronized by passing the current step of the master to the clients. The clients will advance their simulations until they reach the same step as the master, then wait until the master step is incremented. The initial state of the traffic model, which is composed of the cars and the traffic lights state, is sent from the master to the clients at the start of the game. The simulation is deterministic by design: the clients all calculate the same next steps from a common initial configuration, so only the current server step number is necessary.

3.6 Kinematic bicycle model [19]

The model is a simplified car steering model, where the front and back wheels are collapsed into one front wheel and one back wheel. The only steering wheel is the front one.
Known constants:

$$l_r = \text{distance from the center to the rear wheel}$$
$$l_f = \text{distance from the center to the front wheel}$$

State variables:

$$[x, y] = \text{absolute position of the bicycle in the plane.}$$
$$v = \text{velocity of the bicycle.}$$
$$\psi = \text{heading angle}$$

Control variables:

$$u_1 = \text{acceleration}$$
$$u_2 = \text{steering angle of the front wheel.}$$

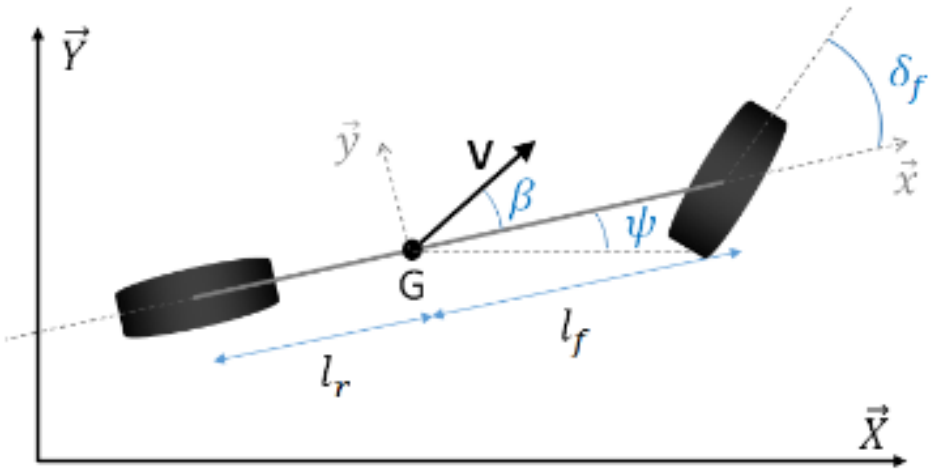


Figure 20: Kinematic bicycle model

The model's differential equations:

$$\dot{x} = v\cos(\psi + \beta(u_2)) \tag{1}$$

$$\dot{y} = v\sin(\psi + \beta(u_2)) \tag{2}$$

$$\dot{v} = u_1 \tag{3}$$

$$\dot{\psi} = \frac{v}{l_r}\sin(\beta(u_2)) \tag{4}$$

where $\beta(u_2)$ is the slip angle given by

$$\beta(u_2) = \arctan(\tan(u_2)\frac{l_r}{l_f + l_r}) \tag{5}$$

The model is integrated discretely.

$$x_{t+1} = x_t + \dot{x}\Delta t \tag{6}$$

$$y_{t+1} = y_t + \dot{y}\Delta t \tag{7}$$

$$\psi_{t+1} = \psi_t + \dot{\psi}\Delta t \tag{8}$$

3.6.1 Player movement

The player is moved following the KBM model. The Unity vertical and horizontal input axis (which contain the standard movement input: arrows, wasd keys and gamepad sticks) are assigned to the control variables. To enforce a soft speed limit and slow down backwards movement the acceleration is modified based on the velocity and a simple friction model is used.

```
float acceleration = Input.GetAxis("Vertical")
    * accelerationSensitivity; // input
acceleration *= 0.2f;
// reduce acceleration as speed increases
acceleration *= Mathf.Exp(-Mathf.Abs(v)*0.1f);
// speed limit when going backwards
if (acceleration < 0 && v > 1)
    acceleration *= Mathf.Sqrt(v);
if (acceleration < 0 && v < 1)
    acceleration *= 0.1f;
v += acceleration * Time.deltaTime;
if (v < 0) v *= 0.99f;
else v *= 0.999f; // friction
```

Player gravity Gravity is handled by incrementing a gravity velocity vector when the player is not on the ground.

```
if (characterController.isGrounded) {
    gravity = new Vector3();
} else {
    gravity += Vector3.down * 0.3f * Time.deltaTime;
}
```

3.7 Game sounds

To make the game more immersive and to make the communication more difficult and strategic, we added audio. The sounds we used are from an online library: Freesound. [3]

Player bike sounds The bike has two sounds: one for acceleration and one for deceleration. The acceleration sound is a sample of a bmx bike constantly pedaling; the volume is proportional to the acceleration amount. The deceleration sound instead is a short sample, cut from a long bike freewheeling. The sample is cut such that it loops seamlessly. The sample volume is set to a constant whenever the bike is not accelerating and the pitch of the sound is modified based on the velocity. This simulates the ratcheting action in the rear bike wheel.

Ambient and traffic sounds The ambient sound is a sample of a calm nightly street and is played constantly in the background at a low volume. The traffic sound is an edited sample of horns and traffic noises. This sample is very loudly played when the player is near to a car and gradually gets quieter the further the player gets from any car. This volume game mechanic forces the players to drive in quiet streets if they have to listen to the master. Moreover, the game audio from the players is picked up and mixed to the radio voice. If the player is in a busy street both the player and the master would have a hard time hearing each other over the traffic noise.

[3]Game sounds provided by Freesound,
`https://freesound.org/people/hisoul`
`https://freesound.org/people/klankbeeld/`
`https://freesound.org/people/14FPanskaBubik_Lukas`

3.8 Radio

The radio in the game is a simulation of a real half-duplex radio. This means that only one player at the time may speak. This is enforced by transmitting white noise while two ore more player are using the radio channel simultaneously. The simulation relies on a server based network to centralize the audio mixing process. All clients send their audio to the server, which mixes all the sources and sends customized mixes to each client and the local speaker. As a convention, all audio passing though the mixer has a sampling rate of 48kHz.

Voice loopback The client audio doesn't loop back to the sender. When the voice returns to the speaker it's delayed, causing confusion. The server processes all audio sources to calculate if noise is needed, then for every client it mixes all sources except the client one.

Buffers The client audio is sent to the server and cached in a buffer in order to account for desynchronizations and network jitters. The server creates and fills a buffer for every client. Periodically, the server polls the buffers and mixes the polled data. The mix is sent to each client where it is cached in a buffer. This buffer is constantly polled by an Unity audio callback. The draining and filling of the buffers is balanced: the microphones produce as many samples as the ones consumed by the speakers, so the intermediate buffers serve as a transfer node.

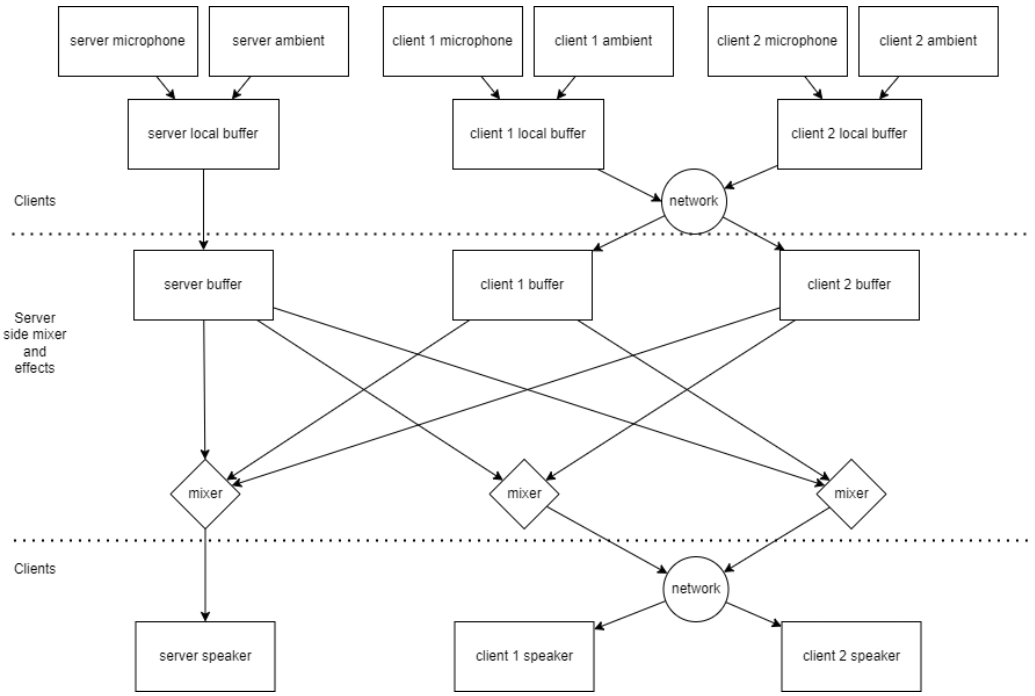


Figure 21: Audio pipeline diagram, the boxes represent the buffers.

Buffer delay Every buffers delays the signal by a set amount. This delay is crucial to alleviate the network variance, which otherwise would cut the signal and override samples. The buffers, when queried to read a certain amount of data, return silence if the data available is less than the requested amount. This ensures that the mixer will always have either a full set of data or silence and that the buffers will not cut the audio.

Sampling frequency and resampling The client's microphone may have a different sampling frequency from each other. The audio sent is resampled to 48 kHz and the audio received is resampled to the speakers sampling frequency. The sampler used is a custom sample and hold algorithm, it inserts or deletes samples following the ratio of sample rates.

Noise The noise generated and mixed when multiple clients are speaking is additive gaussian noise. [20]

```
float[] noise = new float[size];
for (int i = 0; i < size; i++) {
    float u1 = UnityEngine.Random.Range(0, 1f);
    float u2 = UnityEngine.Random.Range(0, 1f);
    float randStdNormal =
        Mathf.Sqrt(-2.0f * Mathf.Log(u1)) *
        Mathf.Sin(2.0f * Mathf.PI * u2);
    noise[i] = randStdNormal * whiteNoiseVolume;
}
```

Which is based on the following equation that samples a gaussian distribution:

$$x = \sqrt{-2 \log u_1} * \sin 2\pi u_2 \tag{9}$$

Filters [21] The audio generated by the clients is filtered by the server with a custom hi pass filter, which attenuates the low frequencies. This effect is done to make the simulation closer to an old style radio.

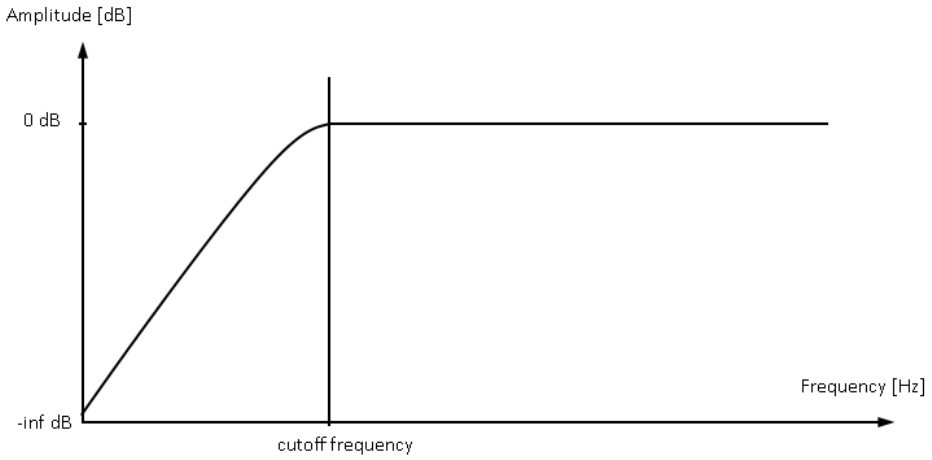


Figure 22: Hi-Pass filter Bode diagram

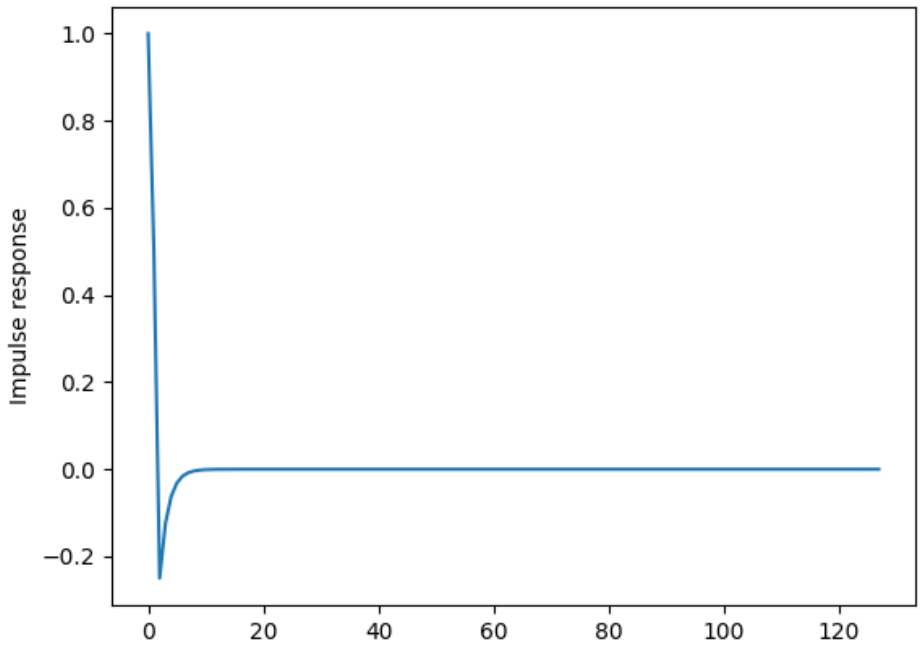


Figure 23: Hi-Pass filter Impulse response

The filter is a recursive digital filter, defined by the following differential equation:

$$y(n) = \alpha(y(n-1) + x(n) - x(n-1)) \tag{10}$$

3.9 Video

The video feed is a frame of the player's camera which is sent to the master by each player. The amount of data required scales rapidly with the resolution of the frame and the frequency of the transmission.

3.10 Editor tools

The development process was sped up by the construction of tools for Unity. This tools are based on editor scripts, which allows to automatize some actions in the editors such as instantiating prefabs (keeping the prefab link) and destroying GameObjects.

Map generator The map generator places prefabs on a grid, randomizes their z rotation snapping on the 4 cardinal directions and swaps these prefabs with their updated ones. This script also places road segments ensuring that no segment overlaps another and giving each segment a unique identifier.

Graph generator This tool generates a graph (a nodes and arcs tuple) from the elements present in the scene. The graph is saved directly in the Assets/Resources folder in json format. The graph can be visualized in the scene with a toggle in the script. The tool also visualizes the traffic rail graph, the cars and the traffic lights state. The traffic simulation can be stepped through directly in the unity editor, which eliminates the need to wait for unity to reset it's state while entering play mode. The tool can also place signs that display the name of the road at each intersection.

4 Testing

4.1 Unit testing

Unit tests are automated tests, they ensure that a small section of code behaves as intended. It's difficult to write testable code and to write tests for all the required functionality if the tests are not included in the design. To make it easier to write a complete test suite (set of tests of a program), a technique called test driven development is used to write code and tests.

4.1.1 Test driven development [14]

The model of the game is developed using the test driven development technique. This technique allows writing testable code by first writing a failing test and then the code to pass it. This makes the code more robust and easier to clean. Other aspect of the game such as the controller, the view or the map tools are not easily testable this way as they depend on the unity framework. One of the goals of TDD is to make it easy to refactor code: if the refactored code passes all tests the code is correct, there's no need to run it to know. This relies on the robustness of the test suite. If an essential functionality is not tested, the test suite is incomplete and by running all tests there is no guarantee of correctness.

4.2 Integration testing

Clumsy is a program that makes the internet network worse. It simulates packet loss, jitter, high latency, small bandwith, packet tampering. This tool is the basis on the game's retransmission and fragmentation algorithm, because it creates the perfect network conditions for testing. The packet tampering option highlighted the need to implement an integrity check for messages, which is necessary if the application is deployed on the web.

4.3 System test and feedback

The game as a whole was tested by independent testers. The testers rotated roles (courier and master), there was no more than one courier at a time and they were all set in the map "Village". The results are summed up in the following table:

Time	Tasks	Video feed	GPS	Completed	Time left
3 minutes	5	yes	yes	no	0
3 minutes	4	yes	yes	no	0
3 minutes	3	yes	yes	yes	26 seconds
5 minutes	3	yes	yes	yes	131 seconds
5 minutes	3	no	yes	yes	95 seconds
5 minutes	3	yes	no	yes	10 seconds
5 minutes	3	no	no	no	0
8 minutes	3	no	no	no	0
8 minutes	3	no	no	yes	39 seconds
8 minutes	4	no	no	yes	154 seconds
8 minutes	5	no	no	yes	112 seconds

The testers started with a simple scenario and little time. They learned how to play (although they pointed out that the game doesn't explain itself properly) and were able to complete the game in under 3 minutes in just 3 tries. The game seems far more difficult when disabling both the video feed and the player's location (GPS). From that moment on, the testers had a much harder time speaking. The master had to rely on the courier telling him were he was otherwise he would have no indication of his position. The testers found the signs of the street names and were starting to learn the map, so they were able to beat the game consistently.

5 Conclusion

In this work we analyzed the usefulness of videogames as a learning tool. We designed and implemented a multiplayer videogame with an integrated radio that forces a rapid exchange of information between players to win. We developed a substitute for the default Unity Networking system (HLAPI) and a custom serializer that support large messages and is stable over adverse network condition. We developed a toolchain to easily produce maps from the Unity Editor. We used test driven development to implement the game rules. We designed and implemented a graph conversion algorithm and a parallel traffic simulation.

The system tests results show a significant improvement of the player's performance in the game.

For what concerns future development we will focus on the following:

- Addition of environmental audio to make the traffic a more effecive obstacle.
- Addition of other obstacles such as railway level passages and pedestrians.
- Improvement of the traffic simulation by making it react to the players and use the current one as a guide for free roaming agents. The rail system can be used as an idealized model of how the cars should move about, each rail agent corresponds to an actual free roaming agent that can solve local navigation and obstacle avoidance.
- Improvement of the traffic simulation by porting the parallelized section to a compute shader. This will enable bigger maps to be feasibly handled.

- Port of the master code in javascript to deploy the master version in a browser. This step has been considered during design and the codebase has been written to make such a port possible.

A Quadratic bezier curves

A bezier curve is defined by the set of control points. The curve starts at the first control point and ends at the last. The other control points are used to shape the curve and are generally not contained in the curve.

A linear bezier curve has two control points and is just a linear interpolation between the control points.
A quadratic bezier curve has three control points $[a, b, c]$ start, control and end. The curve can be defined recursively as an interpolation of interpolations.

Let $f_1(t) = at + b(1-t)$ and $f_2(t) = bt + c(1-t)$ the linear interpolations from a to c and from b to c, $0 <= t <= 1$. The bezier curve from a to c is $b(t) = f_1(t)t + f_2(t)(1-t)$

$$b(t) = at^2 + 2b(1-t)t + c(1-t)^2 \tag{11}$$

The derivative of this function is the velocity vector, which normalized is the direction vector.

$$b'(t) = 2(1-t)(a-b) + 2t(b-c) \tag{12}$$

The lenght of a quadratic bezier is difficult to calculate as it is defined as an integral of the absolute derivative. A simple way to calculate the length is to sample the curve at regularly spaced intervals and sum the lenghts of the segments obtained. This method is fast, but it needs a lot of intervals if the maximum curvature of the curve is large.

B Source code

The source code of the videogame developed for this thesis and this document are publicly available on Github.

Code: `https://github.com/jacopograndi/radio`
Document: `https://github.com/jacopograndi/thesis`

References

[1] Bezier info, an online book on bezier curves. https://pomax.github.io/bezierinfo/l.

[2] Clone hero, a free rhythm game. https://clonehero.net/.

[3] Counterstrike, an fps game franchise. https://blog.counter-strike.net/.

[4] Forged alliance forever, a community driven project based on supreme commander: Forged alliance. https://www.faforever.com/.

[5] Halo franchise. https://www.halowaypoint.com/.

[6] Keep talking and nobody explodes. https://keeptalkinggame.com/.

[7] Keep talking and nobody explodes, the elevator of doom by lthummus. https://www.youtube.com/watch?v=y9zGRy1bl1A.

[8] More brain training. https://www.nintendo.it/Giochi/Nintendo-Switch/Brain-Training-del-Dr-Kawashima-per-Nintendo-Switch-1656777.html.

[9] Starcraft franchise. https://starcraft.com/.

[10] Typeracer, an online typing multiplayer videogame. https://play.typeracer.com/.

[11] Unity high level api documentation. https://docs.unity3d.com/Manual/UNetUsingHLAPI.html.

[12] Unrailed. https://unrailed-game.com/.

[13] Omar Alawajee and Jonathan Delafield-Butt. Minecraft in education benefits learning and social engagement. *International Journal of Game-Based Learning*, 11:19–56, 10 2021.

[14] Kent Beck. *Test-Driven Development: By Example*. Addison-Wesley, 2002.

[15] Damien Djaouti, Julian Alvarez, and Jean-Pierre Jessel. Classifying serious games: the g/p/s model. *Handbook of Research on Improving Learning and Motivation through Educational Games: Multidisciplinary Approaches*, 01 2011.

[16] James D. Foley, Andries van Dam, Steven K. Feiner, and John F. Hughes. *Computer Graphics: Principles and Practice (2nd Ed.)*. Addison-Wesley Longman Publishing Co., Inc., USA, 1990.

[17] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1994.

[18] Changli Jiao and L. Schwiebert. Error masking probability of 1's complement checksums. In *Proceedings Tenth International Conference on Computer Communications and Networks (Cat. No.01EX495)*, pages 505–510, 2001.

[19] Philip Polack, Florent Altché, Brigitte Novel, and Arnaud de La Fortelle. The kinematic bicycle model: A consistent model for planning feasible trajectories for autonomous vehicles? pages 812–818, 06 2017.

[20] B. D. Ripley. Computer generation of random variables: A tutorial. *International Statistical Review / Revue Internationale de Statistique*, 51(3):301–319, 1983.

[21] Julius O. Smith. *Introduction to Digital Filters with Audio Applications.* W3K Publishing, 2007.

[22] Campbell Donald T. *Blind variation and selective retention in creative thought as in other knowledge processes.* 1960.