Networking implementation & optimizations for a VR-AR game with mocap, VoIP and gameplay data

Graduation report

Aldo Leka

Q1-2 2018

Creative Media and Game Technologies

Saxion University of Applied Sciences

Enschede, The Netherlands

Abstract

This paper deals with the networking implementation and optimizations for a VR and AR game with mocap suit, voice over IP and gameplay data like player information, lobby, etc. The logic for implementing these systems in the game is studied.

First, there is a general description of the game so the reader can understand more easily what type of data is expected to be transferred over the network.

Some techniques for compressing mocap data like deflate algorithm and delta compression are studied and tested. The first one is a combination of huffman coding and LZ77 algorithm. Delta compression is a custom made implementation for collecting and sending mocap data that have changed by a given threshold. If threshold is set to 0 this technique is lossless. If it's set to higher than 0 it is lossy.

Then Dissonance is studied and tested for implementing VoIP in the game. Implementation details follow. A lobby is implemented based on a choice from networking options and architecture. A players dictionary is implemented in the server and the clients where the keys are the IDs created by the server upon connecting to a client which are in turn broadcasted. The values are a Player structure with various data like class, name, ready state, spawn position etc.

Then gameplay data is networked including player's spawn positions, and player spawning is implemented at the server and the clients. Loading the game world at the server and at the clients is handled. After the world has loaded at the server, clients fetch tile data from the server at a specific rate (like three times per second) based on their position in the world. Using some hexagonal grid functions it's possible to retrieve the tile where the player is positioned in the server and get the tiles that are in a specific range like six, and send the total amount of tiles to all clients with the ID of the recipient client. The client which has the same ID as the packet ID processes the information and renders its world.

Following are some implementation details on how player movement works with running pathfinding at the server side since the server computer has much more processing power and better specifications than the mobile devices and can better handle player navigation and pathfinding in the game world.

Finally a test is conducted where all these streaming data like mocap, VoIP, tile and movement data is transmitted over the network to see whether the server can handle the network load.

In the end conclusions are derived from the tests and implementation results.

Preface

During this project I had tremendous fun and I loved most of it. I want to thank the teachers at Saxion for making the study exciting and interesting through all this time. I'm grateful to their assistance, especially to my supervisor, Paul Bonsma, with the weekly or biweekly meetings, timely feedback and helpful attitude during the graduation period. I'm also thankful to my supervisor from the company, Robin Kuiper, who was friendly during the whole time and helped me to clarify in my head of what is expected from me during the graduation.

I have to thank Taco van Loon for suggesting me to look into Minor Immersive Media topics to find a graduation spot and got me in touch with Matthijs van Veen who in the end helped me secure a graduation at Twinsense which I preferred the most.

I want to thank my team members, some of whom are close friends by now. At any moment that I needed help from them, they were ready to supply. I'm very thankful to each and everyone of them.

Finally, I want to acknowledge and thank my family who have supported me emotionally and also financially throughout all this time.

I'm proud that I made it this far. I couldn't have done it by myself.

Table of contents

1. Introduction

Project Tabletop RPG with VR & AR is about playing classic Dungeon and Dragons using a VR headset to create the game world that players using AR on their mobile phones can see and interact with. This project has brought together 15 Graduation, Saxion Smart Solutions, and Minor Immersive Media students. The client is Twinsense, an interactive media company based in Enschede. They are doing this project to show what's possible to achieve with the latest technologies in VR and AR via a game which is an interactive media that they never tried making before.

More information on the development on the game can be found on the blog below:

https://tabletoprpgblog.wordpress.com/

In this graduation paper, networking solutions are implemented for realizing the multiplayer aspect of the tabletop VR/AR game using Unity engine. The applications include:

- Mocap data networking and optimizing
- VoIP
- Having a networking solution for a game involving a VR computer application and AR mobile applications.
- Having basic multiplayer components like a lobby, player data synchronization etc.

2. General description

2.1. Graduation assignment

Using VR, the game master (or dungeon master, DM, VR player, server which in this context mean the same thing) can create a world and story for the other players to experience in AR using a hexagonal grid and hexagonal tiles. During the game each AR player (or client) controls their own character to interact with the world, while the gamemaster controls the world and determines the consequences of the player actions.

The gameplay of Project Tabletop can be divided into two segments, the first one being the campaign creation. This segment would only be for the DM who can create campaign using VR. The second segment would be about playing this campaign. In this case the DM still controls and changes the game, but now the AR players are also involved, playing through the campaign created by the DM.

After the dungeon master has created the campaign, the game can begin. The game is based on rounds, every AR (mobile) player controls their own character, which has a certain amount of energy. The players can queue up actions that consume energy (Figure 1 and 2). The actions are executed in the order in which they were added to the queue. Once all the players have used up their energy, or passed, the dungeon master can start the next round when they are ready.

The VR player transmits his movements in real time to AR players on their mobile phones using a mocap suit. This serves to make the experience more immersive. By moving (mocap) and talking (VoIP) the game master can better guide the players throughout the story. This is done via controlling the movements of:

- a giant in the game (Figure 3) who plays the role of the game master and
- various NPCs that players meet in the game world.

This paper's interpretation of the assignment handed in by the company concerns researching and implementing the means of transmitting data (player, mocap, voice over IP, game state, etc.) between the VR player in a computer and AR players in mobile phones.

2.2. Company outline

Twinsense B.V., online brand - was founded in 2005 and is a specialist in the field of online communication. They help their clients by doing virtual reality films, interactive (online) video or 360-degree films.

They are doing this project to showcase what is possible to accomplish with today's technology to their current and new clients using a game setting. Because of that there is a social benefit to society in the sense of innovation.

2.3. Company's objectives

According to company's wishes, there must be a 3D RPG Tabletop game which features virtual reality, augmented reality, full body mocap, and asynchronous multiplayer. Ideally, VoIP should be implemented as well. Expected deliverables include game master application, player application and server application.

2.4. Project boundaries

The project boundaries include:

- Using Unity engine for the VR/AR application development.
- Using a local area network for the networking of the game.
- Using Noitom Perception Neuron mocap suit provided by school.
- Using Dissonance for VoIP provided by the company.

3. Theory

3.1. VR

Virtual reality means using a headset to immerse in a simulated world. When you move your head the simulation changes to adapt to your movement. For that reason there are multiple factors that induce nausea like low framerate of the simulation or high latency of the networked data and these factors have to be taken into account when making a multiplayer VR game.

3.2. AR

Augmented reality (AR) alters one's ongoing perception of the real world environment while virtual reality completely replaces the user's real world environment with a simulation. Much like VR which is a great simulation tool, AR is a novel human-computer interaction tool that overlays computer-generated information on the real scene (Nee, Ong, 2013).

In the context of this paper, AR means using augmented reality features using a mobile phone which is pointed at a marker (page).

3.3. Networking options

- Unity Networking High Level API

Unity's networking has a "high-level" scripting API (HLAPI). Using this means you get access to commands which cover most of the common requirements for multi-user games without needing to worry about the "lower level" implementation details. The HLAPI allows for:

Controlling the networked state of the game using a "Network Manager".

Operating "client hosted" games, where the host is also a player client.

Serializing data using a general-purpose serializer.

Sending and receiving network messages.

Sending networked commands from clients to servers.

Making remote procedure calls (RPCs) from servers to clients.

Sending networked events from servers to clients.

- Unity's Networking Low Level API

In addition to the high level networking API, Unity also provides access to a lower-level networking API called the Transport Layer. It allows for building custom networking systems with more specific or advanced requirements for the game's networking.

The Transport Layer is a thin layer working on top of the operating system's sockets-based networking. It can send and receive messages represented as arrays of bytes, and offers a number of different “quality of service” options to suit different scenarios. It is focused on flexibility and performance, and exposes an API within the NetworkTransport class.

- Third party solution

PlayerIO runs dedicated server code, where clients connect to the server and send messages, and the server processes game logic (Stagner, 2013). PlayerIO also includes a variety of other features such as login/account systems, databases, leaderboards, and so on. PlayerIO is cloud hosted that is, the game runs on a cluster of shared servers. Additionally PlayerIO is room based: Upon connecting to PlayerIO, you specify a room to connect to, and players are segregated into different rooms. For instance, in an MMO there might be different “rooms” for regions in the world. Without a dedicated cluster, rooms are limited to 45 players. PlayerIO supports TCP only as it was originally designed for Flash applications which do not support UDP. They support different platforms including Unity engine via a DLL Unity plugin.

3.4. Networking architecture

In the client-server model, the majority of the logic of the game runs on a single server. Multiple clients can connect to the server in order to take part in the multiplayer game. The client is basically a “dumb” rendering engine that also reads user input and controls the local player character, but otherwise simply renders whatever the server tells it to render (Gregory, 2009).

3.5. Mocap data

Motion capture (sometimes referred as mo-cap or mocap, for short) is the process of recording the movement of objects or people and translating it into other mediums such as video games and movies. It has a variety of uses: from game creation, sports analysis and medical application to robotics and military use (Alexal, 2016).

Noitom's Perception Neuron mocap suit seeks to redefine the motion capture industry by using the world's smallest inertial measurement unit (IMU), or, "neurons," as they call them. Neurons are tiny wireless sensors that actors place on their body (Figure 4) and which connect to a hub. The hub is in turn connected to a computer via WiFi or USB port.

Perception Neuron is delivered with AXIS Neuron, their in-house software that is designed to manage and calibrate the mocap system as well as perform basic motion capture. One of the most important feature of AXIS Neuron is the ability to stream the BVH (Biovision hierarchical, (n.d.)) data stream as well as export files to FBX for use in the most popular 3D software programs. The BVH data stream contains the movement and rotation of neurons.

There are 60 bones in total according to the Neuron API for Unity and 2 float 3D vectors (one for position and one for rotation) are captured for each bone. That means that there are 1440 bytes (60 bones x 2 vectors x 3 float values x 4 bytes per float) transferred per one frame of mocap data. The server sends this data on each Update tick to all the clients which are connected.

There are actually 31 sensors in total (Figure 5), but the data received from the BVH stream from Axis Neuron comes per bone and not per sensor. That is because the data from the sensors is optimized and calibrated and converted to bone data for 60 bones.

According to the API, the position received is in centimeters and rotation is in Euler angles. The format of the data, converted to Unity Engine's coordinate system, is:

Position 1 (float -X, float Y, float Z) → Rotation 1 (float Y, float -X, float -Z) → Position 2 → Rotation 2 → … → Position 60 → Rotation 60

The setup that I am using, is a local area network with one computer application that optimizes and sends the mocap data and multiple mobile phone applications that receive the data, all connected to a Wi-Fi router.

The speed limit of data transmission for the router that I am using is 0.7 Mb/s.

3.5.1. Optimizations

In order to reduce the bandwidth of the mocap data over the network, multiple optimization techniques are available.

- Deflate algorithm (Huffman coding/LZ77)

In Huffman coding, the basic idea is that a binary tree is built where the bytes (or bits) with higher frequencies are stored higher in the tree than bytes with lower frequencies. Frequencies are occurrences of the same bytes (or bits) over the data or file that needs to be compressed. That's why Huffman coding is called a minimum redundancy algorithm and it is a lossless compression algorithm (Huffman, 1952).

LZ77 algorithms achieve compression by replacing repeated occurrences of data with references to a single copy of that data existing earlier in the uncompressed data stream. A match is encoded by a pair of numbers called a [*length-distance*] pair, which is equivalent to the statement "each of the next [*length*] characters is equal to the characters exactly [*distance*] characters behind it in the uncompressed stream" (Ziv, Lempel, 1977).

In the deflate algorithm, which is basically a combination of the two methods discussed above, a compressed data set consists of a series of blocks, corresponding to successive blocks of input data. The block sizes can be defined by the user. Each block consists of two parts: a pair of Huffman code trees that describe the representation of the compressed data part, and a compressed data part. (The Huffman trees themselves are compressed using Huffman encoding.) The compressed data consists of a series of elements of two types: literal bytes (of strings that have not been detected as duplicated within the previous 32K input bytes), and pointers to duplicated strings, where a pointer is represented as a pair <length, backward distance> (Deutsch, 1996).

- Interpolation

In case that the rate of sending mocap data is lower than thirty or sixty frames per second, interpolation methods are needed to make the animation of the model based on mocap data smooth.

Linear Interpolation or lerp is a mathematical function which interpolates between two values. The interpolation amount depends on the passed weight factor along with the start and end values (Glazer, Madhav, 2015). The distance between each step is equal across the entire interpolation in case of positions from the mocap data.

Spherical linear interpolation or slerp is mapped as though on a quarter segment of a circle so there is the slow out and slow in effect. The distance between each step is not equidistant. Slerp can be applied to rotations from mocap data.

3.6. VoIP data

VoIP is short for Voice over Internet Protocol. Voice over Internet Protocol is a category of hardware and software that enables people to use the Internet as the transmission medium for telephone calls by sending voice data in packets using IP rather than by traditional circuit transmissions of the PSTN (public switched telephone network).

- Dissonance

Dissonance is a realtime Voice over IP (VoIP) system designed to be built directly into Unity engine games and it provides:

- Low latency/real-time voice communications.
- Efficient Opus encoding (lossy audio coding format)
- Voice Activation and Push To Talk
- Positional Audio
- Echo cancellation

4. Main and sub questions

How do you implement and optimize a networking solution for a VR-AR game with mocap, VoIP and gameplay data?

4.1. Sub questions

- How can one transmit mocap suit data over the network?
- How can one reduce the bandwidth of mocap data?

- How can one transmit VoIP data over the network?
- What kind of gameplay data is transmitted over the network?
- How is the gameplay data transmitted over the network?

The question on how mocap data and the rest of the data is transmitted over the network is answered in section 5.1.1. The results of reducing the bandwidth of mocap data are discussed in section 5.2.1. How to transmit VoIP data using Dissonance is explained in section 5.4. and the results in section 5.4.1. The type of gameplay data that are transmitted over the network and the way they are transmitted together with optimizations available (with explanations) can be found in section 5.3, 5.5, and 5.6.

5. Implementation and optimizations

5.1. Networking options

There were five test demos prepared to test the different networking technologies. The first uses the transport layer of UNet or low level API. It synchronizes player movement and names. There is no interpolation applied on the movement. See appendix 8.5. for code related to this demo.

Second demo uses UNet high level API. It uses the NetworkManager to control the HLAPI and NetworkManagerHUD for UI to host, join the game and debug networking. This demo is a first-person shooter game where players can shoot each other, die and respawn at preset spawn locations.

The next demo uses a custom server provided by PlayerIO. This demo is very similar to the one above but it differs in the networking technology used, namely using PlayerIO instead of Unity's networking API. The code for synchronizing player movement, rotation and shooting is custom made.

The fourth demo is a network lobby implemented with a custom server provided by PlayerIO. Here players can create rooms with names which are synchronized between all players that connect to the server. See appendix 8.6. for code related to the API used.

The last demo is a test demo for both PC and Android. The server code is uploaded on PlayerIO's servers. After entering their name the players can connect to the same world from mobile or PC. The goal of the game is to collect as much items as possible and the one who has more items by a time mark wins the game. Every iteration of the time mark for example two minutes, the game resets. The players can also chat from PC and view chat from mobile.

5.1.1. Results

Due to the third-party solution using strings for message identification it was discarded in favor of Unity's networking API which uses shorts and uses less bandwidth. Also generally speaking native solutions are preferred to third party solutions in case the latter isn't much better is some way. And it's relatively easy to setup games that connect in a local area network using Unity's networking API.

Due to the low level API needing more code to set up for our general networking needs which were easily fulfilled by Unity's high level API, it was discarded in favor of the latter.

The host uses the UNet HLAPI `NetworkServer.Listen(port)` and the clients use `NetworkClient.Connect(ipAddress, port)`. MessageBase is used to construct the packets that go through the network. There is already network serialization available for Vector3s, Quaternions and other structures. A dictionary can be serialized using an array of keys and an array of values. See appendix 8.7. for link to the demos prepared for the networking options.

5.2. Mocap

- Deflate algorithm

System.IO.Compression from the .NET framework was found which uses Huffman coding with LZ77 algorithm and is open source.

In my good PC at the workplace (Intel Xeon E5-1650 v2 @3.5GHz, 24 GB RAM), compression using this method took 0 ms or 1200-5000 clock ticks (10,000 ticks is 1 ms in the computer) (measured with System.Diagnostics.Stopwatch) and mocap data is compressed from 1440 bytes to 580 bytes. Decompression took 0 ms or 200-1000 ticks.

On my average laptop (Intel Core i5 2.2-2.7GHz, 4 GB RAM) compression took 2-3 ms and decompression took 0-1 ms where data is also compressed from 1440 bytes to 580 bytes (same compression algorithm from System.IO.Compression).

Only decompression is needed on mobile, and it took around 0-8 ms to decompress the Deflate algorithm and less than 1 ms to decompress the bone movement change algorithm discussed in the section below.

- Delta compression: Sending only the bones that moved

Another compression method is to manually send the bones data that changes by a given threshold. Setting the threshold to 0 makes this technique lossless because all bones that move are sent over the network and no data is lost. But making the threshold higher than 0 makes this technique lossy because movement that is below the threshold is cut off. So, if a bone didn't move "that much" it doesn't need to be synchronized anyway.

For this method, a byte needs to be sent for the bone id (example Hips) alongside with a byte for either changing rotation, position or both and after that either 1 or 2 float Vector3s for the bone new rotation and/or position.

So, if only the hand moves, there are in total 18 bones which means that instead of sending 1440 bytes over the network, only 18 * (byte-bone_id + byte-position_or_rotation + 2 * 3 * 4 bytes (floats)) or 468 bytes are sent over the network. Worst case scenario is when most bones move. Best case scenario is when few bones move. The thin line where this algorithm becomes a bad case scenario, considering the highest size for the compressed data is: 60 * (byte + byte + 2 * 3 * float) so the compressed is redundant by 60 * 2 * bytes or 120 bytes. In this scenario the algorithm needs to always send a byte in the beginning of the network message to tell whether the data is compressed or not because compression would not be needed in case most bones move. The "magical" threshold when the algorithm becomes redundant is: 1440 bytes / (byte + byte + 2 * 3 * float) = 55.38 bones.

So if 54 bones are moved and rotated the algorithm has a positive effect as can be seen below:

54 * (byte + byte + 2 * 3 * float) = 1404 bytes which is less than the original 1440 bytes.

To better understand this compression method, please check the Appendix 8.3. and 8.4.

- Lowering rate of sending mocap data

A third compression consideration involves lowering the rate at which mocap data is sent and applying interpolation to the data. Currently the data is transferred at 30 times per second which is bound by Unity's Update tick frequency. That means that without any compression applied there is around 1440 bytes * 30 = 43200 bytes per second or 43 Kbps which is much lower than 700 Kbps limit speed of router. In case the rate at which data is sent becomes 60 times per second, the data would be 86 Kbps. Taking into consideration other data that need to be transferred for the game, it is a good idea to lower the rate of mocap data transfer.

Alongside lowering the data rate, interpolation methods like linear interpolation and spherical linear interpolation are required to make the movement of the bones (from captured mocap data) in the animated model smooth.

5.2.1. Results

Absolute compression (sending original positions and rotations) worked very well for compressing mocap data. Due to the mocap suit being broken through the project, I wasn't able to test with varying levels of thresholds to see which threshold would work best for compression-precision. Relative compression (sending the difference in positions and rotations) was tested and it worked fine for positions and not so well for rotations because the bones would drift off after a while when adding the rotation difference from the last successful rotation. Doing some testing with human perception of the precision of mocap data considering various thresholds and both absolute and relative compression techniques discussed above would be great if the suit would still be working.

The result is partly the code in the appendix 18.2. which describes how the Neuron API should be changed together with two compression algorithms: Deflate and absolute compression.

5.3. Lobby

There is a players dictionary in both the server side and all client sides. The keys are unique identifiers created by the server for all the clients upon connection, and the values are instances of the Player class which have stats like bool ready, string name, and Role (enum) role.

The server sends these unique identifiers to all the clients upon connecting with them and the clients take care of assigning their unique id if it's unassigned and assign it to other players that connect as well.

Upon connecting to a client the server sends all other players information to the new client and the new client id to all the other players. During the demo, as clients change their name, ready state or class, they let the server know about this change and the server broadcasts it to all the clients.

Setting up players might take different amount of time on every peer due to lag, different hardware, or other reasons. To make sure the game will actually start when everyone is ready, pausing the game until all players are ready can be useful. When the server gets the OK from all the peers, it tells them to start the game (Linietsky, Manzur, 2018). This is implemented in the game.

5.4. VoIP

In order to fulfill client's wish to have voice over IP in the game, Dissonance is being used, which is a Unity store plugin. Trying to get the high level API demo working was more difficult and required a Network Manager alongside with other player information. Luckily there was also a low level API integration which was pretty straightforward to implement. `UNetCommsNetwork.StartAsClient` alongside with IP address needs to be called at the client and `UNetCommsNetwork.StartAsServer` needs to be called at the server. It's suggested to start the Dissonance client after the server has acknowledged the new client (for example when server pings the new client with some data like the client's server assigned ID).

It's also needed to add four components to a gameobject in the game, namely `UNetCommsNetwork` (low level API integration), `DissonanceComms` (Dissonance itself), `VoiceReceiptTrigger` (to receive voice data) and `VoiceBroadcastTrigger` (to send voice data).

In order to have a player talk to a specific other player, like the AR players will talk specifically to the game master and not other players – otherwise we would hear an echo since AR players are going to stand next to each other during the game session – some support code is needed to be added to Dissonance.

The server (or the VR player) should send his Dissonance id (hash value) to the client (or the AR player) upon connecting. Afterwards client sets the `VoiceBroadcastTrigger.PlayerId` to the server Dissonance id. Doing that makes it possible that the AR players can talk only to the VR player and not to each other.

Finally, a toggle is needed to be added to the game that enables or disables talking for the players because there is no Push To Talk button on mobile. Doing that proved to be surprisingly hard because it's not possible to change Dissonance code to interact with our code due to project scope issues. The issue was fixed by putting all our scripts under the Plugins folder where Dissonance is located. For this to work, an extra check was added in the Push to Talk (that originally only works with the Input Axis) code to check whether the toggle is activated. Now the AR players can talk to the game master upon activating a toggle and the VR master can talk to everyone via the toggle as well. In order that the VR player can talk to everyone, setting the channel type in `VoiceBroadcastTrigger` to `Room` instead of `Player` is needed.

There was a test conducted with four instances of the player application running on mobile phones which are connected to the game master application on the computer to see whether VoIP data affect the framerate of the VR application in a complex scene.

According to measurements, there are 1.5-2 Kbps voice data transferred from one client to the server. The server transfers data at 2-4 Kbps to one client (Figure 6). According to tests, in case there are 10 clients, server sends 20-40 Kbps of data to the clients and receives 15-20 Kbps data from the clients.

Coupled with the Mocap suit data transfer without any compression to the data the total and maximum would be 40 Kbps (VoIP) + 86 Kbps (MoCap) = 126 Kbps.

5.4.1. Results

Although the expectation was that VoIP data being transmitted between four applications in mobile phones and one application in the computer would affect the frame rate of the VR application running a complex scene, the result was negative. That means that the VoIP network data load was insignificant to the process.

5.5. Tiles data

The condition that clients need to be basic rendering engines without too much logic of their own should be satisfied. For that reason, there needs to be networking code where the server sends tile information to the clients based on where their player is positioned in the world. So two “hexagonal” functions in appendix 8.2. are being used to find tile coordinates based on player position and then find tiles in a specific range (like six) based on tile coordinates.

With a range of six, a total of a hundred twenty seven tiles need to be transferred over the network (five – ninety one, seven – a hundred sixty nine and so on). The tile data message contains this data as of the latest:

```
public class TileDataMessage : MessageBase
{
// recepient
public int Id;
// tile data
public int[] TilesXZ;
public float[] TilesScaleY;
public int[] TilesMaterial;
// objects data...
}
```

So for 127 tiles there would be 4 bytes (Id) + 127 * 2 * 4 bytes (Tile coordinates) + 127 * 4 bytes (Tile material) = 1528 bytes sent over the network. If this data is send every frame, assuming a frame rate of 30 frames per second then there would be 30 * 1528 bytes = 45840 bytes trespassing over the network or 91680 bytes in case of 60 frames per second.

Together with mocap and VoIP there are 126 Kbps + 92 Kbps = 218 Kbps of data transfer over the network.

Although the total is lower than the router's bandwidth limit, it's again a good idea to optimize the code. Among optimizations that can be utilized is lowering the rate of sending tile data to a smaller number like three times per second and limit the speed of the player movement so he doesn't appear at a location before tile information is retrieved from the server for that location.

Also saving the current tile where the player is positioned in the server and sending tile data only when the current tile changes to a new tile, is a good idea in case the player isn't moving constantly.

5.6. Player spawning and movement

First players need to spawn at a tile. Tiles are hexagon objects in the game and they are accessible by two coordinates, namely tile x and y. The game master sets the spawn locations in the level creation mode. The spawn locations are saved in the level file and they're loaded and parsed by the server and stored in an array. Upon connecting or when spawn points are parsed, players get assigned a spawn point which they use to spawn when the game has loaded in both the server and the client.

Unity's own path-finding navigation system is used to calculate the path that players need to take. In the end reinventing the wheel and writing pathfinding code from scratch while taking into account higher steps like hills in the game, would be time-consuming and not as good as Unity's own solution.

Player movement packets look something like this:

```
public class MoveMessage : MessageBase
{
public int Id;
public Vector3 Position;
public float RotationY;
public Vector3[] Path;
}
```

According to the original design of the game movements among other actions need to go through a queuing mechanism that the game master needs to approve before the movement is executed. The players are able to see the path that they or other players want to take but don't move until the action has been approved by the game master via a window (Figure 7).

According to the original design of the networking architecture, the server would create a local client to serve as a host (as can be seen in the demo at Appendix 8.7. Lobby). The local client would interact with the UI elements and the server would only do server logic with complete separation to graphics or UI. But because of this queuing mechanism - the server would need to send a list of the actions that are queued to the local client otherwise called the host, and the host would send back his response.

But this added a layer of unnecessary complexity to the game so there was a decision to drop the host concept and have the server interact with UI elements and other gameplay aspects directly.

Server or the VR player interacts with UI elements to approve, interrupt or deny a movement action. These options remove the current queued movement action from the array and broadcast it to all the other players.

6. Conclusion

Unity's HLAPI was chosen as the favorite option to implement the networking for the game due to using less bandwidth than PlayerIO for message identification and for being the native solution of Unity. For that reason it already serializes multiple common used objects. Also it was chosen for being more straightforward to use than Unity's LLAPI which had more customization than needed.

Tests showed that the "absolute" delta compression method which was developed alongside with Deflate algorithm compression worked very well together on both the compressing and decompressing end. Due to the fact that the suit was broken through the project, further optimizations got low priority.

Changing the networking from having a host of the game which is just a client with privileged commands to having the server directly be the host proved to facilitate a lot of the code that deals with queuing of the AR player actions.

Although the expectation was that VoIP data being transmitted between 4 applications in mobile phones and 1 application in the computer would affect the frame rate of the VR application running a complex scene, the result was that the VoIP data network load was insignificant to the process.

7. References

Ong, S. K., & Nee, A. Y. C. (2013). Virtual and augmented reality applications in manufacturing. Springer Science & Business Media.

Biovision BVH. (n.d.). Retrieved from https://research.cs.wisc.edu/graphics/Courses/cs-838-1999/Jeff/BVH.html.

Alexal. (2016, May 06). MOCAP 101. Retrieved from https://neuronmocap.com/content/mocap-101.

Glazer, J., & Madhav, S. (2015). Multiplayer game programming: Architecting networked games. Addison-Wesley Professional.

Huffman, D. A. (1952). A method for the construction of minimum-redundancy codes. Proceedings of the IRE, 40(9), 1098-1101.

Ziv, J., & Lempel, A. (1977). A universal algorithm for sequential data compression. IEEE Transactions on information theory , 23 (3), 337-343.

Deutsch, P. (1996). DEFLATE compressed data format specification version 1.3 (No. RFC 1951).

Gregory, J. (2014). Game engine architecture. AK Peters/CRC Press, 333-334.

Linietsky, J., & Manzur, A. (2018). High level multiplayer - Synchronizing game start. Retrieved from https://docs.godotengine.org/en/3.0/tutorials/networking/high_level_multiplayer.html

Stagner, A. R. (2013). Unity multiplayer games: Build engaging, fully functional, multiplayer games with Unity engine. Birmingham: Packt Pub. pp. 127.

Mixamo - Demon T Wierzzorek character. (2018). Retrieved from https://www.mixamo.com/

Perception Neuron by Noitom. (2018). Retrieved from https://neuronmocap.com/

Red Blob Games. (2018). Hexagonal Grids. Retrieved from https://www.redblobgames.com/grids/hexagons/

8. Appendix

8.1. Figures

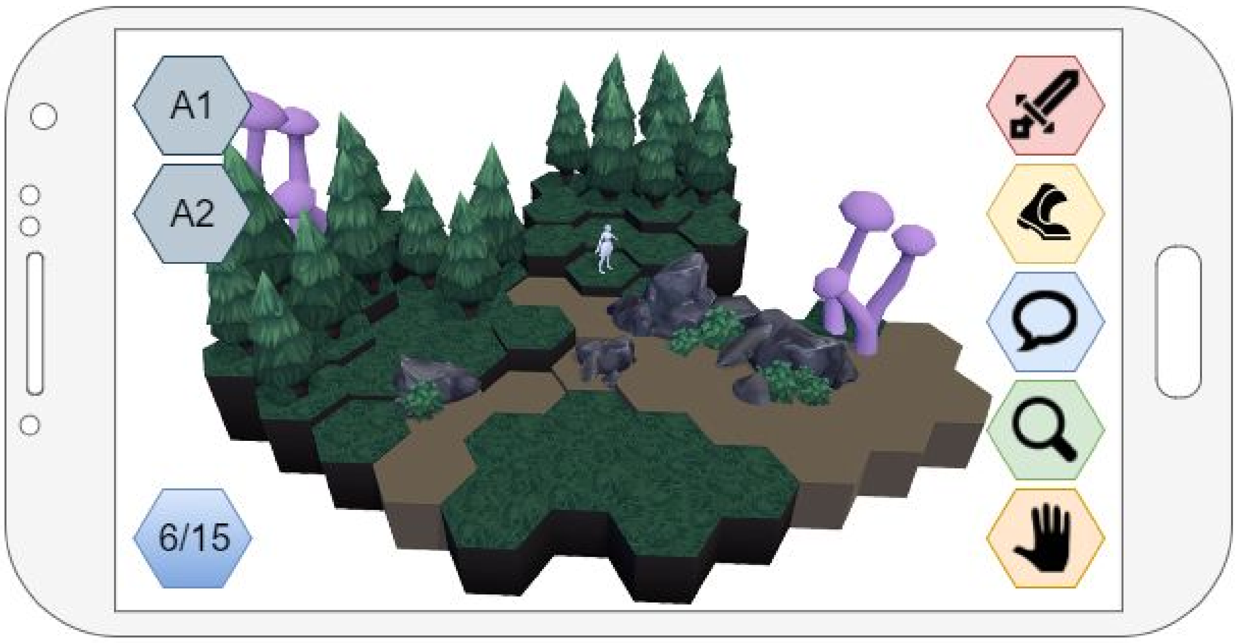


Figure 1 AR Player user interface with the actions that they can queue on the right like attack, inspect, talk to NPC…

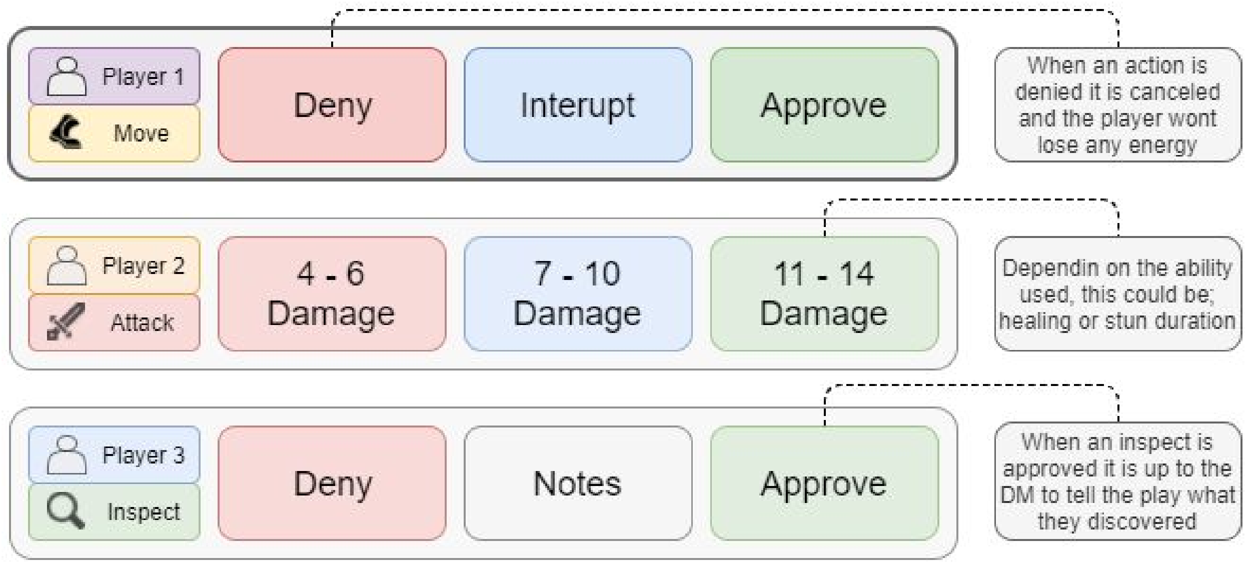
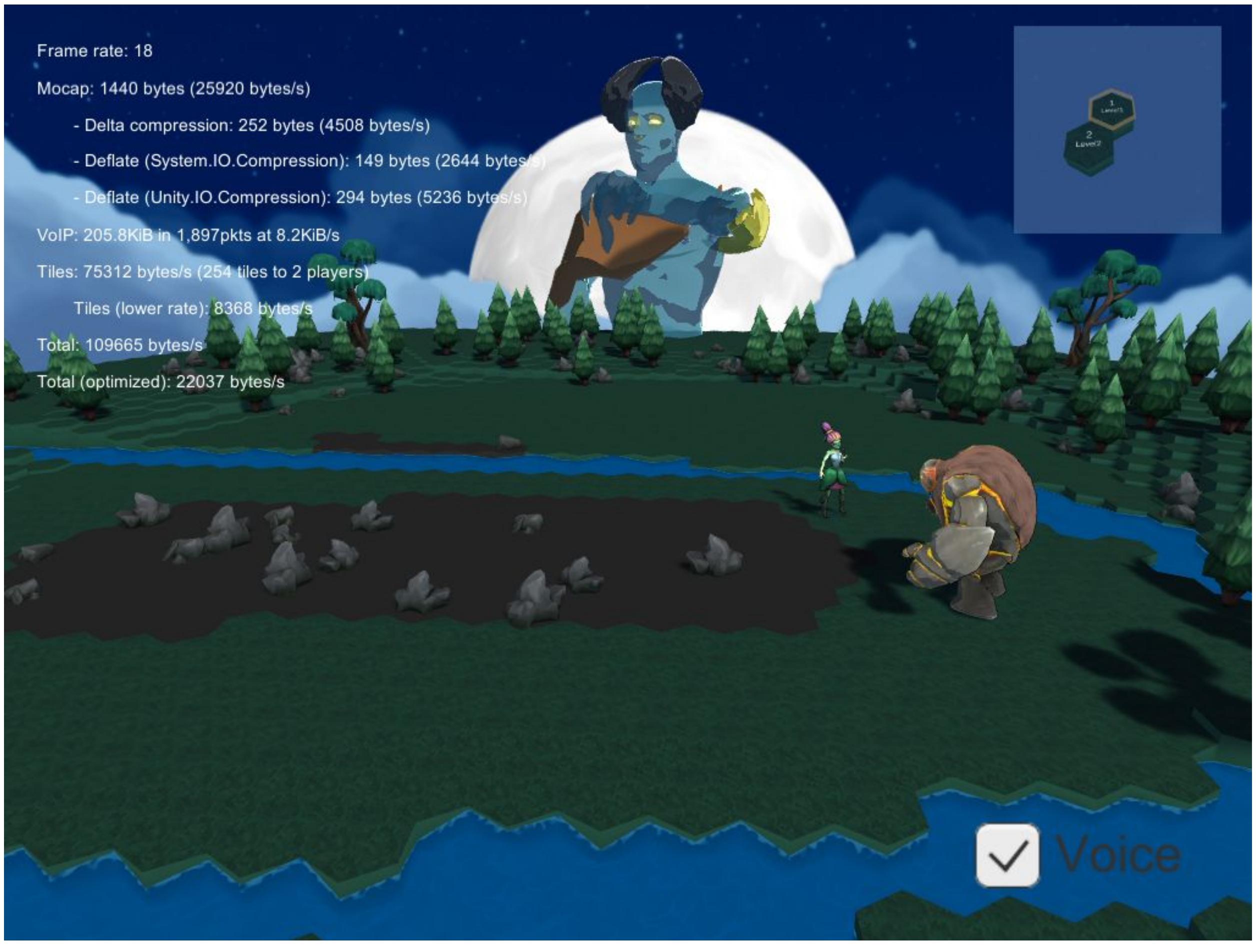

Player 1
Move
Deny
Interupt
Approve
When an action is denied it is canceled and the player wont lose any energy
Player 2
Attack
4 - 6 Damage
7 - 10 Damage
11 - 14 Damage
Dependin on the ability used, this could be; healing or stun duration
Player 3
Inspect
Deny
Notes
Approve
When an inspect is approved it is up to the DM to tell the play what they discovered

Figure 2 Queuing of actions that require game master’s input.



```
Frame rate: 18
Mocap: 1440 bytes (25920 bytes/s)
    - Delta compression: 252 bytes (4508 bytes/s)
    - Deflate (System.IO.Compression): 149 bytes (2644 bytes/s)
    - Deflate (Unity.IO.Compression): 294 bytes (5236 bytes/s)
VoIP: 205.8KiB in 1,897pkts at 8.2KiB/s
Tiles: 75312 bytes/s (254 tiles to 2 players)
    Tiles (lower rate): 8368 bytes/s
Total: 109665 bytes/s
Total (optimized): 22037 bytes/s
```

Figure 3 A concept of the game world and the giant behind being controlled by the VR player using a mocap suit (Mixamo, 2018).

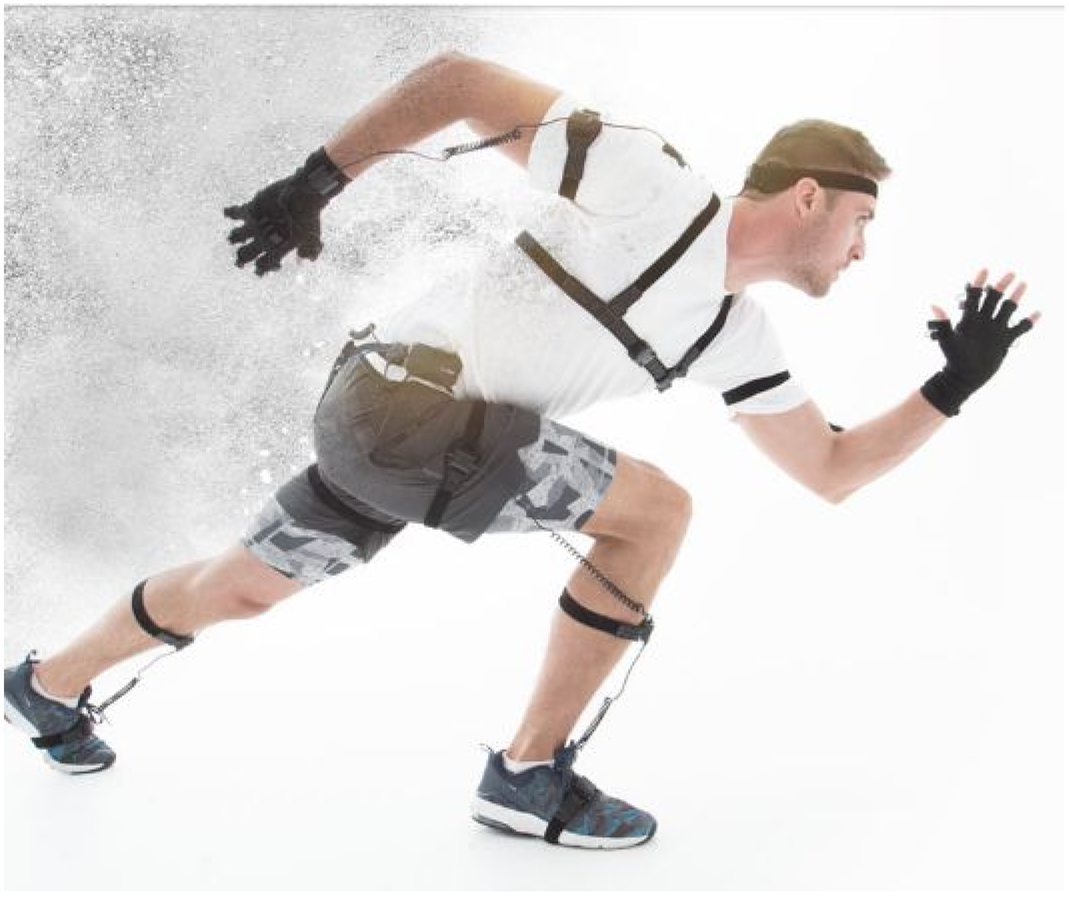


Figure 4 Neuron Perception mocap suit (Noitom, 2018).

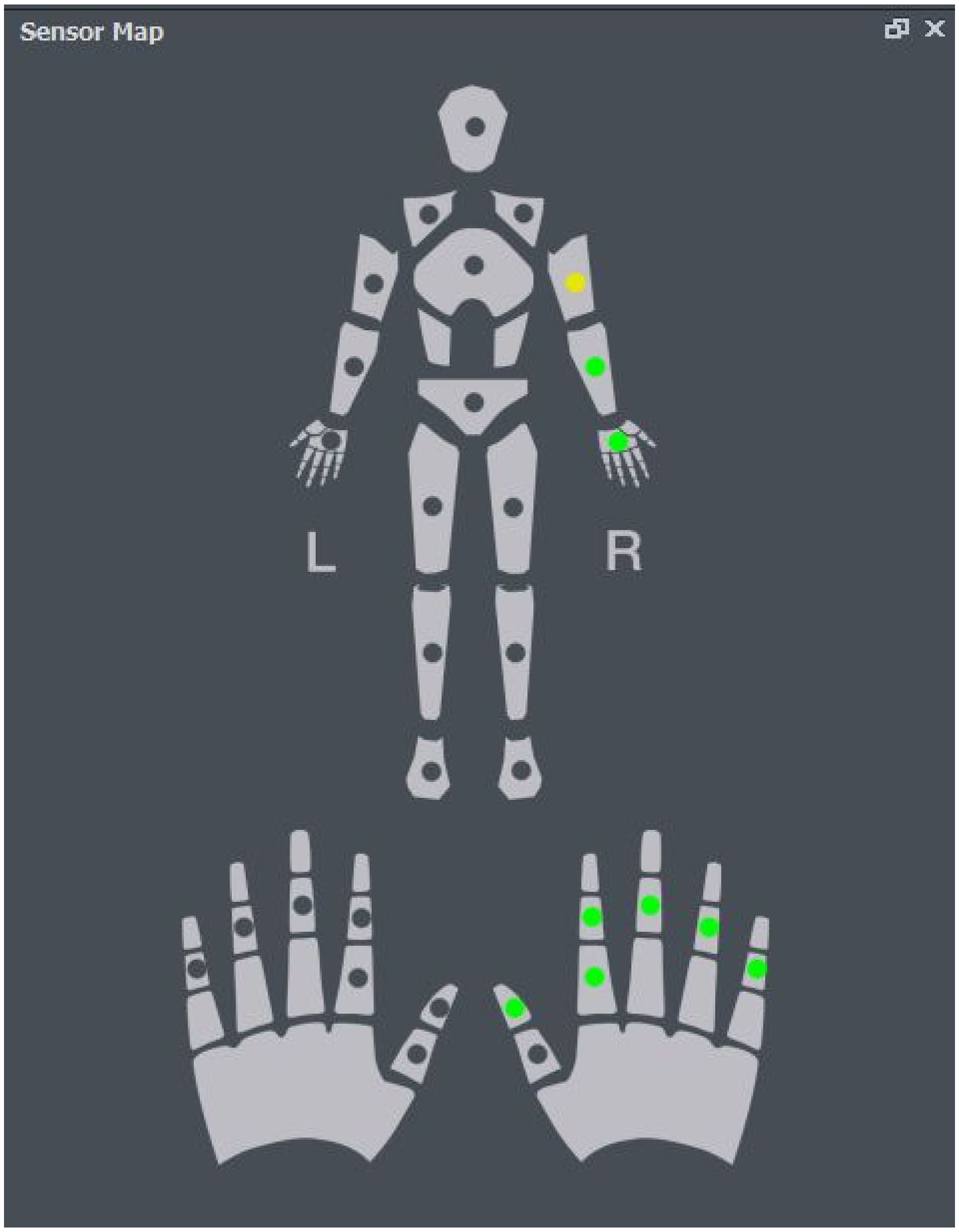


Figure 5 Sensor/neuron map for Neuron Perception mocap suit.

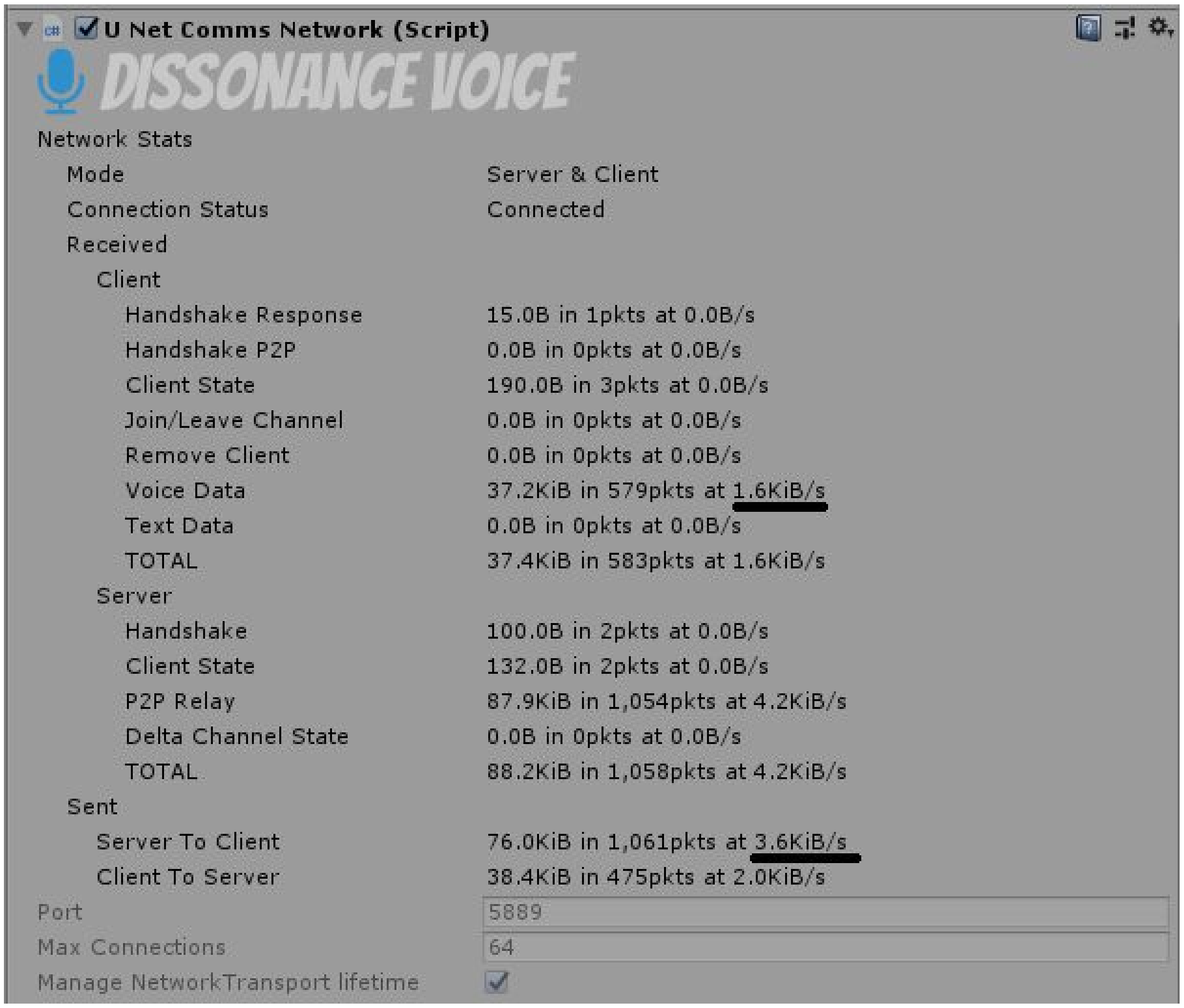

```
U Net Comms Network (Script)
DISSONANCE VOICE
Network Stats
  Mode                          Server & Client
  Connection Status             Connected
  Received
    Client
      Handshake Response        15.0B in 1pkts at 0.0B/s
      Handshake P2P             0.0B in 0pkts at 0.0B/s
      Client State              190.0B in 3pkts at 0.0B/s
      Join/Leave Channel        0.0B in 0pkts at 0.0B/s
      Remove Client             0.0B in 0pkts at 0.0B/s
      Voice Data                37.2KiB in 579pkts at 1.6KiB/s
      Text Data                 0.0B in 0pkts at 0.0B/s
      TOTAL                     37.4KiB in 583pkts at 1.6KiB/s
    Server
      Handshake                 100.0B in 2pkts at 0.0B/s
      Client State              132.0B in 2pkts at 0.0B/s
      P2P Relay                 87.9KiB in 1,054pkts at 4.2KiB/s
      Delta Channel State       0.0B in 0pkts at 0.0B/s
      TOTAL                     88.2KiB in 1,058pkts at 4.2KiB/s
  Sent
    Server To Client            76.0KiB in 1,061pkts at 3.6KiB/s
    Client To Server            38.4KiB in 475pkts at 2.0KiB/s
Port                            5889
Max Connections                 64
Manage NetworkTransport lifetime  ✓
```

Figure 6 VoIP traffic over the network with 1 server and 1 client connected.

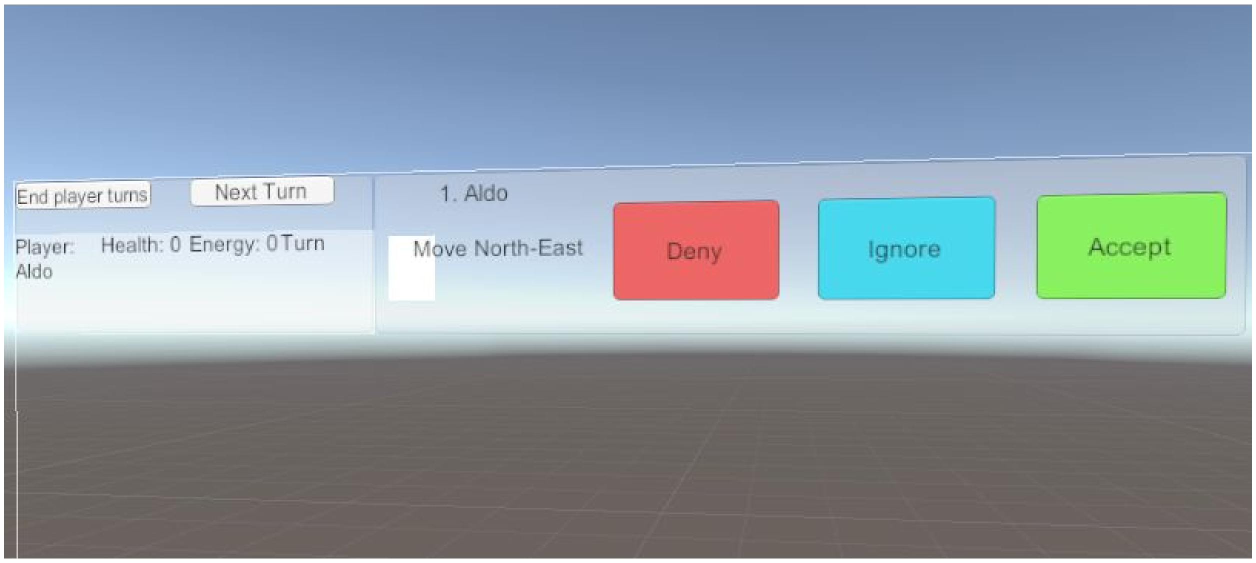


Figure 7 A window at game master's application for approving or denying a movement action of the player.

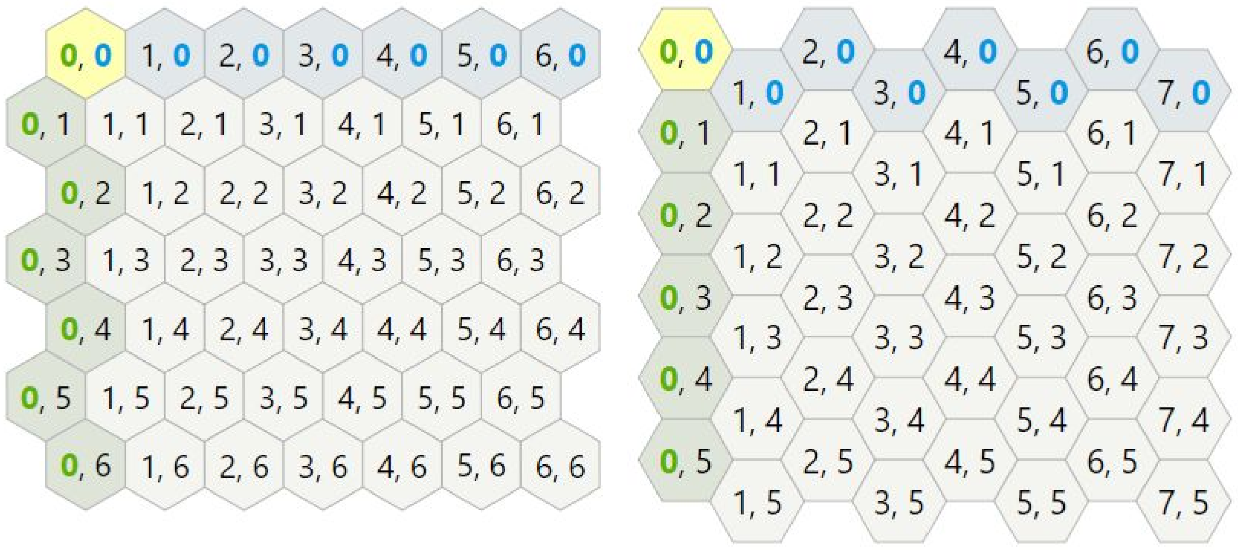


Figure 8 Pointy top hexagonal tiles (left) vs flat top hexagonal tiles (right) (Red Blob Games, 2018).

8.2. Hexagonal grids

Hexagonal grids in the game are composed of hexagons which are 6-sided polygons. Regular hexagons have all the sides the same length. All the hexagons in the game are regular and pointy top hexagons (Figure 8).

The four basic algorithms (pseudo-code) for hexagonal grids that we use in the game, assuming pointy top hexagons are:

- Positioning of hexagon tiles where q and r are tile coordinates and hex radius is the distance between the hexagon center and a corner:

```
var x = hex radius * (sqrt(3) * q  +  sqrt(3)/2 * r)
var z = hex radius * (3./2 * r)
```

- Finding tiles in a range:

```
var results = []
for each -range ≤ x ≤ +range:
    for each max(-range, -x-range) ≤ y ≤ min(+range, -x+range):
        var z = -x-y
        results.append(center + Hex(x, y, z))
```

- Transforming from world position to tile coordinates:

```
var q = (sqrt(3)/3 * point.x  -  1./3 * point.z) / hex radius
var r = (2./3 * point.z) / hex radius
```

The last 2 algorithms are used in order to find the tile corresponding the AR player position and sending the tiles that are in “sight” or range back to him via networking.

8.3. Server-side absolute compression code

```
float[] prevData;
    void AbsoluteCompression()
    {
        var neuronMessage = new NeuronMessageAbsolute();
        List<byte> byteList = new List<byte>();

        int numBones = 60;
        int bonesChanged = 0;

        // iterate through all the bones and construct position/rotation and compare with previous
position rotation by a threshold.
        // send only the bones which moved by the threshold.
        for (int i = 0; i < numBones; i++)
        {
            int offset = 0;
            offset += i * 6;

            Vector3 position = new Vector3(-neuronActor.GetData()[offset], neuronActor.GetData()[offset
+ 1], neuronActor.GetData()[offset + 2]);
            Vector3 prevPosition = new Vector3(-prevData[offset], prevData[offset + 1], prevData[offset
+ 2]);

            offset = 0;
            offset += 3 + i * 6;

            Vector3 rotation = new Vector3(neuronActor.GetData()[offset + 1],
-neuronActor.GetData()[offset], -neuronActor.GetData()[offset + 2]);
            Vector3 prevRotation = new Vector3(prevData[offset + 1], -prevData[offset], -prevData[offset
+ 2]);

            bool positionChanged = false;
            bool rotationChanged = false;

            if (Vector3.Distance(position, prevPosition) > 0 || Vector3.Angle(rotation, prevRotation) >
0)
            {
                bonesChanged++;
            }
            if (Vector3.Distance(position, prevPosition) > 0)
            {
                positionChanged = true;
                position *= NeuronActor.NeuronUnityLinearScale;
            }
            if (Vector3.Distance(rotation, prevRotation) > 0)
            {
                rotationChanged = true;
            }

            // rotation changed
            if (rotationChanged && !positionChanged)
            {
                // 14 bytes
                byteList.Add((byte)i); // bone index
                byteList.Add(0); // what changed
                byteList.AddRange(BitConverter.GetBytes(rotation.x));
                byteList.AddRange(BitConverter.GetBytes(rotation.y));
                byteList.AddRange(BitConverter.GetBytes(rotation.z));
            }
            // position changed
            else if (!rotationChanged && positionChanged)
            {
                // 14 bytes
                byteList.Add((byte)i); // bone index
                byteList.Add(1); // what changed
                byteList.AddRange(BitConverter.GetBytes(position.x));
                byteList.AddRange(BitConverter.GetBytes(position.y));
                byteList.AddRange(BitConverter.GetBytes(position.z));
            }
            // both changed
            else if (rotationChanged && positionChanged)
            {
                // 26 bytes
                byteList.Add((byte)i); // bone index
                byteList.Add(2); // what changed
                byteList.AddRange(BitConverter.GetBytes(rotation.x));
                byteList.AddRange(BitConverter.GetBytes(rotation.y));
                byteList.AddRange(BitConverter.GetBytes(rotation.z));
                byteList.AddRange(BitConverter.GetBytes(position.x));
                byteList.AddRange(BitConverter.GetBytes(position.y));
                byteList.AddRange(BitConverter.GetBytes(position.z));
            }
        }

        // copy current data to previous data.
        Array.Copy(neuronActor.GetData(), prevData, neuronActor.GetData().Length);
        neuronMessage.data = Compress(byteList.ToArray());
        // send the newly constructed packet
        NetworkServer.SendToAll(MyMsgType.NeuronAbsolute, neuronMessage);
    }
    // deflate algorithm
    public static byte[] Compress(byte[] data)
    {
        MemoryStream output = new MemoryStream();
        using (DeflateStream dstream = new DeflateStream(output, CompressionMode.Compress))
        {
            dstream.Write(data, 0, data.Length);
        }
        return output.ToArray();
    }
```

8.4. Client-side absolute decompression code

```
void GotAbsoluteNeuronData(NetworkMessage netMsg)
    {
        var msg = netMsg.ReadMessage<NeuronMessageAbsolute>();

        byte[] decompressed = Decompress(msg.data);

        robotAbsolute.ApplyMotionAbsolute(decompressed);
    }

    public static byte[] Decompress(byte[] data)
    {
        MemoryStream input = new MemoryStream(data);
        MemoryStream output = new MemoryStream();
        using (DeflateStream dstream = new DeflateStream(input, CompressionMode.Decompress))
        {
            CopyTo(dstream, output);
        }
        return output.ToArray();
    }

    public static void CopyTo(Stream input, Stream output)
    {
        byte[] buffer = new byte[16 * 1024];
        int bytesRead;
        while ((bytesRead = input.Read(buffer, 0, buffer.Length)) > 0)
        {
            output.Write(buffer, 0, bytesRead);
        }
    }

    public void ApplyMotionAbsolute(byte[] motionData)
    {
        NeuronAnimatorInstance.ApplyMotionAbsolute(motionData, animator);
    }

    public static void ApplyMotionAbsolute(byte[] motionData, Animator animator)
    {
        int i = 0;
        while (i < motionData.Length)
        {
            // bone id
            byte neuronBoneId = motionData[i++];

            byte rotationOrPosition = motionData[i++];
            // rotation
            if (rotationOrPosition == 0)
            {
                byte[] rotationXBytes = new byte[4];
                rotationXBytes[0] = motionData[i++];
                rotationXBytes[1] = motionData[i++];
                rotationXBytes[2] = motionData[i++];
                rotationXBytes[3] = motionData[i++];
                byte[] rotationYBytes = new byte[4];
                rotationYBytes[0] = motionData[i++];
                rotationYBytes[1] = motionData[i++];
                rotationYBytes[2] = motionData[i++];
                rotationYBytes[3] = motionData[i++];
                byte[] rotationZBytes = new byte[4];
                rotationZBytes[0] = motionData[i++];
                rotationZBytes[1] = motionData[i++];
                rotationZBytes[2] = motionData[i++];
                rotationZBytes[3] = motionData[i++];

                float rotationX = BitConverter.ToSingle(rotationXBytes, 0);
                float rotationY = BitConverter.ToSingle(rotationYBytes, 0);
                float rotationZ = BitConverter.ToSingle(rotationZBytes, 0);

                SetRotation(animator, boneId, new Vector3(rotationX, rotationY, rotationZ));
            }
            // position
            else if (rotationOrPosition == 1)
            {
                byte[] positionXBytes = new byte[4];
                positionXBytes[0] = motionData[i++];
                positionXBytes[1] = motionData[i++];
                positionXBytes[2] = motionData[i++];
                positionXBytes[3] = motionData[i++];
                byte[] positionYBytes = new byte[4];
                positionYBytes[0] = motionData[i++];
                positionYBytes[1] = motionData[i++];
                positionYBytes[2] = motionData[i++];
                positionYBytes[3] = motionData[i++];
                byte[] positionZBytes = new byte[4];
                positionZBytes[0] = motionData[i++];
                positionZBytes[1] = motionData[i++];
                positionZBytes[2] = motionData[i++];
                positionZBytes[3] = motionData[i++];

                float positionX = BitConverter.ToSingle(positionXBytes, 0);
                float positionY = BitConverter.ToSingle(positionYBytes, 0);
                float positionZ = BitConverter.ToSingle(positionZBytes, 0);

                SetPosition(animator, boneId, new Vector3(positionX, positionY, positionZ));
            }
            // rotation and position
            else if (rotationOrPosition == 2)
            {
                byte[] rotationXBytes = new byte[4];
                rotationXBytes[0] = motionData[i++];
                rotationXBytes[1] = motionData[i++];
                rotationXBytes[2] = motionData[i++];
                rotationXBytes[3] = motionData[i++];
                byte[] rotationYBytes = new byte[4];
                rotationYBytes[0] = motionData[i++];
                rotationYBytes[1] = motionData[i++];
                rotationYBytes[2] = motionData[i++];
                rotationYBytes[3] = motionData[i++];
                byte[] rotationZBytes = new byte[4];
                rotationZBytes[0] = motionData[i++];
                rotationZBytes[1] = motionData[i++];
                rotationZBytes[2] = motionData[i++];
                rotationZBytes[3] = motionData[i++];

                float rotationX = BitConverter.ToSingle(rotationXBytes, 0);
                float rotationY = BitConverter.ToSingle(rotationYBytes, 0);
                float rotationZ = BitConverter.ToSingle(rotationZBytes, 0);

                byte[] positionXBytes = new byte[4];
                positionXBytes[0] = motionData[i++];
                positionXBytes[1] = motionData[i++];
                positionXBytes[2] = motionData[i++];
                positionXBytes[3] = motionData[i++];
                byte[] positionYBytes = new byte[4];
                positionYBytes[0] = motionData[i++];
                positionYBytes[1] = motionData[i++];
                positionYBytes[2] = motionData[i++];
                positionYBytes[3] = motionData[i++];
                byte[] positionZBytes = new byte[4];
                positionZBytes[0] = motionData[i++];
                positionZBytes[1] = motionData[i++];
                positionZBytes[2] = motionData[i++];
                positionZBytes[3] = motionData[i++];

                float positionX = BitConverter.ToSingle(positionXBytes, 0);
                float positionY = BitConverter.ToSingle(positionYBytes, 0);
                float positionZ = BitConverter.ToSingle(positionZBytes, 0);

                SetRotation(animator, boneId, new Vector3(rotationX, rotationY, rotationZ));
                SetPosition(animator, boneId, new Vector3(positionX, positionY, positionZ));
            }
        }
    }
```

8.5. Unity's Low Level Networking API

Check the blog post for more info:

https://tabletoprpgblog.wordpress.com/2018/10/01/networking-basics/

```
NetworkTransport.Init();
ConnectionConfig cc = new ConnectionConfig();
reliableChannel = cc.AddChannel(QosType.Reliable);
unreliableChannel = cc.AddChannel(QosType.Unreliable);
HostTopology topo = new HostTopology(cc, MAX_CONNECTION);

hostId = NetworkTransport.AddHost(topo, 0);
connectionId = NetworkTransport.Connect(hostId, "127.0.0.1", port, 0, out error);

private void Update()
{
//...data types
NetworkEventType recData = NetworkTransport.Receive(out recHostId, out connectionId, out channelId,
recBuffer, bufferSize, out dataSize, out error);
switch (recData)
```

```
{
case NetworkEventType.DataEvent:
string msg = Encoding.Unicode.GetString(recBuffer, 0, dataSize);
Debug.Log("Receiving: " + msg);
string[] splitData = msg.Split('|');// what type of message is it?
switch (splitData[0])
{
case "ASKNAME":
OnAskName(splitData);
break;
case "CONNECT":
SpawnPlayer(splitData[1], int.Parse(splitData[2]));
break;
case "DISCONNECT":
PlayerDisconnected(int.Parse(splitData[1]));
break;
case "ASKPOSITION":
OnAskPosition(splitData);
break;
default:
Debug.Log("Invalid message: " + msg);
break;
} //...closure
```

8.6. PlayerIO Networking API

```
PlayerIO.Authenticate(
"rpgtabletopgame-8b8vr6ckqkqlu8eep9hn6a", //Your game id
"public", //Your connection id
new Dictionary<string, string> { //Authentication arguments
{ "userId", playerName },
},
null,
delegate (Client client) {
DebugText("Successfully connected to Player.IO");playerIOClient = client;
client.Multiplayer.DevelopmentServer = new ServerEndpoint("localhost", 8184);
},
delegate (PlayerIOError error) {
DebugText(error.ToString());
}
);

//Create and join the room
playerIOClient.Multiplayer.CreateJoinRoom(
roomId, //Room id. If set to null a random roomid is used
```

```
"RPGTabletopGame", //The room type started on the server
true, //Should the room be visible in the lobby?
null,
null,
delegate (Connection connection) {
DebugText("Joined Room.");// We successfully joined a room so set up the message handler
playerIOConnection = connection;
playerIOConnection.OnMessage += handleMessage;
},
delegate (PlayerIOError error) {
DebugText(error.ToString());
}
);

playerIOClient.Multiplayer.ListRooms("RPGTabletopGame", null, 0, 0, delegate (RoomInfo[] rooms)
{
for (int i = 0; i < 8; i++)
{
GameObject buttonObj = GameObject.Find("Button (" + i + ")");
Button button = buttonObj.GetComponent<Button>();
button.onClick.RemoveAllListeners();
if (i < rooms.Length)
{
buttonObj.GetComponentInChildren<Text>().text = rooms[i].Id;
button.onClick.AddListener(delegate ()
{
roomId = buttonObj.GetComponentInChildren<Text>().text;
DebugText("Clicked button and setting room id to: " + roomId);
playerIOClient.Multiplayer.JoinRoom(roomId, null, delegate (Connection connection)
{
DebugText("Joined Room.");
// We successfully joined a room so set up the message handler
playerIOConnection = connection;
playerIOConnection.OnMessage += handleMessage;
}, // closure
```

8.7. Blog posts and links

Graduation deliverables (professional products, source code, videos and more):
https://drive.google.com/open?id=1k0_Wx3FlAU3tPETeXitIu2aRMdDBwBQ-

Mocap data networking first tests and video
https://tabletoprpgblog.wordpress.com/2018/10/12/sprint-2-programming-networking-mocap-data-test-1/

Game lobby with source code and executable
https://tabletoprpgblog.wordpress.com/2018/10/16/sprint-3-programming-unet-hlapi-lobby/

VoIP implementation
https://tabletoprpgblog.wordpress.com/2018/11/19/sprint-5-programming-voice-over-ip-implementation/

Running path-finding on the server demo video
https://tabletoprpgblog.wordpress.com/2019/01/13/sprint-8-programming-running-path-finding-on-the-server/

All my blog posts
https://tabletoprpgblog.wordpress.com/author/aldoleka/

Belle, C. (2016, April). A Closer Look at Huffman Encoding Using C#. Retrieved from http://www.carlbelle.com/Articles/Article/2016/Apr/6/ACloserLookAtHuffmanEncodingUsingCSharp/95

Huffman Tree: Traversing. (n.d.). Retrieved from https://stackoverflow.com/questions/27104535/huffman-tree-traversing

Huffman Coding - Hong Kong University of science and technology. (n.d.). Retrieved from https://home.cse.ust.hk/~dekai/271/notes/L15/L15.pdf

Huffman Data Compression without needing the tree to decode. (2010). Retrieved from http://code.activestate.com/recipes/577480-huffman-data-compression/

Dissecting the GZIP format. (n.d.). Retrieved from http://www.infinitepartitions.com/art001.html

Huffman Coding: A CS2 Assignment. (n.d.). Retrieved from https://www2.cs.duke.edu/csed/poop/huff/info/

MSFT, S. H. (2007). Network compression: Bitfields. Retrieved from https://blogs.msdn.microsoft.com/shawnhar/2007/12/28/network-compression-bitfields/

Hitcents. (n.d.). GZipStream DeflateStream - Unity.IO.Compression (in case System.IO.Compression doesn’t work on mobile). Retrieved from https://assetstore.unity.com/packages/tools/integration/gzipstream-deflatestream-unity-io-compression-31902