Networked Avatar Meeting: An Example for low-level Network Communication for the Creation of Social Virtual Reality Setups *

Marius Rubo *University of Bern*

This project demonstrates a setup where several avatars, controlled in different programs, meet on a table in a networked environment. The example mimics a social virtual reality (Social VR) setup but runs on ordinary computers, or even a single computer, with no need for VR hardware. It is intended to facilitate the creation of social VR solutions in psychological and neuroscience research rather than gaming or entertainment. Network communication is organized using low-level commands as opposed to higher-level netcode functionalities in order to more flexibly integrate data streams and to bundle data during logfile writing.

Keywords: Social Virtual Reality, Network Communication, Logfile writing

Contents

*Version: 1.0, Release Date: July 25, 2024. Contact: marius.rubo@gmail.com, marius.rubo@unibe.ch

Overview

Features

- Mock social VR setup where two to four avatars meet on a table in a networked environment
- Runs on connected computers or even only one computer with no need for VR hardware
- Demonstrates character control as in a research-grade social VR setup, e.g., aligning a character's head to match the positions of the character's eyes with the user's eyes
- Bundles data streams to store them to disk in an organized format
- Preprocessing of gaze data to conform to standards in eye-tracking research

System Requirements

- Unity 2022 or newer
- Supported Platforms: Should be compatible with all platforms that Unity supports, including Windows, macOS, Linux, iOS, and Android, but tested only on Windows

Installation

- This project provides a self-contained and fully-functional example. See example scenes for details.
- This project does not come with APIs for VR sensors but mimics them. Customized integration into your own VR software is required.

License

- This project is licensed under the GNU General Public License v3.0. See License.txt.

Acknowledgements

- This software uses Fish-Net, a networking solution. See https://github.com/FirstGearGames/FishNet. Fish-Net uses a custom license which permits a "worldwide, non-exclusive, no-charge, and royalty-free license to reproduce, modify, and use the software".
- This software uses Fast IK, an inverse kinematics tool. See https://github.com/ditzel/SimpleIK. Fast IK is used under the MIT License.
- This software uses Quick Outline, an outline shader tool. See https://github.com/chrisnolet/QuickOutline. Quick Outline is used under the MIT License.

- This software uses characters made with the MakeHuman open source 3D computer graphics middleware. See http://www.makehumancommunity.org/ for details. MakeHuman is used under the CC0 license.

Disclaimer

This software is provided "as is", without warranty of any kind, express or implied, including but not limited to the warranties of merchantability, fitness for a particular purpose, and noninfringement.

Getting started

This project demonstrates the architecture of a networked meeting of avatars. Create client instances by disabling the *Server* in the scene *NetworkedMeeting* before building, and create server instances by disabling everything except for the *Server* before building. The server can run in the same program as one of the clients, acting as host. Adapt the 'Server IP' field on the client to match the server's IP address in the local network when running the client on a different computer in that local network.

Scripts

Here I briefly outline the purpose of each script in this project. See the scripts themselves (in Assets/Scripts) and in-line comments for further details.

Character Control

These scripts provide standardizes ways to control both the avatar which a player is controlling (the self) as well as the ones controlled by others (so-called remote players). Using a common intermediate structure for both simplifies the organization of character state exchanges and subsequent logfile writing.

CharacterBehaviorController.cs

This script serves as the center piece in controlling each character model's behavior in the scene. The same script is used for the character which represents the self (where behavior is typically driven by information from the headset's sensors) and for all other characters in the scene, representing other participants (behavior of these characters, called 'remote players' in netcode jargon, are driven by data received from the server).

Using the same center script for controlling both the user's own avatar and the remote players is not strictly necessary when setting up a social VR software. Instead, other solutions may directly drive a user's own avatar by the headset sensors using classes which more strongly focus on modularity and encapsulation (i.e., one module only focussing on transferring head movements, one module only focussing on controlling eye movements etc.). However, by using a center script - the same for the self and for remote players - data can be more easily integrated by subsequent logfile writers.

Note also that this script does not control each component of the character directly, but calls other, specialized scripts, in particular the BodyController, EyeController and FacialExpressionController. This more modular arrangement allows to more easily switch between different character models. For example, while some character models define the direction in which an eye is looking as its "forward" vector, others define it as its "up" vector. Such idiosyncracies are handled in the specific EyeController (which may need to be adapted for other character models) but are hidden from this center script.

BodyController.cs

This script controls and returns the headIK target. Although it does not control the eyes, it needs to also reference them in this setup since the head IK is set so that the character's eyes match the eyes' positions as defined by the eye-tracking API. To achieve this, this script furthermore implements an auxiliary construction to avoid feedback loops between the AdjustHeadToEyesPosition()-function and the head's IK processes. In particular, additional representations of the eyes are instantiated and parented to the headIK object; AdjustHeadToEyesPosition() then obtains the positions of these eye representations which are modulated directly as children of the headIK objects and unaffected by IK algorithms. Without this construction, the interaction between AdjustHeadToEyesPosition() and IK may result in self-reinforcing movement drifts.

EyeController.cs

This script sets and returns the eyes' rotation and returns their positions. It adjusts the transfer of gaze data to a specific character model. The example below handles the case where (as in MakeHuman characters) the eyes' looking direction is indicated by their local up vector. Above the particular implementation of eye rotations in this script, looking directions are consistently viewed as being represented by the eyes' forward directions, and the definition need not be changed if this particular script is adapted for other character models.

FacialExpressionController.cs

This script implements facial expression definitions (e.g., a "smiling" value of 0.4 on a scale from 0 to 1) onto the specific character. In this example using MakeHuman characters, smiling is realized based on a range of hypothetical joints (somewhat misleadingly named 'bones' in computer graphics lingo) inside the skull. Other character models such as those by iClone Character Creator instead use 'blendshapes', which are definitions for how the character's skin is warped and more closely mimic the effects of facial muscle activity. Smiling may be implemented using a single blendshape or a combination of more fine-grained blendshapes. In a typical social VR setup, a larger number of facial expressions may typically be implemented along the same logic as in this example.

CurrentReferencePoint.cs

Here we merely store the current reference point for this character, i.e., the side of the table on which it is sitting. When connecting to the server, the reference point is updated in ClientSetReferencePoint, making sure that only one player sits at each side of the table. The specifics of this script may not be directly relevant when setting up a social VR environment. While it is possible to represent all data relative to the reference point (e.g., to directly compare when users are leaning forward or backward with respect to their side of the table), in this example data are generally handled in global space, and for remote players the reference point is not even directly known.

Control Own Player

These scripts define how a character should move its head and eyes or smile, and transfer data to CharacterBehaviorController.cs which organizes their implementation onto the character. In a social VR setup, data would come from sensors (e.g., eye tracker, head tracker, face tracker). The scripts in this folder therefore only serve to provide a functional example with no need for VR hardware and need to be replaced with the respective VR APIs when transferring the project to a social VR use.

ControlSelf.cs

Data from the headset, the eyes, and facial expression (smiling) are transferred to the character's CharacterBehaviorController which implements data on the character model. For this example the headset, eyes and facial expression are being controlled without VR equipment in the scripts MockMotionTrackingAPI, MockEyeTrackingAPI and MockFaceTrackingAPI, but data transfer to the CharacterBehaviorController are identical when data are instead streamed from actual eye-tracking and face-tracking APIs.

MockMotionTrackingAPI.cs

This script controls the headset to mimic a VR setups. It can return the headset's local up direction much like a motion tracking API used in VR does. In this setup the headset's orientation is sufficient to define its position as well since the head is centered to match the tracked eyes as defined by the eye-tracking API. The specifics of how the mock headset and eyes are being controlled in this example may not be directly relevant when setting up a social VR environment but serve to provide a functional example which can be directly tested even on a single computer without putting into operation VR systems and local networks. In short, the character is controlled by (1) directing its eyes towards a look-at target which is being towards the location pointed at by the mouse cursor in another script (MouseCursorRaycaster), (2) slightly rotate the head towards that direction as well and (3) moving the head laterally by pressing the arrow keys.

MockEyeTrackingAPI.cs

This script mimics an eye-tracking API in that it returns both eyes' positions and rotations. Here we also orient the eyes towards a lookAtTarget. Note that when communicating with an actual eye-tracking API, the process by which it determines each eye's position and orientation is typically not of interest.

MockFaceTrackingAPI.cs

This script controls the character's facial expression to mimic a VR setups. It can return its current smiling on a scale from 0 to 1 much like a face-tracking API does. The specifics of how the facial expression is being controlled in this example may not be directly relevant when setting up a social VR environment but serves as an example. Here, the character smiles when the space bar is being pressed.

MouseCursorRaycaster.cs

A simple script which sets the character's look-at target to where the mouse cursor is pointed on the screen, allowing to simulate head movements.

Network Scripts

While the scripts above only allow to control a character sitting on a table, the network scripts manage connections with the sever, sending of movement data to the server and receiving data from other players as well as their implementation on respective character models in the scene.

StartNetworkServer.cs

The server starts its networking activity. This process does not need any parameters; the server merely becomes reachable to other clients who have its ip address. Authentication is granted to any client who manages to reach the server. This scenario is sufficient for a setup where server and client are located on the same local network. If communication is instead carried out over the internet, additional authentication measures should be taken to avoid potential attacks.

StartNetworkClient.cs

The client starts an attempt to connect to the server, and automatically tries to reconnect if the connection is lost. This script represents the case where the server's ip address is known to the client. When running all instances of the software (server and at least one client) on the same computer, the connection can be established by merely using "localhost" as server ip address. When the server and the client are on different computers but are connected in the same local network, the server's ip address must be inserted here. It can be obtained simply by opening the cmd and typing "ipconfig". The IPv4 address should work fine. However, note that automatically assigned ip addresses can change from time to time. The simplest way to ensure a correct connection between the clients and the server throughout a research study is to assign a static ip address for the server computer.

MyBroadcasts.cs

A collection of basic message types, called broadcasts here, which can be exchanged between clients and the server. Each message type defines what sort of data it can hold, similarly to a form. In this example we only need four types of broadcasts: a message for spawning, a message for despawning, a message to update the reference point, and a message to update the current player state (position, looking direction etc.). Only rather basic data types can be included in a broadcast (specifically, data types which can be serialized/deserialized by Fish-Net's own serializer/deserializer). For example, we can send floats, Vector3s or strings. More complex structures need to be serialized using external serializers (e.g., Odin Serializer) and included as raw bytes in a broadcast.

ServerLogicSimple.cs

Server functionality to organize relaying of information between clients. The server has only few responsibilities in this example: (1) relay information of a new client's spawning to existing players, (2) inform new player of the existence of players who have already been connected, (3) tell each new player where to sit on the table, i.e., its reference point, (4) relay player states between all active players and (5) inform remaining players when a player has disconnected.

The script is called ServerLogic*Simple* because this functionality represents the core of a networked VR setup. The server software could therefore be a small and lightweight program which may easily be hosted on a server structure, although in most laboratory use cases it is even easier to run the server along with one of the clients. Such a relatively simple server software is entirely sufficient to carry out a range of social VR experiments in a laboratory environment, but can be extended in several ways. For example, a server may (1) not only relay data, but also represent the scene which the clients are seeing, thus creating a spectator view of the social situation, (2) note down data to disk, which is currently done on the client side or (3) manage the synchronization of additional objects which can be moved in the scene, including an authority management, i.e., organizing who is able to move which object. With increasing complexity the server logic will typically be spread across several scripts.

ClientSendOwnInfo.cs

Data describing our own avatar's state are continuously sent to the server which relays them to other clients so that they can update their representation of us accordingly.

ClientHandleRemotePlayers.cs

Data regarding other networked users, which are continuously relayed on the server, are being processed here. A new remote player (the local representation of others) is instantiated when the server announces its

connection. The instantiated remote player consists of the actual character model along with an IK setup and scripts which allow to control character behavior in the same fashion as the character representing the self, in particular a CharacterBehaviorController with which this script directly communicates. A dictionary is kept to store what remote players have been instantiated and to what network id they are assigned. This dictionary allows to move all remote players continuously as updated data is being relayed here from the server. Remote players are destroyed when the server announces their disconnecting.

ClientSetReferencePoint.cs

Our own avatar's reference point is reset along a command from the server which ensures that every reference point (i.e., side of the table) is only used by one client. In social VR, this procedure ensures that interaction partners are really located on different sides of the table, also agreeing on who is where and consistently placing everyone in the same location in the scene in all simulations. While the server dictates which client is to use which reference point, clients themselves are responsible for positioning themselves relative to this reference point. An alignment of the physical world with the designated reference point in VR is implemented elsewhere, in VV_Calibration.

Save Data

These scripts set up logfile writers which save data to disk either in a raw format (character state data with no specific highlighting of who is looking at whom) or in a preprocessed data (with eye gaze data transposed with regards to their horizontal and vertical deviations from points of interest). It is here that is may become clear how using a standardized format for referencing both one's own and others' characters simplifies the storing of data in an organized format for subsequent data analysis.

SaveRawData.cs

Noting down state data of the own as well as other's avatars continuously. Data from all characters are saved to the same file in the same format, with one line per character per save time point. Therefore, if others connect or disconnect, we do not obtain new files but there are merely new lines added to each time point. Data storage is carried out on the client here, implying that every client saves all data. This procedure allows to additionally assess concordance between simulations on different client computers. Data storage may be switched to the server software if this feature is not required.

Recorded data incorporate all relevant information but are not yet preprocessed in any way: for instance, while we record where each character is positioned at each moment and where each character is looking, this does not yet tell us directly who is looking at whom. Additional geometric operations are required for such analyses which can be conducted after data storage (e.g., using Matlab, R or Python). Since they are often computationally efficient, they can also be conducted on the fly in this program (see SaveProcessedData) so that stored data can be more directly analysed. Note that most operations in data storage are comparatively lightweight to modern computers (e.g., collecting data from the CharacterBehaviorController component, communicating with the disk itself). A potentially problematic process is string concatenation which produces garbage, which is why the script makes heavy use of StringBuilders.

SavePreprocessedData.cs

Noting down data which are preprocessed for direct analysis. In particular, gaze data are preprocessed with regards to a partner's eyes: For each eye, we note the degrees in the horizontal and vertical direction by which its looking direction misses each of the partner's eyes. This information can be inferred from the data stored by SaveRawData but is efficiently computed in C#, so may likewise be noted down directly.

Depending on the research questions, researchers may note down both raw data and preprocessed gaze data or only one of the two.

GeometryHelpers.cs

Social VR projects may typically involve processing of geometry in 3D. While some functions are specific to one context and they may be more suitably noted in the specific script, others may represent more generic operations and can be noted here.

DissonanceTools

While Dissonance Voice Chat can be directly integrated with Fish-Net Networking using its custom integration with no coding, features such as logfile writing require access to and a manipulation of underlying structures. These scripts can be used after integrating Dissonance into the project and will throw errors when Dissonance is not installed. They are therefore not included in the Unity project under Assets/Scripts but located in an extra folder in the Github repository.

DissonanceManager.cs

Here we keep references of each player's representation in Dissonance voice chat (VoicePlayerState) in order to be able to further process these data (e.g., write them to disk) elsewhere. Importantly, Dissonance is started manually here which allows to link each player's representation with Fish-Net's own id; we would otherwise create the counter-intuitive situation where behavioral and voice data are correctly displayed on each character in the scene, but the logfiles do not allow to relate data streams to each other (e.g., to tell if speech data associated with a specific id represents the self or another person).

To implement this script follow these steps: (1) download and import Dissonance Voice Chat (https://placeholder-software.co.uk/) and the integration for Fish-Net Networking (https://github.com/LambdaTheDev/DissonanceVoiceForFishNet), (2) add all objects to the scene as described in the respective manuals (3) add this script to the object with the DissonanceComms on it, (4) disable DissonanceComms, (5) in VoicePlayback.cs, manually delete "UpdatePositionalPlayback();" (which otherwise sets spatial blend to 0 in this setup, corrupting spatial sound).

While other scripts in this project are organized along whether they execute logic on the client or the server, this script mostly describes client logic but in one instance describes server logic as well. Nothing needs to be changed if you use this script in a host or a client, but see "ServerManager_OnServerConnectionState" if you plan to build a pure server (with no client running in the same program).

SaveSpeechData.cs

Noting down who is speaking at each moment in time and at what amplitude. All data are received from DissonanceManager which started dissonance and keeps track of all players. Importantly, DissonanceManager uses Fish-Nets ids rather than Dissonance's own ids, so that data can be linked with gaze and other behavioral data stored in other scripts (SaveRawData, SavePrepreprocessedData). Similarly as in other logfile writers, data from all users are stored in the same csv file, with one line in the same format for each user at each point in time.