AvatarTransferPro: A Tool for Automatic Transfer of Realistic Personalized Avatars in Networked Social Virtual Reality *

Marius Rubo *University of Bern*

AvatarTransferPro is a software tool for Unity which allows to synchronize the appearance of character models over a networked connection. The use case in mind are Social Virtual Reality (Social VR) applications where users may want to embody avatars which resemble their physical appearance. Many such setups are limited to a set of characters which come with the software pre-installed and may allow for some, but not extended customization. AvatarTransferPro provides tools to store and send all relevant data in characters and apply these data to characters on other computers. This allows for uses cases where a highly personalized character is created on one computer using specified software at any time (in particular after the building and installation of the Social VR software), and is then transmitted to other computers in the Social VR network on the fly.

Keywords: Social Virtual Reality, Personalized Avatars, Computer Characters, Serialization/Deserialization

Contents

*Version: 1.0, Release Date: June 20, 2024. Contact: marius.rubo@gmail.com, marius.rubo@unibe.ch

Overview

Features

- Data structures and functions to hold all relevant data in a character by value (rather than by reference)
- Serialization of character data into a byte[] array and storage in the StreamingAssets folder
- Reading of stored data files, deserialization of byte[] array back into character data, and application onto an existing character
- Example on data transmission in chunks as in network solutions
- Application of new data to existing character largely run async to avoid blocking the main thread
- Includes approaches to reduce data file size (compression, reduction of texture sizes for peripheral objects such as the shoes)
- Designed for easy integration and extendability in existing Social VR setups

System Requirements

- Unity 2021 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 a networking solution but mimics behavior in a networking solution to exemplify its use. Customized integration into your own networking environment (e.g., Fish-Net, Ubiq) is required.

License

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

Acknowledgements

- This software uses the FreeImage open source image library. See http://freeimage.sourceforge.net for details. FreeImage is used under the GNU General Public License v3.0

- This software uses the OdinSerializer open source serializer. See https://github.com/TeamSirenix/odin-serializer/ for details. OdinSerializer is used under the Apache License 2.0.
- 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.
- This software uses the TextMeshPro text solution for Unity. See https://docs.unity3d.com/Manual/com.unity.textmeshpro.html for details. TextMeshPro is used under the Unity Companion License for Unity-dependent projects.

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

The overall goal of the examples is to provide a minimal example for how an avatar is transformed into another avatar (see Figure 1), bypassing the need for a character import using the Editor, keeping all references of a character intact and thereby allowing built executable software to receive additional characters with no need to adapt any source files. Note that while the tool is written with *avatars* in mind (i.e., computer characters which represent a person), the example project uses the more general term *character*. Three example scenes demonstrate how (1) data describing a character's appearance are retrieved from an imported character and stored to disk in a dedicated format (scene *SaveCharacterData.unity*), (2) such data stored on disk is applied directly to an existing character (scene *ApplyCharacterDataSimple.unity*) and (3) how two scripts interact in reading data from disk, transfer them in small chunks, reconstruct and apply them to an existing character, mimicking the typical scenario in networked social VR (scene *ApplyCharacterDataMockNetwork.unity*).

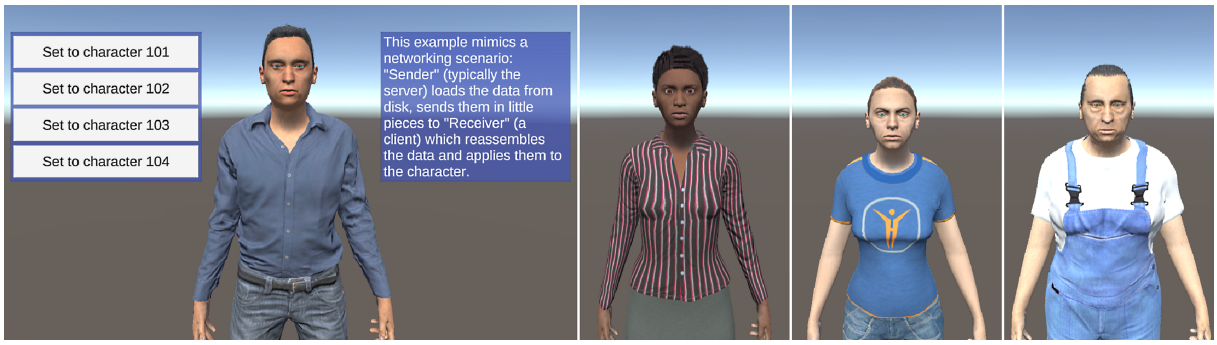

Figure 1: The character on the left is transformed into one of the characters on the right on the fly with no need for character import and leaving all references intact. The examples uses relatively simple open-source characters, but the tool likewise handles higher quality characters.

Example Scene 1: SaveCharacterData.unity

This scene shows how data from an imported character are retrieved, serialized and stored to disk in a dedicated file format within the StreamingAssets folder. Unlike the sending of and applying of new character data, which can be performed by executable software on the fly, importing of characters and, most conveniently, storing of character data is carried out in the Unity Editor. In a typical research scenario, the

creation of personalized avatars as well as its storage will be carried out on a computer belonging to the research laboratory where the Unity Editor is installed. As shown in Figure 2, the script *SaveCharacterData.cs* only needs to reference the character's root object and takes in an identifying number which will determine the resulting file's name.

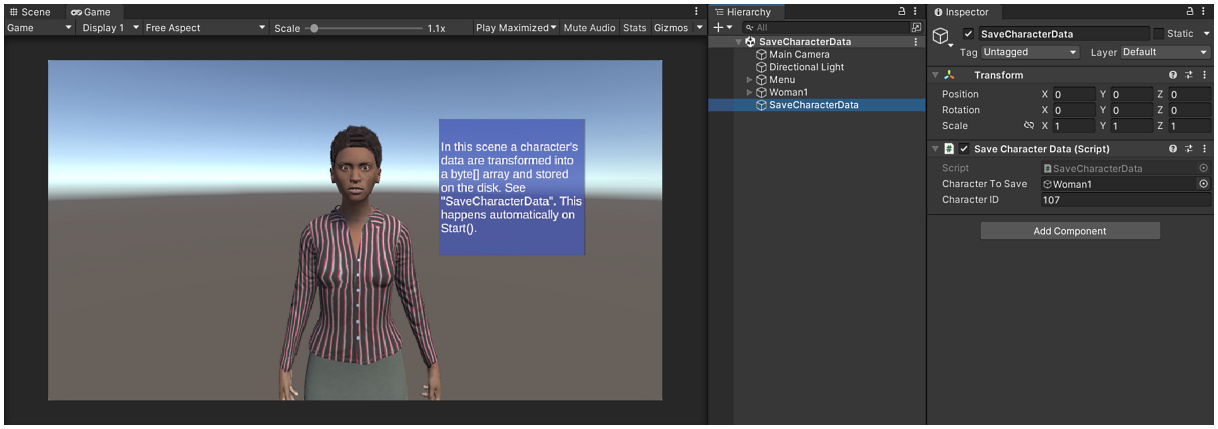

Figure 2: Example scene *SaveCharacterData.unity* where data defining a character's appearance are stored to disk in a dedicated file format.

Example Scene 2: ApplyCharacterDataSimple.unity

This example shows how data files created in *SaveCharacterData.unity* are read from disk and applied to an existing character (see Figure 3). The script *ApplyCharacterData.cs* only needs to reference the imported character's root object. The menu buttons then initiate the script's loading and applying of new character data. Note that while this example shows the procedure of transforming characters in its simplest form, it does not directly correspond to a use case where characters are exchanged over a network.

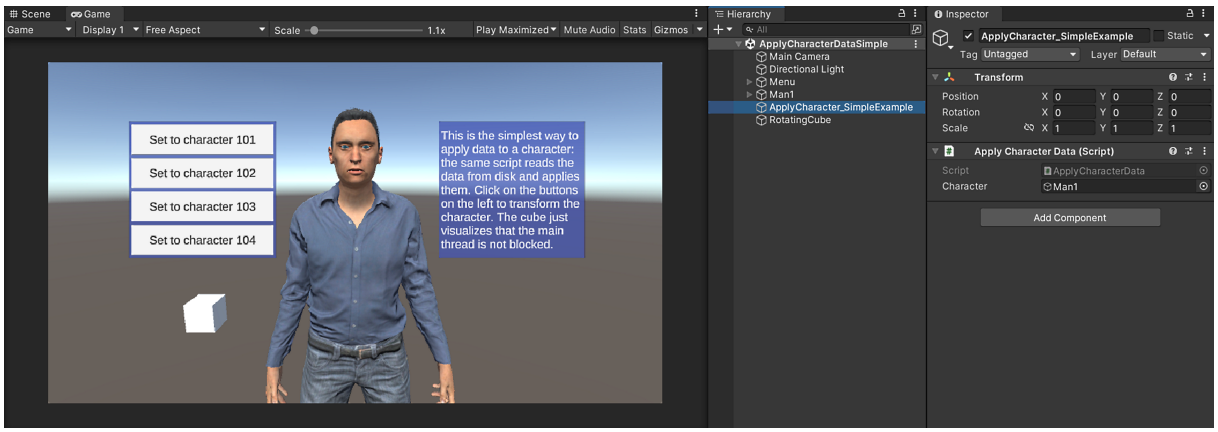

Figure 3: Example scene *ApplyCharacterDataSimple.unity* where a character is transformed into another one in a simple setup, highlighting the tool's core functions but not corresponding to a social VR use case.

Example Scene 3: ApplyCharacterDataMockNetwork.unity

This example extends the previous example *ApplyCharacterDataSimple.unity* in that the transfer of character data is organized in a structure which mimicks a typical networked scenario as in social VR (see

Figure 4). The *Sender* reads character data from disk and sends these data in small chunks to the *Receiver*, which reassembles them, deserialized them and applies them to the existing character. The *Receiver* again needs to reference the existing character's root object while the *Sender* references the *Receiver*, a reference corresponding to an established network connection with defined message types in social VR scenarios.

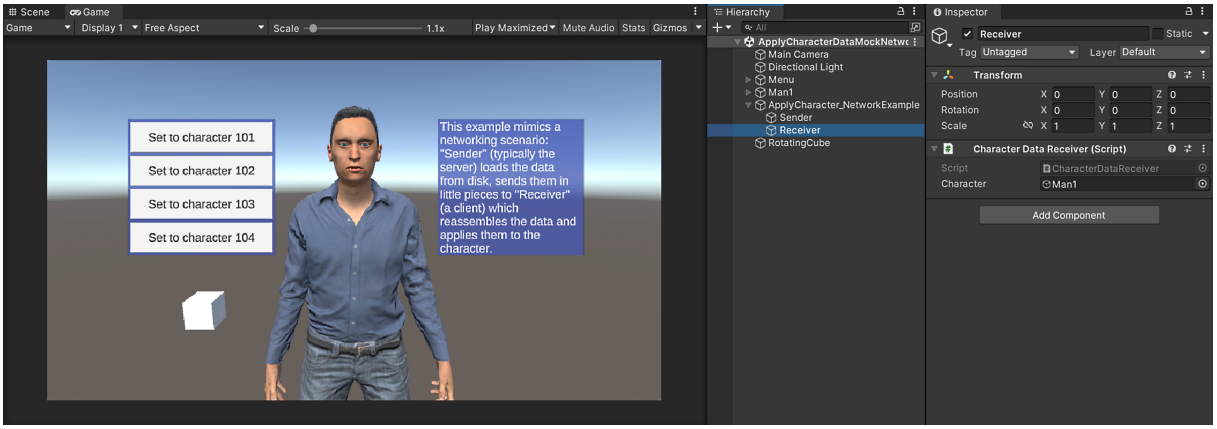

Figure 4: Example scene *ApplyCharacterDataMockNetwork.unity* where the transfer of character data is structured along a typical use case in the context of social VR.

Implementation

The examples demonstrate how the tool is used to transfer novel character data onto an existing character on the fly. Using the tool in an own social VR software requires to implement it with existing structures. AvatarTransferPro is designed to function as an optional extension, leaving existing references intact and allowowing to be switched on and off with no changes needed to other parts of the social VR software.

Integrate with own Social VR Software

AvatarTransferPro can be implemented in social VR software where developers have explicit control over each character's references and can send and receive raw byte data in arbitrary structures. These prerequisites are fulfilled when using dedicated networking solutions such as *Fish-Net networking* (https://github.com/FirstGearGames/FishNet/) or *Ubiq* (https://github.com/ubiq/) but may be more challenging to fulfill when using higher-level social VR templates and can typically not be fulfilled when using existing commercial social VR applications. In a typical use case, the *Sender* in the example scene *ApplyCharacterDataMockNetwork.unity* will be located on a server where a character was imported. Rather than directly calling a function on the *Receiver*, which will typically be located on the clients, data transfer will implemented by means of messages (also called *Broadcasts* in Fish-Net Networking). The rest of the code will continue to work without further adaptations, but may need adaptations depending on the employed characters.

Adapt and Extend

Several situations may require to adapt or extend AvatarTransferPro to directly comply with affordances in a specific social VR software. For example, if characters contain parts not covered here such as sunglasses or scarfs, these must be explicitly referenced. Designers of a social VR environment may furthermore choose to incorporate additional textures or adapt texture resolutions or settings. A description of the scripts

below, but also comments in the scripts themselves give guidance on how to implement such adaptations or extensions.

Scripts

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

CharacterExchange

These scripts hold structures and functions to handle data characters, forming the core of the tool.

CharacterData.cs

The center piece of the software: a class which holds all relevant data of a character by value rather than by reference. Instances of this class can then be serialized into byte[] and be sent over a network. CharacterData consists of instances of three other, more low-level classes which hold data of the skeleton (SkeletonUtilities.BonesData), data of a Mesh of a SkinnedMeshRenderer (MeshUtilities.SkinnedMeshData) and data of the materials of a SkinnedMeshRenderer (MaterialUtilities.MaterialsData). This script furthermore stores a function which applies new CharacterData to an existing and referenced character (ApplyCharacterDataToCharacter).

CharacterReference.cs

A class which stores all relevant parts of a character by reference, i.e., links to those parts. This centralized reference helps to (a) obtain the relevant data by value which are then stored in CharacterData and (b) apply CharacterData onto a character. Note that the character's parts are searched here by name, so the script only works if the names meet the expectations. Common workflows are to either rename all the GameObjects manually to fit the description or to implement different strings in this script to directly match the names in characters as they are exported from the specific character creation tool being used. Note that some character creation tools are not consistent in the naming of parts, e.g., "Shoes" can be called differently depending on what sort of shoes they are. Make sure your approach covers all use cases.

SkeletonUtilities.cs

The Mesh of a SkinnedMeshRenderer is tied to an underlying skeleton structure where the relative positions and rotations of all specified joints are stored (e.g., the position and rotation of the left elbow relative to the left shoulder). Therefore the positions and rotations of these joints, often referred to as bones, must be transmitted when a character is recreated.

MeshUtilities.cs

Functionality to handle data of a mesh by value. A mesh consists of a range of data structures. At its core, one may argue, is the Vector3[] array of the vertices which define the overall shape of the object. In addition, the mesh is linked to bones in the skeleton, is partitioned into submeshes, stores boneweights and bindposes which define how it is streteched when the underlying skeleton moves, can have blendshapes etc. These data structures are all stored here by value so that they can be serialized/deserialized and applied to another SkinnedMeshRenderer.

MaterialUtilities.cs

Functionality to handle data on materials by value. Here we only handle the textures themselves and assume that additional parameters (e.g., glossiness) remain unchanged among different characters of the same type. Also we limit ourselves to handling three textures on a material. Realistic characters often come with more textures per material but may only show marginal improvements in visual appearance when using more than three textures, which may not justify the increase in data transfer. Textures can be easily added if needed. This script makes use of the FreeImage project to perform texture processing operations async which would otherwise result in noticable fps hickups.

TextureProcessing.cs

Low-level functions which process texture data.

Customization.cs

At least one parameter in the process will likely need customization: the names of the individual textures on each material. Therefore, this info is stored in this specific script. For more complex situations, an external database may be more appropriate. This script should work well without modification when using Unity in its Standard Render Pipeline and with all materials using the Standard Shader. More high-quality character creation tools will, however, commonly provide different shaders for different materials (i.e., one shader for the skin, one for the eyes' cornea etc.). The trouble is that textures are named differently in different shaders and we want control over what textures we extract, store and apply. For instance, the albedo map is called "_MainTex" in the Standard Shader but may be called "_DiffuseMap", "_BaseMap" etc. in other shaders. Note that a second area of customization may be the names of all the character's parts which are used in CharacterReference. Since it may be most straightforward to rename them manually on the character itself in most situations, I did not include this here under Customization, but that might be an appropriate extension for some use cases.

Examples

These examples demonstrate how to use the core functionality of the AvatarExchange scripts. They are used in the example scenes.

SaveCharacterData.cs

Saving CharacterData to disk. Relevant data in a specified character are referenced, transformed to CharacterData (which stores information by value), serialized, compressed and stored to disk in its own data format (chardat = character data).

ApplyCharacterData.cs

Simplest example for how data are loaded and applied to a character. Here, everything is done in the same script. in a real-world situation, this would be the case when a client already has the relevant character data stored on its own disk and the server tells it "Please load character data 102 and apply it to character X". In other cases, the client may not have the relevant character data yet but needs to receive them from the server first. For this scenario, see "CharacterDataSender" and "CharacterDataReceiver" and how they are used in the example scene "ApplyCharacterDataMockNetwork".

CharacterDataSender.cs

Character data are read from disk, split into chunks and transmitted to another script. This example shows in priniciple a use case where one computer (typically the server) has character data stored on a disk and sends it to other computers (the clients) which then apply it. For a simpler version of applying new character data to a character, see "ApplyCharacterData" used in the example scene "ApplyCharacterDataSimple".

CharacterDataReceiver.cs

Character data are received in small chunks, reassambled and applied to a character. This example shows in priniciple a use case where one computer (typically the server) has character data stored on a disk and sends it to other computers (the clients) which then apply it. For a simpler version of applying new character data to a character, see "ApplyCharacterData" used in the example scene "ApplyCharacterDataSimple".

MockNetworking

This scripts are used to mimic data transfer within an established networked connection where data can typically only be sent and received in relatively small chunks.

DataChunker.cs

Functions to split larger byte[] arrays into chunks (which are small enough to be sent over a network) and to reassamble the chunks back to the original array.

Message.cs

A mock version of "Messages" as they are used in networking. Used here to show in priniciple how character data can be sent within networking solutions. Alternatively, one can also send the entire byte[] array outside of the netcode, e.g., via HTTPS which is likewise secure and likewise does not block the main thread. Using an established network may just be more convenient in some situations.

BasicDataUtils

Generic scripts to handle data as it could be implemented in any project.

Gzip.cs

Convenience wrapper functions to gzip compress and decompress byte arrays without blocking the main loop. Gzip itself is built-in natively in .NET.

IO.cs

Convenience wrapper functions to read files from disk and write without blocking the main loop.